4 Programación Orientada a Objetos

1. Introducción

La Programación Orientada a Objetos (***POO***) es una técnica de programar aplicaciones basada en una serie de objetos independientes que se comunican entre sí.

A Java se le considera un lenguaje orientado a objetos ya que siempre que se crea un programa en Java, por simple que sea, se necesita declarar una clase, y el concepto de clase pertenece a la programación orientada a objetos.

Un ***objeto*** es un elemento del programa que integra sus propios datos y su propio funcionamiento. Es decir, un objeto está formado por datos (atributos o propiedades) y por las funciones que es capaz de realizar el objeto (métodos). Esta forma de programar se asemeja más al pensamiento humano. La cuestión es detectar adecuadamente los objetos necesarios para una aplicación. De hecho hay que detectar las distintas clases de objetos.

Una ***clase*** es lo que define a un tipo de objeto. Al definir una clase lo que se hace es indicar como funciona un determinado tipo de objeto. Luego, a partir de la clase, podremos crear objetos de esa clase, es decir, la clase es como un molde a partir del cual se crean los objetos que pertenecen a ella. Realmente la programación orientada a objetos es una programación orientada a clases. Es decir, lo que necesitamos programar es como funcionan las clases de objetos.

Por ejemplo, una clase podría ser la clase *Coche*. Cuando se defina esta clase, indicaremos los atributos o propiedades (como el color, modelo, marca, velocidad máxima,...) y los métodos (arrancar, parar, repostar, acelerar, frenar...). Todos los coches, es decir, todos los objetos de la clase *Coche*, tendrán esas propiedades y esos métodos. Para explicar la diferencia entre clase y objeto:

- la clase *Coche* representa a todos los coches.
- un coche concreto es un objeto, es decir, un ejemplar de una clase es un objeto. También se le llama a los objetos *instancias* de la clase. Este término procede del inglés, *instance*, que realmente significa ejemplar.

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearían tantos objetos casilla como casillas tenga el juego. Lo mismo ocurriría con las fichas, la clase ficha definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc), luego se crearían tantos objetos ficha como fichas tenga el juego.

2. Creación de clases

Una clase sirve para definir una serie de objetos con propiedades (atributos), comportamientos (métodos) y semántica comunes. Hay que pensar en una clase como un molde para crear objetos.

La definición de una clase incluye lo siguiente:

- El nombre o identificador de clase. Debe empezar con letra mayúscula y seguirle letras minúsculas, y si consta de varias palabras, se utiliza la notación UpperCamelCase. También pueden contener números pero no como primer carácter. Por ejemplo, `1Coche` no es un identificador válido de clase, pero `Coche1` sí sería válido. Veamos algunos consejos a la hora de elegir los identificadores de las clases:
    - Evitar abreviaturas a favor de la legibilidad del código. Es muy importante que el nombre de las clases sea claro y simbolice perfectamente al tipo de objetos que representa.
    - Evitar nombres excesivamente largos. Aunque parece que se contradice con la norma anterior, se trata de que los nombres sean concisos. No es conveniente que sean descripciones de clase, para eso ya están los comentarios javadoc.
    - Utilizar nombres ya reconocidos. Hay abreviaturas reconocidas como por ejemplo TCP, por eso el nombre de clase `ManejadorTCP` es mejor que `ManejadorProtocoloControlTransmision`.
- Los atributos, también llamados propiedades o campos. Los atributos son variables que poseerá cada objeto de la clase y por lo tanto marcarán el estado de los mismos. Por ejemplo, un coche puede estar parado, en marcha, estropeado, funcionando, sin gasolina, etc. El estado lo marca el valor que tengan los atributos del objeto.
- Los métodos. Son las acciones que pueden realizar los objetos de la clase, es decir, lo que determina el comportamiento de los objetos.

En Java, cada clase se define en un archivo. Además, el nombre de la clase y el del archivo tiene que ser el mismo. Es decir, si queremos definir la clase *Vehicle*, tendremos que hacerlo en un archivo llamado *Vehicle.java*. Dicho archivo contiene la definición de la clase *Vehicle*:

```
class Vehicle {
    int wheelCount; //Número de ruedas
    double speed;   //Velocidad
    String colour;  //Color del vehículo
    void accelerate(double amount){ //acelerar
        speed += amount;
    }
    void brake(double amount){ //frenar
        speed -= amount;
    }
}
```

3. Creación de objetos

Una vez definida la clase, ya se pueden crear objetos de la misma. Para crear un objeto, hay que declarar una variable cuyo tipo será la propia clase:

```
Vehicle car; //car es una variable de tipo Vehicle
```

Una vez definida la variable, se le crea el objeto llamando a un método que se llama constructor. Un constructor es un método que se invoca cuando se crea un objeto y que sirve para inicializar los atributos del objeto y para realizar las acciones pertinentes que requiera el mismo para ser creado. El constructor tiene el mismo nombre que la clase y para invocarlo se utiliza el operador **new**.

```
car = new Vehicle(); //Vehicle() es un método constructor
```

También se puede hacer todo en la misma línea:

```
Vehicle car = new Vehicle();
```

4. Acceso a los atributos y métodos del objeto

Una vez creado el objeto, se puede acceder a sus **atributos** de la siguiente manera: `objeto.atributo`.

```
car.wheelCount = 4;//Se le asigna 4 al atributo número de ruedas de la variable
car
```

Los **métodos** se utilizan de la misma forma que los atributos, a excepción de que los métodos poseen siempre paréntesis ya que son funciones que pertenecen a un objeto: `objeto.método(argumentos)`.

```
car.accelerate(30);
/*El coche incrementa su velocidad en 30. Es decir, si iba a 90km/h, después de
ejecutar el método el coche va a 120km/h */
```

5. Modificadores de acceso

Los modificadores de acceso son palabras reservadas del lenguaje Java que determinan ámbitos de visibilidad de los atributos y métodos de una clase.

Los modificadores de acceso son los siguientes:

- ***Private***: el modificador private en Java es el más restrictivo de todos, cualquier elemento de una clase que sea privado puede ser accedido únicamente por la propia clase. Ninguna otra clase, sin importar la relación que tengan, podrá tener acceso a elementos privados.
- El modificador por defecto: Java nos da la opción de no usar un modificador de acceso y al no hacerlo, el elemento tendrá un acceso conocido como *default* o **friendly**, que permite que tanto la propia clase como las clases del mismo paquete accedan a dichos elementos.
- ***Protected***: lo veremos en el *Tema 7 Herencia 2 Modificadores de acceso* donde se explica el funcionamiento de la herencia en Java.
- ***Public***: el modificador de acceso *public* es el más permisivo de todos, nos permite acceso a los elementos desde cualquier clase incluso de otros paquetes.

Veamos una tabla que resume el funcionamiento de los modificadores de acceso en Java:

	Private	Friendly	Public
Misma clase	X	X	X
Mismo paquete		X	X
Otro paquete			X

El modificador se indica:

- Delante de *class* en la clase.
- Delante del tipo de datos en los atributos y métodos.

Veamos la clase *Vehicle* con los modificadores de acceso:

```
public class Vehicle { //Clase pública

    //Atributos privados:

    private int wheelCount;
    private double speed;
    private String colour;

    //Métodos públicos:

    public void accelerate(double amount){
        speed += amount;
    }
    public void brake(double amount){
        speed -= amount;
    }

}
```

6. Sobrecarga de métodos

Java admite sobrecargar los métodos, es decir, crear distintas variantes del mismo método con el mismo nombre pero que se diferencien en el orden, tipo o número de los parámetros.

Por ejemplo, tenemos el método para sumar `add(int x,int y)`:

- No podríamos definir otro método `add(int a,int b)` porque no varía el tipo ni el número de parámetros.
- Sí podríamos definir `add(int a)` y `add(int a,int b,int c)` porque el número de parámetros varía.
- También podríamos definir `add(int x,double y)` porque aunque no varíe el número de parámetros, sí varía uno de los tipos.

Otro ejemplo donde tenemos el método `add(int x,double y)`:

- Sí podríamos definir el método `add(double x,int y)` porque varía el orden de los parámetros.

El tipo resultado de los métodos sobrecargados puede ser igual o diferente.

7. Atributos, métodos y bloques estáticos

El concepto de estático es que pertenece a la clase, por lo tanto, puede ser accedido o invocado sin la necesidad de tener que instanciar un objeto de la clase. Se indica con la palabra clave **static**.

Los atributos y métodos asociados a los objetos se les conoce como dinámicos.

Los **atributos estáticos** son variables que pertenecen a la clase y son compartidos por todos los objetos de la clase. Son inicializados en el momento en que se carga la clase en memoria, respetando el orden de declaración. Para acceder a un atributo estático, hay que indicar el nombre de la clase: `Clase.AtributoEstático`

Los **métodos estáticos** son métodos que pertenecen a la clase y no al objeto por lo que pueden llamarse sin tener que crear un objeto de dicha clase. Para llamar a un método estático, hay que indicar el nombre de la clase: `Clase.MétodoEstático(Argumentos)`. Un método estático solo puede acceder a datos estáticos y llamar a métodos estáticos. No pueden utilizar el operador *this* ni *super* ya que son conceptos dinámicos. Los métodos estáticos es lo más parecido a lo que son las funciones en los lenguajes estructurados con la diferencia que se encuentran encapsulados en una clase.

Cada vez que creamos un programa en Java debemos especificar el método *main*:

```
public static void main(String[] args)
```

El método *main* es estático para que la máquina virtual de Java pueda llamarlo directamente sin tener que crear un objeto de la clase que lo contiene.

Ejemplos de llamadas a métodos dinámicos y estáticos:

- Llamada a método dinámico: `car.accelerate(30);`
- Llamada a método estático: `Math.pow(2,3);`

El **bloque estático** es un bloque de instrucciones dentro de una clase Java que se ejecutará cuando una clase se cargue por primera vez en la JVM.

```
class MyClass{
    ...
    static{ //Comienzo del bloque estático
        ...
        //Instrucciones
        ...
    }//Fin del bloque estático
    ...
}
```

Ejemplo:

```
package tema4_POO.estatico;

public class MyClass {

    static int a;
    static int b;

    static {
        a = 10;
        b = 20;
    }

}
```

```
package tema4_POO.estatico;

public class Main {

    public void showStatic() {

        System.out.printf("Valor de a: %d", MyClass.a);
        System.out.printf("\nvalor de b: %d", MyClass.b);

    }

    public static void main(String[] args) {

        new Main().showStatic();

    }

}
```

Salida por consola:

```
Valor de a: 10
Valor de b: 20
```

8. JavaBean

Los JavaBeans son un modelo de componentes para la construcción de aplicaciones en Java. Se usan para encapsular varios objetos en un único objeto (la vaina o Bean en inglés), para hacer uso de un solo objeto en lugar de varios más simples. La especificación de JavaBeans los define como "componentes de software reutilizables que se puedan manipular visualmente en una herramienta de construcción".

Para funcionar como una clase JavaBean, una clase debe obedecer ciertas convenciones sobre nomenclatura de métodos, construcción y comportamiento. Estas convenciones permiten tener herramientas que puedan utilizar, reutilizar, sustituir y conectar JavaBeans.

Las convenciones requeridas son:

- Debe tener un constructor sin argumentos.
- Sus atributos de clase deben ser privados.
- Sus propiedades deben ser accesibles mediante métodos *get* y *set* que siguen una convención de nomenclatura estándar.
- Debe ser serializable.

Dentro de un JavaBean podemos distinguir tres partes:

- Propiedades: Los atributos que contiene.
- Métodos: Se establecen los métodos *get* y *set* para acceder y modificar los atributos.
- Eventos: Permiten comunicar con otros JavaBeans.

9. Métodos get y set

Los métodos *get* y *set* son métodos de las clases para mostrar o modificar el valor de un atributo. Para mostrar se utiliza el método *get* y para modificar el método *set*.

Según las convenciones JavaBean, la nomenclatura de ambos debe ser la siguiente:

- **get**:
    - Debe ser declarado con el modificador de acceso *public*.
    - El nombre del método comienza con *get* y le sigue el nombre del atributo en UpperCamelCase. En el caso de los booleanos, el nombre del método comienza con *is*.
    - El tipo de retorno del método debe ser el mismo que el tipo del atributo.
- **set**:
    - Debe ser declarado con el modificador de acceso *public*.
    - El nombre del método comienza con *set* y le sigue el nombre del atributo en UpperCamelCase.
    - El tipo de retorno del método debe ser *void*.
    - El tipo del parámetro del método debe ser el mismo que el tipo del atributo.

Hacer uso de este convenio nos facilitará trabajar con el resto del mundo y nos permitirá ampliar las capacidades de nuestro código utilizando frameworks existentes que hacen uso del convenio y que si no seguimos no podremos utilizar.

Es muy importante no utilizar estas nomenclaturas para métodos que en realidad no sean métodos get ni set.

En el Eclipse se pueden generar ambos métodos automáticamente en el menú *Source* → *Generate Getters and Setters*.

Vamos a incluir en la clase *Vehicle* los *getters* y *setters*. Para ello, nos tenemos que plantear qué atributos vamos a permitir consultar y modificar. En cuanto a la consulta, vamos a permitir consultar todos los atributos por lo que tendremos que realizar 3 *getters*. En cuanto a las modificaciones, el número de ruedas de un vehículo no cambia a lo largo del tiempo por lo que no hace falta ponerle un método *set*. La velocidad se va modificando con los métodos de acelerar y frenar por lo que tampoco le vamos a poner un método *set*. En cuanto al color, vamos a permitir que un vehículo pueda cambiar de color por lo que sí vamos a hacerle un método *set* al color.

```
public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public int getWheelCount() { //Método para consultar el número de ruedas
        return wheelCount;
    }

    public double getSpeed() { //Método para consultar la velocidad
        return speed;
    }

    public String getColour() { //Método para consultar el color
        return colour;
    }

    public void setColour(String colour) { //Método para modificar el color
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

Fijémonos en el método *setColour*:

```
public void setColour(String colour) { //Método para modificar el color
    this.colour = colour;
}
```

Tenemos dos variables que se llaman igual: *colour*. Una es un parámetro y otra es un atributo. Para diferenciarlas, el atributo se utiliza con la palabra *this*. El uso de *this* se explica con detenimiento más adelante.

En el caso de los booleanos, el getter comienza por *is*. Si la clase *Vehicle* tuviera un atributo booleano llamado *empty* para detectar cuándo está vacío el depósito de gasolina, el getter se llamaría *isEmpty()*:

```
private boolean empty;
public boolean isEmpty() {
    return empty;
}
```

10. Constructores

Un ***constructor*** es un método que se invoca cuando se crea un objeto y que sirve para inicializar los atributos del objeto y para realizar las acciones pertinentes que requiera el mismo para ser creado. El constructor tiene el mismo nombre que la clase y para invocarlo se utiliza el operador **new**.

```
Vehicle car = new Vehicle();//Vehicle() es un constructor de la clase Vehicle
```

En los constructores no se especifica tipo de retorno:

```
package tema4_POO.constructorPorDefecto;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle() { //No se especifica tipo de retorno
        wheelCount = 4;
        speed = 0;
        colour = "blanco";
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }
```

```
    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema4_POO.constructorPorDefecto;

public class Main {

    public void showDefaultConstructor() {

        Vehicle car1, car2;
        car1 = new Vehicle();
        car2 = new Vehicle();
        System.out.printf("Coche1: %d ruedas y de color %s",
car1.getWheelCount(), car1.getColour());
        System.out.printf("\nCoche2: %d ruedas y de color %s",
car2.getWheelCount(), car2.getColour());
        System.out.printf("\nVelocidad del coche1: %.2f km/h", car1.getSpeed());
        System.out.println("\nAceleramos el coche1 90,50 km/h");
        car1.accelerate(90.5);
        System.out.printf("Velocidad del coche1: %.2f km/h", car1.getSpeed());
        System.out.println("\nFrenamos el coche1 20,30 km/h");
        car1.brake(20.30);
        System.out.printf("Velocidad del coche1: %.2f km/h", car1.getSpeed());

    }

    public static void main(String[] args) {

        new Main().showDefaultConstructor();

    }

}
```

La salida por consola es la siguiente:

```
Coche1: 4 ruedas y de color blanco
Coche2: 4 ruedas y de color blanco
Velocidad del coche1: 0,00 km/h
Aceleramos el coche1 90,50 km/h
Velocidad del coche1: 90,50 km/h
Frenamos el coche1 20,30 km/h
Velocidad del coche1: 70,20 km/h
```

El constructor que no tiene parámetros se llama ***constructor por defecto***, como por ejemplo *Vehicle()*. Si nos fijamos en la salida por consola, los dos coches tienen los mismos datos ya que el constructor por defecto crea todos los objetos de la misma manera. Para poder crear objetos diferentes, tendremos que usar un constructor con parámetros:

```java
package tema4_POO.constructorConParametros;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) { //Constructor con parámetros
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```java
package tema4_POO.constructorConParametros;

public class Main {

    public void showParameterizedConstructor() {

        Vehicle car, moto;
        car = new Vehicle(4, "azul");
        moto = new Vehicle(2, "rojo");
        System.out.printf("Coche: %d ruedas y de color %s", car.getWheelCount(),
car.getColour());
        System.out.printf("\nMoto: %d ruedas y de color %s",
moto.getWheelCount(), moto.getColour());

    }
```

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

        new Main().showParameterizedConstructor();

    }

}
```

Salida por consola:

```
Coche: 4 ruedas y de color azul
Moto: 2 ruedas y de color rojo
```

Al tener parámetros el constructor, esto nos permite poder crear objetos diferentes.

En el Eclipse, se puede generar el constructor con parámetros en Menú *Source → Generate Constructor using Fields*.

Los constructores se pueden sobrecargar, por lo tanto, una clase puede tener varios constructores:

```
package tema4_POO.constructores1;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle() { //Constructor por defecto
        wheelCount = 4;
        speed = 0;
        colour = "blanco";
    }

    public Vehicle(int wheelCount, String colour) { //Constructor con parámetros
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
```

```
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema4_POO.constructores1;

public class Main {

    public void showConstructors1() {

        Vehicle car, moto;
        car = new Vehicle();
        moto = new Vehicle(2, "rojo");
        System.out.printf("Coche: %d ruedas y de color %s", car.getWheelCount(),
car.getColour());
        System.out.printf("\nMoto: %d ruedas y de color %s",
moto.getWheelCount(), moto.getColour());

    }

    public static void main(String[] args) {

        new Main().showConstructors1();

    }

}
```

Salida por consola:

```
Coche: 4 ruedas y de color blanco
Moto: 2 ruedas y de color rojo
```

Si una clase no tiene constructor, Java crea uno por defecto. Los atributos se inicializan a su valor por defecto en función del tipo que posean:

- Tipos numéricos enteros: 0
- Tipos numéricos decimales: 0.0
- Caracteres: carácter nulo (Código Unicode 0)
- Booleanos: false
- Referencias a objetos: null

Solo se inicializan los atributos, las variables locales de los métodos no son inicializadas por defecto.

```java
package tema4_POO.javaConstructor;

public class Vehicle { //Clase sin constructor. Java crea un constructor por defecto

    private int wheelCount;
    private double speed;
    private String colour;

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```java
package tema4_POO.javaConstructor;

public class Main {

    public void showJavaConstructor() {

        Vehicle car;
        car = new Vehicle();
        System.out.printf("Color: %s", car.getColour());
        System.out.printf("\n%d ruedas", car.getWheelCount());
        System.out.printf("\nVelocidad: %.2f km/h", car.getSpeed());

    }

    public static void main(String[] args) {

        new Main().showJavaConstructor();

    }
```

```
}
```

Salida por consola:

```
Color: null
0 ruedas
Velocidad: 0,00 km/h
```

Java solamente crea un constructor cuando la clase no tiene ninguno, pero si la clase tiene un constructor, aunque sea con parámetros, Java ya no crea ninguno por defecto.

```
package tema4_POO.constructores2;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) { //Java no crea el constructor
por defecto
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema4_POO.constructores2;
```

```
public class Main {

    public void showConstructors2() {

        Vehicle moto;
        Vehicle car = new Vehicle();//Error de compilación: el constructor
Vehicle() no está definido
        moto = new Vehicle(2, "rojo");
        System.out.printf("Coche: %d ruedas y de color %s", car.getWheelCount(),
car.getColour());
        System.out.printf("\nMoto: %d ruedas y de color %s",
moto.getWheelCount(), moto.getColour());

    }

    public static void main(String[] args) {

        new Main().showConstructors2();

    }

}
```

Cuando se intenta crear el coche con el constructor por defecto, el compilador nos da un mensaje de error de que el constructor *Vehicle()* no está definido.

11. Visibilidad

La visibilidad de los atributos y métodos de una clase se establece utilizando los modificadores de acceso.

Los modificadores de acceso permiten dar un nivel de seguridad mayor a nuestras aplicaciones restringiendo el acceso a diferentes atributos y métodos asegurándonos que el cliente no va a consultar el valor de un atributo que no debe, no va a modificar incorrectamente el valor de un atributo o que no va a utilizar un método que no le esté permitido. Cuando nos referimos al término cliente es cualquier programador que utilice nuestras clases.

Tal y como vimos en el apartado *5. Modificadores de acceso*, se resume su funcionamiento en la siguiente tabla:

	Private	**Friendly**	**Public**
Misma clase	X	X	X
Mismo paquete		X	X
Otro paquete			X

Veamos un ejemplo de la clase *Vehicle* que está en el mismo paquete que una clase cliente *Main*. Vamos a comprobar la visibilidad que tiene la clase *Main* sobre la clase *Vehicle* en función de los modificadores de acceso. Tal y como podemos observar en la tabla anterior, tendrá acceso a *public* y a *friendly* pero no a *private*.

```java
package tema4_POO.visibilidad;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    int getWheelCount() {//Modificador de acceso friendly
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```java
package tema4_POO.visibilidad;

public class Main {

    public void showVisibility() {

        Vehicle car;
        car = new Vehicle(4, "azul");
        //getWheelCount() es visible porque es friendly y la clase Main está en
el mismo paquete que la clase Vehicle
        System.out.printf("El coche tiene %d ruedas", car.getWheelCount());
        //accelerate(100) es visible porque es public
        car.accelerate(100);
        //Error de compilación: speed no es visible porque es private
        System.out.printf("\nEl coche va a %.2f km/h", car.speed);
```

```
    }

    public static void main(String[] args) {

        new Main().showVisibility();

    }

}
```

Podemos observar lo siguiente en el código anterior:

- El método *getWheelCount()* es visible porque es *friendly* y la clase *Main* está en el mismo paquete que la clase *Vehicle*.
- El método *accelerate(100)* es visible porque es *public*.
- El atributo *speed* no es visible porque es *private*.

12. instanceof

El operador *instanceof* permite comprobar si un determinado objeto pertenece a una clase concreta. Se utiliza de esta forma:

```
objeto instanceof clase
```

Devuelve *true* si el objeto pertenece a dicha clase.

```
package tema4_POO.instanceofObjetos;

public class Main {

    public void showInstanceof() {

        Vehicle vehicle;
        vehicle = new Vehicle(2, "azul");
        System.out.println(vehicle instanceof Vehicle);//true

    }

    public static void main(String[] args) {

        new Main().showInstanceof();

    }

}
```

13. Referencias

Una ***referencia*** en cualquier lenguaje de programación hace alusión a la posición en memoria RAM que tiene una variable.

Una variable es conceptualmente un recipiente donde guardamos un dato. Java es un lenguaje fuertemente tipado. Las variables se definen siempre con un tipo asociado. No se puede meter en una variable ningún dato que no sea del tipo con que se definió.

Una variable es un lugar en la memoria donde se guarda un dato. Para ser exacto, este lugar en la memoria es la *Pila o Stack*. En el caso de los datos primitivos, como en `int i = 5;` hay cuatro bytes en la *Pila* donde se almacena el número 5. Pero cuando se crea un objeto en Java, el objeto se guarda en una parte de la memoria llamada *Heap*. Cuando asignamos el objeto a una variable como en `Vehicle car=new Vehicle();`, lo que guardamos en *car* es la dirección de memoria *Heap* donde está el objeto.

Stack en java es una sección de memoria que contiene métodos, variables locales y variables de referencia. La memoria de pila siempre se referencia en el orden de último en entrar primero en salir. Las variables locales se crean en la pila.

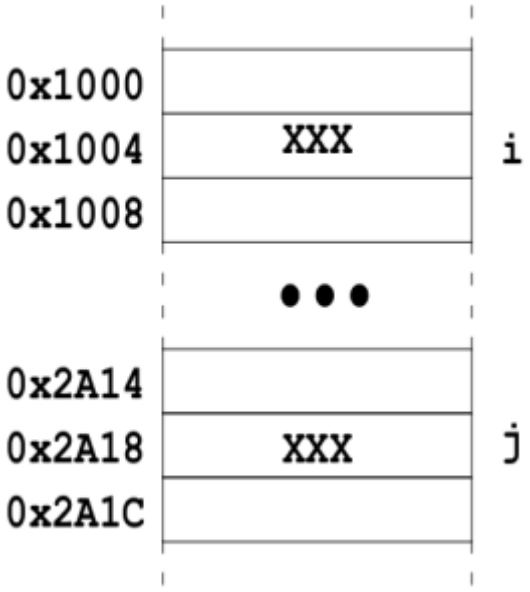
Heap es una sección de memoria que contiene objetos y también puede contener variables de referencia. Los atributos se crean en el *heap*.

13.1 Representación en memoria de tipos primitivos

Veámoslo paso a paso con un ejemplo:

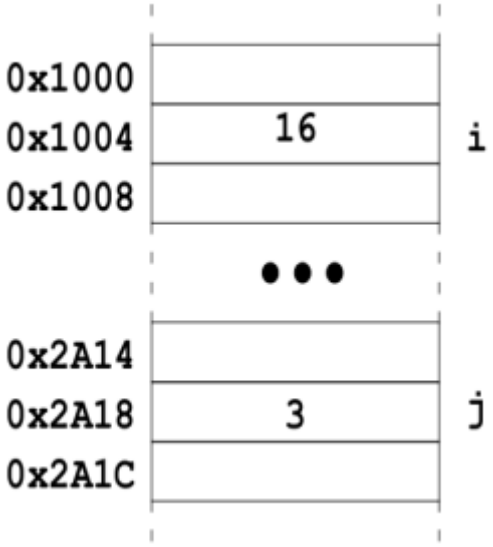
- Cuando se declara una variable de un tipo primitivo, el compilador reserva un área de memoria para ella:

```
int i , j;
```



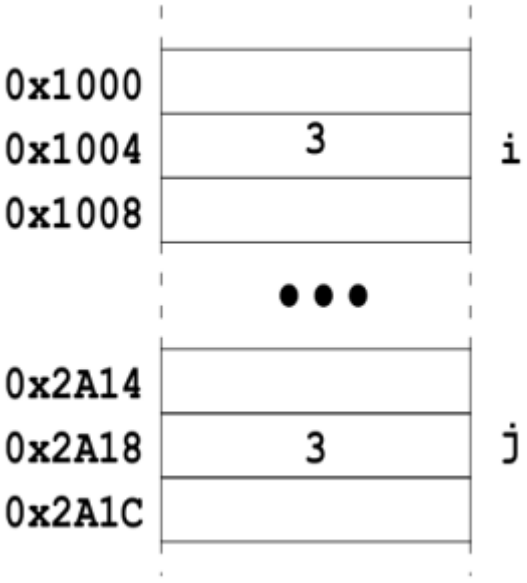
- Cuando se asigna un valor, éste es escrito en el área reservada:

```
i=16;
j=3;
```



- La asignación entre variables significa copiar el contenido de una variable en la otra:

```
i = j;
```



- La comparación entre variables compara los contenidos de las mismas:

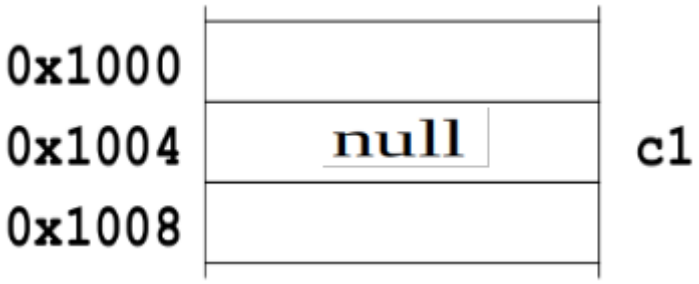
```
i == j; //true porque 3 es igual a 3
```

13.2 Representación en memoria de objetos

Supongamos que tenemos una clase *Complex* con dos atributos: *r*, *i*

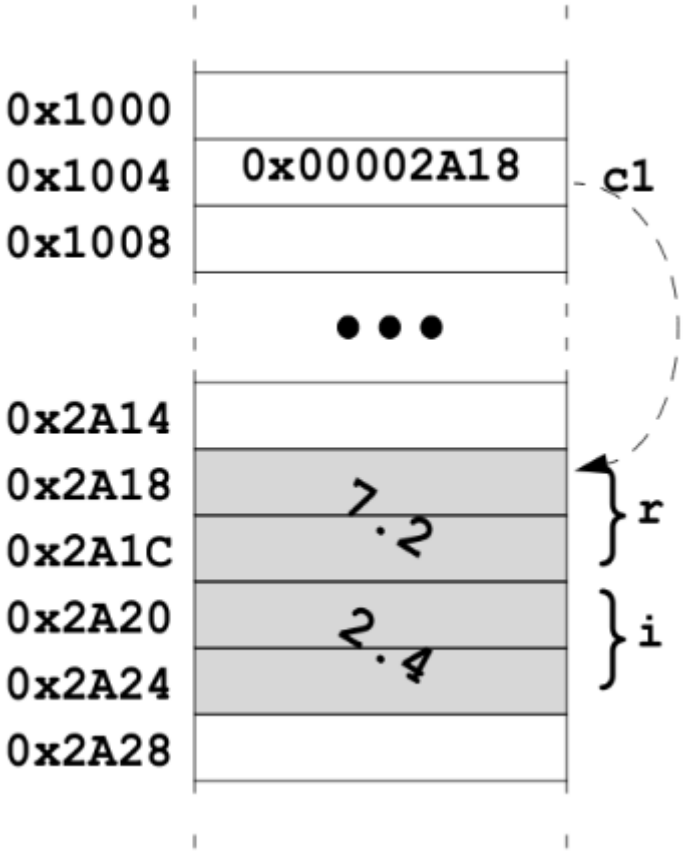
- Cuando se declara una variable de un objeto, el compilador reserva un área de memoria para ella. Mientras no se cree ningún objeto con el operador *new*, lo que contendrá será *null*. La palabra reservada *null* indica que una variable que referencia a un objeto se encuentra de momento sin referenciar a ninguno.

```
Complex c1;
```



- Cuando se crea el objeto llamando al constructor con el operador *new*, lo que se guarda en la variable *c1* es la dirección de memoria donde se ha almacenado el objeto en el *heap*. Es decir, la diferencia entre los tipos primitivos y los objetos es que en los primitivos la variable almacena el valor y en los objetos, la variable almacena la dirección de memoria donde se encuentra el objeto.

```
c1 = new Complex(7.2 , 2.4);
```



- La asignación entre variables objetos significa copiar la dirección de memoria donde se encuentra el objeto:

```
Complex c2 = c1;
```

  En este caso, el valor de la variable *c2* es ***0x2A18***, por lo tanto, ambas variables apuntan al mismo objeto. Si una de ellas modifica algún valor del objeto, también le afectará a la otra variable. Por ejemplo, si a *c2* le cambiamos el valor de *r* a 9.7, si consultamos el valor de *r* de *c1*, también valdrá 9.7, en lugar de 7.2.

- La comparación entre referencias no compara los contenidos de los objetos sino las direcciones de memoria, es decir, si apuntan al mismo sitio:

```
c2 == c1; //true porque apuntan al mismo sitio
Complex c3 = new Complex(7.2 , 2.4);
c3 == c1;//false porque aunque los objetos tengan los mismos valores de r e
i, c3 y c1 no apuntan al mismo sitio, es decir, no tienen la misma dirección
de memoria.
```

Para comparar los contenidos de los objetos, se utiliza el método *equals*:

```
c3.equals(c1); //true porque los contenidos de los objetos es el mismo
```

13.3 Garbage Collector

Garbage Collector (recolector de basura) es un programa que se ejecuta en el Java Virtual Machine que elimina objetos que ya no están siendo utilizados por una aplicación Java. Es una forma de gestión automática de la memoria.

Un objeto es ***elegible para el recolector de basura*** cuando deja de existir alguna referencia hacia él. Veamos algunos ejemplos:

```
Complex c1 = new Complex(7.2 , 2.4);
....
c1 = null;
....
```

Existe un objeto referenciado por la variable *c1*. Cuando la variable *c1* pierde la referencia porque se le asigna el *null*, se pierde cualquier forma de acceder al objeto, de modo que pasa a ser elegible para el recolector de basura.

```
Complex c1 = new Complex(7.2 , 2.4);
Complex c2 = c1;
....
c1 = null;
....
```

En este segundo ejemplo, el objeto no pasa a ser elegible para ser recolectado pues aunque la variable *c1* haya perdido la referencia porque se le asigna el *null*, todavía existe una referencia hacia el objeto por la variable *c2*.

```
Complex c1 = new Complex(7.2 , 2.4);
....
c1 = new Complex(8.3 , 2.7);
....
```

En este otro ejemplo, la variable *c1* es reasignada, es decir, se le asigna otro objeto mediante el operador *new*, por lo tanto se pierde cualquier referencia al objeto creado al principio `new Complex(7.2 , 2.4)` por lo que dicho objeto pasa a ser elegible para el recolector de basura.

14. This

Cuando se llama a un método, se pasa automáticamente un argumento implícito que es una referencia al objeto invocado, es decir, el objeto sobre el que se llama el método. Esta referencia se llama *this*, es decir, *this* es una variable que hace referencia al objeto actual.

Veamos los distintos usos que puede tener:

14.1 Para acceder a un atributo del objeto actual

Fijémonos de nuevo en la clase *Vehicle*:

```
public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

Centrémonos en el método getter del color:

```
public String getColour() {
    return colour;
}
```

colour es un atributo de la clase *Vehicle*. El método *getColour()* de la clase *Vehicle* puede acceder perfectamente al atributo *colour* ya que cuando el método *getColour()* es llamado, recibe de forma implícita el objeto actual, es decir, el objeto sobre el que se llama el método. Esto le permite al método *getColour()* poder acceder a todos los atributos de la clase y llamar a cualquier método de

la misma.

Como *colour* es un atributo de la clase *Vehicle*, también podíamos haber accedido a él como *this.colour*. Pero en realidad, cuando desde un método se accede a un atributo, se usa *this* de forma implícita, es decir, que aunque no escribamos *this*, el compilador lo sobreentiende. Por eso en la práctica, solo se indica si es imprescindible. Veamos un ejemplo en el que sea imprescindible:

```
public void setColour(String colour) {
        this.colour = colour;
}
```

En este caso, nos encontramos con dos variables que se llaman *colour*: el atributo y el parámetro del método. El ***shadow*** (*sombra*) de variables se refiere a la práctica en programación de utilizar dos variables con el mismo nombre dentro de ámbitos que se superponen. La variable con el alcance de nivel superior se oculta porque la variable con el alcance de nivel inferior la anula. La variable de nivel superior se dice entonces que es "*sombreada*". En nuestro ejemplo, el atributo es *"sombreado"* por el parámetro, es decir, cuando en el método accedemos a *colour*, es el parámetro al que nos estamos refiriendo. Para poder acceder al atributo, necesitamos utilizar la palabra this: `this.colour`. ¿Qué es lo que estamos haciendo entonces con esta línea de código: `this.colour = colour` ? Pues le estamos asignando el valor del parámetro al atributo, es decir, estamos actualizando el color del objeto actual con el color pasado por parámetro al método *setColour*.

14.2 Para invocar a un constructor

Un constructor puede llamar a otro constructor de la clase utilizando *this*, pero esta llamada solamente puede estar en la primera línea de código.

```
package tema4_POO.thisConstructor;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle() { //Constructor por defecto

        this(4, "blanco");
        /*Llama al constructor con parámetros. Esta llamada se tiene
        que hacer en la primera línea de código*/
    }

    public Vehicle(int wheelCount, String colour) { //Constructor con parámetros
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }
```

```
    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema4_POO.thisConstructor;

public class Main {

    public void showThisConstructor() {

        Vehicle car, moto;
        car = new Vehicle();
        moto = new Vehicle(2, "rojo");
        System.out.printf("Coche: %d ruedas y de color %s", car.getWheelCount(),
car.getColour());
        System.out.printf("\nMoto: %d ruedas y de color %s",
moto.getWheelCount(), moto.getColour());

    }

    public static void main(String[] args) {

        new Main().showThisConstructor();

    }

}
```

Salida por consola:

```
Coche: 4 ruedas y de color blanco
Moto: 2 ruedas y de color rojo
```

14.3 Como referencia al objeto actual

Dentro de la clase, siempre que queramos obtener una referencia al objeto actual, podemos utilizar *this*.

Supongamos que queremos modificar el método acelerar de los vehículos para que devuelvan el objeto actual con la velocidad modificada por la aceleración:

```
public Vehicle accelerate(double amount) {
        speed += amount;//Se incrementa la velocidad del objeto actual con la
cantidad pasada por parámetro
        return this; //Se devuelve el objeto actual
}
```

Veamos el ejemplo completo:

```
package tema4_POO.thisObjetoActual1;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public Vehicle accelerate(double amount) {
        speed += amount;
        return this; //Se devuelve una referencia al objeto actual
    }

    public void brake(double amount) {
        speed -= amount;
    }
```

```
}
```

```
package tema4_POO.thisObjetoActual1;

public class Main {

    public void showthisCurrentObject1() {

        Vehicle car1, car2;
        car1 = new Vehicle(4, "azul");
        System.out.printf("Coche1: %d ruedas, color %s, velocidad %.2f",
car1.getWheelCount(), car1.getColour(), car1.getSpeed());
        car2 = car1.accelerate(90);
        System.out.println("\nAceleramos el coche1 y se lo asignamos al coche2");
        System.out.printf("Coche1: %d ruedas, color %s, velocidad %.2f",
car1.getWheelCount(), car1.getColour(), car1.getSpeed());
        System.out.printf("\nCoche2: %d ruedas, color %s, velocidad %.2f",
car2.getWheelCount(), car2.getColour(), car2.getSpeed());
    }

    public static void main(String[] args) {

        new Main().showthisCurrentObject1();

    }

}
```

Salida por consola:

```
Coche1: 4 ruedas, color azul, velocidad 0,00
Aceleramos el coche1 y se lo asignamos al coche2
Coche1: 4 ruedas, color azul, velocidad 90,00
Coche2: 4 ruedas, color azul, velocidad 90,00
```

Veamos más ejemplos del uso de *this* como referencia al objeto actual. Vamos a añadir dos métodos nuevos a la clase *Vehicle*:

- *addSpeeds*: devuelve la suma de dos velocidades, la del objeto actual y la del vehículo pasado como parámetro.
- *doubleSpeed*: dobla la velocidad del objeto actual utilizando el método *addSpeeds*.

```
public double addSpeeds(Vehicle vehicle) {//Devuelve la suma de dos velocidades
    return speed + vehicle.speed;
}

public void doubleSpeed() {
    speed = addSpeeds(this); //Se le pasa al método addSpeeds la referencia al
objeto actual
}
```

Para doblar la velocidad utilizando el método *addSpeeds*, tenemos que pasarle a este método la referencia al objeto actual para que sume la velocidad consigo mismo, doblando de esta forma su propia velocidad.

```
package tema4_POO.thisObjetoActual2;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    public double addSpeeds(Vehicle vehicle) {//Devuelve la suma de dos
velocidades
        return speed + vehicle.speed;
    }

    public void doubleSpeed() {
        speed = addSpeeds(this); //Se le pasa al método addSpeeds la referencia
al objeto actual
    }

}
```

```
package tema4_POO.thisObjetoActual2;

public class Main {

    public void showthisCurrentObject2() {

        Vehicle car1, car2;
        car1 = new Vehicle(4, "azul");
        car2 = new Vehicle(4, "rojo");
        car1.accelerate(90.3);
        car2.accelerate(120.5);
        System.out.printf("Velocidad del coche1: %.2f", car1.getSpeed());
        System.out.printf("\nVelocidad del coche2: %.2f", car2.getSpeed());
        System.out.printf("\nSuma de las velocidades de los dos coches: %.2f",
car1.addSpeeds(car2));
        car1.doubleSpeed();
        System.out.printf("\nEl coche1 ha doblado su velocidad: %.2f",
car1.getSpeed());

    }

    public static void main(String[] args) {

        new Main().showthisCurrentObject2();

    }

}
```

Salida por consola:

```
Velocidad del coche1: 90,30
Velocidad del coche2: 120,50
Suma de las velocidades de los dos coches: 210,80
El coche1 ha doblado su velocidad: 180,60
```

Fijémonos de nuevo en el método de doblar la velocidad:

```
public void doubleSpeed() {
    speed = addSpeeds(this);
}
```

El método *doubleSpeed* de la clase *Vehicle* puede llamar perfectamente al método *addSpeeds* ya que cuando el método *doubleSpeed* es llamado, recibe de forma implícita el objeto actual, es decir, el objeto sobre el que se llama el método. Esto le permite al método *doubleSpeed* poder acceder a todos los atributos de la clase y llamar a cualquier método de la misma.

Como *addSpeeds* es un método de la clase *Vehicle*, el método *doubleSpeed* podía haberlo llamado también utilizando el this: *this.addSpeeds(this)*. Pero en realidad, cuando desde un método se llama a otro de la clase, se usa *this* de forma implícita, es decir, que aunque no escribamos *this*, el compilador lo sobreentiende. Por eso en la práctica, solo se indica si es imprescindible.

15. Encadenamiento de llamadas a métodos

Se emplea cuando invocamos a un método de un objeto que nos devuelve como resultado otro objeto al que podemos volver a invocar otro método y así encadenar varias operaciones.

```
package tema4_POO;

public class CallsToMethods {

    public void showCallsToMethods() {

        Boolean b;
        String string;

        string = "EntornosDeDesarrollo";
        System.out.println(string.substring(10).toUpperCase()); //DESARROLLO

        b = Boolean.TRUE;
        System.out.println(b.toString().charAt(2)); //u

    }

    public static void main(String[] args) {

        new CallsToMethods().showCallsToMethods();

    }

}
```

En este ejemplo, el método `substring(10)` está devolviendo una subcadena de la cadena string a partir del carácter 10 empezando en 0, es decir, "Desarrollo", al que se le invoca luego el método `toUpperCase` devolviendo como resultado la cadena "DESARROLLO".

Y el método `toString()` de la variable b de tipo Boolean está devolviendo la cadena "true" a la que se le encadena el método `charAt(2)` devolviendo el carácter 'u'.

16. Sintaxis fluida

Cuando un método modifica algún atributo del objeto, se puede devolver el objeto con un return para que dicho método pueda insertarse en una expresión. Esto permite encadenar otras llamadas de métodos consiguiendo que el código sea más corto, más legible y más fácil de manejar para los programadores.

Veamos un ejemplo de dos clases, una con sintaxis fluida y otra sin ella para ver la diferencia:

```
package tema4_POO.sintaxisFluida;

public class Vehicle1 { //Clase que no utiliza sintaxis fluida

    private int wheelCount;
    private double speed;
    private String colour;
```

```
    public Vehicle1(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema4_POO.sintaxisFluida;

public class Vehicle2 { //Clase que utiliza sintaxis fluida en los métodos
accelerate y brake

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle2(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }
```

```
    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public Vehicle2 accelerate(double amount) {
        speed += amount;
        return this; //Se devuelve una referencia al objeto actual para poder
utilizar sintaxis fluida
    }

    public Vehicle2 brake(double amount) {
        speed -= amount;
        return this; //Se devuelve una referencia al objeto actual para poder
utilizar sintaxis fluida
    }

}
```

```
package tema4_POO.sintaxisFluida;

public class Main {

    public void showfluidSintax() {

        Vehicle1 car1;
        Vehicle2 car2;
        car1 = new Vehicle1(4, "rojo");
        car2 = new Vehicle2(4, "azul");

        car1.accelerate(120.55);
        car1.brake(20.32);
        System.out.printf("Velocidad del coche1: %.2f", car1.getSpeed());

        /* car2 hace en una línea de código lo mismo que car1
         * hace en 3 líneas de código debido a que car2 utiliza
         * sintaxis fluida:
         */
        System.out.printf("\nVelocidad del coche2: %.2f",
car2.accelerate(120.55).brake(20.32).getSpeed());

    }

    public static void main(String[] args) {

        new Main().showfluidSintax();

    }

}
```

Salida por consola:

```
Velocidad del coche1: 100,23
Velocidad del coche2: 100,23
```

17. Invariante de una clase

El invariante de una clase es el conjunto de restricciones que deben cumplir todos los objetos que se instancien de dicha clase, como por ejemplo:

- Restricciones sobre los valores que pueden tomar los atributos de la clase. Ejemplo: el atributo *wheelCount* de la clase *Vehicle* tiene que ser mayor que 0.
- Restricciones que afecten a más de un atributo. Ejemplo: si una clase modela un rango de valores, el atributo que corresponda al límite superior del rango debe ser forzosamente mayor o igual que el atributo que corresponda al límite inferior del rango, dada la definición matemática de rango.
- Restricciones con respecto a los objetos con los que se relaciona. Ejemplo: un empleado tiene un atributo que es un objeto de la clase *Empresa* donde trabaja. Una restricción posible sería que dicha empresa no pueda ser nula.

Los métodos constructores de una clase deben respetar el invariante de la clase a la hora de construir los objetos.

Los métodos públicos de la clase también deben respetar el invariante. Un método puede no respetar el invariante en el transcurso de la ejecución, pero cuando el método finalice, el invariante se tiene que cumplir. Es decir, es perfectamente viable que un método para alcanzar su objetivo pueda perder de forma temporal el invariante pero siempre y cuando finalice con el invariante cumplido, es decir, antes de la llamada el invariante se debe cumplir y después de la llamada también, durante la ejecución del mismo puede no satisfacerse.

Únicamente se deben satisfacer los invariantes en las llamadas a métodos públicos, la ejecución de métodos privados de la misma clase pueden saltarse esta norma aunque no es aconsejable.

Definir invariantes de clase puede ayudar a los programadores y controladores de calidad a localizar más errores durante las pruebas de software.

18. Encapsulamiento

La encapsulación es un principio fundamental de la programación orientada a objetos que consiste en ocultar el estado o los atributos de un objeto y obligar a que toda interacción se realice a través de los métodos del objeto definidos en su clase para conservar su invariante.

El encapsulamiento se consigue utilizando los modificadores de acceso. Se recomienda que los atributos de una clase sean privados, por lo tanto, aquellos atributos que se permitan consultar deben tener sus propios métodos get, y los que se permitan modificar, deben tener sus propios métodos set. Hacer uso de este convenio nos facilitará trabajar con el resto del mundo y nos permitirá ampliar las capacidades de nuestro código utilizando frameworks existentes que hacen uso del convenio y que si no seguimos no podremos utilizar.

Veamos un ejemplo que no cumple el principio de encapsulamiento:

```
package tema4_POO.noEncapsulacion;

public class Vehicle { //Clase que no utiliza encapsulación
```

```
    public int wheelCount;
    public double speed;
    public String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema4_POO.noEncapsulacion;

public class Main {

    public void showNonEncapsulation() {

        Vehicle car;
        car = new Vehicle(4, "azul");
        car.accelerate(90.54);
        System.out.printf("Velocidad: %.2f", car.speed);
        car.speed = 120;//El cliente está accediendo al atributo directamente no
cumpliendo el encapsulamiento
        System.out.printf("\nVelocidad: %.2f", car.speed);

    }

    public static void main(String[] args) {

        new Main().showNonEncapsulation();

    }

}
```

Salida por consola:

```
Velocidad: 90,54
Velocidad: 120,00
```

Podemos observar que no se está cumpliendo el principio de encapsulamiento ya que el cliente está accediendo directamente al atributo *speed* modificando incorrectamente su valor ya que la velocidad solamente se debe modificar acelerando o frenando el vehículo.

Vamos a modificar el ejemplo anterior para que sí cumpla con el principio de encapsulamiento, ocultando el estado o los atributos del objeto y obligando a que toda interacción se realice a través de los métodos del objeto definidos en su clase:

```
package tema4_POO.encapsulacion;

public class Vehicle { //Clase que utiliza encapsulación

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

}
```

```
package tema4_POO.encapsulacion;

public class Main {

    public void showEncapsulation() {

        Vehicle car;
        car = new Vehicle(4, "azul");
        car.accelerate(90.54);
        System.out.printf("Velocidad: %.2f", car.getSpeed());
```

```
        car.speed = 120;//Error de compilación: el atributo speed no es visible

    }

    public static void main(String[] args) {

        new Main().showEncapsulation();

    }

}
```

Al cliente no se le permite modificar directamente el atributo *speed* ya que éste tiene como modificador de acceso *private*. Si quiere modificar la velocidad tiene que hacerlo a través de los métodos *accelerate* y *brake*, cumpliendo de esta forma con el principio de encapsulamiento.

19. El método toString

El método `toString()` se utiliza para obtener una cadena de texto que represente al objeto. Lo veremos en más profundidad en el *Tema 7 Herencia 6 El método toString*. Veamos su uso:

```
package tema4_POO.toString;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }
```

```
    public void brake(double amount) {
        speed -= amount;
    }

    //Método toString para obtener una cadena de texto que represente al objeto
    @Override
    public String toString() {
        return String.format("Número de ruedas: %d, color: %s, velocidad: %.2f",
wheelCount, colour, speed);
    }

}
```

```
package tema4_POO.toString;

public class Main {

    public void showToString() {

        Vehicle vehicle;

        vehicle = new Vehicle(2, "azul");
        System.out.println(vehicle.toString());
        System.out.println(vehicle);//hace lo mismo que vehicle.toString()

    }

    public static void main(String[] args) {

        new Main().showToString();

    }

}
```

Salida por consola:

```
Número de ruedas: 2, color: azul, velocidad: 0,00
Número de ruedas: 2, color: azul, velocidad: 0,00
```

El método *toString()* también se puede generar con el Eclipse en el menú *Source* → *Generate toString()*.

Cuando se saca por consola un objeto y solo se pone el nombre de la variable del objeto, se llama al toString del objeto tal y como podemos observar en la línea de código:

```
System.out.println(vehicle);//hace lo mismo que vehicle.toString()
```

20. Control de nulidad

El método `Objects.requireNonNull` es una utilidad estándar en Java que pertenece a la clase `java.util.Objects`. Su principal función es validar que un objeto no sea `null`. Si el objeto es `null`, lanza una excepción de tipo `NullPointerException`. Esto lo hace útil para asegurar que un parámetro, variable o referencia no pueda ser `null` en tiempo de ejecución. Si el objeto no es `null`, simplemente lo retorna, lo que permite que se siga utilizando en el código de manera segura. Ejemplo:

```
import java.util.Objects;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = Objects.requireNonNull(colour, "El color no puede ser
nulo");
        speed = 0;
    }
}
```

¿Cuándo utilizar `Objects.requireNonNull`?:

- Validación de parámetros: es muy común usarlo en los constructores y en los métodos que requieren que ciertos parámetros no sean nulos.
- Asegurar inmutabilidad: en clases que siguen el patrón de objetos inmutables, se puede asegurar que los atributos esenciales no sean `null` desde el inicio.
- Seguridad en tiempo de ejecución: es una manera de asegurarse en tiempo de ejecución de que efectivamente un valor no es `null`.

21. Uso de las excepciones para el control del invariante de una clase

Veamos un ejemplo de invariante en la clase *Vehicle* con respecto al color: un vehículo en todo momento tiene que tener información sobre el color, no puede estar vacío dicho campo. En la clase *Vehicle* hay dos sitios que afectan a la modificación del color: en el constructor y en el método *setColour*. En este caso, controlamos el invariante en el método *setColour* y el constructor llama al *setColour*. En el caso de un atributo que no tuviera método *set*, el control del invariante se haría en el constructor.

```
package tema4_POO.invariante;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;
```

```
    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        setColour(colour);//El constructor llama al método setColour
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        if(colour.isEmpty()) {//Control del invariante
            throw new IllegalArgumentException();
        }
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {
        return String.format("Número de ruedas: %d, color: %s, velocidad: %.2f",
wheelCount, colour, speed);
    }

}
```

```
package tema4_POO.invariante;

public class Invariance1 {

    public void show() {

        Vehicle vehicle = new Vehicle(4, "");
        System.out.println(vehicle);

    }

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

```
        new Invariance1().show();

    }

}
```

En este código estamos intentando crear un vehículo con un color vacío por lo que abortará:

```
Exception in thread "main" java.lang.IllegalArgumentException
    at tema4_POO.invariante.Vehicle.setColour(Vehicle.java:29)
    at tema4_POO.invariante.Vehicle.<init>(Vehicle.java:11)
    at tema4_POO.invariante.Invariance1.show(Invariance1.java:7)
    at tema4_POO.invariante.Invariance1.main(Invariance1.java:14)
```

De esta manera controlamos que no se va a crear un vehículo sin color. Veamos un ejemplo donde el vehículo lo creamos con un color y luego intentamos modificarlo con un color vacío:

```
package tema4_POO.invariante;

public class Invariance2 {

    public void show() {

        Vehicle vehicle = new Vehicle(4, "azul");
        System.out.println(vehicle);
        vehicle.setColour("");

    }

    public static void main(String[] args) {

        new Invariance2().show();

    }

}
```

En este caso también abortará porque el método *setColour* controla que no se modifique un vehículo con un color vacío:

```
Número de ruedas: 4, color: azul, velocidad: 0,00
Exception in thread "main" java.lang.IllegalArgumentException
    at tema4_POO.invariante.Vehicle.setColour(Vehicle.java:29)
    at tema4_POO.invariante.Invariance2.show(Invariance2.java:9)
    at tema4_POO.invariante.Invariance2.main(Invariance2.java:15)
```

En ambos casos, el cliente puede capturar la excepción y dar un mensaje al usuario:

```
package tema4_POO.invariante;

import java.util.Scanner;

public class Invariance3 {
```

```
    @SuppressWarnings("resource")
    public void show() {

        Vehicle vehicle;
        String colour;
        Scanner keyboard = new Scanner(System.in);

        try {
            System.out.println("Introduzca un color para el vehículo: ");
            colour = keyboard.nextLine();
            vehicle = new Vehicle(4, colour);
            System.out.println(vehicle);
            System.out.println("Introduzca el nuevo color del vehículo: ");
            colour = keyboard.nextLine();
            vehicle.setColour(colour);
            System.out.println(vehicle);
        } catch (IllegalArgumentException e) {
            System.err.println("El color del vehículo no puede estar vacío");
        }

    }

    public static void main(String[] args) {

        new Invariance3().show();

    }

}
```

22. Records

Los registros (*records*) son una nueva clase especial en Java que simplifica la creación de clases que son principalmente "contenedores de datos", es decir, clases cuyos principales propósitos son almacenar valores. Los records contienen datos inmutables.

Un ejemplo típico de un record es un `Point` que representa una clase inmutable utilizada para almacenar coordenadas en un espacio bidimensional (con `x` e `y` como componentes). La definición de dicho record sería la siguiente:

```
public record Point(int x, int y) {}
```

Este código genera automáticamente:

- Un constructor con `x` e `y` como parámetros llamado *constructor canónico*.
- Métodos getters para `x` e `y`. El nombre de los getters de los records no sigue las convenciones JavaBean `getX()` y `getY()` sino mediante métodos con el mismo nombre que el campo: `x()` e `y()`.
- Implementaciones de los métodos *equals*, *hashCode* y *toString* basados en `x` e `y`.

Estos métodos también se pueden anular si hiciera falta una implementación diferente.

Cualquier registro extiende implícitamente de `java.lang.Record`, al igual que cualquier tipo enumerado extiende implícitamente de java.lang.Enum. La superclase Record no tiene estado y sólo tiene métodos abstractos *equals*, *hashCode* y *toString*.

Veamos un ejemplo de utilización del record `Point`:

```
package tema4_POO.records;

public record Point(int x, int y) {}
```

```
package tema4_POO.records;

public class showRecords {

    public void show() {

        Point p = new Point(3,4);
        System.out.println(p);//Llama por defecto al toString
        System.out.println(p.x());
        System.out.println(p.y());
        System.out.println(p.equals(new Point(3,4)));
        System.out.println(p.hashCode());

    }

    public static void main(String[] args) {

        new showRecords().show();

    }

}
```

Salida por consola:

```
Point[x=3, y=4]
3
4
true
97
```

Se puede añadir lógica personalizada en el constructor canónico de un record, por ejemplo, validar que los valores de `x` e `y` sean positivos en el momento de la creación de un `Point`:

```java
package tema4_POO.recordsConstructorCanonico;

public record Point(int x, int y) {

    public Point {
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException("Coordinates must be positive");
        }
    }

}
```

```java
package tema4_POO.recordsConstructorCanonico;

public class showRecordsCanonicalConstructor {

    public void show() {

        Point p1,p2;
        p1 = new Point(3,4);
        System.out.println(p1);
        p2 = new Point(5,-2);//Lanza la excepción IllegalArgumentException
        System.out.println(p2);

    }

    public static void main(String[] args) {

        new showRecordsCanonicalConstructor().show();

    }

}
```

Además del constructor canónico, se pueden añadir otros constructores cuya primera sentencia debe invocar a otro constructor, de forma que en última instancia se invoque al constructor canónico. Veamos el siguiente ejemplo donde hay dos constructores: el constructor canónico y otro constructor que devuelve el origen:

```java
package tema4_POO.recordsConstructores;

public record Point(int x, int y) {

    public Point {
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException("Coordinates must be positive");
        }
    }

     public Point() {
         this(0, 0);
     }
```

```
}
```

```
package tema4_POO.recordsConstructores;

public class showRecordsConstructors {

    public void show() {

        Point p1,p2;
        p1 = new Point(3,4);
        System.out.println(p1);
        p2 = new Point();
        System.out.println(p2);

    }

    public static void main(String[] args) {

        new showRecordsConstructors().show();

    }

}
```

Salida por consola:

```
Point[x=3, y=4]
Point[x=0, y=0]
```

El *constructor canónico* asigna los parámetros `x` e `y` a los atributos `this.x` y `this.y`. Esta asignación se produce al final del constructor canónico, es decir, si añadimos código a este constructor, después de dicho código se realizará dicha asignación:

```
this.x = x;
this.y = y;
```

Esto se hace automáticamente, es decir, el programador no tiene que hacerlo, de hecho, el programador no puede leer ni modificar los atributos en el cuerpo del constructor canónico. Pero lo que sí puede modificar es los parámetros del constructor canónico antes de asignarlos a los atributos. Por ejemplo, verificamos si los parámetros `x` o `y` son negativos y en dicho caso se ajustan a `0`. Esta modificación ocurre antes de la asignación a los atributos. Después de modificar los valores de los parámetros, el compilador se encarga de asignar `x` e `y` a los atributos correspondientes, lo que garantiza que las modificaciones se reflejen en el objeto creado:

```
package tema4_POO.recordsParametrosConstructor;

public record Point(int x, int y) {

    public Point {

        if (x < 0) {
            x = 0;
```

```
        }
        if (y < 0) {
            y = 0;
        }

    }

}
```

```
package tema4_POO.recordsParametrosConstructor;

public class showRecordsConstructorParameters {

    public void show() {

        Point p1 = new Point(3, 4);
        Point p2 = new Point(-5, 10);
        Point p3 = new Point(7, -8);

        System.out.println(p1);
        System.out.println(p2);
        System.out.println(p3);

    }

    public static void main(String[] args) {

        new showRecordsConstructorParameters().show();

    }

}
```

Salida por consola:

```
Point[x=3, y=4]
Point[x=0, y=10]
Point[x=7, y=0]
```

Veamos aspectos que **sí** se pueden hacer con los records:

- Añadir métodos adicionales igual que en una clase normal.
- Implementar interfaces.
- Se pueden declarar tanto en su propio archivo .java como dentro de otra clase o de otro record. La elección de cómo declararlo depende del contexto y del propósito para el que lo estás utilizando.
- Pueden contener atributos y métodos estáticos.
- Pueden definirse localmente dentro de un método.
- Pueden ser genéricos.

Veamos aspectos que **no** se pueden hacer con los records:

- No pueden heredar de otra clase ni de otro record.

- No pueden tener más atributos que las variables declaradas con el constructor canónico, es decir, en el caso de `record Point(int x, int y) {}`, el record Point no puede tener más atributos aparte de `x` e `y`.
- No existen los *inner records* (registros internos), es decir, un registro que se define dentro de otra clase o método es automáticamente estático, es decir, no tiene una referencia a la clase que lo contiene.
- El constructor canónico no puede lanzar excepciones checked:

```
record SleepyPoint(double x, double y) {
   public SleepyPoint throws InterruptedException { // Error
      Thread.sleep(1000);
   }
}
```