# Metodología de la Programación

Apuntes hechos por Brianda García, Lorena Ballesteros, Lía de Miguel) y Marina Sancho

# Sesión I: Lenguajes de Programación 💻

## 1. ¿Lenguajes?

Existen dos tipos de lenguaje:

- Naturales (No Formales):
  Dependientes del contexto, se debe tener un nexo común → Español, Italiano, Inglés …
- Formales:
  Un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados. No necesitan contexto → Lenguajes de programación …

## 2. ¿Por qué hay lenguajes de programación?

- **Lenguajes Máquina**
  Son los lenguajes de más bajo nivel: secuencias binarias de ceros y unos con tarjetas perforadas (0 sin agujero, 1 con)
  Históricamente, los primeros

- **Lenguajes Ensambladores**
  Segunda generación de lenguajes
  Versión simbólica de los lenguajes máquina (MOV, ADD, etc).

- **Lenguajes de Alto Nivel**
  Lenguajes de tercera generación (3GL), no hay flexibilidad en la máquina pero si la hay a la hora de desarrollar la creatividad, no es tan eficiente que el ensamblador.
  Estructuras de control, Variables de tipo, Recursividad, etc.
  Ej.: C, Pascal, C++, Java, etc

- **Lenguajes Orientados a Problemas: describen la solución, no cómo conseguirla**
  Describen la solución, no como conseguirla, es un paradigma sencillo
  Lenguajes de cuarta generación (4GL) Ej. SQL

## 3. ¿Necesitamos más lenguajes?

Los lenguajes de programación representan un conjunto de construcciones abstractas centradas en resolver problemas más o menos genéricos, en base a una serie de paradigmas de organización del pensamiento y su desarrollo.

Si los problemas evolucionan, es natural que los lenguajes para resolverlos también, e incluso aparezcan nuevos lenguajes, por tanto, a medida que evoluciona la tecnología y sus problemas también deben hacerlo los lenguajes

ELM, DART, TypeScript, Go, Babel, Kotlin, Rust, Crystal, Elixir,… son lenguajes que en enero de 2010 no existían.

# 4. Traductores, intérpretes y compiladores

**Traductor:**

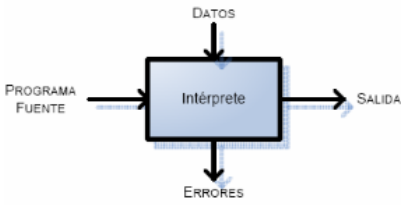
Un traductor es un programa que lee un programa escrito en lenguaje fuente de alto nivel, y lo traduce a un lenguaje objeto. Según sus funcionalidades puede ser de dos tipos:

- Intérprete
  Ejecuta directamente lo que traduce, sin almacenar en disco la traducción realizada.
- Compilador
  Genera un fichero del lenguaje objeto estipulado, que puede posteriormente ser ejecutado tantas veces se necesite sin volver a traducir.
    - Como beneficio adicional, el compilador informa de los errores del programa fuente, ya que lo analiza al completo

# 5. Tipos de traductor

## 1. Intérprete

1. El intérprete ejecuta el código según lo va leyendo.
2. Cada vez que se escribe una línea el programa comprueba si es correcta, sí lo es, la ejecuta.
3. La ejecución es interactiva.
4. Los intérpretes más puros no guardan copia del programa que se está escribiendo
    - **Ejemplos**: Procesos por lotes, JavaScript, CAML, etc.
    - El intérprete siempre debe estar presente para la traducción y ejecución.




## 2. Compilador

Un compilador es un programa que lee un programa escrito en lenguaje fuente, y lo traduce a un lenguaje objeto de bajo nivel. Además generará una lista de los posibles errores que tenga el programa fuente.


- El compilador es el traductor más extendido
- El programa ejecutable, una vez creado, no necesita el compilado para funcionari

### 2.1 Representación de un compilador

Los compiladores pueden estar implementados en lenguajes de programación distintos del lenguaje fuente (Source) y del lenguaje objeto(Target). A este tercer lenguaje se le denomina lenguaje anfitrión (Host). Por tanto, para representar en abstracto un compilador suele emplearse una representación en forma de T que incluye los 3 lenguajes que lo caracterizan.

## 6. Entorno de ejecución

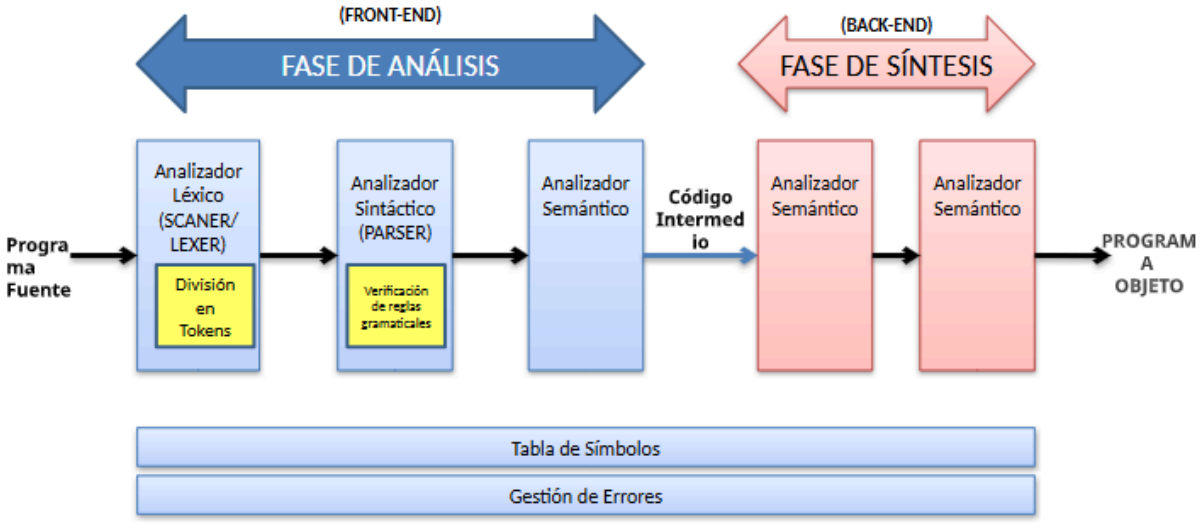
**Según implementación:**

- En metal (máquina física)
- En máquina virtual

**Según tipo de ejecución:**

- En máquina de pila
- En máquina de registros (von Newmann)

## 7. Procesamiento de un lenguaje



# Sesión II: Tipos de Programación

## 1. Escuelas de pensamiento

Para abordar los problemas de la programación, existen dos grandes escuelas de pensamiento:

- Programación funcional: es un paradigma que se basa en el uso de funciones matemáticas para transformar datos. En este enfoque, los programas se construyen mediante funciones puras que no tienen efectos secundarios ni modifican el estado del sistema. Todo el cálculo se realiza a través de la evaluación de funciones que pueden ser componibles y anidadas. Este paradigma enfatiza la inmutabilidad de los datos, lo que favorece la concurrencia y facilita la razonabilidad del código. La programación funcional es un paradigma declarativo, centrado en el "qué hacer" más que en el "cómo hacerlo", lo que permite escribir programas más predictivos y fáciles de testear.

- Programación Orientada a Objetos: organiza el software en torno a objetos, que son instancias de clases que encapsulan tanto datos (atributos) como comportamientos (métodos). Este paradigma es más intuitivo, ya que refleja cómo los humanos tienden a entender y organizar los problemas, representando entidades del mundo real. Los objetos interactúan entre sí, permitiendo segmentar problemas de manera modular. Los conceptos clave de la POO incluyen herencia, polimorfismo, encapsulamiento y abstracción, los cuales permiten crear programas más reusables, fáciles de mantener y ampliar, facilitando la resolución de problemas complejos mediante la organización y la reutilización del código.

La abstracción en Programación Orientada a Objetos (POO) oculta los detalles internos de los objetos y muestra solo lo esencial para interactuar con ellos. Las clases definen los objetos y sus métodos, pero el programador no necesita conocer cómo se implementan esos métodos, solo qué hacen. Esto facilita la creación de programas más simples y reutilizables, ya que se puede usar un objeto sin conocer su implementación interna.

## 2. Conceptos clave

- **Programación orientada a componentes:**

  La programación orientada a componentes es un paradigma que facilita el desarrollo de software modular, reutilizable y mantenible. Se basa en la creación de componentes independientes que pueden interactuar entre sí.

- **Interfaces (GUI).**

  Las interfaces gráficas de usuario (GUI) actúan como la frontera de comunicación entre diferentes componentes o sistemas. Son esenciales para establecer contratos de interacción definidos por las empresas.

- **Abstracción.**

  La abstracción permite centrarse en la funcionalidad sin preocuparse por los detalles internos de la implementación. Sin embargo, debe usarse con moderación, ya que una abstracción excesiva puede dificultar el desarrollo y mantenimiento del software.

- **Reutilización del código.**

  Uno de los principales beneficios de la programación orientada a componentes es la posibilidad de reutilizar código. Esto reduce la duplicidad, mejora la eficiencia y facilita la mantenibilidad del software.

- **Encapsulamiento.**

  El encapsulamiento permite ocultar los detalles internos de un componente, asegurando que solo se exponga la funcionalidad necesaria. Esto se logra mediante:

  - **Principio de ocultación**: Impide el acceso directo a los detalles internos de un componente, garantizando seguridad y reduciendo la complejidad.
  - **Reducción de puntos de exposición**: Se limita el acceso a los elementos esenciales, evitando interferencias no deseadas.
  -

- **Modularidad.**

  Cuando trabajamos con programación, trabajamos con objetos. Para que funcione debe de trabajar todos juntos y moldearlos para que trabajen de forma independiente pero juntos.

  La modularidad implica dividir un sistema en módulos independientes que trabajan de forma conjunta. Cada módulo tiene una responsabilidad específica y se diseña para interactuar con otros sin perder independencia. Siendo un programador pensamos en cómo unir esos módulos para que puedan trabajar con ellos y actúen de forma coordinada.

- **Herencia y Polimorfismo.**

  Son herramientas esenciales en el diseño de software orientado a objetos:

  - **Herencia:** Permite establecer relaciones jerárquicas entre clases, facilitando la reutilización de código y la extensibilidad del sistema.
  - **Polimorfismo**: Permite que diferentes clases respondan de manera distinta a un mismo mensaje o método, mejorando la flexibilidad y escalabilidad del software.

  **Ejemplo:** Dos clases pueden compartir el mismo nombre de método, pero implementar funcionalidades distintas según su estructura de datos.

- **Recolección de basura.**

  Es una funcionalidad propia de los lenguajes orientados a objetos que gestiona automáticamente la memoria. Cuando se crean nuevos objetos, el sistema asigna memoria; cuando estos objetos dejan de ser utilizados, la recolección de basura los elimina para liberar espacio.

Este proceso se ejecuta periódicamente y optimiza el uso de recursos del sistema

- **Programación orientada a objetos**

    - **Clase**: es una plantilla que define los atributos (propiedades) y métodos (comportamientos) de los objetos que se crean a partir de ella. No es un objeto en sí, sino una descripción general de sus características y funcionalidades. Los objetos son instancias concretas de una clase, con valores específicos para sus atributos y la capacidad de ejecutar sus métodos.

    - **Objeto:** es una instancia concreta de una clase, almacenada en memoria, y con la cual podemos interactuar y rastrear.

    - **Métodos**: son las funcionalidades que definen lo que un objeto puede hacer, es decir son las acciones que un objeto puede ejecutar.

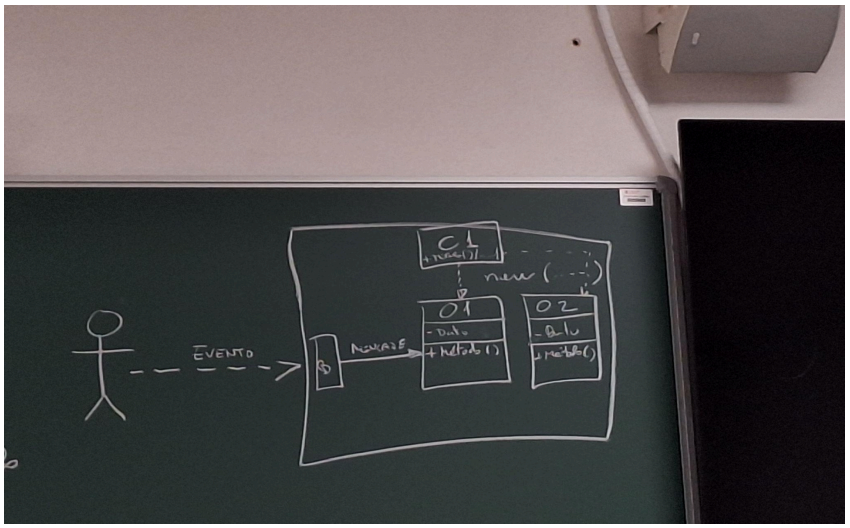
    - **Mensaje**: es la invocación de un método, es decir, cuando pedimos a un objeto que ejecute una acción.

    - **Evento**: es un mensaje externo que provoca una acción en el objeto. Los eventos suelen ser generados por el usuario o por otras fuentes externas.

    - **Atributo/propiedad**: son los datos variables que describen las características de un objeto. Los atributos se almacenan dentro del objeto y pueden cambiar a lo largo del tiempo.

    - **Identificador**: Es el número único asignado a cada objeto en memoria para poder identificarlo y rastrearlo.

    - **Estado interno**: son los valores actuales de los atributos de un objeto, que determinan el comportamiento del objeto en un momento dado.



## 3. Fases canónicas del desarrollo software

**Definición:** son un conjunto de etapas estándar que guían el proceso de creación de un sistema de software desde su concepción hasta su entrega y mantenimiento.

1. **Análisis:** en esta fase, se identifican y definen **los requisitos** del software, es decir, lo que debe hacer el sistema. Los **usuarios** e **interesados** proporcionan información sobre las necesidades y expectativas del software. El objetivo principal es **comprender el problema** y documentar los requisitos de manera detallada, para asegurar que el desarrollo se enfoque en las soluciones correctas.

2. **Diseño:** se define cómo va a estar estructurado el software para cumplir con los requisitos identificados. Se crea una arquitectura del sistema y se diseñan los componentes del software, como las clases, interfaces y bases de datos. El diseño también incluye la planificación de la interacción entre los módulos del sistema y la definición de las tecnologías y herramientas que se usarán.

3. **Programación:** es la fase en la que los desarrolladores escriben el código del software según los diseños definidos. En esta etapa, se implementan las funcionalidades del sistema utilizando el lenguaje de programación y las herramientas seleccionadas durante la fase de diseño. Es una fase crítica, ya que es donde el sistema empieza a tomar forma como un software funcional.

4. **Pruebas:** una vez que el software está desarrollado, se realiza una serie de pruebas para verificar que el sistema cumple con los requisitos especificados en la fase de análisis y funciona de acuerdo con lo esperado. Esto incluye pruebas unitarias (para partes individuales del código), pruebas de integración (para asegurar que los módulos funcionan juntos), pruebas de sistema (para probar el software en su conjunto) y pruebas de aceptación (para confirmar que el software cumple con las expectativas del usuario).

Estas fases no siempre se siguen de forma estricta, ya que el desarrollo de software puede ser iterativo y puede involucrar revisiones y mejoras durante el proceso. Sin embargo, estas etapas son fundamentales para garantizar que el producto final sea de calidad y cumpla con las necesidades del usuario.

**Observación:** Es muy difícil si te encuentras en un nivel bajo arreglar algo de arriba porque tienes que subir y luego bajar.

# 4. Diagramas

Los diagramas son representaciones visuales utilizadas en programación para planificar, diseñar y documentar sistemas. Facilitan la comprensión del código, la comunicación entre desarrolladores y la organización del proyecto. Existen distintos tipos según lo que se quiera representar, como la estructura, el comportamiento o la interacción del software.

- **Diagrama de Cosas de Uso**
  - Permite visualizar gráficamente el funcionamiento del código.
  - Representa las interacciones entre los actores (usuarios o sistemas externos) y el software.
  - Se utiliza en la fase de análisis para entender los requisitos del sistema.
- **Diagrama de Clases**
  - Proporciona una imagen estática y general del programa que se va a construir.
  - Muestra las clases, atributos, métodos y las relaciones entre ellas.
  - Es fundamental en el diseño orientado a objetos.

- **Diagrama de Secuencia**
  - Ilustra qué ocurre cuando se ejecuta el programa correctamente.
  - Muestra la interacción entre objetos a lo largo del tiempo.
  - Es útil para comprender el flujo de mensajes y la secuencia de eventos en un sistema.

- **Diagrama de Estados**
  - Representa objetos con un estado interno especial que modifica el comportamiento de la clase.
  - Describe cómo un objeto cambia de estado en respuesta a eventos.
  - Se usa en sistemas con ciclos de vida complejos o máquinas de estados.
  -
- **Diagrama de Componentes**
  - Representa los módulos agrupados y sus relaciones.
  - Muestra la estructura física del software, indicando cómo los distintos componentes se conectan.
  - Es clave en la arquitectura del software para visualizar dependencias y modularidad.

# SESIÓN 3: Modificadores

## Modificadores de Acceso

En Java, los modificadores de acceso determinan la visibilidad de los atributos y métodos dentro y fuera de una clase. Son fundamentales para el encapsulamiento y la protección de datos dentro de un programa orientado a objetos. Existen tres principales:

- **Public (Público)**: Permite el acceso desde cualquier parte del programa, ya sea dentro o fuera del paquete donde se definió la clase.
- **Private (Privado)**: Restringe el acceso únicamente a la misma clase donde se definió el atributo o método. No puede ser accedido ni por subclases ni por otras clases dentro del mismo paquete.
- **Protected (Protegido)**: Permite el acceso dentro de la misma clase, en subclases y dentro del mismo paquete. Es útil cuando se quiere que una subclase tenga acceso a los atributos y métodos sin hacerlos completamente públicos.

### Ejemplo:

```
package es.uah.matcomp.mp.el1.eja.e1;

	public class MiClase {
		public int dato;   //Visible desde cualquier parte
		private String secreto;  // Solo accesible dentro de esta clase
		protected int valorProtegido;  // Visible dentro del paquete y por subclases
	}
```

## Paquetes en Java

Un paquete es un mecanismo para organizar y agrupar clases relacionadas de manera estructurada. Los paquetes ayudan a evitar conflictos de nombres y a mejorar la mantenibilidad del código. En Java, los paquetes se definen con la palabra clave `package` al inicio del archivo fuente.

**Ejemplo:**

```
package es.uah.matcomp.mp.el1.eja.e1;

        public class MiClase {
                public int dato;
        }
```

Para compilar y ejecutar una clase dentro de un paquete, se debe seguir la estructura de directorios correspondiente y utilizar el nombre del paquete en la declaración de la clase.

## Anotaciones en Java

Las anotaciones son metadatos que proporcionan información adicional al compilador y a las herramientas de tiempo de ejecución. Se utilizan para mejorar la legibilidad del código y facilitar la automatización de ciertas tareas. Algunas anotaciones comunes en Java son:

- @Override: Indica que un método sobrescribe uno de la superclase.
- @Deprecated: Marca un método o clase como obsoleto, sugiriendo no usarlo en nuevas implementaciones.
- @SuppressWarnings: Suprime advertencias específicas del compilador.

**Ejemplo:**

```
@Override
   public String toString() {
          return "Representación en cadena del objeto";
}.
```

# Herencia en Java

La herencia es un mecanismo que permite que una clase (subclase) adquiera los atributos y métodos de otra clase (superclase). Esto favorece la reutilización del código y la organización jerárquica de las clases.

## Ejemplo:

```
public class Vehiculo {
        private String matricula;

        public String getMatricula() {
           return matricula;
    }
        public void setMatricula(String m) {
           this.matricula = m;
    }
}
```

## Para crear una subclase que herede de Vehículo:

```
public class VRuedas extends Vehiculo {
        private int numRuedas;

        public int getNumRuedas() {
           return numRuedas;
    }

        public void setNumRuedas(int n) {
           this.numRuedas = n;
    }
}
```

## Ejemplo de herencia múltiple usando interfaces:

```
interface Terrestre {
        void conducir();
}

        interface Acuatico {
           void navegar();
}

  class Anfibio implements Terrestre, Acuatico {
        public void conducir() {
          System.out.println("Conduciendo en tierra");
    }
        public void navegar() {
          System.out.println("Navegando en agua");
    }
}
```

## Clases Abstractas

Las clases abstractas son aquellas que no pueden ser instanciadas directamente. Sirven como modelo para otras clases y pueden contener métodos abstractos, es decir, métodos sin implementación que deben ser definidos en las subclases.

### Ejemplo:

```
abstract class Vehiculo {
    private String matricula;

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String m) {
        this.matricula = m;
    }

    abstract void mover();  // Método abstracto que debe implementarse en subclases
}
```

Las subclases que hereden de Vehículo deben proporcionar una implementación para el método mover()

## Test Unitarios

Las pruebas unitarias permiten verificar que las clases y métodos funcionan correctamente. En Java, se utiliza la biblioteca JUnit para escribir y ejecutar pruebas automáticas.

## Ejemplo de prueba unitaria con JUnit:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class VehiculoTest {
    @Test
    public void testSetMatricula() {
        Vehiculo v = new Coche();
        v.setMatricula("1234ABC");
        assertEquals("1234ABC", v.getMatricula());
    }
}
```

Esto ejecuta una prueba para verificar que el método setMatricula funciona correctamente.

# SESIÓN 4: Multiplicidad y cardinalidad

## 1. Multiplicidad

La multiplicidad en una relación indica el número de instancias de una clase que pueden estar asociadas con una instancia de otra clase.

### 1.1. Tipos de multiplicidad

Cuando hablamos de relación, puede haberla o no haberla. Para el máximo, puede ser que tenga relación para dos instancias concretas o puede ser que tenga múltiples relaciones (podemos tener un coche (1) o muchos (n)). No representan números concretos, sino valores del entorno.

- (0,1): Mínimo cero, máximo uno. Puede o no haber relación.
- (0,n): Mínimo cero, máximo muchos. Puede no haber relación o haber varias.
- (1,1): Exactamente uno.
- (1,n): Mínimo uno, máximo muchos. Siempre existe al menos una relación.

La correcta elección de multiplicidad depende de las condiciones del problema.

### 1.2. Factores a considerar en la multiplicidad

- Un **cero** representa la ausencia de relación.
- Un **uno** indica que debe haber al menos una relación.
- Un **n** indica que pueden existir varias relaciones.

**Ejemplo:**

poseeCoche → ( , 1 ) → Clases (una persona puede poseer un solo coche).

poseeCoche → ( , n ) → Clases (una persona puede poseer varios coches).

Para evitar que el primer valor sea nulo, se puede manejar a través del constructor para anular valores manualmente.

## 2. Cardinalidad

La cardinalidad también se expresa en el sentido contrario de la relación, es decir, cuántas instancias de una entidad pueden estar relacionadas con una instancia de otra entidad.

### 2.1. Las principales categorías de cardinalidad

1. **1:1** - Ambos lados tienen el mismo valor en el segundo dígito (por ejemplo, (0,1) y (0,1) o (0,n) y (0,n)).
2. **1:N** - Una instancia de una entidad está relacionada con múltiples instancias de otra entidad.

3. **N:M** - Múltiples instancias de una entidad están relacionadas con múltiples instancias de otra entidad.

**Ejemplo:**

poseeCoche [ ] —> esPoseidoPor [ ] → Esto sería un **1:N**, pero si se extiende, puede formar un **N:M**.

# 3. Relaciones

## 3.1. Relación de asociación

Es la relación más general entre clases y se establece mediante variables en cada clase.

**Ejemplo:**

```
class Persona {
      Coche coche; // Relación 1:1
}
```

Si se desea permitir múltiples relaciones, se usa una lista o array:

```
class Persona {
   List<Coche> coches; // Relación 1:N
}
```

## 3.2. Relaciones de agregación

- Se representa con un **rombo vacío**.
- Indica una relación de "tiene un" pero donde la existencia de los objetos es independiente.
- Ejemplo: Un coche **tiene** un motor, pero el motor puede existir independientemente del coche.
- UML:

Coche ◇—> Motor

**Ejemplo:**

```
class Coche {
   Motor motor;
   public Coche(Motor motor) {
         this.motor = motor;
   }
}
```

### 3.3. Relaciones de composición

- Se representa con un **rombo relleno**.
- Indica una relación de "es parte de" donde la existencia de los objetos depende uno del otro.
- Ejemplo: Un coche **contiene** asientos, y los asientos no existen sin el coche.
- UML:

Coche ◆——> Asiento

**Ejemplo:**

```
class Coche {
    private List<Asiento> asiento
    public Coche() {
          this.asientos = new ArrayList<>();
          asientos.add(new Asiento()); // Creando asientos al crear coche
    }
}
```

## 4. Mínimo y máximo

### 4.1. ¿Por qué es importante definirlos?

- Determinan cuántas instancias pueden estar relacionadas.
- Evitan valores incorrectos en el diseño de la base de datos o el modelo de objetos.
- Aseguran que los objetos sean creados con las relaciones correctas.

**Ejemplo práctico en programa:**

```
class Persona {
      private Coche coche;

      public Persona(Coche coche) {
            if (coche == null) {
                throw new IllegalArgumentException("Una persona debe tener un coche");
             }
             this.coche = coche;
      }
}
```

# Prácticas

## 1. Prácticas generales

### - Variables y tipos de datos

| TIPO | DESCRIPCIÓN | EJEMPLO |
|---|---|---|
| int | Números enteros | int edad = 25; |
| double | Números decimales | double precio = 12.99; |
| char | Un solo carácter | char letra = “B”; |
| boolean | Verdadero o falso | boolean activo = true; |
| String | Texto (no primitivo, es un objeto) | String nombre = “Juan” |

### - Operadores

| OPERADOR | DESCRIPCIÓN | EJEMPLO |
|---|---|---|
| + | Suma | int suma = 5+3: |
| - | Resta | int resta= 5-3: |
| * | Multiplicación | int multiplicación= 5*3: |
| / | División | double division = 10/3 |
| % | Módulo (resto) | int resto= 5%3: |
| == | Comparación (igual) | if (a == b) |
| != | Diferente | if (a != b) |
| && | AND lógico | if (a > 0 && b < 10) |

## - Estructuras de control

CONDICIONAL ( if - else )

```
int edad = 18;
if (edad >= 18) {
  System.out.println("Eres mayor de edad.");
 } else {
       System.out.println("Eres menor de edad.");
 }
```

SWITCH (para múltiples opciones)

```
int dia = 3;
switch (dia)  {
   case 1:
      System.out.println("Lunes");
      break;
    case 2:
      System.out.println("Martes");
      break;
    default:
       System.out.println("Otro día"); }
```

- BUCLES JAVA

BUCLE for

```
for (int i = 0; i < 5; i++) {
     System.out.println("Número: " + i); }
```

BUCLE while

```
int i = 0;
while (i < 5) {
     System.out.println("Número: " + i);
     i ++ ;
}
```

BUCLE do-while (ejecuta al menos una vez)

```
int i = 0;
do {
     System.out.println("Número: " + i);
     i ++ ;
} while (i < 5);
```

- **MÉTODOS EN JAVA**

Los métodos permiten reutilizar código y dividir el programa en partes más pequeñas.

```
public class Ejemplo {
      // Método sin retorno
      public static void saludar() {
            System.out.println("¡Hola!");
      }
      // Método con retorno
      public static int sumar(int a, int b) {
            return a + b;
      }
      public static void main(String[] args) {
            saludar();
            int resultado = sumar(5, 3);
            System.out.println("Suma: " +
resultado);
      }
   }
```

- **PROGRAMACIÓN ORIENTADA A OBJETOS (POO)**

**Clases y objetos**

```
public class Persona {
      String nombre;
      int edad;
      // Constructor
      public Persona(String nombre, int edad) {
            this.nombre = nombre;
            this.edad = edad;
      }
       // Método
      public void saludar() {
          System.out.println("Hola, soy " + nombre);
      }
      public static void main(String[] args) {
          Persona p1 = new Persona("Juan", 25);
      p1.saludar();
      }
}
```

- this.nombre = nombre; → Se usa this para referirse a la variable de instancia.
- Persona p1 = new Persona("Juan", 25); → Se crea un objeto de

# - Funciones

## 1. STRING.FORMAT()

Se usa para formatear cadenas de texto. Te permite insertar valores en una cadena con un formato específico.

### 1.1. Formato de números decimales

```
double value = 123.4567;
String formatted = String.format("%.2f", value); // Redondea a 2 decimales
System.out.println(formatted); // Salida: 123.46
```

### 1.2. Formato con múltiples valores

```
String name = "Alice";
int age = 25;
String output = String.format("Name: %s, Age: %d", name, age);
System.out.println(output); // Salida: Name: Alice, Age: 25
```

### 1.3. Alineación y relleno con espacios

```
System.out.println(String.format("|%-10s|%10s|", "left", "right"));
```

**SALIDA**

```
|left      |     right|
```

### 1.4. Formato de números con ceros a la izquierda

```
int number = 5;
String padded = String.format("%04d", number);
System.out.println(padded); // Salida: 0005
```

## 2. MATH.ROUND(), MATH.CEIL(), MATH.FLOOR()

```
System.out.println(Math.round(4.6));  // Salida: 5
System.out.println(Math.ceil(4.2));   // Salida: 5
System.out.println(Math.floor(4.8));  // Salida: 4
```

## 3. SUBSTRING(), TOUPPERCASE(), TOLOWERCASE()

```
String text = "Hello World";
System.out.println(text.substring(0, 5)); // Salida: Hello
System.out.println(text.toUpperCase());  // Salida: HELLO WORLD
System.out.println(text.toLowerCase());  // Salida: hello world
```

### 4. REPLACE()

```
String replaced = "apple".replace("a", "o");
System.out.println(replaced); // Salida: opple
```

### 5. CHARAT(INT INDEX) - OBTENER UN CARÁCTER EN UNA POSICIÓN ESPECÍFICA

```
String text = "Java";
System.out.println(text.charAt(1)); // Salida: 'a'
```

### 6. LENGTH() - OBTENER LA LONGITUD DE LA CADENA

```
String text = "Hola Mundo";
System.out.println(text.length()); // Salida: 10
```

### 7. INDEXOF(STRING S) - ENCONTRAR LA POSICIÓN DE UN CARÁCTER O PALABRA

```
String text = "Programación";
System.out.println(text.indexOf("g"));  // Salida: 3
System.out.println(text.indexOf("ción")); // Salida: 7
System.out.println(text.indexOf("x")); // Salida: -1 (no encontrado)
```

### 8. EQUALS() Y EQUALSIGNORECASE() - COMPARAR CADENAS

```
String a = "Hola";
String b = "hola";

System.out.println(a.equals(b)); // Salida: false
System.out.println(a.equalsIgnoreCase(b)); // Salida: true
```

### 9. TRIM() - ELIMINAR ESPACIOS EN BLANCO AL INICIO Y AL FINAL

```
String text = "   Hola   ";
System.out.println(text.trim()); // Salida: "Hola"
```

### 10. SPLIT(STRING REGEX) - DIVIDIR UNA CADENA EN UN ARRAY

```
String text = "rojo,azul,verde";
String [ ] colores = text.split(",");

for (String color : colores) {
```

```
    System.out.println(color);
}
// Salida:
// rojo
// azul
// verde
```

## 11. Funciones matemáticas (Math)

### - Math.max() y Math.min() - Máximo y mínimo entre dos números

```
System.out.println(Math.max(10, 20)); // Salida: 20
System.out.println(Math.min(10, 20)); // Salida: 10
```

### - Math.pow(base, exponente) - Potencia

```
System.out.println(Math.pow(2, 3)); // Salida: 8.0
```

### - Math.sqrt(numero) - Raíz cuadrada

```
System.out.println(Math.sqrt(16)); // Salida: 4.0
```

### - Math.random() - Generar un número aleatorio entre 0 y 1

```
System.out.println(Math.random()); // Salida: Un número aleatorio entre 0.0 y 1.0
```

### - Si quieres un número entre 1 y 100:

```
int randomNumber = (int) (Math.random() * 100) + 1;
System.out.println(randomNumber);
```

### - ¿Cuándo utilizar Get, Set o ambos?

Los métodos **get** y **set** (también llamados getters y setters) se utilizan para acceder y modificar atributos privados en una clase.

## Regla básica:

- Usa get cuando quieras permitir la lectura del atributo.
- Usa set cuando quieras permitir la modificación del atributo.
- Usa ambos (get y set) si el atributo debe ser leído y modificado.

## Casos en los que se usa get, set o ambos:

- Solo get (Solo lectura)

Cuando un valor **no debe cambiar después de ser inicializado**. Ejemplo: Un número de identificación (DNI, ID de empleado, etc.):

```
class Empleado {

        private final String dni;

        public Empleado(String dni) {

                this.dni = dni; // Se asigna en el constructor y no cambia}

        public String getDni() {

                 return dni;}}
```

- Solo set (Solo escritura)

Cuando quieres modificar un valor, pero no permitir que lo lean directamente. Ejemplo: Un sistema donde se almacena una contraseña.

```
class Usuario {
        private String contrasena;

        public void setContrasena(String contrasena) {
                this.contrasena = contrasena;}}
```

## 5. ¿Cómo hacer un Test Main?

Un TestMain en Java es una clase con un método main() que se usa para probar si otras clases funcionan correctamente. Aquí te explico paso a paso cómo hacerlo.

- **ESTRUCTURA BÁSICA DE UN TestMain**

```
public class TestMain {
        public static void main(String[] args) {
            // 1. Crear objetos de la clase a probar
            Clase objeto = new Clase();

            // 2. Usar métodos de la clase
            objeto.metodo();

            // 3. Imprimir resultados para verificar si funcionan correctamente
            System.out.println(objeto);
        }}
```

- **EJEMPLO CON CLASE Persona”**

```
class Persona {
        private String nombre;
           private int edad;
         public Persona(String nombre, int edad) {
              this.nombre = nombre;
              this.edad = edad;
        }
        public String getNombre() {
           return nombre;
        }
        public int getEdad() {
         return edad;
        }
        public void cumplirAños() { edad++; } @Override
        public String toString() {
        return "Persona[nombre=" + nombre + ", edad=" +
        edad + "]"; }

}
```