Añadir la clase Comparable y como hacer overright del método EqualsTo

Crea un proyecto llamado JavaCourse y crea un paquete para cada ejercicio.

ⓘ **tipos primitivos y CLASES WRAPPER**

Los tipos primitivos son los tipos de datos más básicos, por ejemplo: int, double, boolean, char, etc…
Las Clases Wrapper (wrap significa envoltura. El Crunchwrap de Taco Bell significa Envoltura Crujiente, de hecho), nos permiten envolver a los tipos primitivos, permitiéndonos así tratarlos como si fueran objetos.

¿Diferencia real entre tipos primitivos y Clases Wrapper? Al permitir que las Clase Wrapper actúen como si fueran objetos, estos pueden tener un valor más (a parte del propio) y es null; un tipo primitivo nunca puede ser null.

Entonces:

❌ Esto da error en la tercera línea porque int es un tipo primitivo y no permite introducir un null:

```
int numero = 1;
numero = 43;
numero = null;
```

☑ Esto está bien escrito porque al ser una Clase Wrapper no hay ningun tipo de problema:

```
Integer numeraso = 200;
numeraso = null;
numeraso = 3;
```

1. **ejercicio1: Clase Wrapper a tipo primitivo:**

   → Tienes dos variables, una es de tipo texto cuyo valor es 72 y el otro es un entero cuyo valor es 33. Tienes que pintar por pantalla la suma de ambos números.

2. **ejercicio2: tipo primitivo a Clase Wrapper:**

   → Tienes una variable de un número entero que vale 11, tienes que pasar el valor de esa variable a otra variable Clase Wraper y mostrar el resultado de esta segunda variable por consola.
   → Hay dos formas de pasar el valor de una variable a otra:

   a) una forma es más “manual” o directa, que es parseando el valor con “()”, como forzandole a que cambie de un tipo de variable a otra
   b) y la otra forma es más óptima, con un método ya creado (tú no tienes que crear ningún método: el método se encuentra ya creado)

   Haz la conversión de las dos formas en la misma clase y asegúrate de comprobar que las dos formas funcionan

Diferencia entre JDK, JRE y JVM

- JDK (Java Development Kit): Es un conjunto de herramientas que permite desarrollar aplicaciones Java. Incluye el compilador Java (javac), las bibliotecas Java estándar (como java.lang y java.util) y otros utilitarios para desarrolladores.

- JRE (Java Runtime Environment): Es un entorno de ejecución que proporciona la máquina virtual Java (JVM) y las bibliotecas Java necesarias para ejecutar aplicaciones Java. El JRE no incluye las herramientas de desarrollo como el compilador.

- La JVM (Java Virtual Machine) es una máquina virtual que ejecuta programas Java convertidos en bytecode. Es una parte esencial del entorno de ejecución de Java y es responsable de interpretar y ejecutar el código bytecode de Java en cualquier plataforma que la admita.

Entonces la JVM está dentro del JRE, el cual es necesario para ejecutar aplicaciones Java. Si por otro lado quieres desarrollar aplicaciones Java entonces necesitas el JDK, el cual incluye el JRE.

La jerarquía de posesión es la siguiente:

JDK

- **JRE**
  - JVM

ⓘ Java SE vs Java EE

Java SE (Standard Edition) y **Java EE**(Enterprise Edition) son distintas plataformas (versiones) de Java.

- **Java SE** proporciona las capacidades básicas estándar del lenguaje Java.
- **Java EE** implementa facilidades para desarrollar aplicaciones web con Java.

A la hora de desarrollar una aplicación a nivel empresarial se usaba **Java EE,** pero un pibardo re astuto creó un **framework** llamado **Spring** y ahora se usa ese framework para crear las apps con Java.

ⓘ Hardcodear (Hard coding)

Esto se usa para hacer referencia a: "introducir un valor directamente de forma manual en la app, quitando así la posibilidad de hacer que el valor pueda ser dinámico". **Esto, en general, es una mala práctica**, porque le quitas a la variable la posibilidad de cambiar de valor.

Ejemplo de valor dinámico:

```
// variable dinámica
int edad = 0;

Scanner sc = new Scanner(System.in);

System.out.println("¿Cuál es tu edad?");
edad = sc.nextInt();

System.out.println("- Tu edad es: " + edad);
```

Ejemplo de valor hardcodeado:

```
// variable estática. El valor de la edad siempre va a ser 55:
// se ha hardcodeado el valor de la edad como 55, y ese valor no se puede cambiar dentro del programa
int edad = 0;

System.out.println("¿Cuál es tu edad?");
edad = 55;

System.out.println("- Tu edad es: " + edad);
```

☑ La finalidad de estas definiciones es que las cosas te suenen, no debes aprenderlo de memoría ni tampoco son conceptos super importantes: Solo leelo 1 o 2 veces y pasa a lo siguiente

☑ **Ahora la práctica.**

Crea un paquete para cada tipo de bucle/condicional.

↗ Encapsulación (setters y getters)

Los getters y setters son una forma de acceder a los atributos de una clase. ¿Por qué se usan? Porque acceder directamente a un atributo es un poco peligroso (no hay ningún tipo de control ni impedimento), lo que se suele hacer es implementar alguna condición (if) dentro del set para no permitir que se ponga cualquier dato a la hora de modificar un valor; sólo unos valores concretos permitidos

3. **ejercicio3: Acceder a los atributos de un objeto mediante getters & setters**

   → Crea una clase que se llame **Vamost** desde la cual se ejecutará el programa de Java.
   → Crea una clase que se llame **Pizzero** que tenga los atributos **nombre**, **edad** y **sueldo**. Los valores por defecto deben ser: sin nombre, edad 0 y sueldo 3000.
   → En la clase **Pizzero**, crea los **getters** y los **setters** (no uses atajos) y llama a los métodos con "obtener" y con "modificar" en vez de con "get" y "set";
   ejemplo con el atributo nombre: **obtenerNombre** y **modificarNombre** en vez de *getNombre* y *setNombre*;

   → En la clase desde la cual se ejecutará el programa, **Vamost**, crea un objeto de la clase **Pizzero** que se llame Antonio y que tenga 35 años.
   → Ahora modifica el sueldo de Antonio para que tenga 200. Imprime por consola el nombre del pizzero, su edad y su sueldo multiplicado x 5.

4. **ejercicio4: Modificar un atributo de una clase solo cuando se cumplan los requerimientos**

   → Crea una clase que se llame **Programa** desde la cual se ejecutará el programa de Java.

   → Crea una clase que se llame Persona y crea un atributo de numeros enteros que se llame edad y dale el valor por defecto de 0. Crea los setters y getters para ese atributo (atajo de teclado: ***shift + alt + S***).
   → Ahora ve al setter y modificalo para hacer que solo se pueda modificar la edad si la edad es mayor de 50.

   → Vuelve a la clase Programa y crea un objeto de la clase Persona. Ahora modifica la edad de ese objeto (por medio del setter) e introduce

el valor 30. Ahora escribe un Syso que obtenga la edad del objeto de la clase Persona. Ejecuta el programa

*Te debería dar **0** por consola, que es el valor por defecto del atributo edad, puesto que el valor 30 no pasa la condición que has debido introducir en el setter*

→ Ahora cambia ese 30 que pasas por el setter y pon un 70 y ejecuta el programa

Te debería dar 70 por consola, puesto que el valor 70 si pasa la condición que has debido introducir en el setter

↗ Estructuras de Datos (general)

Las Interfaces Maps, Listas y Conjuntos son estructuras de datos comunes en Java que se utilizan para almacenar y manipular colecciones de elementos de manera eficiente.

-1 Map

La Interface Map (java.io.Map) en Java, nos permite representar una estructura de datos para almacenar pares "clave/valor"; de tal manera que para una clave solamente tenemos un valor.

HashMap, TreeMap y LinkedHashMap

- Implementan la interfaz Map.
- Almacenan pares clave-valor.
- Permiten claves y valores nulos

Diferencias:

- HashMap: No mantiene ningún orden específico.
- TreeMap: Mantiene los elementos ordenados por sus claves.
- LinkedHashMap: Mantiene el orden de inserción de los elementos.

-2 List

En Java, las listas son una estructura de datos que almacena una colección ordenada de elementos **donde se permite duplicados**

- Implementan la interfaz List.
- Permiten elementos duplicados.

ArrayList y LinkedList son básicamente lo mismo. LinkedList es casi idéntico a un ArrayList: puedes guardar tantos objetos del mismo tipo como quieras, además puedes usar los mismos métodos que ArrayList ya que ambos implementan la interface **List**. **La única diferencia es que, en términos generales,**

ArrayList es más eficiente, pero LinkedList provee bastantes métodos para hacer ciertas operaciones más eficientemente:

Se puede decir que se debe usar `ArrayList` para guardar y acceder a los datos, y `LinkedList` para manipular los datos.

¿Cómo funciona un ArrayList?
Cuando un elemento es añadido, se crea un nuevo array que ya contiene el nuevo elemento y este nuevo array sustituye al antiguo. De esta forma se "actualiza" el ArrayList.

¿Cómo funciona una LinkedList?
Una LinkedList almacena elementos en containers. La LinkedList tiene en enlace al primer container y cada container tiene un enlace hacia el próximo container. Para añadir un nuevo elemento a la LinkedList, el elemento se coloca en un nuevo container y ese container es linkeado con otro container en la LinkedList.

Métodos extra que provee ArrayList:

Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

Diferencias:

- ArrayList: Rápido para acceso aleatorio pero lento para inserciones y eliminaciones en el medio de la lista.
- LinkedList: Rápido para inserciones y eliminaciones en el medio de la lista pero lento para acceso aleatorio.

-3️⃣ Conjuntos

Las clases HashSet, TreeSet y LinkedHashSet implementan la interfaz Set en Java. La interfaz Set es una interfaz de la biblioteca estándar de Java que representa una colección de elementos únicos, es decir, no permite elementos duplicados. La interfaz Set proporciona

operaciones para agregar, eliminar y verificar la existencia de elementos en el conjunto, así como también operaciones de conjunto como unión, intersección, diferencia, etc.

- Implementan la interfaz Set.
- Almacenan elementos únicos.
- No permite elementos duplicados.

Diferencias:

- HashSet: No mantiene ningún orden específico.
- TreeSet: Mantiene los elementos ordenados según su valor.
- LinkedHashSet: Mantiene el orden de inserción de los elementos.

↗ Arrays & Arrays Bidimensionales

Sabiendo recorrer un array con un bucle for o for each, ya sabes el 80% de lo relativo a arrays.

- ¿Cómo recorrer un array? Con un bucle **For** o **For Each**
- ¿Cómo saber el tamaño de un array? Escribiendo **.length**
- **¿Se puede añadir un elemento al array? No, nunca.** Si se **instancia** (instanciar = crear) un **array** con 10 **elementos** (elementos = huecos), entonces siempre tendrá 10 elementos

¿Qué es un **array bidimensional**?

Esto es un array de X elementos, donde cada uno de esos elementos tiene dentro otro array.

```
int[][] arrayPrincipal = new int[4][3];

arrayPrincipal[2][1] = 3;

arrayPrincipal = [ ] [ ] [ ] [ ]
                 [ ] [ ] [3] [ ]
                 [ ] [ ] [ ] [ ]
```

Como puedes ver esto de los arrays bidimensionales se usa para tableros. Por ejemplo para el ajedrez, el hundir la flota o juegos como la serpiente de los móviles antiguos.

↗ ArrayList

Un **ArrayList** es exactamente lo mismo que un **Array**, solo que la cantidad de elementos que tiene un **ArrayList** es modificable. O sea se puede crear el

ArrayList con ningún elemento y por medio del método **.add()** podemos añadir un elemento al **ArrayList** . O sea que es la hostia porque es dinámico.

> Crear el ArrayList:

```
ArrayList<String> cars = new ArrayList<String>();
```

> Añadir items:

```
cars.add("BMW");
```

> Acceder a un item:

```
cars.get(0);
```

> Cambiar un item:

```
cars.set(0, "Opel");
```

> Borrar un item:

```
cars.remove(0);
```

> Borrar todos los items:

```
cars.clear();
```

> Tamaño del ArrayList:

```
cars.size();
```

> Ordenar el ArrayList:

```
Collections.sort(cars);
```

5. **ejercicio13:**

- Instancia un **ArrayList** vacío que guardará los nombres de los empleados.
- Añade un empleado e imprime por consola todos los elementos del **ArrayList.**
- Luego añade otro nombre y saca todos los elementos del **ArrayList.**
- Ahora borra el primer elemento del **ArrayList** y saca todos los elementos del **ArrayList.**

https://www.w3schools.com/java/java_try_catch.asp

↗ Controles del flujo del programa: Bucles y Condicionales

1 IF / ELSE

Esta sentencia nos permite elegir entre dos alternativas de flujo diferentes, dependerá de si la condición se cumple o no.

6. **ejercicio7:** Rrealizar los 20 ejercicios de esta página sin ver las soluciones, claro, dentro del *paquete de ejercicio7*:

Ejercicios IF / ELSE

2 SWITCH

Esta sentencia nos permite tener múltiples alternativas en base al valor que pueda tener una variable

7. **ejercicio8:**

Considera que estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo **Motor** (que representa el motor de una bomba para mover fluidos). Define una clase **Motor** considerando los siguientes **atributos** de clase: tipoBomba (int), tipoFluido (String), combustible (String). Define un **constructor** asignando unos valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos. Crea un **método** tipo procedimiento denominado **dimeTipoMotor**() donde a través de un condicional switch hagas lo siguiente:

a) Si el tipo de motor es 0, mostrar un mensaje por consola indicando "No hay establecido un valor definido para el tipo de bomba".

b) Si el tipo de motor es 1, mostrar un mensaje por consola indicando "La bomba es una bomba de agua".

c) Si el tipo de motor es 2, mostrar un mensaje por consola indicando "La bomba es una bomba de gasolina".

d) Si el tipo de motor es 3, mostrar un mensaje por consola indicando "La bomba es una bomba de hormigón".

e) Si el tipo de motor es 4,mostrar un mensaje por consola indicando "La bomba es una bomba de pasta alimenticia".

f) Si no se cumple ninguno de los valores anteriores mostrar el mensaje "No existe un valor válido para tipo de bomba".

Compila el código para comprobar que no presenta errores, crea un objeto, usa sus métodos y
comprueba que aparezcan correctamente los mensajes por consola

3 WHILE vs DO WHILE

- La sentencia **While** se ejecuta una y otra vez mientras que la condición sea verdadera.
- La sentencia **Do While** se ejecuta una y otra vez mientras que la condición sea verdadera pero además se ejecutará al principio una vez si o si

¿Cuál es diferencia entre **while** y **do while**?

While puede que no se ejecute ninguna vez, pues **lo primero que se lee es la condición** y si esta no es verdadera entonces no se ejecutará nunca; en el caso de que la condición sea verdadera, entonces si se ejecuta

Do While siempre se ejecutará una vez. **Lo primero que hace es ejecutarse una vez y luego se checkeará la condición** (si es verdadera, se ejecutará otra vez; si es falsa, no se ejecutará), con lo cual el bucle **Do While siempre se ejecutará al menos una vez.**

❗ *Cuidado con **While** y **Do While** porque se ejecutarán en bucle siempre que la condición sea verdadera, esto quiere decir que si no se programa algo que rompa el bucle se puede dar el caso de que se cree un bucle infinito (esto es un error muy común al empezar con Java)*

8. **ejercicio9: (while)** Pedimos por consola un **valor inicial** y un **valor final** por consola, e incrementemos el valor inicial dado hasta el tope establecido, mostrando el valor de la variable contadora en cada iteracion.

9. **ejercicio10: (do while)** Escribir un programa que solicite que se introduzca un número entre 0 y 999, y nos muestre un mensaje de cuantos dígitos tiene el mismo. Finalizar el programa cuando se cargue el valor 0.

4 FOR vs FOR EACH

Estas sentencias son como un while con la condición de ruptura del bucle incluida en la cabecera de la sentencia. **For** y **For Each** se usa sobre todo para recorrer los **arrays**.

Así se recorren los índices de un array de 2 en 2:

```
// recorrer bucle for de 2 en 2
int m[] = {1,2,3,4,5};

for(int i=0; i<5; i+=2) {
      System.out.println(m[i]);
}
```

10. **ejercicio11:**

    Pinta por consola esta figura:

    ```
    *
    **
    ***
    ****
    ```

    Teniendo en cuenta que la base de asteriscos debe incluir tantos asteriscos como el usuario introduzca por consola.

11. **ejercicio12:** Dado el siguiente **array**:

    ```
    String[] nombres = {"Antonio", "Bernarda", "Camilo", "Diana", "Ernesto"};
    ```

    Recorrelo con un **for** y luego también recorrelo con **for each**. En ambos casos saca por consola el valor de cada elemento del array.

↗ Excepciones

En Java los errores en tiempo de ejecución (cuando se está ejecutando el programa) se denominan **excepciones**.

El error se captura con catch y se lanza a una capa superior donde se trata. Ejemplo:

Tenemos un programa con dos clases: la clase Parking y la clase Programa que es donde se encuentra el main, por lo tanto donde se ejecuta toda la app.

Clase Parking

```
import java.util.ArrayList;

public class Parking {
```

```
    // attributes
    int plaza = 0;
    String nombre = "";
    ArrayList<String> matriculas = new ArrayList<String>();

    // constructors
    public Parking(String nombre, int plaza) {
        this.nombre = nombre;
        this.plaza = plaza;
    }

    // methods                  // 2.- lanzamos la excepción fuera del método
    public void entrada(String matricula, int plaza) throws Exception {

        if(matricula==null || matricula.length()<4) {
            // 1.- creamos una excepción
            throw new Exception("Matricula incorrecta");
        }

        if(matriculas.get(plaza)!=null) {
            throw new Exception("Matricula repetida");
        }

        matriculas.set(plaza, matricula);
    }
}
```

Ahora vamos a la clase Programa, que es donde hemos creado un objeto de la clase Parking, y dónde también hemos llamado al método entrada.

Clase Programa

```
import java.util.Scanner;

public class Programa {

    // attributes
    static Scanner sc = new Scanner(System.in);
    static Parking parking = new Parking("Centro", 10);

    // main
    public static void main(String[] args) {
        int opcion = 0;
```

```
        do {
// alternativa: podrías lanzar la excepción en el método "menu" (punto 3)
   y controlar la excepción en el main, o sea añadir un try / catch aquí
            opcion = menu();
        } while (opcion != 0);
    }

    // método para llamar al menú
    public static int menu() {
        int opcion = 0;
        System.out.println("\n- 1. Entrada coche");
        System.out.println("- 2. Salida coche");
        System.out.println("- 0. Salir");
        System.out.println("¿Qué quieres hacer?");

        opcion = sc.nextInt();

        switch (opcion ) {
                case 1 -> entradaCoche();
                case 2 -> salidaCoche();
        }
        return opcion ;
    }

    // methods
    public static void entradaCoche() {
        System.out.println("\nIntroduce matricula");
        String matricula = sc.nextLine();

        System.out.println("\nIntroduce una plaza");
        int plaza = sc.nextInt();

// 4.- controlamos las posibles excepciones que podría devolver el método
           "entrada" de la clase Parking mediante un try / catch
        try {
            parking.entrada(matricula, plaza);
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}
```

Prueba a ejecutar el camino alternativo que se encuentra en el main de la clase Programa. Al final, la posible excepción a ido viajando desde más adentro hacia la capa más externa:

método **entrada** de la clase **Parking** ->
-> método **entradaCoche** de la clase **Programa** ->
-> alternativa: main

Prueba esto para poder llegar a atender la diferencia entre lanzar una excepción hacia una capa más externa y controlar la excepción en el punto que se desea con try / catch.

↗ Ficheros: Grabar y leer ficheros

Existen varias formas de grabar datos en Java, desde almacenar bytes hasta almacenar objetos directamente, pasando por almacenar cadenas de caracteres (Strings).

- Se puede leer y grabar por lineas de String con BufferedReader / BufferedWriter.
- También se puede leer y escribir objetos directamente con FileOutput / FileInput.

GrabarFichero:

```
String contenido = "Este es el contenido que queremos escribir en el archivo.";

        try {
            File f = new File("K:/_Workspaces/Java/Apuntes/src/files/archivo-escritura.txt");
            FileWriter fw = new FileWriter(f);
            BufferedWriter bw = new BufferedWriter(fw);

            bw.write(contenido);

            bw.close();
            fw.close();
            System.out.println("Se ha escrito el archivo correctamente.");
        } catch (IOException e) {
            System.out.println("Ocurrió un error al escribir el archivo: " + e.getMessage());
        }
```

LeerFichero:

```
        try {
            File f = new File("K:/_Workspaces/Java/Apuntes/src/files/archivo.txt");
            FileReader fr = new FileReader(f);
            BufferedReader br = new BufferedReader(fr);

            String linea = br.readLine();

            while (linea != null) {
                System.out.println(linea);
                linea = br.readLine();
            }

            br.close();
            fr.close();
        } catch (IOException e) {
            System.out.println("Ocurrió un error al leer el archivo: " + e.getMessage());
        }
```

↗ Static

Este modificador de acceso se aplica para:

- Clases
- Variables
- Métodos

- Bloques

La palabra clave static tiene varias funcionalidades y se puede aplicar a diferentes elementos. Aquí tienes algunas de las posibles funcionalidades de static con ejemplos de código:

1. **Métodos estáticos:** Los métodos estáticos pertenecen a la clase en lugar de a las instancias de la clase, lo que significa que pueden ser llamados sin crear una instancia de la clase (como la clase Math)

```
public class MyClass {
    public static void staticMethod() {
        System.out.println("Este es un método estático.");
    }

    public static void main(String[] args) {
        MyClass.staticMethod(); // Llamada al método estático sin
instancia de la clase.
    }
}
```

2. **Variables estáticas:** Las variables estáticas pertenecen a la clase en lugar de a las instancias de la clase. Todas las instancias de la clase comparten la misma variable estática.

```
public class Counter {
    public static int count = 0; // Variable estática

    public Counter() {
        count++; // Incrementa la variable estática en cada creación de
objeto
    }

    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();

        System.out.println("Número de instancias creadas: " +
Counter.count); // Imprime 3
    }
}
```

3. **Bloques estáticos:** Los bloques estáticos se ejecutan solo una vez cuando la clase se carga en la memoria, generalmente se utilizan para inicializar variables estáticas.

```
public class ClaseCualquiera{
    static {
        System.out.println("Este es un bloque estático.");
```

```
        }

        public static void main(String[] args) {
            // No se necesita instanciar la clase para que se ejecute el
bloque estático -> aparecerá por consola: "Este es un bloque estático."
        }
    }
```

4. **Clases internas estáticas:** Las clases internas estáticas no requieren una instancia de la clase externa y pueden ser instanciadas directamente.

```
    public class OuterClass {
        static class StaticInnerClass {
            public void innerMethod() {
                System.out.println("Método de la clase interna estática.");
            }
        }

        public static void main(String[] args) {
            OuterClass.StaticInnerClass inner = new
OuterClass.StaticInnerClass();
            inner.innerMethod(); // Llamada al método de la clase interna
estática.
        }
    }
```

12. **ejercicio5: Crear una clase embedida (una clase dentro de otra)**
    → Crea una clase llamada Estatica y otra clase que es donde se va a ejecutar el programa que se llame Principal.
    → En la clase Estatica crea un atributo estático de tipo texto que se llame mensaje y que contenga "ADIOS". Ahora crea un método estático que contenga un Syso con "HOLA"
    → En la clase Principal llama al método de la clase **Estatica** sin crear ningún objeto de esa clase. Luego escribe un Syso y pinta el atributo mensaje de la clase Estatica sin crear ningun objeto de esta clase)

13. **ejercicio6: Crear una clase embedida (una clase dentro de otra)**
    → Crea una clase que se llame **Telefono** y dentro de esa clase crea otra que se llame **SistemaOperativo**
    → Crea un atributo en la clase **SistemaOperativo** que se llame **mensajePrincipal** con el siguiente contenido: "ANDROID" y luego crea un método que se llame **encenderTelefono** que muestra un Syso con "iniciando el Sistema" y muestra el valor del atributo **mensajePrincipal**
    → Ahora, dentro de la clase Telefono, crea un objeto de la clase **SistemaOperativo** y llama al método **encenderTelefono**

↗ Overloading / Sobrecarga de Métodos VS Overriding / Polimorfismo

Overloading sucede cuando, dentro de la misma clase, creas varios métodos con el mismo nombre pero con distintos parámetros, (esto es básicamente el)por ejemplo:

- Public void myMethod(int num){}
- Public void myMethod(int num1, int num2){}
- Public void myMethod(double dec1, int num1, String name){}

Overriding ocurre cuando tienes diferentes clases y todas ellas heredan de una clase padre, entonces cada subclase (o clase hija) puede sobreescribir el método heredado de la clase padre, modificando el contenido del método para recibir diferentes resultados:

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }

}
```

Polimorfismo ocurre cuando diferentes clases implementan una interfaz que tiene un método sin definir: entonces todas las clases que implementan esta interfaz pueden (y deben) modificar el método adaptándolo a sus necesidades, con lo cual, aunque tienen todas las clases un método con el mismo nombre, cada método realiza unas funciones diferentes. De ahí su nombre: polimorfismo, varias formas.

```java
// Interfaz que define un método
interface Saludable {
    void saludar();
}
// Clase que implementa la interfaz Saludable
class Persona implements Saludable {
    @Override
    public void saludar() {
        System.out.println("Hola, soy una persona.");
    }
}
// Otra clase que implementa la interfaz Saludable
class Perro implements Saludable {
    @Override
    public void saludar() {
        System.out.println("¡Guau, guau!");
    }
}
public class PruebaPolimorfismo {
    public static void main(String[] args) {
        Saludable serSaludable;

        serSaludable = new Persona();
        serSaludable.saludar(); // Llamada al método saludar de Persona

        serSaludable = new Perro();
        serSaludable.saludar(); // Llamada al método saludar de Perro
```

```
    }
}
```

↗ Incrementos / Decrementos

Para incrementar o decrementar se usa ++ o -- pero hay que tener cuidado donde se colocan estos signos:

- Si se colocan antes del número entonces el valor se ve afectado antes de realizar la operación o ser guardado.
- Si se colocan después del valor entonces se usa o guarda el valor que tenía y luego se modifica el valor (este valor puede usarse más adelante quizá)

- Opción 1

```
int a=3, b=5, c;
// se asigna "b" a "c" y luego se resta uno, pero ese restado se pierde
c = (a>b) ? a*a : b--;

// f vale 5
System.out.println(c);
```

- Opción 2

```
int d=3, e=5, f;
// se resta a "b" y luego se asigna "b" a "c"
f = (d>e) ? d*d : --e;

// f vale 4
System.out.println(f);
```

ⓘ Inferencia de tipos

Se pueden crear variables y dejar que Java asigne el tipo de variable sola con la partícula "var", pero no se puede usar en atributos de clase:

```
var array = new int[] {3,4,5};
```

```
        for (var i : array) {
            System.out.println(i);
        }
```

↗ Scope

Es la zona de actuación de una variable. O sea, delimita desde que región puede ser accesible cada variable.

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

    // Code here CANNOT use x

        // Code here CANNOT use x

      int x = 100;

      // Code here CAN use x

      System.out.println(x);

  // Code here CANNOT use x

  }
```

↗ Distintos objetos misma referencia

Misma referencia

```
/* aquí pasas la referencia del objeto p1 a un nuevo objeto llamado p2.
Ambos objetos apuntan a la misma referencia y lo que modifiques en uno se
modifica en el otro */
        Persona p1 = new Persona();
        Persona p2 = p1;
        p1.nombre = "Antoniou";

        System.out.println("P1 nombre: " + p1.nombre);
        System.out.println("P2 nombre: " + p2.nombre);
```

Lo que obtenemos por consola es:

```
P1 nombre: Antoniou
P2 nombre: Antoniou
```

Distinta referencia

En cambio si creamos dos objetos distintos:

```
/* creando otro objeto así lo estás creando de cero y se crea como un
objeto diferente */
        Persona p1 = new Persona();
        Persona p2 = new Persona();
        p1.nombre = "Antoniou";

        System.out.println("P1 nombre: " + p1.nombre);
        System.out.println("P2 nombre: " + p2.nombre);
```

Entonces el resultado es:

```
P1 nombre: Antoniou
P2 nombre:
```

↗ Recursion

Es una técnica que permite hacer que una función o método se llame a sí mismo hasta que se rompa el bucle.

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

            int result = sum(5);
          System.out.println(result);

      }

      public static int sum(int k) {

            if (k > 0) {
            // k = 5 + 4 + 3 + 2 + 1
            return k + sum(k - 1);
          } else {
            return 0;
          }
        }
```

↗ Encapsulation

La encapsulación protege los atributos para que no puedan ser accedidos directamente, sino que se deba pasar por un método público para poder acceder a un atributo.

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }

}
```

↗ Unboxing / Autoboxing

```
/*1. autoboxing
se puede asignar directamemte el tipo primitivo a la variable objeto: */
        Integer ent = 200; //autoboxing
        Double db = 45.7; //autoboxing

/*2. unboxing
se recupera el tipo primitivo asignando directamente la variable objeto a
la variable primitiva: */
        int n = ent; //unboxing
        Integer k = 30; //autoboxing
        k++; //unboxing + autoboxing
```

↗ Enums

Un ENUM es una clase especial que representa un grupo de constantes (como variables finales). Al ser considerados estos valores como constantes, pueden ser accedidos directamente sin necesidad de crear un objeto de la clase del enum.

```
enum Profesion{
  PROGRAMADOR,
```

```
    ARQUITECTO,
    DOCTOR

}
```

La forma de seleccionar un ENUM dentro de una clase:

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

        Persona p = new Persona();

        System.out.println(p.profesion); //PROGRAMADOR

        p.cambiarProfesion(Profesion.ARQUITECTO);

        System.out.println(p.profesion); //ARQUITECTO
}
```

↗ Scanner

La clase Scanner sirve para permitir al usuario introducir valores por teclado.

```
        Scanner sc = new Scanner(System.in);

        String userName = sc.nextLine();
```

↗ Ternary condition

Este operador se usa para crear expresiones condicionales en una sola linea (como un if pero mucho más reducido)

```
condition ? valueIfTrue : valueIfFalse

int value1 = 10;
int value2 = 20;

int maxValue = (value1 > value2) ? value1 : value2;
```

ⓘ Date

Tienes que importar java.time para hacer funcionar alguna de sus clases:

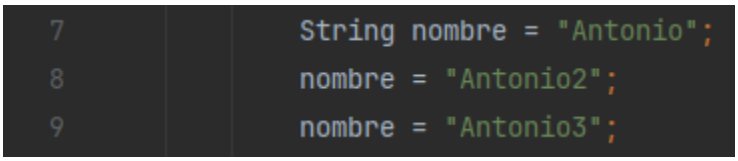
Class	Description
`LocalDate`	Represents a date (year, month, day (yyyy-MM-dd))
`LocalTime`	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
`LocalDateTime`	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
`DateTimeFormatter`	Formatter for displaying and parsing date-time objects

Por ejemplo, para mostrar la fecha actual usamos la clase `LocalDate` y llamamos al método `now()`:

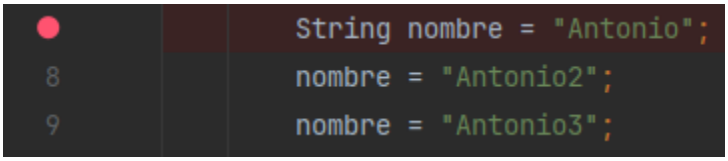
```
import java.time.LocalDate; // import the LocalDate class

public class Main {
  public static void main(String[] args) {
    LocalDate myObj = LocalDate.now(); // Create a date object
    System.out.println(myObj); // Display the current date
  }

}
```

↗ Debug

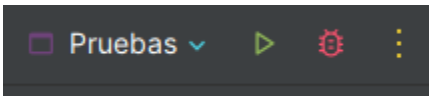
Debugear es muy fácil, solo debes crear un breakpoint (punto de ruptura) en una línea para que el programa entre en modo "debug" en la línea indicada.



Vamos a crear un breakpoint en la línea 7:



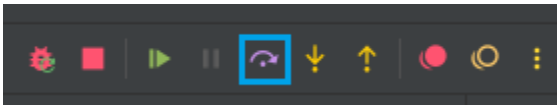
Ahora ejecutamos la app pero en modo "debug" (icono insecto rojo):



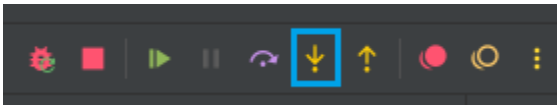
Una vez iniciado el debuggeo, veremos las variables que se están instanciando y los valores de cada una:

Cuando debuggeamos, las variables se actualizan a linea pasada, es decir, una vez hemos pasado a la siguiente línea entonces la anterior se lee y se actualizan todas las variables modificadas.

Botones:



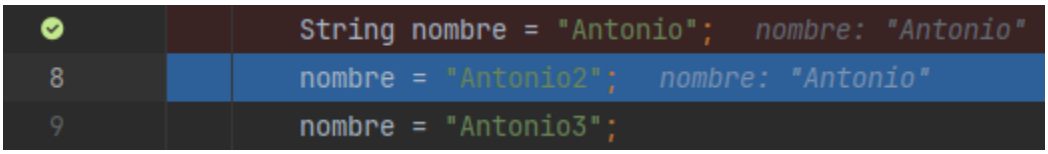
Para pasar a la siguiente línea debemos hacer click en la flecha de "Step Over":

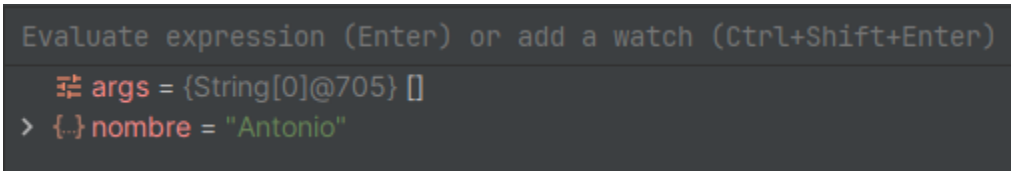


Para meterte dentro de un método que se esté ejecutando en esa línea debemos hacer click en el icono de la flecha "Step Into"

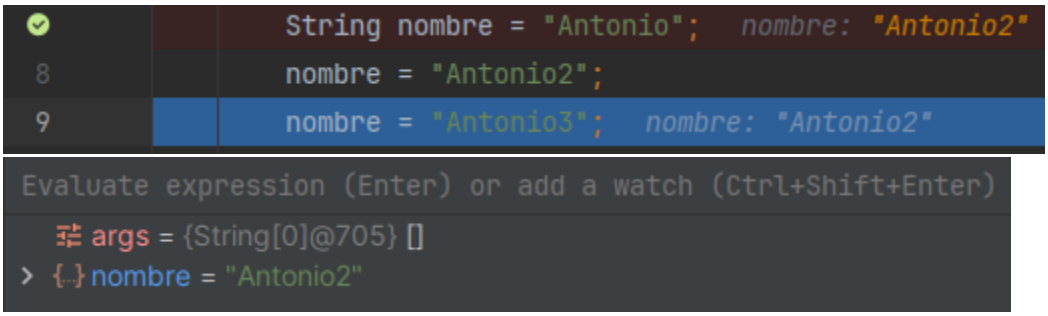
Si hacemos click en "Step Over", entonces veremos como se ejecuta la próxima línea y la variable modificada en la línea anterior (7) se actualiza.

o el valor "Antonio2" se guarda en la variable "nombre", y así sucesivamente.





Si ahora volvemos a ejecutar el botón "Step Over" veremos como el valor "Antonio2" se guarda en la variable "nombre", y así sucesivamente.



De esta forma podemos ir comprobando los valores para cada variable para detectar posibles errores o asignaciones de valores a variables incorrectas.

↗ Herencias, Clases Abstractas e Interfaces

Superclass and Subclass (extends) → a child class (subclass) can inherit attributes and methods from the parent class (superclass). If you don't want other classes to inherit from a class, use the 'final' keyword in the parent class.

The abstract keyword is a non-access modifier, used for classes and methods:

1. Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class). Basically you force the child classes - which inherit from the abstract class - to complete the methods of its parent.

2. Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

You create an "Interface" file - which is basically a completely abstract class (like a template) - where you type the methods that have to be coded. Then create a "Class" file which implements the "Interface" file and all the methods are automatically displayed in the "Class" file: now you only have to complete - by coding - those methods.

Una clase puede heredar (**extends**) de otra solo una vez; mientras que esa misma clase puede implementar (implements) ilimitadas interfaces.

↗ Modifiers

There two types of modifiers, the 'access modifiers' and the 'non-access modifiers':

≡ Access Modifier

> for classes:

→ default: the class is only accessible by classes in the same package.

→ public: the class is accessible by any other class.

→ final: if a class has this modifier, you won't be able to inherit from that class.

> for attributes, constructors and methods:

→ default: the code is only accessible in the same package.

→ public: the class is accessible for all classes.

→ private: the code is only accessible within the declared class.

→ protected: the code is accessible in the same package and subclasses. *(bueno para **inheritance** con extends)*

≡ Non-Access Modifier

> for classes:

→ 1. final: the class cannot be inherited by other classes.

→ 2. abstract: the class cannot be used to create objects (to access an abstract class, it must be inherited from another class)

> for attributes, constructors and methods:

→ 1. final: attributes and methods cannot be overridden/modified.

→ 2. static: attributes and methods belong to the class, rather than an object.

→ 3. abstract: can only be used in an abstract class, and can only be used on methods. The method does not have a body and the body is provided by the subclass. For example: `abstract void run();`.

→ *4. transient: attributes and methods are skipped when serializing the object containing them. (never used)*

→ 5. synchronized: methods can only be accessed by one thread at a time. *(never used)*

→ 6. volatile: the value of an attribute is not cached thread-locally, and is always read from the "main memory". *(never used)*

ⓘ Polymorphism

This happens when you have different classes and all of them inherit from the same class, so each subclass can overwrite the same method (also known as) to receive different outcomes.

↗ Inner Classes

Esto es crear una clase dentro de otra clase. El propósito es agrupar clases que funcionan juntas, por ejemplo:

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

↗ Iterator

Un Iterator es un objeto que es usado para recorrer colecciones, como un ArrayList o un HashSet.

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

   // Creamos el ArrayList
   ArrayList<String> cars = new ArrayList<String>();
   cars.add("Volvo");
   cars.add("BMW");
   cars.add("Ford");
```

```
    cars.add("Mazda");

    // Creamos el objeto Iterator y le asignamos el contenido del ArrayList
    Iterator<String> it = cars.iterator();

    // Recorre el ArrayList
    while(it.hasNext()) {
      System.out.println(it.next());
    }
  }
```

Es simplemente otra forma de recorrer las distintas colecciones. Esto puede ser útil para comprobar si hay siguiente elemento y actuar conforme a ello:

```
while(it.hasNext()) {

        Integer i = it.next();

        if(i < 10) {
                it.remove();
        }
}
```

ⓘ Clean Code

El **código limpio** está relacionado con la aplicación de buenas prácticas al momento de escribir código, tales como:

- Estructurar el proyecto adecuadamente.
- Escribir nombre de clases/variables/métodos descriptivos y claros.
- Evitar los side effects.
- Evitar la duplicación de código.
- Añadir comentarios al código.
- Evitar el hardcoding.
- Evitar demasiados parámetros en un método (en la medida de lo posible).
- Presentar un buen formato de código
- Crear funciones que hagan una sola cosa.

ⓘ ¿Qué es un Framework?

Un **Framework** es una estructura base utilizada como punto de partida para elaborar un proyecto con objetivos específicos, es como una especie de **plantilla** desde la que se parte para crear las apps. Los **frameworks** también brindan herramientas para hacer que el proceso de la creación del software sea más rápido y fácil (facilita la vida al programador)

ⓘ Principio DRY (Don't Repeat Yourself)

El **principio DRY (Don't Repeat Yourself)** hace referencia a que debemos eliminar las repeticiones de código de nuestro programa. Por ejemplo si tenemos este código:

```
public static void main(String[] args) {
        System.out.println("hola");
        System.out.println("hola");
        System.out.println("adios");
        System.out.println("adios");
        System.out.println("hola");
        System.out.println("hola");
        System.out.println("adios");
        System.out.println("adios");
    }
```

Hay muchos bloques repetidos. Si aplicamos el **principio DRY** podemos generar un par de funciones que se encarguen de eliminar la repetición:

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

    private static void adios() {
        System.out.println("adios");
        System.out.println("adios");
    }

    private static void hola() {
        System.out.println("hola");
        System.out.println("hola");
    }
```

ⓘ Principio YAGNI (You Aren't Gonna Need It)

Este principio se refiere a codificar solo aquello que no vamos a necesitar, en español significa "**No vas a necesitarlo**". Los programadores en ocasiones agregamos en las aplicaciones lógica "por si acaso", pensando que en el futuro la vamos a necesitar, ignorando que estamos agregando complejidad adicional con algo que no aporta valor directo a la solución.

ⓘ Principio KISS (Keep It Simple, Stupid)

KISS es un principio aplicado al desarrollo de software y a otros ámbitos. Significa «Keep It Simple, Stupid», en español "Mantenlo simple y estúpido", y nos quiere decir que las cosas sencillas funcionan mejor. O sea, que mantengamos un código simple.

ⓘ Principio DAMP (Descriptive And Meaningful Phrases)

El **principio DAMP** hace referencia a que debemos construir un código lo suficientemente descriptivo como para que pueda ser comprendido rápidamente.

```
@Test
    public void test_Nueva_Noticia() {

        SimpleDateFormat f = new SimpleDateFormat("yyyy-mm-dd");

        HashMap<String, String> json = new HashMap<>();
        json.put("titulo", "noticia4");
        json.put("autor", "cecilio");
        json.put("fecha", f.format(new Date()));
        System.out.println(json);

given().contentType(ContentType.JSON).body(json).post("api/noticias");

        get("/api/noticias").then().body("_embedded.noticias",
IsCollectionWithSize.hasSize(4));

}
```

Esto es una prueba unitaria para comprobar que un nuevo usuario ha sido insertado. El programador lo ha debido ver más claro de esta forma y así ha creado el código.

ⓘ **Principio SOLID**

Un programa va aumentando su complejidad a medida que se implementan funcionalidades, por lo que Robert C. Martin estableció 5 directrices (conocidas como el **principio SOLID**) para facilitarnos a los desarrolladores la labor de crear programas legibles y mantenibles.

El principio SOLID está compuesto por:

1. **S: Single responsibility principle** o Principio de responsabilidad única

Como su propio nombre indica, establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término "decoupled" en inglés). Si por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.

Considera este ejemplo:

```
class Coche {

    String marca;

    Coche(String marca){
      this.marca = marca;
    }

    String getMarcaCoche(){
      return marca;
    }

    void guardarCocheDB(Coche coche){
      ...
    }
}
```

💭¿Por qué este código viola el principio de responsabilidad única?

Como podemos observar, la clase Coche permite tanto el acceso a las propiedades de la clase como a realizar operaciones sobre la BBDD, por lo que la clase ya tiene más de una responsabilidad.

Para evitar esto, debemos separar las responsabilidades de la clase, por lo que podemos crear otra clase que se encargue de las operaciones a la BBDD.

2. **O: Open/closed principle** o Principio de abierto/cerrado

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación.

3. **L: Liskov substitution principle** o Principio de sustitución de Liskov

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto, el programa falla, estaremos violando este principio.

4. **I: Interface segregation principle** o Principio de segregación de la interfaz

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en dicha interfaz.

Imaginemos que queremos definir las clases necesarias para albergar algunos tipos de aves. Por ejemplo, tendríamos loros, tucanes y halcones.

```
interface IAve {
    void volar();
    void comer();
}

class Loro implements IAve{

    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //..
    }
}

class Tucan implements IAve{
    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //..
    }
}
```

Hasta aquí todo bien. Pero ahora imaginemos que queremos añadir a los pingüinos. Estos son aves, pero además tienen la habilidad de nadar. Podríamos hacer esto:

```java
interface IAve {
    void volar();
    void comer();
    void nadar();
}

class Loro implements IAve{

    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }

    @Override
    public void nadar() {
        //...
    }
}

class Pinguino implements IAve{

    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }

    @Override
    public void nadar() {
        //...
    }
}
```

El problema es que el loro no nada, y el pingüino no vuela, por lo que tendríamos que añadir una excepción o aviso si se intenta llamar a estos métodos. Además, si quisiéramos añadir otro método a la interfaz IAve, tendríamos que recorrer cada una de las clases que la implementa e ir añadiendo la implementación de dicho método en todas ellas. Esto viola el principio de segregación de interfaz, ya que estas clases (los clientes) no tienen por qué depender de métodos que no usan.
Lo más correcto sería segregar más las interfaces, tanto como sea necesario. En este caso podríamos hacer lo siguiente:

```java
interface IAve {
    void comer();
}
interface IAveVoladora {
    void volar();
}

interface IAveNadadora {
    void nadar();
}

class Loro implements IAve, IAveVoladora{

    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }
}

class Pinguino implements IAve, IAveNadadora{

    @Override
    public void nadar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }
}
```

Así, cada clase implementa las interfaces de la que realmente necesita implementar sus métodos. A la hora de añadir nuevas funcionalidades, esto nos ahorrará bastante tiempo, y además, cumplimos con el primer principio (Responsabilidad Única).

5. **D: Dependency inversion principle** o Principio de inversión de dependencia

Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:

Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.
En algún momento nuestro programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase, es cuando debemos usar inyección de dependencias, lo que nos permitirá controlar las funcionalidades desde un sitio concreto en vez de tenerlas esparcidas por todo el programa. Además, este aislamiento nos permitirá realizar testing mucho más fácilmente.

Supongamos que tenemos una clase para realizar el acceso a datos, y lo hacemos a través de una BBDD:

```
class DatabaseService{
    //...
    void getDatos(){ //... }
}

class AccesoADatos {

    private DatabaseService databaseService;

    public AccesoADatos(DatabaseService databaseService){
        this.databaseService = databaseService;
    }

    Dato getDatos(){
        databaseService.getDatos();
        //...
    }
}
```

Imaginemos que en el futuro queremos cambiar el servicio de BBDD por un servicio que conecta con una API. Para un minuto a pensar qué habría que hacer... ¿Ves el problema? Tendríamos que ir modificando todas las instancias de la clase AccesoADatos, una por una.

Esto es debido a que nuestro módulo de alto nivel (AccesoADatos) depende de un módulo de más bajo nivel (DatabaseService), violando así el principio de inversión de dependencias. El módulo de alto nivel debería depender de abstracciones.

Para arreglar esto, podemos hacer que el módulo AccesoADatos dependa de una abstracción más genérica:

```java
interface Conexion {
    Dato getDatos();
    void setDatos();
}

class AccesoADatos {

    private Conexion conexion;

    public AccesoADatos(Conexion conexion){
        this.conexion = conexion;
    }

    Dato getDatos(){
        conexion.getDatos();
    }
}
```

Así, sin importar el tipo de conexión que se le pase al módulo AccesoADatos, ni este ni sus instancias tendrán que cambiar, por lo que nos ahorraremos mucho trabajo.

Ahora, cada servicio que queramos pasar a AccesoADatos deberá implementar la interfaz Conexion:

```java
class DatabaseService implements Conexion {

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}

class APIService implements Conexion{

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}
```

Así, tanto el módulo de alto nivel como el de bajo nivel dependen de abstracciones, por lo que cumplimos el principio de inversión de dependencias. Además, esto nos forzará a cumplir el principio de Liskov, ya que los tipos derivados de Conexion (DatabaseService y APIService) son sustituibles por su abstracción (interfaz Conexion).

ⓘ Programación Imperativa vs Declarativa

Todos estamos habituados a programar y cuando lo hacemos habitualmente usamos un enfoque imperativo.

¿Qué es la programación Imperativa?
En este tipo de programación indicamos, por medio del código, como tiene que realizar el programa cada uno de los pasos. Le decimos que variables usar, que bucles y sentencias etc. Es decir construimos un algoritmo muy concreto para solventar un problema.

```
Persona p1= new Persona("pepe",20);
 Persona p2= new Persona("juan",12);
 Persona p3= new Persona("angela",30);

 List<Persona> lista=new ArrayList<Persona>();

 lista.add(p1);
 lista.add(p2);
 lista.add(p3);

 int totalEdad=0;
 int totalPersonas=0;

 for (Persona p: lista) {
     if (p.getEdad()>=18) {

         totalEdad+=p.getEdad();
         totalPersonas++;
     }
 }

 System.out.println(totalEdad/totalPersonas);
```

¿Qué es la programación Declarativa?
En el caso de la programación Declarativa lo que hacemos es solicitar al programa que realice una serie de tareas

```
Persona p1= new Persona("pepe",20);
 Persona p2= new Persona("juan",12);
 Persona p3= new Persona("angela",30);

 List<Persona> lista=new ArrayList<Persona>();

 lista.add(p1);
 lista.add(p2);
 lista.add(p3);

 OptionalDouble resultado=lista.stream().filter(persona->persona.getEdad()>=18)
               .mapToInt(persona->persona.getEdad())
               .average();

 if(resultado.isPresent()) {
     System.out.println(resultado.getAsDouble());
 }
```

↗ Clase Comparable
Pdggd

↗ Sobreescribir método EqualsTo

Pdggd

↗ TEST - Pruebas Unitarias

Pdggd

↗ TEST - Pruebas de Integración

Pdggd

↗ Test - TDD (Driven Development)

Pfvdfg

↗ Mockito (Mocks)

Pfvdfg