Curso Java SE

# 1 - Introducción a Java SE

## ***¿Qué es Java?***

Java es un lenguaje de programación orientado a objetos de alto nivel, creado en 1991 por James Goslim mientras trabajaba en Sun Microsystems, este lenguaje fue creado para ser multiplataforma y poder funcionar en diferentes sistemas operativos mediante una máquina virtual que se instala en cada equipo, este lenguaje fue comprado por Oracle en el año de 2009 pero personas o comunidades Open source le dan mantenimiento a Java haciendolo mas accesible a todas las personas.

Java sigue la **filosofía** de Write Once, Run Anywhere (WORA) gracias a su máquina virtual, también todo lo que aprendas en Java SE lo podrás aplicar también con Java EE.

Java tiene dos categorías:

* Standard Edition - Para construir aplicaciones de escritorio o consola.
* Enterprise Edition - Para que las empresas trabajen aplicaciones web de última generación.

## ***Versiones***

Las APIs de desarrollo con Java han evolucionado con el tiempo, por lo que existen diferentes versiones de java que puedes utilizar. La versión que más elevó la popularidad y las ofertas de trabajo con Java fue Java SE 6.

En Java SE 9 anunciaron que las actualizaciones ocurrirían cada 6 meses, pero las versiones LTS (Long Time Support) tendrán mantenimiento por 3 años, así que las actualizaciones son necesarias, pero no urgentes.

En este curso vamos a trabajar con la versión Java SE 11 LTS, la primera versión de Java con licencia. Solo podremos usarlo gratis cuando trabajamos en ambientes de desarrollo y testing. De otra forma, debemos pagar 2.5 USD al mes por usuario de escritorio y 25 USD por procesador para aplicaciones de servidor.

Afortunadamente, OpenJDK es una versión gratis y open source de usar Java SE Platform Edition.

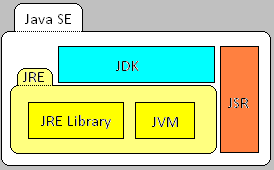
## ***Arquitectura***

A continuación vamos a ver de qué componentes está compuesto Java SE revisando su arquitectura y describiendo la función de cada capa o componente para su mejor comprensión.

El JDK o Java Development Kit el conjunto de componentes o herramientas que permiten que se pueda programar en Java este se compone de los siguientes elementos:

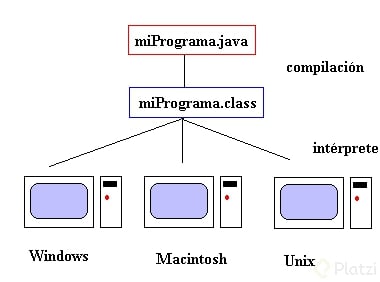
* **Compilador de Java**: El encargado de traducir nuestro código en Java a un lenguaje bytecode que puede entender e interpretar nuestra máquina virtual.
* **Java Runtime Environment (JRE)**: Es un entorno de ejecución que permite ejecutar aplicaciones Java. Incluye la Máquina Virtual de Java (**JVM**), las bibliotecas estándar y otras herramientas necesarias para interpretar y ejecutar el código **bytecode** generado por el compilador de Java.
  + La JVM, como parte del JRE, crea una capa de abstracción independiente del sistema operativo del host, lo que garantiza que el **bytecode** se ejecute de manera consistente en cualquier máquina que tenga un JRE compatible. Sin embargo, el JRE en sí no compila ni genera bytecode; solo se encarga de ejecutarlo.

Esta es una imagen donde se representa toda la arquitectura y entorno de Java SE.

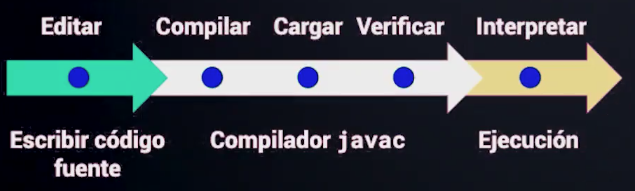


## ***¿Por qué Java es multiplataforma?***

El lenguaje Java es a la vez compilado e interpretado. Con el compilador se convierte el código fuente que reside en archivos cuya extensión es .java, a un conjunto de instrucciones que recibe el nombre de bytecodes que se guardan en un archivo cuya extensión es .class. Estas instrucciones son independientes del tipo de ordenador. El intérprete ejecuta cada una de estas instrucciones en un ordenador específico (Windows, Macintosh, etc). Solamente es necesario, por tanto, compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un ordenador.

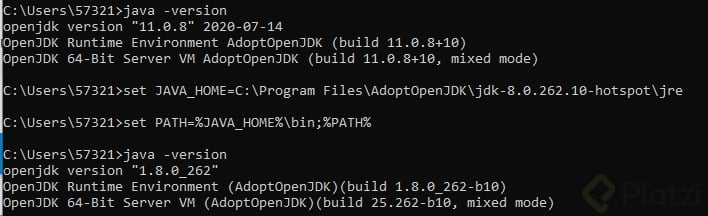


Cada intérprete de Java es una implementación de la máquina virtual Java (JVM).

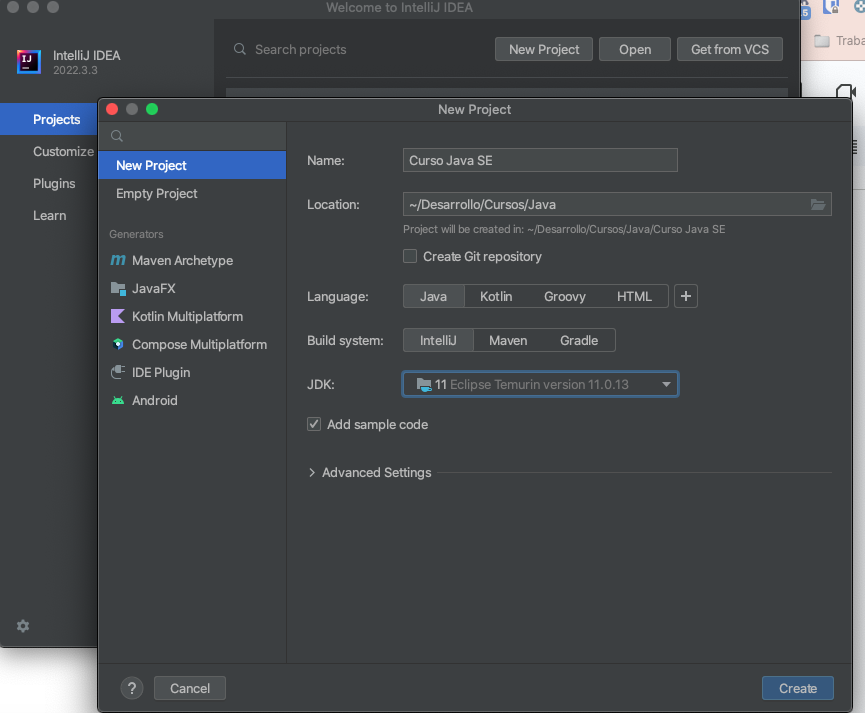


## ***Cambiar de versiones Java***

Si cuentas con varias versiones de Java instaladas en tu ordenador y deseas hacer un cambio de versión para poder trabajar con una en específico solamente se tiene que cambiar los valores de las variables de entorno:



Otra alternativa es descargar y seleccionar el JDK requerido fácilmente la versión de java usar mediante el IDE Intellij IDEA al crear nuestro proyecto.



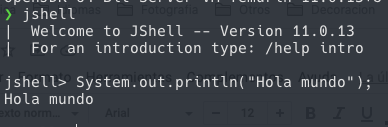
## ***JShell***

Sabías que Java tiene una herramienta interactiva en dónde puedes ir probando segmentos de código en vez de realizar todo el proceso de creación de un programa en Java. Escribir, compilar y correr.

Su nombre es jshell y está disponible desde la versión 9 de Java.

Para acceder a ella solamente desde consola se escribe el comando:

"jshell"



## ***Archivos JAR***

Los ficheros Jar (Java Archives) permiten recopilar en un sólo fichero varios ficheros .class, reuniendolos en un formato comprimido para que ocupen menos espacio. Es por tanto, algo similar a un fichero .zip (de hecho están basados en ficheros .zip) para luego ser ejecutados por la máquina virtual (JVM).

La particularidad de los ficheros .jar es que no necesitan ser descomprimidos para ser usados, es decir que el intérprete de Java es capaz de ejecutar los archivos comprimidos en un archivo jar directamente.

Para poder correr estos archivos .jar es necesario que la máquina virtual instalada en el equipo sea igual o mayor a la versión usada para compilarse el .jar.

# 2 - Fundamentos

## ***Variables***

Una variable es un contenedor que almacena datos que pueden ser utilizados y modificados durante la ejecución del programa. En Java, cada variable debe tener un tipo definido y un nombre.

### **Variables locales**

Declaradas dentro de un método o bloque, solo son accesibles dentro de ese método o bloque.

public void ejemplo() {

int numero = 10; // Variable local

System.*out*.println(numero);

}

### **Variables de instancia**

Declaradas dentro de un método o bloque, solo son accesibles dentro de ese método o bloque.

public class Persona {

String nombre; // Variable de instancia

int edad; // Variable de instancia

}

### **Variables estáticas**

Declaradas con la palabra clave static. Estas son compartidas por todas las instancias de la clase.

public class Contador {

static int *total* = 0; // Variable estática

}

Ejemplo de declaración de variables:

public class EjemploVariables {

public static void main(String[] args) {

// Variable local

int edad = 25;

// Variables de instancia (se declaran dentro de una clase, fuera de métodos)

Persona persona = new Persona();

persona.nombre = "Juan";

persona.edad = 30;

// Variable estática

Contador.total = 1;

// Constante

final double PI = 3.14159;

// Imprimir variables

System.*out*.println("Edad: " + edad);

System.*out*.println("Nombre: " + persona.nombre);

System.*out*.println("Total: " + Contador.total);

System.*out*.println("PI: " + PI);

}

}

| **⚠️ Nota:**   * Desde la versión de Java10 se implementó un tipo de dato genérico que se puede usar para todo tipo de variables. * var saludo = "Hola mundo"; var dinero = 10.5; |
| --- |

**Buenas prácticas al declarar variables**

**Nombres descriptivos:**Usa nombres significativos que describen claramente el propósito de la variable.

* ✅ int numeroEstudiantes = 25;
* ❌ int x = 25;

**Usa camelCase:**Sigue la convención de usar letras minúsculas para la primera palabra y mayúsculas para las siguientes.

* ✅ int cantidadLibros;
* ❌ int Cantidad\_Libros;

**Declara las variables lo más cerca de su uso:**Define variables en el contexto donde las necesitas, para mejorar la legibilidad del código.

* ✅

if (esMayor) {

int edad = 18;

System.out.println(edad);

}

* ❌

int edad;

if (esMayor) {

edad = 18;

}

System.out.println(edad);

**Inicializa las variables:**Siempre inicializa las variables, ya que usar variables sin inicializar puede causar errores.

* ✅ int numero = 0;
* ❌ int numero;

**Minimiza el uso de variables estáticas:**Evita abusar de las variables estáticas, ya que comparten su valor entre todas las instancias y pueden complicar la gestión del estado.

**Documenta las variables importantes:**Agrega comentarios para variables cuyo propósito no sea evidente de inmediato.

// Tiempo máximo en segundos que un usuario puede estar inactivo

final int TIEMPO\_MAX\_INACTIVIDAD = 300;

## ***Tipos de datos primitivos***

Los tipos de datos primitivos que Java utiliza como se observa en la imagen anterior, son los tipos de datos simples que al declararlos no es necesaria una invocación y estos carecen de métodos.



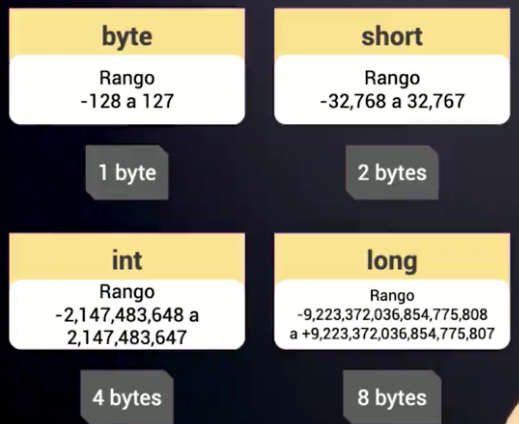
### **Enteros**

Estos tipos de datos se utilizan para guardar cifras y se dividen en dos categorías "enteros" y "decimales", estos tipos de datos tienen la característica de que tienen un límite de almacenamiento, si el valor que se le asigna sobrepasa el límite ocasionará un desbordamiento y el código fuente marcara errores, por eso es muy importante elegir el tipo de dato que se utiliza y analizar si el valor incrementará con el tiempo.

En Java, se pueden utilizar los siguientes tipos de datos numéricos para números enteros:

1. byte : ocupa 1 byte de memoria.
2. short: ocupa 2 bytes de memoria.
3. int : ocupa 4 bytes de memoria
4. long : ocupa 8 bytes de memoria.

| **⚠️ Nota:**   * Para diferenciar un int de un long es necesario poner una ‘L’ al final del número de tipo long. * long numLong = 1234L; |
| --- |

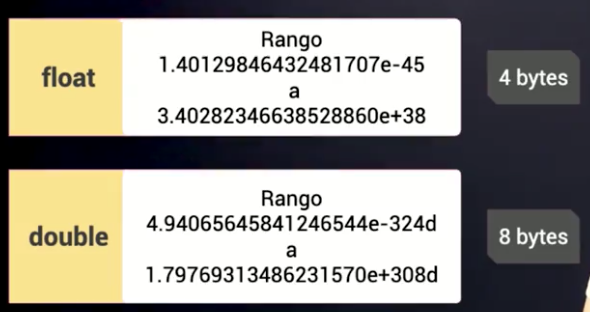


### **Decimales**

Para datos numéricos con punto decimal o flotante existen dos opciones: float y double.

1. float : ocupa 4 bytes de memoria y no tiene tanta precisión como double.
2. double: ocupa 8 bytes de memoria y tiene más precisión qué float.

| **⚠️ Nota:**   * Para hacer la distinción entre estos, es necesario poner una ‘F/f’ al final del número flotante o ‘D/d’. * float numf = 1234F; * double numd = 1234D; |
| --- |



### **Carácter**

Para datos de tipo texto o más bien caracteres el único tipo de dato primitivo que podemos utilizar es el char y solo podrá contener un carácter en su valor.

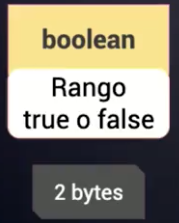
1. char : ocupa 2 bytes de memoria y es de rango unicode.

| **⚠️ Nota:**   * El valor de este dato siempre debe declararse con comillas simples y alternativamente podrías declararlo en su valor ASCII. * char letra = 'a'; * char letraAscii = 97; //Valor de a en ASCII |
| --- |

### 

### **Lógicos**

Para utilizar datos lógicos y poder representar decisiones se utiliza el tipo de dato boolean donde su valor puede ser: false y true.



Ejemplo de datos primitivos

public class TiposPrimitivos {

public static void main(String[] args) {

// Enteros

byte edad = 25;

int poblacion = 1000000;

long distancia = 9876543210L;

// Decimales

float precio = 19.99f;

double pi = 3.141592653589793;

// Caracteres y booleanos

char inicial = 'J';

boolean esActivo = true;

}

}

## ***Constantes***

En Java, una constante es una variable cuyo valor no puede cambiar después de ser inicializado. Para definir constantes se utiliza la palabra clave final.

| **⚠️ Nota:**   * Por convención al nombrar constantes siempre será en mayúsculas para identificar rápidamente el tipo. * *final* int HORASDIA = 24; |
| --- |

### 

### **Constantes primitivas**

final int EDAD\_MINIMA = 18;

final double PI = 3.14159;

### 

### **Constantes de objetos**

final String MENSAJE\_BIENVENIDA = "Bienvenido a Java!";

final List<String> DIAS\_SEMANA = Arrays.asList("Lunes", "Martes", "Miércoles");

### **Constantes estáticas**

Si una constante debe ser accesible globalmente, se combina final con static.

public class Configuracion {

public static final int *MAX\_USUARIOS* = 100;

public static final String *URL\_BASE* = "https://api.miapp.com/";

}

Ejemplo de declaración y uso de constantes

public class EjemploConstantes {

// Constantes globales

public static final int *MAX\_EDAD* = 120;

public static final String *SALUDO* = "¡Hola, Mundo!";

public static void main(String[] args) {

// Constantes locales

final double GRAVEDAD\_TIERRA = 9.81;

System.*out*.println("Máxima edad permitida: " + *MAX\_EDAD*);

System.*out*.println(*SALUDO*);

System.*out*.println("Gravedad en la Tierra: " + GRAVEDAD\_TIERRA);

// Error: no se puede modificar una constante

// MAX\_EDAD = 130; // <- Esto genera un error de compilación

}

}

**Buenas prácticas con constantes**

**Usa final siempre para valores constantes:**Esto garantiza que no puedan ser modificados después de la inicialización.

**Combina static con final para constantes globales:**Si una constante no depende de una instancia de la clase, hazla estática para evitar la creación de copias innecesarias.

public static final String *APP\_NAME* = "MiAplicacion";

**Nombres en mayúsculas con guiones bajos:**Sigue la convención de escribir los nombres de las constantes en **MAYÚSCULAS** y separar palabras con guiones bajos.

* ✅ final int MAX\_LONGITUD = 100;
* ❌ final int MaxLongitud = 100;

**Declara constantes como privadas si no se usan fuera de la clase:**Restringe el acceso si la constante no debe ser visible para otras clases.

private static final double *TASA\_INTERES* = 3.5;

**Agrupa constantes relacionadas en una clase o interfaz:**

Para mejorar la organización, agrupa constantes en una clase específica.

public class ConstantesErrores {

public static final String *ERROR\_404* = "Página no encontrada";

public static final String *ERROR\_500* = "Error interno del servidor";

}

**Evita redefinir valores mágicos:**Usa constantes en lugar de valores mágicos (números o cadenas "hardcodeados"). Esto facilita la legibilidad y mantenimiento.

* ❌ if (edad > 18)
* ✅ if (edad > EDAD\_MINIMA)

**Evita inicializaciones complejas:**Las constantes deben tener valores simples y no depender de cálculos complejos en tiempo de ejecución. Si es necesario, usa inicializadores estáticos.

public static final double *RAIZ\_2*;

static {

RAIZ\_2 = Math.sqrt(2);

}

## ***Técnicas de Naming***

### **Upper Camel Case**

También conocido como **Pascal Case**.

En esta convención, **la primera letra de cada palabra es mayúscula**, incluidas la primera palabra y las subsecuentes.

Se utiliza principalmente para:

* **Nombres de clases:**

public class MiClaseEjemplo { }

public class UsuarioRegistrado { }

* **Nombres de interfaces**

public interface MiInterfaz { }

public interface CalculadoraAvanzada { }

### **Lower Camel Case**

En esta convención, **la primera letra de la primera palabra es minúscula** y **la primera letra de cada palabra subsiguiente es mayúscula**.

Se utiliza principalmente para:

* **Nombres de variables**

String nombreUsuario;

int cantidadProductos;

double precioTotal;

* **Nombres de métodos**

public void calcularPromedio() { }

public int obtenerEdad() { }

## ***Casteo de variables***

El casting o casteo es un procedimiento para transformar una variable primitiva de un tipo a otro. También se utiliza para transformar un objeto de una clase a otra clase siempre y cuando haya una relación de herencia entre ambas.

En la programación hay situaciones en las que se necesita cambiar el tipo de dato esto dependiendo de nuestra necesidad, podemos realizar castings por:

***Estimación :*** Se utiliza cuando no nos importa la cantidad de decimales que tiene un número, sino que se busca representar la parte entera del número.

***Exactitud :*** Se hace uso de esta para realizar un alcance más preciso posible del número.

Dentro del casteo se distinguen dos clases:

***Casting implícito***: Aquí no se necesita escribir código. Ocurre cuando se lleva a cabo una conversión ancha, es decir, cuando se coloca un valor pequeño en un contenedor grande, estos casos no son susceptibles a pérdida de datos.

Orden de jerarquía de tipos numéricos:

byte → short → int → long → float → double

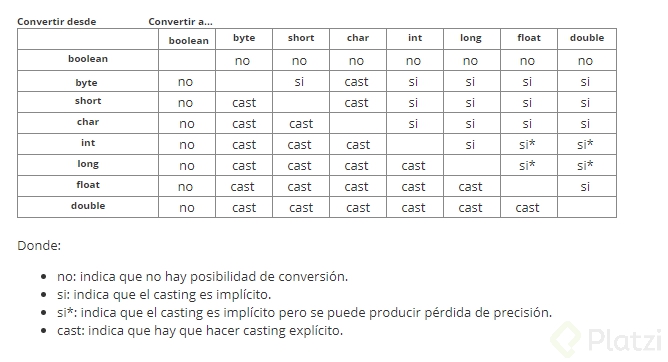
int num1 = 100;

long num2 = num1; // un int cabe en un long

***Casting explícito***: Aquí si es necesario escribir código. Ocurre cuando se realiza una conversión estrecha, es decir, cuando se coloca un valor grande en un contenedor pequeño, son susceptibles a pérdida de datos.

double d = 86.45;

int i = (int) d; //86 convertimos un decimal a entero



***Conversión entre tipos primitivo y objeto (Boxing/Unboxing)***

**Boxing (Autoboxing)**

Conversión automática de un tipo primitivo a su clase envoltura (wrapper class).

int numero = 10;

Integer numeroEnvoltorio = numero; // Autoboxing

**Unboxing**

Conversión automática de una clase envoltura a su tipo primitivo.

Integer numeroEnvoltorio = 10;

int numero = numeroEnvoltorio; // Unboxing

**Conversión entre tipos no compatibles**

Para tipos no directamente compatibles, como de String a int o viceversa, necesitas usar métodos auxiliares.

**De String a Número:**

String texto = "123";

int numero = Integer.*parseInt*(texto); // Método estático de Integer

System.out.println(numero + 10); // 133

**De Número a String:**

int numero = 123;

String texto = String.*valueOf*(numero);

System.out.println("Número como texto: " + texto); // "Número como texto: 123"

**Buenas prácticas para el casteo de variables**

**Evita el casteo innecesario:**Solo convierte cuando sea realmente necesario.

**Comprueba rangos antes del casteo:**Asegúrate de que el valor que estás convirtiendo cabe en el tipo de destino.

long numeroLargo = 300L;

if (numeroLargo <= Byte.MAX\_VALUE && numeroLargo >= Byte.MIN\_VALUE) {

byte numeroPequeno = (byte) numeroLargo;

}

**Ten cuidado con la pérdida de precisión:**

Al convertir tipos de punto flotante a enteros, los decimales se truncan.

double pi = 3.14159;

int enteroPi = (int) pi; // Resultado: 3

**Manejo de excepciones en conversiones de String:**

Usa try-catch para manejar errores en conversiones no válidas.

try {

String texto = "abc";

int numero = Integer.parseInt(texto); // Lanza NumberFormatException

} catch (NumberFormatException e) {

System.out.println("Error: No se puede convertir el texto a número.");

}

**Ejemplo práctico:**

public class CasteoEjemplo {

public static void main(String[] args) {

// Casteo implícito

int entero = 100;

double decimal = entero; // Automático

System.*out*.println("Casteo implícito: " + decimal);

// Casteo explícito

double numeroDecimal = 99.99;

int numeroEntero = (int) numeroDecimal; // Manual

System.*out*.println("Casteo explícito: " + numeroEntero);

// Conversión entre tipos no compatibles

String texto = "456";

int numero = Integer.*parseInt*(texto); // String a int

System.*out*.println("Conversión de String a int: " + numero);

// Autoboxing y Unboxing

Integer envoltura = numero; // Autoboxing

int primitivo = envoltura; // Unboxing

System.*out*.println("Autoboxing y Unboxing: " + primitivo);

}

}

## ***Operadores***

### **Aritméticos**

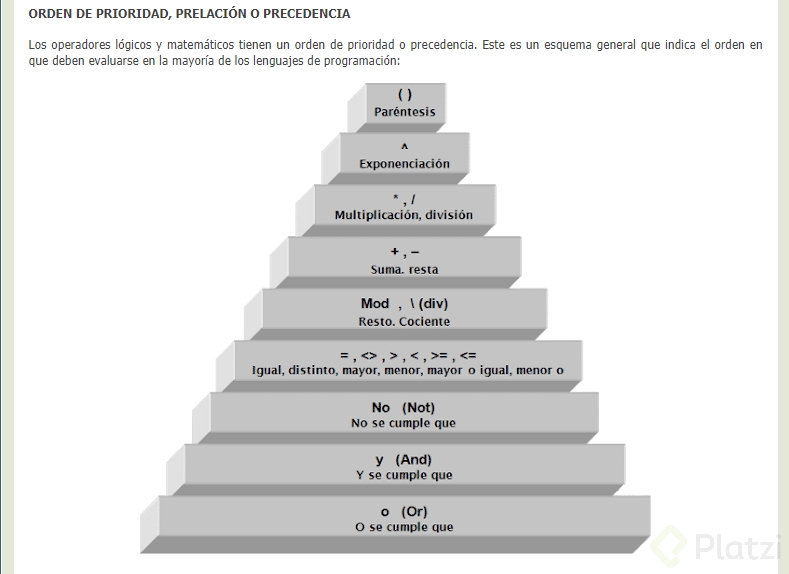
Se utilizan para realizar operaciones matemáticas básicas:

| **Operador** | **Función** | **Ejemplo** |
| --- | --- | --- |
| + | Suma | a + b |
| - | Resta | a - b |
| \* | Multiplicación | a \* b |
| / | División | a / b |
| % | Módulo (residuo) | a % b |

### **Lógicos**

| **Operador** | **Función** | **Ejemplo** | **Resultado** |
| --- | --- | --- | --- |
| && | AND lógico | a && b | true si ambos son true |
| || | OR lógico | a || b | true si uno u otro se cumple |
| ! | NOT lógico | !a | Invierte el valor booleano de a |

### 



### 

### **Asignación**

Los operadores de asignación son muy útiles al momento de querer dar un valor a una variable o reasignar un valor, como se puede apreciar en la siguiente tabla hay abreviaciones para poder agilizar el tiempo de desarrollo pero es importante comprender lo que cada operador realiza internamente.

### 

| **Operador** | **Función** | **Ejemplo** | **Equivalente a** |
| --- | --- | --- | --- |
| = | Asignación simple | a = b | - |
| += | Asignación con suma | a += b | a = a + b |
| -= | Asignación con resta | a -= b | a = a - b |
| \*= | Asignación con multiplicación | a \*= b | a = a \* b |
| /= | Asignación con división | a /= b | a = a / b |
| %= | Asignación con módulo | a %= b | a = a % b |

### 

### **Unarios**

Estos son sumamente útiles para definir valores que van incrementando su valor o por lo contrario reduciendo su valor.

| **Operador** | **Función** | **Ejemplo** | **Resultado** |
| --- | --- | --- | --- |
| + | Positivo | +a | Retorna el valor de a tal cual |
| - | Negativo | -a | Invierte el signo de a |
| ++ | Incremento | a++ o ++a | Incrementa a en 1 |
| -- | Decremento | a-- o --a | Decrementa a en 1 |
| ! | Negación lógica | !true | false |

| **⚠️ Nota:**   * ++a y --a (preincremento o predecremento) se evalúan antes de usar el valor. * a++ y a-- (postincremento o postdecremento) se evalúan después de usar el valor. |
| --- |

### **Relacionales**

| **Operador** | **Función** | **Ejemplo** | **Resultado** |
| --- | --- | --- | --- |
| == | Igual a | a == b | true si a es igual a b |
| != | No igual a | a != b | true si a no es igual a b |
| > | Mayor que | a > b | true si a es mayor que b |
| < | Menor que | a < b | true si a es menor que b |
| >= | Mayor o igual que | a >= b | true si a es mayor o igual a b |
| <= | Menor o igual que | a <= b | true si a es menor o igual a b |

### **Ternarios**

| **Operador** | **Función** | **Ejemplo** | **Resultado** |
| --- | --- | --- | --- |
| ? : | Condicional ternario | a = (b > c) ? b : c; | Asigna b si b > c, si no c |

### **Especiales**

| **Operador** | **Función** | **Ejemplo** | **Resultado** |
| --- | --- | --- | --- |
| . | Acceso a miembro | obj.propiedad | Accede a propiedad de obj |
| [] | Indexación de arreglo | arreglo[0] | Accede al primer elemento |
| () | Llamada a método | obj.metodo() | Llama a metodo de obj |
| new | Creación de objeto | new Clase() | Crea una nueva instancia |
| (tipo) | Casting | (int) 3.14 | Convierte 3.14 a entero |

## 

# 3 - Control de flujo

## ***Condicionales***

Los condicionales son la forma en que las computadoras toman decisiones, evaluarán si la condición para ejecutar una parte del código se cumple. Si el resultado de la operación es verdadero ejecutarán esta parte del código, en caso de que no, seguirán con las siguientes instrucciones. Son fundamentales para la programación estructurada.

### **If / else**

En Java, las estructuras condicionales if y else permiten tomar decisiones en el flujo del programa basadas en condiciones booleanas. Son fundamentales para la programación estructurada.

**if**

Evalúa una condición booleana y ejecuta el bloque de código si la condición es true.

if (condición) {

// Código a ejecutar si la condición es verdadera

}

**if-else**

Proporciona una alternativa para ejecutar código si la condición es false.

if (condicion) {

// Código a ejecutar si la condición es verdadera

} else {

// Código a ejecutar si la condición es falsa

}

**else-if**

Permite evaluar múltiples condiciones de manera secuencial.

if (condicion1) {

// Código para condición1

} else if (condicion2) {

// Código para condición2

} else {

// Código si ninguna condición es verdadera

}

}

Ejemplo práctico:

public class Condicionales {

public static void main(String[] args) {

int edad = 20;

if (edad < 18) {

System.*out*.println("Eres menor de edad.");

} else if (edad >= 18 && edad < 65) {

System.*out*.println("Eres adulto.");

} else {

System.*out*.println("Eres mayor de edad.");

}

}

}

### 

**Buenas prácticas usando If-else**

**Mantén las condiciones simples y legibles:**

Divide condiciones complejas en variables auxiliares descriptivas.

boolean esMayorDeEdad = edad >= 18;

if (esMayorDeEdad) {

System.out.println("Eres adulto.");

}

**Evita anidamientos excesivos:**

Anidar múltiples if dentro de otros dificulta la lectura. Usa retornos anticipados o estructura el código para minimizar anidaciones.

// Mal

if (a > 0) {

if (b > 0) {

System.out.println("Ambos son positivos.");

}

}

// Mejor

if (a > 0 && b > 0) {

System.out.println("Ambos son positivos.");

}

**Usa bloques else solo si es necesario:**

Si el bloque else no agrega valor, omítelo para evitar código innecesario.

**Evita repetir lógica similar:**

Extrae lógica repetida en funciones reutilizables.

// Mal

if (edad >= 18) {

System.out.println("Acceso permitido.");

} else {

System.out.println("Acceso denegado.");

}

// Mejor

String mensaje = (edad >= 18) ? "Acceso permitido." : "Acceso denegado.";

System.out.println(mensaje);

**Usa paréntesis {} incluso para bloques de una línea:**

Aunque son opcionales, los paréntesis mejoran la legibilidad y previenen errores al agregar nuevas líneas de código.

**Evita comparar directamente con valores booleanos:**

// Mal

if (esActivo == true) { ... }

// Mejor

if (esActivo) { ... }

**Prefiere switch para múltiples condiciones similares:**

Si tienes varias condiciones comparando el mismo valor, considera usar switch.

**Ejemplo usando buenas prácticas**

public class EjemploPractico {

public static void main(String[] args) {

int edad = 20;

boolean tieneLicencia = true;

if (edad >= 18) {

if (tieneLicencia) {

System.*out*.println("Puedes conducir.");

} else {

System.*out*.println("Necesitas una licencia para conducir.");

}

} else {

System.*out*.println("Eres menor de edad y no puedes conducir.");

}

}

}

### **Ternario**

Los operadores ternarios son otra forma de evaluar condiciones, es una alternativa compacta a if-else para condiciones simples.

Sintaxis:

variable = (condicion) ? valorSiTrue : valorSiFalse;

Ejemplo:

int edad = 20;

String mensaje = (edad >= 18) ? "Eres adulto" : "Eres menor de edad";

System.out.println(mensaje);

**Buenas prácticas usando ternarios**

**Mantén las expresiones simples**

El operador ternario es ideal para casos sencillos donde las condiciones y valores son cortos y directos. Si la expresión se vuelve compleja, es mejor usar un if-else tradicional.

Ejemplo simple y legible

String resultado = (edad >= 18) ? "Adulto" : "Menor de edad";

Evita usarlo en condiciones complejas

// Mal: difícil de leer

String resultado = (edad >= 18 && puntos >= 50) ? "Aprobado" : "Reprobado";

// Mejor: usa variables descriptivas

boolean esAdulto = edad >= 18;

boolean tienePuntosSuficientes = puntos >= 50;

String resultado = (esAdulto && tienePuntosSuficientes) ? "Aprobado" : "Reprobado";

**Evita anidaciones de operadores ternarios**

// Mal

String resultado = (nota > 90) ? "Excelente" : (nota > 70) ? "Bueno" : "Regular";

// Mejor

if (nota > 90) {

resultado = "Excelente";

} else if (nota > 70) {

resultado = "Bueno";

} else {

resultado = "Regular";

}

**Usa el operador ternario para asignaciones**

El uso más común y adecuado del operador ternario es para asignar un valor a una variable en función de una condición.

// Ejemplo correcto

int max = (a > b) ? a : b;

// Evitar: esto no es una buena práctica

(a > b) ? System.out.println("a es mayor") : System.out.println("b es mayor");

// Mejor: usa un if-else en este caso

if (a > b) {

System.out.println("a es mayor");

} else {

System.out.println("b es mayor");

}

**Úsalo con tipos compatibles**

El operador ternario debe devolver valores del mismo tipo o compatibles. Si los tipos son diferentes, pueden ocurrir errores de compilación o resultados inesperados.

// Ejemplo incorrecto

Object resultado = (edad >= 18) ? "Adulto" : 1; // Error: tipos incompatibles

// Ejemplo correcto

String resultado = (edad >= 18) ? "Adulto" : "Menor de edad";

### **Switch**

Es una estructura de control que permite comparar un valor con múltiples casos predefinidos, proporcionando una alternativa más legible y eficiente que múltiples sentencias if-else if en situaciones específicas.

Sintaxis basica:

switch (expresion) {

case valor1:

// Código a ejecutar si expresion == valor1

break; // Finaliza el caso

case valor2:

// Código a ejecutar si expresion == valor2

break;

default:

// Código a ejecutar si ningún caso coincide

break;

}

Características clave:

* **Expresión:** Puede ser de tipo int, short, byte, char, String (desde Java 7), o enum.
* **case:** Debe contener valores constantes o literales.
* **break:** Evita que se ejecuten los casos siguientes. Si se omite, el control "cae" al siguiente caso (fall-through).
* **default:** Es opcional, pero se ejecuta si no coincide en ningún caso.

Switch hasta Java 11:

public class EjemploSwitch {

public static void main(String[] args) {

int dia = 3;

switch (dia) {

case 1:

System.*out*.println("Lunes");

break;

case 2:

System.*out*.println("Martes");

break;

case 3:

System.*out*.println("Miércoles");

break;

default:

System.*out*.println("Día no válido");

}

}

} // salida “Miércoles”

Switch desde Java 12 y si necesitaras lógica en un case solo añade {}:

switch (edad) {

case 1 -> {System.out.println("¡Tienes 1 año!");}

case 20 -> System.out.println("Tienes 20 años!");

default -> System.out.println("Tu edad no es 1 ni 20");

}

| **⚠️ Nota:**  La sentencia SWITCH solo admite los tipos de datos:   * byte y Byte * short y Short * int y Integer * char y Character * String * enum   No admite:   * boolean y Boolean * long y Long * float y Float * double y Double |
| --- |

A partir de java 14+ podemos hacer switch con agrupación:

public static void main(String[] args) {

int diaSemana = 1;

switch (diaSemana) {

case 1, 2, 3, 4, 5:

System.*out*.println("Dia laboral");

break;

case 6 ,7:

System.*out*.println("No laboral");

break;

default:

System.*out*.println("Dia no válido");

}

}

}

**Buenas prácticas usando switch**

**Evita la lógica compleja dentro del switch:**

Mantén las operaciones dentro de los casos simples y delega lógica compleja a métodos.

switch (opcion) {

case 1:

realizarOperacion1();

break;

case 2:

realizarOperacion2();

break;

}

**Prefiere enum en lugar de constantes:**

Usar enum hace que el código sea más robusto y legible.

enum Dia {

*LUNES*, *MARTES*, *MIERCOLES*

}

Dia dia = Dia.*LUNES*;

switch (dia) {

case LUNES:

System.out.println("Inicio de semana.");

break;

case MARTES:

System.out.println("Segundo día.");

break;

}

## 

## ***Bucles***

Los bucles (ciclos) nos ayudan a ejecutar una parte de nuestro código una cantidad de veces hasta que se cumpla alguna condición y podamos continuar con la ejecución de nuestro código.

Los ciclos evaluarán si la condición se cumple y cuando deje de hacerlo no ejecutarán más el código del ciclo. Las instrucciones son las encargadas de que esta condición cambie de verdadero a falso. De otra forma, si las instrucciones nunca cambian la condición, el ciclo no se detendrá nunca, lo que conocemos como un ciclo infinito.

### **While**

Evalúa la condición antes de ejecutar el bloque de código.

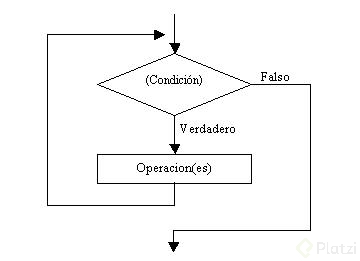
Si la condición es falsa desde el principio, el código no se ejecuta ni una sola vez.

while (condición) {

// Código a ejecutar mientras la condición sea verdadera

}

El ciclo while ejecuta primero la evaluación para posteriormente ejecutar la operación.



Ejemplo:

int contador = 0;

while (contador < 5) {

System.out.println("Contador: " + contador);

contador++;

}

Salida:

Contador: 0

Contador: 1

Contador: 2

Contador: 3

Contador: 4

### **do-while**

Evalúa la condición después de ejecutar el bloque de código.

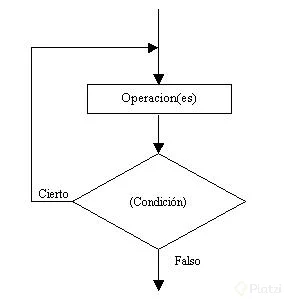
El código se ejecuta al menos una vez, incluso si la condición es falsa desde el principio.

do {

// Código a ejecutar al menos una vez

} while (condición);

El ciclo do while ejecuta primero la operación para posteriormente realizar la validación.



Ejemplo:

int contador = 0;

do {

System.out.println("Contador: " + contador);

contador++;

} while (contador < 5);

Salida:

Contador: 0

Contador: 1

Contador: 2

Contador: 3

Contador: 4

**Buenas prácticas usando while / do-while**

**Evitar ciclos infinitos involuntarios:**

Asegúrate de que la condición se actualice correctamente dentro del ciclo para evitar bucles que nunca terminan.

**Prefiere while para condiciones preconocidas:**Usa while si sabes que el bloque puede no necesitar ejecutarse en absoluto.

**Usa do-while si el bloque debe ejecutarse al menos una vez:**Esto es útil, por ejemplo, para validaciones o menús interactivos.

int opcion;

do {

System.out.println("Menú:");

System.out.println("1. Opción 1");

System.out.println("2. Opción 2");

System.out.println("0. Salir");

opcion = scanner.nextInt();

} while (opcion != 0);

### 

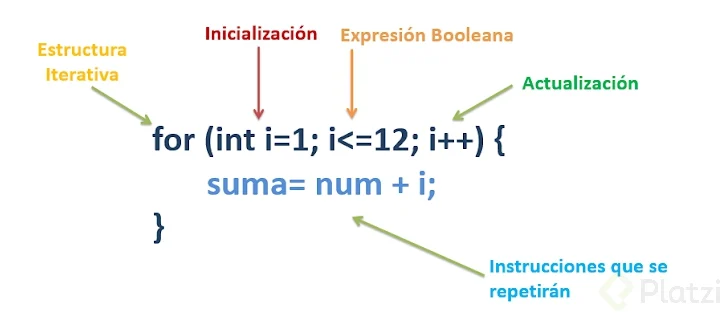
### **For**

El Ciclo For también nos ayuda a ejecutar una parte de nuestro código las veces que sean necesarias para que se cumpla una condición. De hecho, el ciclo FOR nos da muchas ayudas para lograr este resultado de la forma más fácil posible:

for (inicialización; condición; actualización) {

// Código a ejecutar en cada iteración

}



1. **Inicialización:** Define e inicializa las variables de control.
2. **Condición:** Evalúa si se debe ejecutar el bloque de código. Si la condición es false, el ciclo termina.
3. **Actualización:** Se ejecuta al final de cada iteración para modificar la variable de control.

Ejemplo básico:

for (int i = 0; i < 5; i++) {

System.out.println("Iteración: " + i);

}

Salida:

Iteración: 0

Iteración: 1

Iteración: 2

Iteración: 3

Iteración: 4

Variaciones del ciclo for:

Contador decreciente

for (int i = 5; i > 0; i--) {

System.out.println("Cuenta regresiva: " + i);

}

Incrementos personalizados

for (int i = 0; i <= 10; i += 2) {

System.out.println("Número par: " + i);

}

Anidamientos

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

System.out.println("i: " + i + ", j: " + j);

}

}

**Buenas prácticas usando For**

**Evita modificar la variable de control dentro del bloque:** Cambiar manualmente la variable de control dentro del ciclo puede ser confuso y propenso a errores.

for (int i = 0; i < 5; i++) {

// No cambies i aquí manualmente

}

**Usa nombres de variables descriptivos:** En lugar de usar i, j, o k en ciclos complejos, usa nombres más representativos para mejorar la legibilidad.

for (int fila = 0; fila < 5; fila++) {

for (int columna = 0; columna < 5; columna++) {

System.out.println("Fila: " + fila + ", Columna: " + columna);

}

}

### 

**Prefiere for-each cuando no necesitas índices:** Usa for-each para colecciones y arreglos, especialmente si solo necesitas acceder a los elementos.

List<String> nombres = Arrays.asList("Ana", "Luis", "Carlos");

for (String nombre : nombres) {

System.out.println("Nombre: " + nombre);

}

**Evita ciclos innecesarios o anidados excesivamente:** Los ciclos anidados pueden ser costosos en términos de rendimiento. Si son inevitables, considera optimizar la lógica o dividir el problema.

**Cuidado con los límites:** Asegúrate de que la condición del ciclo sea correcta para evitar errores como IndexOutOfBoundsException en arreglos o listas.

**Reduce el alcance de las variables:** Declara las variables del ciclo dentro del for si no se usan fuera del mismo.

**Evitar ciclos infinitos involuntarios:** Asegúrate de que la condición del ciclo eventualmente se cumpla.

for (int i = 0; i < 5; ) { // Falta el incremento

System.out.println("Esto nunca terminará");

}

### 

### **Sentencias de control**

#### Break

En Java esta sentencia la verás en dos situaciones específicamente:

1. En un Switch: en esta situación break hace que el flujo del switch no continúe ejecutándose a la siguiente comparación, esto con el objetivo de que solo se cumpla una sola condición:

switch (colorModeSelected){

case "Light":

System.out.println("Seleccionaste Light Mode");

break;

case "Night": //Ambar

System.out.println("Seleccionaste Night Mode");

break;

case "Blue Dark":

System.out.println("Seleccionaste Blue Dark Mode");

break;

}

1. Para salir de un bucle: Como acabamos de ver un break es capaz de detener el flujo en el código, en este caso detendremos el ciclo como tal terminándolo y haciendo que saltemos a la siguiente instrucción después del ciclo.

for (int i = 1; i <= 10; i++) {

printSOS();

break;

}

#### 

#### Continue

Continue en cierto modo también nos va a servir para detener un ciclo pero en lugar de terminarlo como en el caso de break, este omitie la ejecución de las líneas restantes en la iteración actual y salta a la siguiente iteración del ciclo.

for (int i = 0; i < 10; i++) {

if (i % 2 == 0) {

continue; // Omite los números pares

}

System.out.println("i: " + i);

}

Salida:

i: 1 i: 3 i: 5 i: 7 i: 9

#### Return

Finaliza inmediatamente la ejecución del método actual y devuelve opcionalmente un valor al llamador del método.

En ciclos, return también detiene el ciclo si está dentro de un método.

return; // Si el método no devuelve nada (void)

return valor; // Si el método devuelve un valor

Ejemplo metodo void:

public void mostrarMensaje(int numero) {

if (numero < 0) {

return; // Termina el método si el número es negativo

}

System.*out*.println("Número: " + numero);

}

Ejemplo método con Retorno:

public int encontrarPrimeroPar(int[] numeros) {

for (int numero : numeros) {

if (numero % 2 == 0) {

return numero; // Devuelve el primer número par encontrado

}

}

return -1; // Devuelve -1 si no hay números pares

}

Ejemplo práctico aplicando los tres controladores de flujo:

public class EjemploControlFlujo {

public static void main(String[] args) {

int[] numeros = {1, 3, 5, 7, 2, 9};

for (int numero : numeros) {

if (numero == 5) {

continue; // Omite el número 5

}

if (numero == 7) {

break; // Termina el ciclo cuando encuentra 7

}

if (numero % 2 == 0) {

System.*out*.println("Primer número par encontrado: " + numero);

return; // Sale del método principal

}

System.*out*.println("Número: " + numero);

}

System.*out*.println("Esto no se imprimirá si se encuentra un número par");

}

}

Salida:

Número: 1

Número: 3

Primer número par encontrado: 2

#### Tags

Una etiqueta es un identificador seguido de dos puntos : que precede a un bloque de código (generalmente un ciclo).

etiqueta: // Identificador seguido de dos puntos

for (inicialización; condición; actualización) {

// Bloque de código

}

break puede usarse con etiquetas para salir de un ciclo específico, incluso si está anidado dentro de otros.

etiqueta: // Etiqueta del ciclo externo

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (i == 1 && j == 1) {

break etiqueta; // Sale del ciclo etiquetado (externo)

}

System.out.println("i: " + i + ", j: " + j);

}

}

continue puede usarse con etiquetas para omitir la iteración de un ciclo externo etiquetado, en lugar de solo el ciclo interno.

etiqueta: // Etiqueta del ciclo externo

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (i == 1 && j == 1) {

continue etiqueta; // Salta a la siguiente iteración del ciclo etiquetado (externo)

}

System.out.println("i: " + i + ", j: " + j);

}

}

**Buenas prácticas usando Break, Continue y Return**

**Usa break con moderación en ciclos grandes:**

* Si el ciclo tiene múltiples condiciones complejas, usar break repetidamente puede hacer el código difícil de leer.
* Prefiere organizar la lógica para evitar necesitarlo, pero úsalo cuando sea claro y necesario.

**Usa continue para flujos simples:**

* Úsalo para evitar ejecutar parte del cuerpo del ciclo si una condición no se cumple.
* No abuses de continue en ciclos anidados, ya que puede confundir el flujo de control.

**Evita usar return dentro de ciclos si es confuso:**

* Si usas return en un ciclo, asegúrate de que sea fácil de entender.
* Por ejemplo, si hay múltiples return en un método, documenta las condiciones.

**Siempre incluye un caso predeterminado con return:**

* Si tu método tiene varias rutas de salida, asegúrate de que siempre haya un return claro al final para manejar casos no contemplados.

public int dividir(int a, int b) {

if (b == 0) {

return -1; // Devuelve un valor por defecto si b es 0

}

return a / b;

}

**Comentarios en casos complejos:**

* Si usas break, continue o return en un contexto complicado, añade comentarios para explicar su propósito.

**Evita sobreusar controladores de flujo:**

* Si un método tiene demasiados break, continue, o return, considera refactorizar la lógica.

# 4 - Java SE con POO

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el código en torno a objetos, los cuales representan entidades del mundo real con atributos (estado) y comportamientos (métodos). Java es un lenguaje diseñado desde cero para ser orientado a objetos, lo que significa que prácticamente todo en Java se basa en este paradigma.

**Principios Fundamentales de la POO**

1. **Abstracción:**Oculta los detalles internos de los objetos y muestra solo lo esencial. Por ejemplo, al usar un automóvil, no necesitas saber cómo funciona el motor, solo cómo conducirlo.
2. **Encapsulación:**Agrupa datos y métodos que trabajan sobre esos datos dentro de una clase, restringiendo el acceso a ciertos componentes mediante modificadores de acceso (private, public, etc.).
3. **Herencia:**Permite que una clase (clase hija) reutilice las propiedades y métodos de otra clase (clase padre). Facilita la reutilización del código y la creación de jerarquías.
4. **Polimorfismo:**Permite que un mismo método se comporte de diferentes maneras dependiendo del objeto que lo invoque. Se puede lograr a través de la **sobrecarga** y la **sobrescritura** de métodos.

**Conceptos Básicos de la POO en Java**

1. **Clases:**Son plantillas que definen las propiedades (atributos) y métodos (comportamientos) de un tipo de objeto.

public class Persona {

String nombre;

int edad;

public void saludar() {

System.*out*.println("Hola, mi nombre es " + nombre);

}

}

1. **Objetos:** Son instancias de una clase que contienen valores específicos para los atributos.

public class Main {

public static void main(String[] args) {

Persona persona1 = new Persona();

persona1.nombre = "Juan";

persona1.edad = 30;

persona1.saludar(); // Salida: Hola, mi nombre es Juan

}

}

1. **Atributos:** Variables que representan el estado de un objeto.

String nombre; // Atributo de una clase

1. **Métodos:** Definen el comportamiento de los objetos.

public void saludar() {

System.*out*.println("Hola, soy un método.");

}

1. **Constructores:** Métodos especiales que se usan para inicializar objetos.

public Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

1. **Modificadores de acceso:**Controlan quién puede acceder a los miembros de una clase.

* public: Accesible desde cualquier lugar.
* private: Accesible sólo dentro de la clase.
* protected: Accesible dentro del paquete y subclases.
* Sin modificador (*default*): Accesible sólo dentro del paquete.

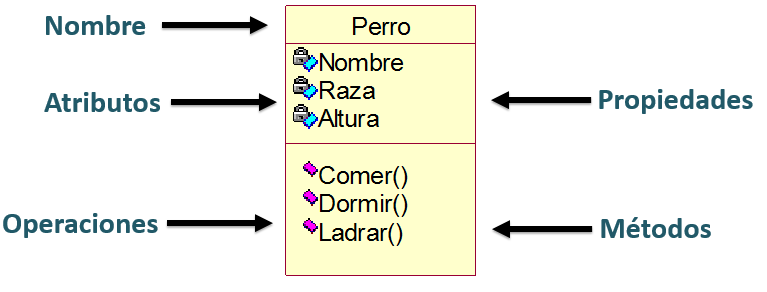
**Ventajas de la POO en Java**

1. **Reutilización del código:**La herencia y las clases reutilizables reducen la duplicación de código.
2. **Mantenimiento sencillo:**El código estructurado en objetos es más fácil de entender, mantener y extender.
3. **Flexibilidad:**El polimorfismo y la abstracción facilitan adaptarse a nuevos requisitos.
4. **Mayor modularidad:**Permite dividir el programa en partes independientes (objetos) que interactúan entre sí.
5. **Escalabilidad:**Ideal para desarrollar aplicaciones grandes y complejas.

## ***Clases***

En Java, una **clase** es la estructura fundamental de la programación orientada a objetos. Define un modelo o plantilla que describe el estado (atributos) y el comportamiento (métodos) que los objetos de ese tipo tendrán.

**Estructura de una Clase**



La estructura básica de una clase en Java incluye que más adelante veremos profundidad:

1. **Modificadores de acceso:**Controlan la visibilidad de la clase (public, default).
2. **Declaración de la clase:**Incluye la palabra clave class seguida del nombre de la clase.
3. **Atributos (propiedades):**Variables que almacenan el estado del objeto.
4. **Constructores:**Métodos especiales que inicializan los objetos de la clase.
5. **Métodos:**Define el comportamiento de los objetos.

Ejemplo básico de una clase:

public class Persona {

// Atributos

String nombre;

int edad;

// Constructor

public Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

// Métodos

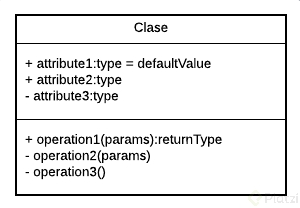
public void saludar() {

System.*out*.println("Hola, mi nombre es " + nombre + " y tengo " + edad + " años.");

}

}

Una clase se puede representar en UML (*Unified Modeling Language*) de la siguiente forma:



En la parte superior se colocan los **atributos** o **propiedades**, y debajo las **operaciones** o **métodos** de la clase.

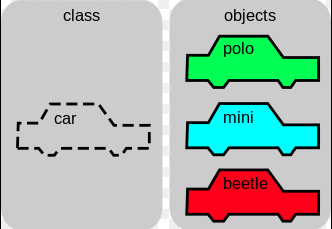
Notarás que el primer carácter con el que empiezan es un símbolo. Esto denotará la visibilidad del atributo o método, esto es un término que tiene que ver con Encapsulamiento y veremos más adelante a detalle.

### **Objetos**

Para poder hacer uso de una clase primero se debe crear un objeto a este proceso se le denomina (instanciación) no hay límite para el número de instancias pero se sugiere que sea instancie solo las necesarias para no impactar en el rendimiento del programa puesto que cada instanciación o creación de objeto tiene asignado un peso en memoria. Para instanciar se hace con la palabra clave new.

Ejemplo:

Coche miCoche = new Coche("Toyota", "Corolla", 2022);



Ejemplo:

// Definimos una clase "Coche"

class Coche {

// Atributos (Estado del objeto)

String marca;

String modelo;

int anio;

// Constructor

Coche(String marca, String modelo, int anio) {

this.marca = marca;

this.modelo = modelo;

this.anio = anio;

}

// Método (Comportamiento del objeto)

void mostrarInfo() {

System.*out*.println("Marca: " + marca + ", Modelo: " + modelo + ", Año: " + anio);

}

}

// Uso de la clase (Crear objetos)

public class Main {

public static void main(String[] args) {

Coche miCoche = new Coche("Toyota", "Corolla", 2022);

miCoche.mostrarInfo();

}

}

### **Gestión de memoria**

Cuando creamos objetos mediante la instanciación o creamos variables de tipo objeto que veremos más adelante, se involucran dos áreas principales de la memoria.

* Stack (Pila)
* Heap (Montículo)

Podría decirse que JVM usa memoria de tu computadora (RAM), pero la gestiona internamente en áreas como Stack y Heap.

#### *Memoria Stack*

La memoria Stack se usa para almacenar las variables locales (cuyo ámbito de acción está limitada sólo a la función donde se declaró) y también las llamadas de funciones en Java. Las variables almacenadas en esta memoria por lo general tienen un periodo de vida corto, viven hasta que termina la función o método en el que se están ejecutando.

La memoria stack en Java se caracteriza por:

* Almacenar variables y sus valores de tipos de datos primitivos, como booleanos, números, en general **variables locales y referencias a objetos**.
* Gestionar las invocaciones de métodos y las variables locales según el método **LIFO (último en entrar, primero en salir)** .
* Ser más rápida que otros espacios de memoria.
* Se limpia automáticamente cuando un método termina.

#### *Memoria Heap*

Por otro lado, la memoria Heap es utilizada para almacenar los objetos (incluyendo sus atributos), los objetos almacenados en este espacio de memoria normalmente tienen un tiempo de duración más prolongado que los almacenados en Stack.

* Es donde se almacenan **objetos creados con new**.
* Los objetos permanecen en Heap hasta que el **Garbage Collector** los elimina.
* Es más grande que el Stack pero más lento en acceso.

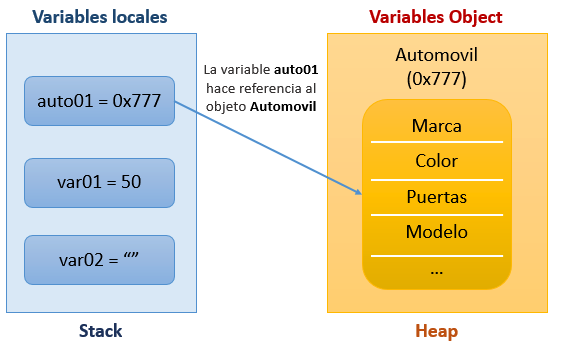
Hasta este punto, ya debes haber recordado que las variables de un programa no almacenan el objeto, más bien guardan la referencia del objeto.

Veamos el siguiente gráfico:

Automovil auto01 = new Automovil();

int var01 = 50;

String var02 = "";



Podemos observar que la referencia del objeto que es representada por un hexadecimal por ejemplo el valor **‘0x777’**, esta contiene la referencia de memoria donde está almacenado este objeto **Automóvil**, por lo tanto la variable local **auto01** almacena únicamente esta referencia de memoria donde está almacenado, donde fue creado este objeto. Aprovechando este mismo gráfico, podemos ver que variables de tipo primitivo como **var01** y **var02**, almacenan directamente el valor donde está creada esta variable.

Si la variable local **auto01** no tuviera referencia a memoria (por ejemplo igual a Null) será eliminado por el **Garbage Collector** (recolector de basura), cuya función es eliminar los objetos que no estén siendo apuntados por ninguna variable, es decir que este objeto que se ha creado de tipo **Automovil** como está siendo apuntado o referenciado por la variable **auto01**, no será eliminado. Pero si la variable auto01 se libera o es igualado a Null **(quitamos la referencia de memoria al objeto Automovil)**, nuestra variable será candidata a ser borrada por el **Garbage Collector.** En este caso la memoria **Heap** que es donde se almacenan los objetos en Java, de esta manera el **Garbage Collector** (recolector de basura) puede buscar aquellos objetos en la memoria Heap que ya no estén referenciados por ninguna otra variable y finalmente liberar el espacio en memoria que ocupaba dicho objeto.

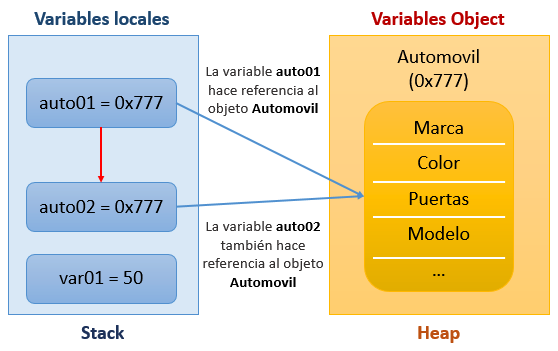
**¿Cómo se asigna la referencia en memoria?**

Ahora veamos otro ejemplo, en este código creamos una variable **auto01** la cual está apuntando a una referencia de un objeto que se está creando de tipo **Automovil**, el cual tiene el valor **0x777** de memoria. Este mismo valor lo podemos asignar a otra variable también del tipo Automovil, a esta nueva variable la llamaremos **auto02**:

Automovil auto01 = new Automovil();

Automovil auto02 = auto01;

int var01 = 50;



De esta forma, las dos variables estarían apuntando al mismo objeto que se ha creado en este momento.

En resumen, estamos creando un objeto de tipo **Automovil**, donde dos variables, **auto01** y **auto02**, almacenan referencia a ese objeto **Automovil**.

La variable creada **auto02** también se le asigna la misma referencia y por lo tanto existen dos variables **auto01** y **auto02** que apuntan al mismo objeto. Es importante que tengas en cuenta que cualquier cambio que apliquemos a una variable, se aplicará o verá reflejado automáticamente en la otra variable, debido a que lo que se modifica es el objeto y no la referencia que almacena la variable en sí misma.

Por lo tanto ahora son dos variables las que están apuntando al mismo objeto (**Automovil**), las dos variables podrán acceder a lo que el objeto les permita, es decir según como se haya definido la clase.

Por ejemplo sus atributos o métodos que haya definido es importante observar que no se está creando un segundo objeto ya que solamente hay una llamada de la palabra New y por lo tanto sólo hay un objeto creado lo que almacena la variable P2 es sólo el valor de la referencia en memoria del objeto persona creado cada vez que utilicemos la palabra New.

Sobre el **Garbage Collector**, este se ejecuta automáticamente (trabaja solo), y no hay necesidad de administrarlo o programarlo (salvo una excepción que usted quiera manejar el borrado de memoria), el **Garbage Collector** va observando el **Heap** (ver gráfico) para identificar qué objetos le hacen referencia y cuáles no, eliminando los no utilizados.

**¿Cómo se recibe una nueva referencia de memoria?**

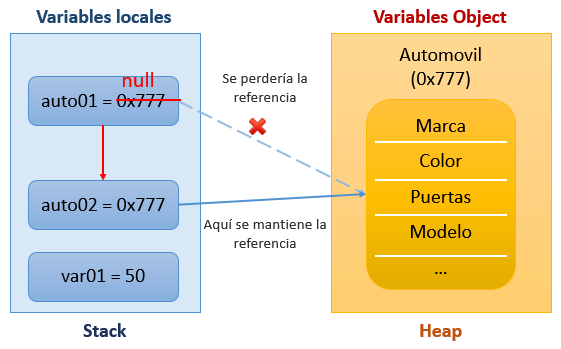
Se recibe una nueva referencia cada vez que usemos la palabra New, aquí es donde se creará un nuevo objeto.

Continuando con el ejemplo anterior, si igualamos nuestra variable **auto01** a **NULL** entonces se perdería la referencia al objeto **Automovil**.

Automovil auto01 = new Automovil();

Automovil auto02 = auto01;

auto01 = null;



Y aquí se almacenaría el valor de Null en esta variable **auto01** y por lo tanto solamente la variable **auto02** mantendría el acceso al objeto **Automovil** pero la variable auto01 ya no podría acceder a los atributos o métodos del objeto creado (Marca, Color, Puertas, etc.).

Ahora bien, pueden ver que hasta este momento la variable **auto02** aún apunta al objeto, por lo tanto el recolector de basura no puede eliminar el objeto **Automovil**, por eso aunque eliminemos esta referencia del objeto (de una variable) aún existe otra variable que sigue apuntando a este objeto.

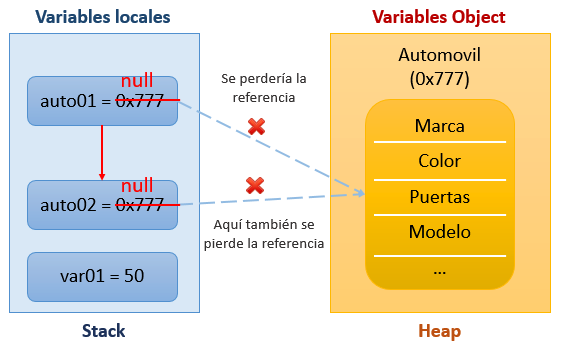
Pero si a la segunda variable **auto02** también la igualamos a **NULL**:

Automovil auto01 = new Automovil();

Automovil auto02 = auto01;

auto01 = null;

auto02 = null;



Ahora sí, tanto la variable **auto01** como la variable **auto02** ya no apuntan al objeto **Automovil** y por lo tanto al romperse las referencias a este objeto entonces ahora si este objeto es un candidato para eliminarse por el recolector de basura de la memoria **Heap**.

Así que un objeto es candidato a eliminarse de la memoria hasta que ninguna variable esté apuntando a dicho objeto es decir que ninguna variable contenga su referencia.

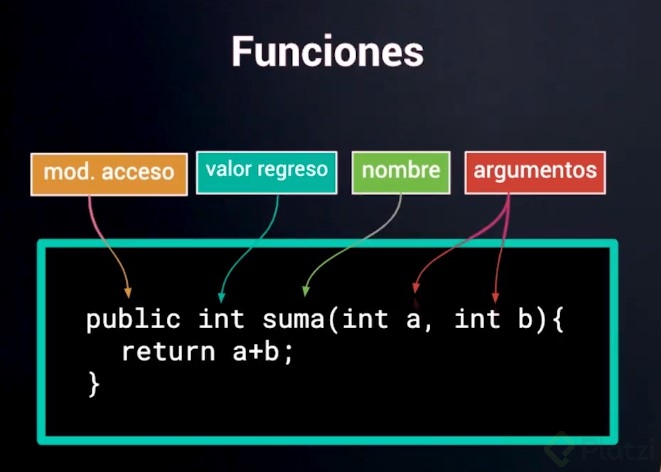
El proceso de recolección de basura es un proceso que demanda muchos recursos por lo que llamar manualmente (o sea nosotros lo programemos) al método **System.gc()**, que es la forma de llamar al Garbage Collector, no garantiza que se ejecute el recolector de basura.

Sin embargo con la llamada a esta función le estamos indicando a Java que en cuanto sea posible ejecutar el recolector y por tanto elimine los objetos de memoria que ya no están siendo utilizados. Es decir que ya no exista ninguna variable apuntando a estos objetos.

| **⚠️ Nota:**   * Por defecto, la JVM asigna una cantidad limitada de memoria según la configuración y el hardware disponible. * Puedes ajustar la memoria máxima y mínima con estos parámetros al ejecutar un programa Java:   java -Xms512m -Xmx2g MiPrograma   1. **Xms512m:** Mínimo 512 MB de Heap **Xmx2g:** Máximo 2 GB de Heap   Si la JVM necesita más memoria de la asignada, puede lanzar un *OutOfMemoryError* |
| --- |

### **Métodos**

En Java, las **funciones** se conocen como **métodos**. Un método es un bloque de código que realiza una tarea específica, puede ser llamado desde otros lugares del programa y puede o no devolver un valor como resultado. Los métodos son esenciales para estructurar el código en Java, ya que permiten la reutilización, organización y encapsulación del comportamiento.



| **⚠️ Nota:**   * En Java, **métodos y funciones son equivalentes en funcionalidad**, pero siempre los llamamos **métodos** porque están encapsulados dentro de una clase. En otros lenguajes (como C, Python o JavaScript), las funciones pueden existir de manera independiente y no estar vinculadas a objetos o clases. |
| --- |

**Elementos Clave**

1. **Modificadores de acceso**Controlan quién puede acceder al método:
   * public: Accesible desde cualquier parte del programa.
   * protected: Accesible dentro del paquete y por subclases.
   * private: Solo accesible dentro de la misma clase.
   * (Por defecto): Solo accesible dentro del mismo paquete.
2. **Modificadores opcionales**Controlan características adicionales:
   * static: Indica que el método pertenece a la clase y no a una instancia.
   * final: El método no puede ser sobrescrito.
   * abstract: El método no tiene cuerpo (declarado en clases abstractas).
3. **Tipo de retorno**Define el tipo de dato que devuelve el método (int, String, void, etc.). Si no devuelve nada, se usa void.
4. **Nombre del método**Debe seguir las convenciones de nombres (usar *lowerCamelCase*).
5. **Parámetros**Una lista opcional de entradas que el método puede recibir. Cada parámetro tiene un tipo y un nombre.
6. **Cuerpo del método**Contiene el código que se ejecuta cuando se llama al método.

#### *Métodos sin retorno*

No devuelven un valor. Se usan para realizar acciones como puedes observar se diferencian de los demás por que usan la palabra void en vez de un tipo de dato que en teoría debería retornar y carecen de la aplicación de la palabra reservada return.

public void imprimirMensaje() {

System.*out*.println("Hola, este es un método.");

}

### 

#### *Métodos con retorno*

Devuelven un valor al lugar desde donde se llaman y el tipo de dato retornado debe coincidir con el tipo de dato de la cabecera del método y se usa la palabra **return** para indicar el dato a retornar y también para indicar el final del método.

public int sumar(int a, int b) {

return a + b; // Devuelve la suma de a y b

}

### 

#### *Métodos estáticos*

Pertenecen a la clase y no requieren una instancia para ser llamados.

public static void mostrarMensaje() {

System.*out*.println("Este es un método estático.");

}

### 

#### *Métodos sobrecargados*

Permiten definir múltiples métodos con el mismo nombre pero diferentes parámetros a esto se le llama overloading.

public int multiplicar(int a, int b) {

return a \* b;

}

public double multiplicar(double a, double b) {

return a \* b;

}

### 

#### *Métodos recursivos*

Se llaman a sí mismos para resolver problemas que pueden dividirse en subproblemas más pequeños.

public int factorial(int n) {

if (n == 0) {

return 1;

}

return n \* factorial(n - 1);

}

Ejemplo práctico:

public class EjemploMetodos {

// Método principal

public static void main(String[] args) {

EjemploMetodos ejemplo = new EjemploMetodos();

ejemplo.imprimirMensaje();

int suma = ejemplo.sumar(5, 10);

System.*out*.println("La suma es: " + suma);

int resultadoFactorial = ejemplo.factorial(5);

System.*out*.println("El factorial es: " + resultadoFactorial);

}

// Método sin retorno

public void imprimirMensaje() {

System.*out*.println("Hola, bienvenido a los métodos en Java.");

}

// Método con retorno

public int sumar(int a, int b) {

return a + b;

}

// Método recursivo

public int factorial(int n) {

if (n == 0) {

return 1;

}

return n \* factorial(n - 1);

}

}

| **⚠️ Nota:**   * Para acceder rápidamente a un método de una clase se puede omitir la creación de la variable asignada a la instancia: * return new Persona().saludar(); |
| --- |

**Buenas prácticas con Métodos**

**Nombres descriptivos:**Usa nombres que indiquen claramente la acción del método. Por ejemplo, calcularArea es mejor que calcular.

**Una tarea por método:**  
Cada método debe realizar una tarea específica. Evita métodos largos y complejos.

**Parámetros claros:**Evita usar demasiados parámetros en un método. Si hay muchos, considera agruparlos en una clase.

**Comentarios cuando sea necesario:**Si el propósito del método no es obvio, incluye un comentario para explicar qué hace.

**Métodos reutilizables:**Escribe métodos que puedan ser reutilizados en lugar de duplicar código.

**Prefiere métodos estáticos para utilidades comunes:**Si el método no depende del estado de la clase, considera hacerlo estático.

#### *Paso por valor o referencia*

En Java, el paso de **parámetros** es un tema fundamental que a menudo genera confusión, especialmente cuando se trata de entender si los argumentos se pasan por valor o por referencia. Vamos a aclarar este concepto con ejemplos y buenas prácticas.

**Paso por valor:**

* Se pasa una copia del valor del argumento.
* Los cambios realizados en el parámetro no afectan al argumento original.

En Java, **todos los parámetros se pasan por valor.** Sin embargo, el comportamiento puede parecer diferente dependiendo de si el parámetro es de tipo primitivo o de tipo objeto.

**a) Paso por valor con tipos primitivos**

Cuando pasas un tipo primitivo (como int, double, boolean, etc.), se pasa una copia del valor. Los cambios en el parámetro no afectan al valor original.

Ejemplo:

public class Main {

public static void main(String[] args) {

int x = 10;

System.*out*.println("Antes de modificar: " + x); // Imprime 10

*modificarValor*(x);

System.*out*.println("Después de modificar: " + x); // Imprime 10

}

public static void modificarValor(int valor) {

valor = 20; // Modifica la copia, no el valor original

}

}

Explicación:

* El valor de x se copia en el parámetro valor.
* Cambiar valor no afecta a x.

**b) Paso por referencia con tipos de objeto**

Cuando pasas un objeto (como un String, un ArrayList, o cualquier instancia de clase), se pasa una copia de la referencia al objeto. Esto significa que:

* No puedes cambiar la referencia original (porque es una copia).
* Pero puedes modificar el estado del objeto al que apunta la referencia.

Ejemplo 1: Modificar el estado del objeto

public class Main {

public static void main(String[] args) {

int[] numeros = {1, 2, 3};

System.*out*.println("Antes de modificar: " + Arrays.toString(numeros)); // [1, 2, 3]

*modificarArray*(numeros);

System.*out*.println("Después de modificar: " + Arrays.toString(numeros)); // [10, 2, 3]

}

public static void modificarArray(int[] arr) {

arr[0] = 10; // Modifica el estado del objeto

}

}

Explicación:

* numeros es una referencia a un array.
* Se pasa una copia de la referencia (arr), pero ambas referencias apuntan al mismo objeto.
* Cambiar el estado del objeto afecta al objeto original.

Ejemplo 2: Intercambiar la referencia

public class Main {

public static void main(String[] args) {

String mensaje = "Hola";

System.*out*.println("Antes de modificar: " + mensaje); // Imprime "Hola"

*modificarString*(mensaje);

System.*out*.println("Después de modificar: " + mensaje); // Imprime "Hola"

}

public static void modificarString(String str) {

str = "Mundo"; // Cambia la copia de la referencia, no el objeto original

}

}

Explicación:

* mensaje es una referencia a un objeto String.
* Se pasa una copia de la referencia (str).
* Cambiar str no afecta a mensaje, porque solo estás modificando la copia de la referencia.

Ejemplo Completo:

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

// Paso por valor con tipos primitivos

int x = 10;

System.*out*.println("Antes de modificar (primitivo): " + x); // 10

*modificarValor*(x);

System.*out*.println("Después de modificar (primitivo): " + x); // 10

// Paso por valor con tipos de objeto

int[] numeros = {1, 2, 3};

System.*out*.println("Antes de modificar (objeto): " + Arrays.toString(numeros)); // [1, 2, 3]

*modificarArray*(numeros);

System.*out*.println("Después de modificar (objeto): " + Arrays.toString(numeros)); // [10, 2, 3]

}

public static void modificarValor(int valor) {

valor = 20; // No afecta al valor original

}

public static void modificarArray(int[] arr) {

arr[0] = 10; // Afecta al objeto original

}

}

**Buenas prácticas**

**a) Evita modificar objetos pasados como parámetros**

* Si un método recibe un objeto como parámetro, evita modificarlo directamente a menos que sea explícitamente necesario.
* En su lugar, devuelve un nuevo objeto con los cambios.

Ejemplo:

public List<String> agregarElemento(List<String> lista, String elemento) {

List<String> nuevaLista = new ArrayList<>(lista); // Copia la lista

nuevaLista.add(elemento);

return nuevaLista; // Devuelve la nueva lista

}

**b) Usa tipos inmutables cuando sea posible**

* Los tipos inmutables (como String, LocalDate, etc.) evitan problemas relacionados con la modificación accidental de objetos.

**c) Documenta el comportamiento de los métodos**

* Si un método modifica el estado de un objeto recibido como parámetro, documéntalo claramente.

Ejemplo:

*/\*\**

*\* Modifica el array recibido como parámetro.*

*\** ***@param*** *arr El array que será modificado.*

*\*/*

public void modificarArray(int[] arr) {

arr[0] = 10;

}

**d) Considera el uso de final en parámetros**

* Usar final en parámetros evita que se reasignen accidentalmente dentro del método.

Ejemplo:

public void procesar(final int valor) {

// valor = 20; // Error: no se puede reasignar

}

### **Constructores**

Un constructor en Java es un método especial dentro de una clase que se ejecuta automáticamente cuando se crea un objeto de esa clase. Su propósito principal es inicializar los atributos del objeto.

**Características de los Constructores**

* Se llaman automáticamente al instanciar un objeto con new.
* Tienen el mismo nombre que la clase.
* No tienen ningún tipo de retorno (ni siquiera void).
* Pueden sobrecargarse (tener varios constructores con diferentes parámetros).

#### *Constructor por defecto*

Si no se define un constructor en la clase, Java proporciona automáticamente un constructor vacío por defecto.

class Persona {

String nombre;

// Constructor por defecto (Java lo genera automáticamente si no lo declaramos)

Persona() {

System.*out*.println("Constructor por defecto ejecutado");

}

}

public class Main {

public static void main(String[] args) {

Persona p = new Persona(); // Llama al constructor por defecto

}

}

#### 

#### *Constructor con parámetros*

Permite inicializar los atributos de la clase al momento de la creación del objeto.

class Persona {

String nombre;

// Constructor con parámetros

Persona(String nombre) {

this.nombre = nombre; // 'this' se usa para referirse al atributo de la clase

}

void mostrarNombre() {

System.*out*.println("Nombre: " + nombre);

}

}

public class Main {

public static void main(String[] args) {

Persona p = new Persona("Juan");

p.mostrarNombre();

}

}

#### *Sobrecarga de constructores*

Se pueden definir múltiples constructores con diferentes parámetros para mayor flexibilidad.

class Persona {

String nombre;

int edad;

// Constructor principal

Persona() {

this("Desconocido", 0); // Llama al constructor con dos parámetros

}

// Constructor con un parámetro

Persona(String nombre) {

this(nombre, 0); // Llama al constructor con dos parámetros

}

// Constructor con dos parámetros

Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

void mostrarDatos() {

System.*out*.println("Nombre: " + nombre + ", Edad: " + edad);

}

}

public class Main {

public static void main(String[] args) {

Persona p1 = new Persona();

Persona p2 = new Persona("Carlos");

Persona p3 = new Persona("Ana", 25);

p1.mostrarDatos();

p2.mostrarDatos();

p3.mostrarDatos();

}

}

**Ventaja**: Evita duplicación de código y mejora la mantenibilidad.

#### *Constructores en clase con herencia*

Cuando una clase hereda de otra, el constructor de la subclase puede llamar al constructor de la superclase usando super().

class Animal {

String especie;

// Constructor de la superclase

Animal(String especie) {

this.especie = especie;

}

}

class Perro extends Animal {

String raza;

// Constructor de la subclase

Perro(String especie, String raza) {

super(especie); // Llama al constructor de Animal

this.raza = raza;

}

void mostrarDatos() {

System.*out*.println("Especie: " + especie + ", Raza: " + raza);

}

}

public class Main {

public static void main(String[] args) {

Perro p = new Perro("Mamífero", "Labrador");

p.mostrarDatos();

}

}

| **⚠️ Nota:**   * Si la superclase tiene un constructor sin parámetros, no es necesario llamar a super(), ya que Java lo hace automáticamente. |
| --- |

**Buenas Prácticas al Usar Constructores**

* Usar this para evitar confusión entre variables locales y atributos de la clase.
* Reutilizar código con this() para reducir redundancias.
* Evitar lógica compleja dentro del constructor; si es necesario, usar métodos auxiliares.
* Siempre validar datos en los constructores antes de asignarlos a los atributos.
* Si deseas crear una clase static y no crear instancias puedes hacer el constructor private.

Ejemplo de validación en el constructor:

class Producto {

String nombre;

double precio;

Producto(String nombre, double precio) {

if (precio < 0) {

throw new IllegalArgumentException("El precio no puede ser negativo");

}

this.nombre = nombre;

this.precio = precio;

}

}

### **This**

En Java, this es una referencia al objeto actual dentro de una clase. Se usa principalmente para:

* Diferenciar atributos de variables locales con el mismo nombre.
* Llamar a otro constructor dentro de la misma clase.
* Pasar la instancia actual a otro método o constructor.
* Devolver la instancia actual desde un método.

#### 

#### *Distinguir atributos de variables*

Cuando los nombres de los parámetros de un constructor o método son iguales a los atributos de la clase, this se usa para diferenciarlos.

Ejemplo:

class Persona {

private String nombre;

private int edad;

public Persona(String nombre, int edad) {

this.nombre = nombre; // `this` evita la confusión con el parámetro

this.edad = edad;

}

public void mostrarDatos() {

System.*out*.println("Nombre: " + this.nombre + ", Edad: " + this.edad);

}

}

public class Main {

public static void main(String[] args) {

Persona p = new Persona("Carlos", 30);

p.mostrarDatos();

}

}

**¿Qué pasaría si no usáramos this?**

El compilador asumiría que nombre y edad dentro del constructor se refieren a los parámetros del método, en lugar de los atributos de la clase.

#### *Llamar a otro constructor*

Podemos llamar a otro constructor de la misma clase con this(), evitando repetir código.

Ejemplo:

class Persona {

private String nombre;

private int edad;

// Constructor por defecto

public Persona() {

this("Desconocido", 0); // Llama al constructor con parámetros

}

// Constructor con parámetros

public Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

public void mostrarDatos() {

System.*out*.println("Nombre: " + nombre + ", Edad: " + edad);

}

}

public class Main {

public static void main(String[] args) {

Persona p1 = new Persona(); // Llama al constructor por defecto

Persona p2 = new Persona("Ana", 25); // Llama al constructor con parámetros

p1.mostrarDatos(); // Nombre: Desconocido, Edad: 0

p2.mostrarDatos(); // Nombre: Ana, Edad: 25

}

}

**Reglas importantes:**

* this() **debe ser la primera línea** dentro del constructor.
* No puedes usar this() en métodos normales, solo en constructores.
* El ámbito de this se limita solo al ámbito de clase.

### 

#### *Pasar la instancia como argumento*

Cuando queremos pasar el objeto actual a otro método o clase, usamos this.

Ejemplo: Pasar this a otro método dentro de la misma clase

class Persona {

private String nombre;

public Persona(String nombre) {

this.nombre = nombre;

}

public void imprimir() {

mostrarDatos(this); // Pasamos la instancia actual al método

}

public void mostrarDatos(Persona p) {

System.*out*.println("Nombre: " + p.nombre);

}

}

public class Main {

public static void main(String[] args) {

Persona p1 = new Persona("Lucas");

p1.imprimir(); // Nombre: Lucas

}

}

Ejemplo: Pasar this a otra clase

class A {

public void mostrar(B obj) {

System.*out*.println("Método llamado desde la clase A usando `this`");

}

}

class B {

void metodo() {

A a = new A();

a.mostrar(this); // Pasamos la instancia actual

}

}

public class Main {

public static void main(String[] args) {

B b = new B();

b.metodo();

}

}

**¿Cuándo usar esto?**

* Cuando una clase necesita acceso al objeto actual en otro método.
* Para registrar el objeto en otra clase, como en **listeners** o **callbacks**.

### 

#### *Retornar la instancia actual*

Hacer el retorno de la instancia actual es muy utilizado en el encadenamiento de métodos.

Ejemplo: Encadenamiento de métodos (Method Chaining)

class Persona {

private String nombre;

public Persona setNombre(String nombre) {

this.nombre = nombre;

return this; // Devuelve el objeto actual

}

public void mostrar() {

System.*out*.println("Nombre: " + nombre);

}

}

public class Main {

public static void main(String[] args) {

new Persona().setNombre("Leo").mostrar();

}

}

**¿Cuándo es útil?**

* Permite encadenar llamadas a métodos en **una sola línea**, útil en librerías como **StringBuilder** o **Streams**.

**Buenas prácticas con this**

**Usa this solo cuando sea necesario**

* Si no hay ambigüedad entre parámetros y atributos, no hace falta usar this.
* 🚫 **Ejemplo innecesario:**

public void setNombre(String nombre) {

this.nombre = nombre; // ✅ Correcto

}

public void mostrarDatos() {

System.*out*.println(this.nombre); // 🚫 No es necesario `this` aquí

**Usa this() para evitar código repetitivo**

* Si hay constructores similares, usa this() para llamar a otro constructor y reducir código duplicado.

**Encadenamiento de métodos (Method Chaining) sólo si aporta claridad**

* Si mejora la legibilidad, es una buena práctica. Pero no abuses si hace el código confuso.

| Uso de this | Explicación |
| --- | --- |
| Diferenciar atributos | this.nombre = nombre; |
| Llamar a otro constructor | this("Desconocido", 0); |
| Pasar la instancia actual | obj.metodo(this); |
| Devolver la instancia actual | return this; |

## 

## ***Encapsulamiento***

El encapsulamiento es uno de los cuatro pilares de la Programación Orientada a Objetos (POO). Consiste en:

* Ocultar los detalles internos de una clase (atributos y métodos) para proteger la integridad de los datos.
* Exponer una interfaz controlada (métodos públicos) para interactuar con los datos.

**Beneficios del encapsulamiento:**

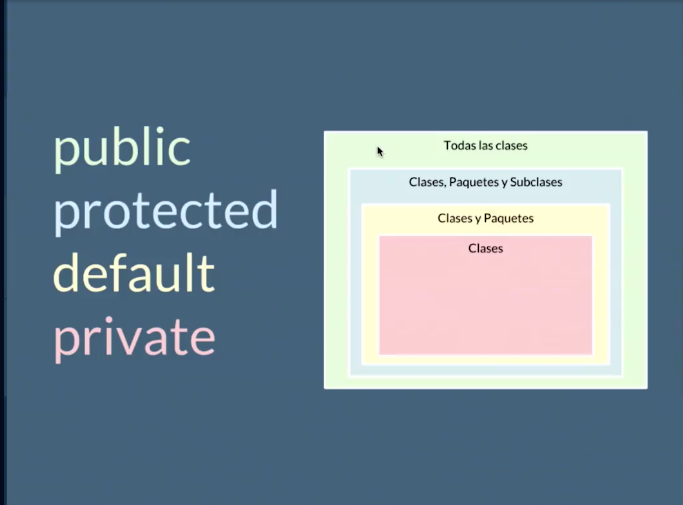
* **Control de acceso:** Permite restringir el acceso directo a los atributos de una clase.
* **Flexibilidad:** Puedes cambiar la implementación interna de una clase sin afectar el código que la usa.
* **Mantenibilidad:** Facilita la depuración y el mantenimiento del código.
* **Seguridad:** Evita que los datos se modifiquen de manera incorrecta.

### **Modificadores de acceso**

Un modificador de acceso define el alcance y visibilidad de un miembro de clase.

Los modificadores de acceso son los siguientes y su área de acción:

| **Modificador** | **Alcance** |
| --- | --- |
| public | Accesible desde cualquier parte del programa. |
| protected | Accesible en la misma clase, en **subclases** y dentro del **mismo paquete**. |
| *Sin modificador* (default) | Accesible sólo dentro del mismo paquete. |
| private | Solo accesible dentro de la misma clase. |



| **⚠️ Nota:**   * Si a nuestras clases y variables no le asignamos un modificador de acceso por defecto su modificador de acceso será el “default” de allí el nombre. |
| --- |

Ejemplo explicado:

class Persona {

public String nombre; // Visible en todo el programa.

private int edad; // Solo accesible dentro de esta clase

protected String direccion; // Accesible en subclases y en el mismo paquete

// Constructor

public Persona(String nombre, int edad, String direccion) {

this.nombre = nombre;

this.edad = edad;

this.direccion = direccion;

}

// Método público para acceder a 'edad'

public int getEdad() {

return edad;

}

// Método público para modificar 'edad' con validación

public void setEdad(int edad) {

if (edad > 0) {

this.edad = edad;

} else {

System.*out*.println("Edad no válida.");

}

}

}

// Subclase en el mismo paquete

class Estudiante extends Persona {

public String escuela;

// Constructor

public Estudiante(String nombre, int edad, String direccion, String escuela) {

super(nombre, edad, direccion);

this.escuela = escuela;

}

public void mostrarInfo() {

System.*out*.println("Nombre: " + nombre); // ✅ Accesible (public)

// System.out.println("Edad: " + edad); ❌ ERROR (private)

System.*out*.println("Dirección: " + direccion); // ✅ Accesible (protected)

}

}

public class Main {

public static void main(String[] args) {

Persona p1 = new Persona("Juan", 30, "Calle 123");

System.*out*.println(p1.nombre); // ✅ Accesible (public)

// System.out.println(p1.edad); ❌ ERROR (private)

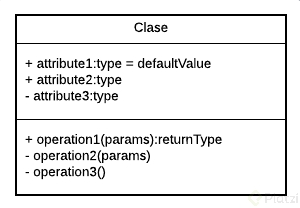
System.*out*.println(p1.getEdad()); // ✅ Accesible a través del getter

p1.setEdad(35); // ✅ Modifica con validación

}

}

Los modificadores de acceso en **UML** se representan con símbolos a un lado de cada atributo de una clase y determinará la visibilidad de cada uno dependiendo cada símbolo se le asigne.



Estos son los niveles de visibilidad que puedes tener:

- private

+ public

# protected

~ default

**Buenas prácticas con modificadores de acceso**

1. **Usa private para proteger datos sensibles y prioriza su uso**

* Evita modificar atributos directamente.
* Proporciona métodos getter y setter con validaciones.

1. **Usa protected solo cuando sea necesario**

* Permite que subclases accedan a atributos sin hacerlos public.
* Pero no es totalmente seguro, porque cualquier clase en el mismo paquete también puede acceder.

1. **Usa public solo si realmente es necesario**

* No hagas variables globales públicas a menos que deban serlo.
* Para métodos, sí es común usar public.

1. **Evita el uso de default en paquetes grandes**

* Si omites un modificador, la visibilidad será default (accesible sólo dentro del mismo paquete).
* En proyectos grandes, puede generar dependencias innecesarias entre clases.

### **Getters y Setters**

Los getters y setters son muy importantes porque nos permiten aplicar el principio de encapsulamiento permitiéndonos trabajar con propiedades que son privadas, pero no de manera directa, sino que a través de un método denominado **setter** donde podemos asignar o actualizar el valor de un dato y con método llamado **getter** podemos obtener o validar ese dato que estamos recibiendo esto gracias a que convertimos el atributo en un método que dentro de él podemos meter más validaciones y reglas de negocio.

**a) Getters:**

* Métodos que obtienen el valor de un atributo privado.
* Suelen tener la forma: public Tipo getNombreAtributo().

**b) Setters:**

* Métodos que establecen el valor de un atributo privado.
* Suelen tener la forma: public void setNombreAtributo(Tipo valor).

Ejemplo de Setters y Getters en java:

public class Persona {

// Atributos privados

private String nombre;

private int edad;

// Getter para nombre

public String getNombre() {

return nombre;

}

// Setter para nombre

public void setNombre(String nombre) {

this.nombre = nombre;

}

// Getter para edad

public int getEdad() {

return edad;

}

// Setter para edad

public void setEdad(int edad) {

if (edad >= 0) { // Validación

this.edad = edad;

} else {

System.*out*.println("Edad no válida");

}

}

}

**Cuándo Usar Constructores y Cuándo Métodos set?**

| Situación | Uso Recomendado |
| --- | --- |
| Crear un objeto y asegurarse de que tenga valores iniciales | Constructor |
| Modificar atributos después de la creación del objeto | Métodos set |
| Evitar la creación de objetos sin datos esenciales | Constructor obligatorio con parámetros |
| Proporcionar valores opcionales con valores por defecto | Sobrecarga de constructores o métodos set |

**Buenas prácticas con getters y setters**

**a) Validación en setters:**

* Usa los setters para validar los datos antes de asignarlos a los atributos.

Ejemplo:

public void setEdad(int edad) {

if (edad >= 0) {

this.edad = edad;

} else {

throw new IllegalArgumentException("La edad no puede ser negativa");

}

}

**b) No exponer atributos innecesariamente:**

* Si un atributo no necesita ser modificado, no proporciones un setter.

Ejemplo:

public class CuentaBancaria {

private double saldo;

// Getter para saldo

public double getSaldo() {

return saldo;

}

// No hay setter para saldo, ya que no debe ser modificado directamente

}

**c) Uso de final para atributos inmutables:**

* Si un atributo no debe cambiar después de la creación del objeto, hazlo al final y no proporciones un setter.

Ejemplo:

public class Persona {

private final String id; // Atributo inmutable

public Persona(String id) {

this.id = id;

}

public String getId() {

return id;

}

}

### **JavaBeans**

Un JavaBean es una clase en Java que sigue ciertas convenciones para representar objetos reutilizables y encapsulados. Son muy usados en frameworks como Spring y Java EE.

**Características de un JavaBean:**

* **Encapsulación:** Todos los atributos deben ser private.
* **Métodos getter y setter:** Se usan para acceder y modificar los atributos.
* **Constructor sin parámetros:** Debe haber un constructor vacío además de otros opcionales.
* **Serializable:** Implementa Serializable para permitir la persistencia del objeto.

Si te has dado cuenta y has visto todo el tema de encapsulamiento técnicamente todas las clases que hemos realizado son casi idénticas a un JavaBean solo con excepción de la implementación del serializable que veremos más adelante.

Ejemplo:

import java.io.Serializable;

public class PersonaBean implements Serializable {

private String nombre;

private int edad;

// Constructor vacío obligatorio

public PersonaBean() {}

// Constructor con parámetros (opcional)

public PersonaBean(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

// Getter y Setter

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public int getEdad() {

return edad;

}

public void setEdad(int edad) {

if (edad > 0) {

this.edad = edad;

} else {

System.*out*.println("Edad no válida.");

}

}

}

**Relación con la Encapsulación**

* **Uso de private en atributos:** No se puede acceder directamente, sólo a través de métodos.
* **Métodos getter y setter:** Proporcionan control sobre cómo se accede o modifica el objeto.
* **Seguridad y Mantenimiento:** Permite validar datos antes de asignarlos (setEdad(int edad)).

Ejemplo de uso de JavaBean:

public class Main {

public static void main(String[] args) {

PersonaBean persona = new PersonaBean();

// Modificar atributos con setters

persona.setNombre("Ana");

persona.setEdad(25);

// Obtener valores con getters

System.*out*.println("Nombre: " + persona.getNombre());

System.*out*.println("Edad: " + persona.getEdad());

}

}

**Buenas prácticas con JavaBeans**

1. Usa private para los atributos y public para los métodos getter/setter.
2. Evita lógica compleja en setters (para eso usa métodos de negocio aparte).
3. Si el objeto no debe cambiar, usa final en atributos y elimina setters.
4. Si usas Java 14+, puedes usar record en lugar de JavaBeans (si el objeto es inmutable).

### **Validación de atributos**

Cuando trabajamos con atributos en una clase, es importante asegurarnos de que sus valores sean correctos antes de asignarlos. La validación de atributos evita datos inconsistentes o errores en la ejecución del programa.

1. **Validación en los Setters**

Una forma común es validar los valores en los setters antes de asignarlos.

Ejemplo: Validación de edad en una clase Persona

public class Persona {

private String nombre;

private int edad;

public Persona(String nombre, int edad) {

this.nombre = nombre;

setEdad(edad); // Usamos el setter para aplicar validación

}

public void setEdad(int edad) {

if (edad < 0 || edad > 150) {

throw new IllegalArgumentException("La edad debe estar entre 0 y 150.");

}

this.edad = edad;

}

public int getEdad() {

return edad;

}

}

**Pros:** Fácil de entender y aplicar.

**Contras:** Se debe llamar siempre al setter dentro del constructor para aplicar la validación.

1. **Validación en el Constructor**

Si queremos asegurarnos de que un objeto nunca tenga valores inválidos, podemos validar los atributos directamente en el constructor.

Ejemplo: Validación en el constructor sin usar setters

public class Producto {

private String nombre;

private double precio;

public Producto(String nombre, double precio) {

if (nombre == null || nombre.isBlank()) {

throw new IllegalArgumentException("El nombre no puede estar vacío.");

}

if (precio <= 0) {

throw new IllegalArgumentException("El precio debe ser mayor a 0.");

}

this.nombre = nombre;

this.precio = precio;

}

public String getNombre() {

return nombre;

}

public double getPrecio() {

return precio;

}

}

**Pros:** Asegura que los objetos siempre sean creados con valores válidos.

**Contras:** No se puede modificar el valor después de la creación sin añadir setters.

1. **Validación con Objetos Inmutables (final)**

Si un atributo no debe cambiar después de ser inicializado, podemos declararlo como final y validarlo solo en el constructor.

Ejemplo: Persona con nombre inmutable

public class Persona {

private final String nombre;

public Persona(String nombre) {

if (nombre == null || nombre.isBlank()) {

throw new IllegalArgumentException("El nombre no puede estar vacío.");

}

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

}

**Pros:** Seguridad y consistencia de datos.

**Contras:** No se puede modificar el valor después de la inicialización.

**4. Validación con Métodos Auxiliares**

Para evitar duplicar código en el constructor y los setters, podemos usar métodos auxiliares.

Ejemplo: Método validarEdad() separado

public class Persona {

private String nombre;

private int edad;

public Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = validarEdad(edad);

}

public void setEdad(int edad) {

this.edad = validarEdad(edad);

}

private int validarEdad(int edad) {

if (edad < 0 || edad > 150) {

throw new IllegalArgumentException("La edad debe estar entre 0 y 150.");

}

return edad;

}

public int getEdad() {

return edad;

}

}

**Pros**: Evita código repetido.

**Contras:** Requiere más estructura en la clase.

**5. Validación con Anotaciones y javax.validation**

Para proyectos más grandes, podemos usar validaciones automáticas con javax.validation.

Ejemplo: Uso de @NotNull y @Min / @Max

import jakarta.validation.constraints.\*;

public class Usuario {

@NotBlank(message = "El nombre no puede estar vacío.")

private String nombre;

@Min(value = 18, message = "Debe ser mayor de edad.")

@Max(value = 100, message = "Debe ser menor de 100 años.")

private int edad;

public Usuario(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

public String getNombre() {

return nombre;

}

public int getEdad() {

return edad;

}

}

**Pros:** Separación clara de responsabilidades.

**Contras:** Necesita dependencias adicionales (Jakarta Validation).

**Buenas prácticas para validación de atributos**

* **Para proyectos pequeños:** Validar en setters o en el constructor.
* **Para proyectos medianos:** Usar métodos auxiliares para evitar duplicación de código.
* **Para proyectos grandes:** Usar anotaciones (javax.validation) y técnicas avanzadas como objetos inmutables.

## ***Ámbito o scope de las variables***

### **Ámbito de clase**

Las variables definidas dentro de una clase, pero fuera de cualquier método, constructor o bloque, tienen ámbito a nivel de clase.

Estas variables son accesibles por todos los métodos de la clase.

Pueden ser:

* **Variables de instancia:** Pertenecen a cada objeto de la clase.
* **Variables estáticas:** Pertenecen a la clase y no a una instancia específica.

public class Persona {

String nombre; // Variable de instancia

static String *especie* = "Humano"; // Variable estática

public void mostrarNombre() {

System.*out*.println("Nombre: " + nombre);

}

public static void mostrarEspecie() {

System.*out*.println("Especie: " + *especie*);

}

}

### **Ámbito de método**

Las variables declaradas dentro de un método tienen ámbito solo dentro de ese método.

* Declaradas dentro de un método, constructor o bloque.
* No son accesibles fuera del método donde fueron definidas.
* Deben inicializarse antes de usarse.

public void calcularSuma() {

int suma = 0; // Variable local

for (int i = 1; i <= 5; i++) {

suma += i;

}

System.*out*.println("Suma: " + suma);

// System.out.println(i); // Error: i no está disponible aquí

}

### **Ámbito de bloque**

Las variables declaradas dentro de un bloque (como if, for, while) tienen ámbito limitado al bloque donde se declaran.

* Declaradas dentro de un bloque delimitado por {}.
* Accesibles sólo dentro del bloque donde se declaran.

public void verificarEdad(int edad) {

if (edad >= 18) {

String mensaje = "Eres mayor de edad."; // Variable de bloque

System.*out*.println(mensaje);

}

// System.out.println(mensaje); // Error: mensaje no es accesible aquí

}

**Reglas del Abmito**

1. No se puede usar una variable fuera de su ámbito.

{

int x = 10;

}

// System.out.println(x); // Error: x está fuera de alcance

1. No se pueden declarar dos variables con el mismo nombre en el mismo ámbito.

public void ejemplo() {

int x = 5;

// int x = 10; // Error: x ya está declarada

}

1. Variables locales no tienen valores predeterminados:

Deben inicializarse antes de ser usadas.

public void calcular() {

int x;

// System.out.println(x); // Error: x no está inicializada

}

**Buenas prácticas sobre el Ámbito**

1. **Usa el ámbito más pequeño posible:**

* Declara variables solo donde las necesites y limita su alcance para mejorar la legibilidad y evitar errores.

for (int i = 0; i < 10; i++) {

System.out.println("Número: " + i);

}

// i no está disponible aquí

1. **Evita usar variables globales innecesarias:**

* Minimiza las variables a nivel de clase a menos que sean estrictamente necesarias.

1. **Inicializa variables locales antes de usarlas:**

* Esto evita errores de compilación y asegura que las variables tengan valores válidos.

1. **Usa nombres descriptivos para evitar conflictos:**

* Si hay variables en distintos ámbitos con nombres similares, asegúrate de que sus nombres sean descriptivos.

public void ejemplo() {

int contador = 0; // Variable local

for (int contador = 0; contador < 5; contador++) { // Error

System.*out*.println(contador);

}

}

1. **Usa this para distinguir variables de instancia de variables locales:**

public class Persona {

String nombre;

public void setNombre(String nombre) {

this.nombre = nombre; // Usa `this` para referirse a la variable de instancia

}

}

## ***Static***

Si te has dado cuenta en temas anteriores has visto la palabra static en este tema vamos de una vez por todas definir para que se usa y en qué contextos se puede aplicar.

La palabra clave static en Java se usa para definir **atributos, métodos, bloques de código y clases anidadas** que pertenecen a la clase en sí, en lugar de a las instancias de la clase.

Cuando un miembro de una clase es static, **no pertenece a ninguna instancia específica**, sino a la clase en general. Esto significa que:

* Se puede acceder a él **sin necesidad de crear objetos**.
* Su valor **es compartido por todas las instancias** de la clase.
* Es útil para **constantes, métodos utilitarios y gestión de recursos compartidos**.

Ejemplo básico:

class Ejemplo {

static int *contador* = 0; // Variable compartida por todas las instancias

public Ejemplo() {

*contador*++;

}

}

public class Main {

public static void main(String[] args) {

new Ejemplo();

new Ejemplo();

System.*out*.println(Ejemplo.*contador*); // Output: 2

}

}

### **En atributos**

Una variable estática se declara dentro de una clase, pero fuera de los métodos.

Ejemplo:

class Config {

static String *empresa* = "empresa"; // Se comparte entre todas las instancias

}

Uso:

System.out.println(Config.empresa); // No necesita instancias

**Buenas prácticas en atributos static**

* Se usan para **constantes globales** (final static).
* Se utilizan para **atributos compartidos** entre objetos.
* **No abusar** de static, ya que puede dificultar pruebas unitarias y generar código acoplado.

### **En métodos**

Un método static puede **acceder solo a miembros estáticos** y **no puede usar this o super**.

Ejemplo:

class Utilidades {

static int suma(int a, int b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

System.*out*.println(Utilidades.*suma*(5, 3)); // Output: 8

}

}

**Buenas prácticas en métodos static**

* Útiles para **métodos utilitarios** como en Math.
* Usados en **factories** para crear objetos.
* **No usarlos si dependen del estado interno** de la clase.

Ejemplo incorrecto:

class Contador {

static int *valor* = 0;

static void incrementar() {

*valor*++; // Esto hace que el estado de la clase cambie

}

}

⚠ Este patrón puede causar problemas en entornos concurrentes.

### **En bloques de inicialización**

Un bloque static se ejecuta una sola vez cuando la clase se carga en memoria.

Ejemplo:

class Ejemplo {

static int *numero*;

static {

*numero* = 42; // Se ejecuta al cargar la clase

System.*out*.println("Bloque static ejecutado.");

}

}

Salida:

Bloque static ejecutado.

**Buenas prácticas en bloques static**

* Usarlos para **cargar configuraciones** iniciales.
* **No abusar**, puede hacer el código difícil de leer.

### **En clases anidadas**

Si una clase anidada es static, no necesita una instancia de la clase externa para existir.

Ejemplo:

class Externa {

static class Interna {

void mostrar() {

System.*out*.println("Hola desde la clase interna estática.");

}

}

}

public class Main {

public static void main(String[] args) {

Externa.Interna obj = new Externa.Interna();

obj.mostrar();

}

}

Salida:

Hola desde la clase interna estática.

**Buenas prácticas en clases static anidadas**

* Útiles cuando la clase interna **no necesita acceso a la instancia de la externa**.
* Si necesitas acceso a atributos de la clase externa, **no hagas la clase** static.

### **En clase main**

El método main de Java siempre es static:

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

Permite que la JVM lo ejecute **sin necesidad de crear una instancia de la clase**.

**Diferencias entre** static **y** final

| **Característica** | **static** | **final** |
| --- | --- | --- |
| Pertenece a la clase | ✅ Sí | ❌ No |
| Compartido entre instancias | ✅ Sí | ❌ No |
| Se puede cambiar su valor | ✅ Sí | ❌ No (si es una variable) |

Ejemplo de combinación:

class Config {

static final double *PI* = 3.1416; // Constante

}

✔ static es útil para **atributos y métodos que deben compartirse** entre instancias.

✔ Se usa en **métodos utilitarios, clases anidadas y bloques de inicialización**.

✔ No abusar de static, ya que puede generar código difícil de probar y mantener.

Ejemplo completo:

class Utilidades {

static final String *EMPRESA* = "empresa"; // Constante

static int sumar(int a, int b) { // Método estático

return a + b;

}

static {

System.*out*.println("Inicializando clase Utilidades...");

}

static class Interna {

void mostrar() {

System.*out*.println("Hola desde la clase interna estática.");

}

}

}

public class Main {

public static void main(String[] args) {

System.*out*.println(Utilidades.*EMPRESA*);

System.*out*.println(Utilidades.*sumar*(10, 5));

Utilidades.Interna obj = new Utilidades.Interna();

obj.mostrar();

}

}

Salida:

Inicializando clase Utilidades...

empresa

15

Hola desde la clase interna estática.

## ***Records***

Los **Records** en Java son una forma concisa de definir clases inmutables para representar datos. Fueron introducidos en **Java 14** como una funcionalidad preview y se estabilizaron en **Java 16**.

Un **Record** es una **clase especial** diseñada para contener datos sin la necesidad de escribir código repetitivo como:

* **Constructores**
* **Getters**
* **Métodos toString(), equals() y hashCode()**

Básicamente, los **Records** son **inmutables** y simplifican la creación de **clases DTO (Data Transfer Object)** o **clases modelo**.

Ejemplo básico de record:

public record Persona(String nombre, int edad) {}

Esto equivale a escribir una clase normal así:

public final class Persona {

private final String nombre;

private final int edad;

public Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

public String nombre() { return nombre; }

public int edad() { return edad; }

@Override

public String toString() {

return "Persona[nombre=" + nombre + ", edad=" + edad + "]";

}

@Override

public boolean equals(Object o) { /\* Implementación automática \*/ }

@Override

public int hashCode() { /\* Implementación automática \*/ }

}

Como ves, **con un Record** evitamos escribir todo ese código manualmente.

**Características de los Records**

✅ Inmutables: No se pueden modificar después de su creación.

✅ No permiten herencia: Siempre son final.

✅ Generan automáticamente toString(), equals() y hashCode().

✅ Constructor implícito: Java lo genera automáticamente.

**Cómo usar un Record en Java**

1. **Creación de un Record**

public record Producto(String nombre, double precio, int stock) {}

2. **Uso en el código**

public class Main {

public static void main(String[] args) {

Producto p1 = new Producto("Laptop", 1200.99, 5);

System.*out*.println(p1); // Producto[nombre=Laptop, precio=1200.99, stock=5]

// Acceso a los datos

System.*out*.println("Nombre: " + p1.nombre());

System.*out*.println("Precio: " + p1.precio());

}

}

### **Constructores personalizados**

Aunque los Records generan un constructor automáticamente, podemos definir el nuestro si necesitamos validaciones.

public record Usuario(String nombre, String email) {

public Usuario {

if (nombre == null || nombre.isBlank()) {

throw new IllegalArgumentException("El nombre no puede estar vacío.");

}

if (!email.contains("@")) {

throw new IllegalArgumentException("Email no válido.");

}

}

}

### **Añadir métodos**

También puedes agregar métodos adicionales en un **Record**, pero las variables siguen siendo inmutables.

public record Rectangulo(double ancho, double alto) {

public double area() {

return ancho \* alto;

}

}

Uso:

Rectangulo r = new Rectangulo(5, 10);

System.out.println("Área: " + r.area()); // 50.0

### **equals() y hashCode()**

Los **Records** generan equals() y hashCode() automáticamente.

Producto p1 = new Producto("Laptop", 1200.99, 5);

Producto p2 = new Producto("Laptop", 1200.99, 5);

System.out.println(p1.equals(p2)); // true

System.out.println(p1.hashCode() == p2.hashCode()); // true

### **Atributos y métodos static**

Los records no pueden tener atributos de instancia mutables, pero sí pueden contener constantes (static final) y métodos static**.**

Ejemplo**:**

public record Persona(String nombre, int edad) {

// Atributo estático

public static final String *DEFAULT\_NAME* = "Desconocido";

// Método estático

public static Persona createDefault() {

return new Persona(*DEFAULT\_NAME*, 0);

}

// Método de instancia

public String saludo() {

return "Hola, soy " + nombre;

}

}

**¿Cuándo usar un Record en lugar de una clase normal?**

Usa un Record si:

✔️ Tu clase solo representa datos y no requiere lógica compleja.

✔️ Quieres menos código repetitivo y un código más limpio.

✔️ Necesitas inmutabilidad por seguridad y estabilidad.

Evita Records si:

❌ Necesitas herencia (extends).

❌ Vas a modificar los valores después de la creación.

❌ Requieres lógica de negocio compleja dentro de la clase.

**Buenas prácticas al usar Records**

✅ Úsalos para modelar datos inmutables.

✅ Evita poner lógica de negocio dentro de un Record.

✅ No los uses si necesitas herencia (los Records son final).

✅ Si necesitas mutabilidad, usa clases normales en su lugar.

## ***Relaciones entre clases***

Las relaciones entre clases se podría definir como la interacción entre dos clases estas pueden ser de diferentes tipos y se pueden clasificar en:

### **Asociación (Relación general)**

* Es **cualquier tipo de relación entre dos clases** donde un objeto usa o interactúa con otro.
* Puede ser **unidireccional o bidireccional**.

Ejemplo: Un Profesor y una Escuela

class Escuela {

private String nombre;

public Escuela(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

}

class Profesor {

private String nombre;

private Escuela escuela; // Asociación (el profesor pertenece a una escuela)

public Profesor(String nombre, Escuela escuela) {

this.nombre = nombre;

this.escuela = escuela;

}

public void mostrarInfo() {

System.*out*.println(nombre + " trabaja en " + escuela.getNombre());

}

}

public class Main {

public static void main(String[] args) {

Escuela escuela = new Escuela("Instituto Nacional");

Profesor profesor = new Profesor("Juan Pérez", escuela);

profesor.mostrarInfo();

}

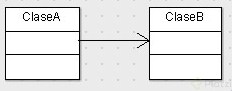
}

Aquí, Profesor **tiene una asociación con** Escuela, pero ambos pueden existir sin depender el uno del otro.

La asociación en UML se representa mediante este tipo de flecha

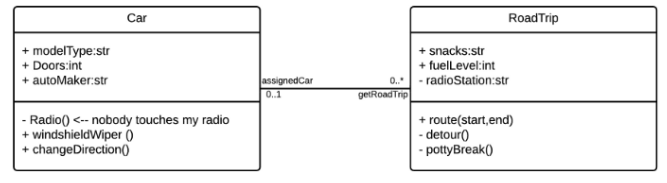


Como su nombre lo dice, notarás que cada vez que esté referenciada este tipo de flecha significa que ese elemento contiene al otro en su definición. La flecha apuntará hacia la dependencia



Con esto vemos que la clase A está asociada a la clase B.

La asociación **Bidireccional** se representa con una línea recta.



### **Agregación (Asociación débil / “Tiene un - unos”)**

* Es un **tipo especial de asociación** donde un objeto **tiene otro objeto, pero este puede existir por separado**.
* **Ejemplo:** Un Equipo tiene Jugadores, pero los jugadores pueden existir sin el equipo.

En muchos casos, la **agregación** se usa para representar una **relación de uno a muchos**, pero **no siempre es así**. La clave de la **agregación** es que representa una relación **débil** entre objetos, lo que significa que los objetos pueden existir de manera independiente.

Ejemplo de agregación **Uno a Muchos**:

**Una Universidad tiene muchos Estudiantes**, pero los estudiantes pueden existir sin la universidad.

class Universidad {

private String nombre;

private List<Estudiante> estudiantes; // Agregación (Uno a Muchos)

public Universidad(String nombre) {

this.nombre = nombre;

this.estudiantes = new ArrayList<>();

}

public void agregarEstudiante(Estudiante estudiante) {

estudiantes.add(estudiante);

}

public void mostrarEstudiantes() {

System.*out*.println("Estudiantes de la universidad " + nombre + ":");

for (Estudiante e : estudiantes) {

System.*out*.println("- " + e.getNombre());

}

}

}

public class Main {

public static void main(String[] args) {

Estudiante maria = new Estudiante("María");

Estudiante juan = new Estudiante("Juan");

Universidad universidad = new Universidad("UPV");

universidad.agregarEstudiante(maria);

universidad.agregarEstudiante(juan);

universidad.mostrarEstudiantes();

}

}

Aquí, la universidad tiene muchos estudiantes (**Uno a Muchos**), pero los estudiantes pueden existir sin la universidad.

Ejemplo de agregación (**Uno a Uno**)

**Una Persona tiene un Pasaporte**, pero el pasaporte sigue existiendo aunque la persona fallezca.

class Pasaporte {

private String numero;

public Pasaporte(String numero) {

this.numero = numero;

}

public String getNumero() {

return numero;

}

}

class Persona {

private String nombre;

private Pasaporte pasaporte; // Agregación (Uno a Uno)

public Persona(String nombre, Pasaporte pasaporte) {

this.nombre = nombre;

this.pasaporte = pasaporte;

}

public void mostrarInfo() {

System.*out*.println(nombre + " tiene el pasaporte: " + pasaporte.getNumero());

}

}

public class Main {

public static void main(String[] args) {

Pasaporte pasaporte = new Pasaporte("ABC123");

Persona persona = new Persona("Carlos", pasaporte);

persona.mostrarInfo();

}

}

Aquí, una persona tiene un pasaporte (**Uno a Uno**), pero el pasaporte puede seguir existiendo aunque la persona desaparezca.

Ejemplo de agregación (**Muchos a Muchos**)

**Un** Profesor puede enseñar **varias** Materias, y una Materia puede tener **varios** Profesores.

import java.util.List;

import java.util.ArrayList;

class Profesor {

private String nombre;

private List<Materia> materias;

public Profesor(String nombre) {

this.nombre = nombre;

this.materias = new ArrayList<>();

}

public void agregarMateria(Materia materia) {

materias.add(materia);

}

public void mostrarMaterias() {

System.*out*.println(nombre + " enseña:");

for (Materia m : materias) {

System.*out*.println("- " + m.getNombre());

}

}

}

class Materia {

private String nombre;

public Materia(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

}

public class Main {

public static void main(String[] args) {

Profesor profeJuan = new Profesor("Juan");

Profesor profeAna = new Profesor("Ana");

Materia matematicas = new Materia("Matemáticas");

Materia fisica = new Materia("Física");

profeJuan.agregarMateria(matematicas);

profeJuan.agregarMateria(fisica);

profeAna.agregarMateria(matematicas);

profeJuan.mostrarMaterias();

profeAna.mostrarMaterias();

}

}

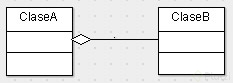
Aquí, un profesor puede tener muchas materias, y una materia puede tener muchos profesores (**Muchos a Muchos**).

La agregación se representa en **UML** con esta flecha:

agregacion.png

Esto se parece a la asociación en que un elemento dependerá del otro, pero en este caso será:

* Uno a uno
* Uno a muchos
* Muchos a muchos



Con esto decimos que la ClaseA contiene varios elementos de la Clase B.

### **Composición (Asociación fuerte / “Contiene un”)**

Se usa cuando un objeto contiene instancias de otras clases como atributos, en lugar de heredar de ellas. Esto promueve el principio **"composición sobre herencia"**, lo que hace que el código sea más flexible y fácil de mantener.

La **composición** es una relación **"tiene un"** (HAS-A), en contraste con la herencia, que representa una relación **"es un"** (IS-A).

Por ejemplo:

* **Un coche tiene un motor** → Composición.
* **Un coche es un vehículo** → Herencia.

Cuando un objeto se destruye, sus partes también pueden destruirse si la composición es **fuerte** (cuando la vida útil de los objetos dependientes depende del objeto principal).

Ejemplo: Clase motor (**Componente**)

public class Motor {

private String tipo;

public Motor(String tipo) {

this.tipo = tipo;

}

public void encender() {

System.*out*.println("El motor " + tipo + " está encendido.");

}

public void apagar() {

System.*out*.println("El motor " + tipo + " está apagado.");

}

public String getTipo() {

return tipo;

}

}

Clase Carro (**Objeto que compone**)

public class Carro {

private String marca;

private Motor motor; // Relación de composición

public Carro(String marca, Motor motor) {

this.marca = marca;

this.motor = motor;

}

public void arrancar() {

System.*out*.println("Arrancando el carro " + marca);

motor.encender();

}

public void apagar() {

System.*out*.println("Apagando el carro " + marca);

motor.apagar();

}

public String getMarca() {

return marca;

}

}

Uso en main:

public class Main {

public static void main(String[] args) {

Motor motorV8 = new Motor("V8");

Carro mustang = new Carro("Ford Mustang", motorV8);

mustang.arrancar();

mustang.apagar();

}

}

Salida:

Arrancando el carro Ford Mustang

El motor V8 está encendido.

Apagando el carro Ford Mustang

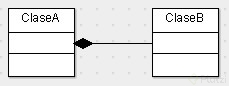
El motor V8 está apagado.

Aquí, Carro **contiene** un Motor, y **el motor no puede existir sin el carro**. Esto es **composición**.

La composición se puede representar en UML por medio de este conector:

**composicion.png**

Este es similar al anterior solo que su relación es totalmente compenetrada de tal modo que conceptualmente una de estas clases no podría vivir si no existiera la otra.



**Buenas prácticas en composición**

✅ **Encapsulación**: Usa private en los atributos y proporciona métodos getters y setters solo si son necesarios.

✅ **Favor composición sobre herencia**: Reduce acoplamiento y hace el código más flexible.

✅ **Inyección de dependencias**: Permite que un objeto reciba sus dependencias desde afuera (como en el constructor del Carro).

✅ **Evita crear objetos innecesarios dentro de la clase**: Usa inyección de dependencias en lugar de this.motor = new Motor("V8");.

### **Delegación**

* Es cuando un objeto **pide a otro que realice una tarea por él**.

**Ejemplo:** Un Cliente le pide a un Cajero que haga una transacción.

import java.util.List;

import java.util.ArrayList;

class Estudiante {

private String nombre;

public Estudiante(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

}

class Cajero {

public void procesarPago(double monto) {

System.*out*.println("Procesando pago de $" + monto);

}

}

class Cliente {

private Cajero cajero; // Delegación

public Cliente(Cajero cajero) {

this.cajero = cajero;

}

public void realizarPago(double monto) {

cajero.procesarPago(monto); // El cliente delega la tarea al cajero

}

}

public class Main {

public static void main(String[] args) {

Cajero cajero = new Cajero();

Cliente cliente = new Cliente(cajero);

cliente.realizarPago(100.0);

}

}

Aquí, Cliente **no maneja el pago directamente, sino que delega la tarea al Cajero**.

### **Dependencia**

* Se da cuando **un método de una clase usa otra clase temporalmente**.
* No es una relación permanente como la asociación, agregación o composición.

Ejemplo:

class Producto {

private String nombre;

public Producto(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

}

class Tienda {

public void venderProducto(Producto producto) { // Dependencia

System.*out*.println("Vendiendo producto: " + producto.getNombre());

}

}

public class Main {

public static void main(String[] args) {

Producto laptop = new Producto("Laptop Dell");

Tienda tienda = new Tienda();

tienda.venderProducto(laptop); // Se usa temporalmente

}

}

Aquí, Tienda usa un Producto **solo en el método**, sin almacenarlo como atributo.

## ***Herencia***

La **herencia** es un pilar fundamental de la **Programación Orientada a Objetos (POO)**. Permite que una clase (subclase) herede atributos y métodos de otra (superclase), fomentando la **reutilización de código** y facilitando el mantenimiento.

La **herencia** permite que una clase nueva tome propiedades y comportamientos de una clase existente.  
La clase que **hereda** se llama **subclase** (o clase hija), y la clase de la que hereda se llama **superclase** (o clase padre).

Ejemplo básico de herencia:

// Superclase

class Animal {

String nombre;

void hacerSonido() {

System.*out*.println("Hace un sonido");

}

}

// Subclase (hereda de Animal)

class Perro extends Animal {

void ladrar() {

System.*out*.println("Guau Guau!");

}

}

// Uso de la herencia

public class Main {

public static void main(String[] args) {

Perro miPerro = new Perro();

miPerro.nombre = "Firulais";

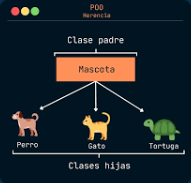
miPerro.hacerSonido(); // Heredado de Animal

miPerro.ladrar(); // Método propio de Perro

}

}

Perro hereda de Animal, lo que significa que puede usar métodos y atributos de Animal sin volverlos a escribir.



### **Extends**

Para que una clase herede de otra, usamos la palabra clave **extends**.

class Superclase {

// Código de la superclase

}

class Subclase extends Superclase {

// Código adicional de la subclase

}

Una subclase puede añadir nuevos métodos y atributos, pero también puede modificar los heredados.

### **Super**

La palabra clave **super** sirve para:

1. **Acceder a métodos y atributos de la superclase**.
2. **Llamar al constructor de la superclase**.

Ejemplo:

class Animal {

String nombre;

Animal(String nombre) {

this.nombre = nombre;

}

void hacerSonido() {

System.*out*.println("Hace un sonido");

}

}

class Perro extends Animal {

Perro(String nombre) {

super(nombre); // Llamamos al constructor de la superclase

}

void mostrarNombre() {

System.*out*.println("El perro se llama: " + super.nombre);

}

}

public class Main {

public static void main(String[] args) {

Perro miPerro = new Perro("Rex");

miPerro.mostrarNombre();

}

}

**super(nombre);** llama al constructor de Animal desde Perro.

**super.nombre** accede al atributo nombre de la superclase.

### **@Override**

La **sobreescritura de métodos** permite a la subclase cambiar la implementación de un método heredado.

Ejemplo de sobreescritura **(@Override):**

class Animal {

void hacerSonido() {

System.*out*.println("Hace un sonido genérico");

}

}

class Perro extends Animal {

@Override

void hacerSonido() {

System.*out*.println("Guau Guau!");

}

}

public class Main {

public static void main(String[] args) {

Animal miAnimal = new Perro();

miAnimal.hacerSonido(); // Llamará al método de Perro

}

}

**@Override** indica que estamos **redefiniendo un método heredado**.

Si la subclase no lo redefiniera, usaría el método de la superclase.

### **Evitar herencia**

Si queremos que **una clase no pueda ser heredada**, usamos la palabra clave **final**.

Ejemplo de clase final:

final class Animal {

void hacerSonido() {

System.*out*.println("Hace un sonido");

}

}

// Esto generará un error

class Perro extends Animal {

void ladrar() {

System.*out*.println("Guau Guau!");

}

}

**No se puede heredar de una clase final**.  
También se puede aplicar final a métodos para evitar su sobreescritura.

Ejemplo de método final:

class Animal {

final void hacerSonido() {

System.*out*.println("Hace un sonido");

}

}

class Perro extends Animal {

// Esto dará error

void hacerSonido() {

System.*out*.println("Guau Guau!");

}

}

### **Herencia múltiple**

Java no permite herencia múltiple directa, es decir, una clase no puede heredar de más de una clase a la vez.

Sin embargo, podemos usar interfaces para simular la herencia múltiple.

Ejemplo de múltiples interfaces:

interface Volador {

void volar();

}

interface Nadador {

void nadar();

}

// Una clase puede implementar múltiples interfaces

class Pato implements Volador, Nadador {

@Override

public void volar() {

System.*out*.println("El pato vuela.");

}

@Override

public void nadar() {

System.*out*.println("El pato nada.");

}

}

public class Main {

public static void main(String[] args) {

Pato miPato = new Pato();

miPato.volar();

miPato.nadar();

}

}

Se usa **implements** en lugar de **extends** cuando queremos usar múltiples comportamientos.

### **Herencia vs Composición**

**Herencia:** "Un Perro es un Animal".  
**Composición:** "Un Coche tiene un Motor".

**Herencia:** “Es un”

**Composición:** “Tiene un/unos”

**Prefiere composición** si la relación **no es una especialización clara**.

Ejemplo de composición en lugar de herencia:

class Motor {

void encender() {

System.*out*.println("Motor encendido");

}

}

class Coche {

private Motor motor = new Motor();

void encender() {

motor.encender();

}

}

**Buenas prácticas en herencia**

✅ Usar herencia solo cuando tenga sentido (no fuerces la relación).

✅ Evitar herencias profundas (demasiados niveles dificultan el mantenimiento).

✅ Prefiere composición sobre herencia cuando sea posible.

✅ Usar @Override siempre que sobreescribas un método.

✅ Evitar heredar constructores directamente, usa super() cuando sea necesario.

✅ Si una clase no debe ser heredada, usa final.

## ***Polimorfismo***

El **polimorfismo** es un concepto clave en la Programación Orientada a Objetos (POO) que permite que un mismo método pueda comportarse de manera diferente según el objeto que lo invoque. En Java, se pueden distinguir dos tipos de polimorfismo:

1. **Polimorfismo en tiempo de compilación (Sobrecarga de métodos)**
2. **Polimorfismo en tiempo de ejecución (Sobreescritura de métodos)**

### **En tiempo de compilación**

La **sobrecarga de métodos** ocurre cuando varios métodos tienen el mismo nombre pero diferentes parámetros. Esto se decide en **tiempo de compilación**.

Ejemplo:

class Calculadora {

// Método para sumar dos números enteros

int sumar(int a, int b) {

return a + b;

}

// Método sobrecargado para sumar tres números enteros

int sumar(int a, int b, int c) {

return a + b + c;

}

// Método sobrecargado para sumar números decimales

double sumar(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

Calculadora calc = new Calculadora();

System.*out*.println(calc.sumar(2, 3)); // 5

System.*out*.println(calc.sumar(2, 3, 4)); // 9

System.*out*.println(calc.sumar(2.5, 3.2)); // 5.7

}

}

**Buenas prácticas:**

✅ Usa la sobrecarga cuando tenga sentido semántico.

✅ Mantén la coherencia en el propósito de los métodos.

❌ No abuses de la sobrecarga con demasiadas variaciones de parámetros.

### **En tiempo de ejecución**

La **sobreescritura de métodos** permite a una subclase proporcionar su propia implementación de un método que ya está definido en su superclase. Se decide en **tiempo de ejecución**.

Ejemplo:

class Perro extends Animal {

@Override

void hacerSonido() {

System.*out*.println("El perro ladra");

}

}

class Gato extends Animal {

@Override

void hacerSonido() {

System.*out*.println("El gato maúlla");

}

}

public class Main {

public static void main(String[] args) {

Animal miAnimal1 = new Perro();

Animal miAnimal2 = new Gato();

miAnimal1.hacerSonido(); // El perro ladra

miAnimal2.hacerSonido(); // El gato maúlla

}

}

**Buenas prácticas:**

✅ Usa @Override para indicar que estás sobrescribiendo un método.

✅ Respeta el principio de sustitución de Liskov (una subclase debe poder reemplazar a su superclase sin alterar el comportamiento del programa).

❌ No uses la sobrescritura sin necesidad.

### **Con interfaces**

El polimorfismo también se logra con **interfaces**, lo que permite a diferentes clases implementar los mismos métodos de distintas maneras.

Ejemplo:

interface Figura {

double calcularArea();

}

class Circulo implements Figura {

private double radio;

Circulo(double radio) {

this.radio = radio;

}

@Override

public double calcularArea() {

return Math.*PI* \* radio \* radio;

}

}

class Rectangulo implements Figura {

private double ancho, alto;

Rectangulo(double ancho, double alto) {

this.ancho = ancho;

this.alto = alto;

}

@Override

public double calcularArea() {

return ancho \* alto;

}

}

public class Main {

public static void main(String[] args) {

Figura f1 = new Circulo(5);

Figura f2 = new Rectangulo(4, 6);

System.*out*.println("Área del círculo: " + f1.calcularArea()); // 78.54

System.*out*.println("Área del rectángulo: " + f2.calcularArea()); // 24.0

}

}

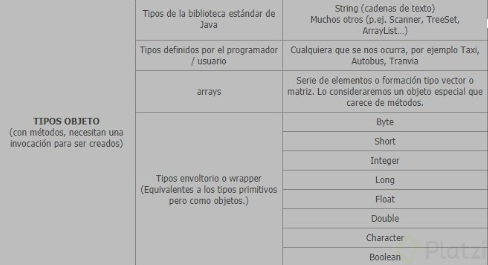
**Buenas prácticas:**✅ Usa interfaces para definir comportamientos comunes.  
✅ Mantén la lógica separada en clases concretas.  
❌ No fuerces el uso de interfaces si no hay múltiples implementaciones previstas.

## 

## ***Abstracción***

## ***Tipos de dato Objeto***

En Java, los tipos de datos de objeto son referencias a instancias de clases. A diferencia de los tipos primitivos, que almacenan directamente valores, los tipos de objeto almacenan referencias a ubicaciones en memoria donde se encuentran los datos reales.



### 

### **Definidas por Java**

Ejemplo: String, Integer, Double, Boolean, etc.

String nombre = "Juan";

Integer edad = 30; // Envuelve un int

Double precio = 99.99; // Envuelve un double

### **Definidas por el usuario**

Instancias de clases creadas por el programador.

class Persona {

String nombre;

int edad;

}

Persona persona = new Persona();

persona.nombre = "Ana";

persona.edad = 25;

### **Arreglos / Arrays / Vectores**

Estructuras que almacenan múltiples valores del mismo tipo.

int[] numeros = {1, 2, 3, 4};

String[] nombres = {"Juan", "Ana", "Luis"};

### 

### **Colecciones (List, Set, Map, etc)**

Parte de la biblioteca java.util. Permiten almacenar y gestionar conjuntos de datos.

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

lista.add("Java");

lista.add("Python");

Map<String, Integer> mapa = new HashMap<>();

mapa.put("Edad", 30);

### 

### **Envoltorios (Wrappers)**

Convierte tipos primitivos en objetos. Ejemplo: Integer, Double, Character.

Integer numero = Integer.*valueOf*(100); // Convierte un int en un objeto Integer

int valor = numero.intValue(); // Devuelve el valor primitivo

Ejemplo práctico

public class TiposObjeto {

public static void main(String[] args) {

// String (clase predefinida)

String saludo = "¡Hola, Mundo!";

System.*out*.println(saludo);

// Clase definida por el usuario

Persona persona = new Persona("Carlos", 28);

System.*out*.println("Nombre: " + persona.nombre + ", Edad: " + persona.edad);

// Arreglo

int[] numeros = {1, 2, 3, 4};

System.*out*.println("Primer número: " + numeros[0]);

// Lista

List<String> frutas = new ArrayList<>();

frutas.add("Manzana");

frutas.add("Banana");

System.*out*.println("Frutas: " + frutas);

// Mapa

Map<String, String> mapa = new HashMap<>();

mapa.put("Clave1", "Valor1");

mapa.put("Clave2", "Valor2");

System.*out*.println("Mapa: " + mapa);

}

}

class Persona {

String nombre;

int edad;

Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

}

# 5 - Estructuras de datos

Las estructuras de datos en Java son fundamentales para organizar y manipular datos de manera eficiente. Java proporciona tanto estructuras básicas (como arrays) como estructuras avanzadas (como las proporcionadas por la librería de colecciones de Java).

## ***Arrays***

Los arreglos o arrays son objetos en los que podemos guardar más de una variable, una lista de elementos. Los arrays son de una sola dimensión, pero si guardamos arrays dentro de otros arrays podemos obtener arrays multidimensionales.

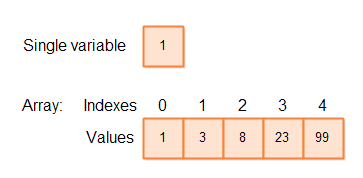
Podríamos decir que son un conjunto de datos que se almacenan en memoria de manera contigua con el mismo nombre, pero con diferentes índices para diferenciar cada uno de ellos.

Los arreglos son estructuras fijas que se declaran y que mantienen su tamaño durante todo su ciclo de vida. Si se declara un arreglo de 5 posiciones, mantendrá esas 5 posiciones siempre.

Cada arreglo que se cree puede ser de un determinado tipo de dato y puede almacenar sólo valores de ese mismo tipo.

Características

* **Definición**: Es una estructura de datos de tamaño fijo que almacena elementos del mismo tipo.
* **Indexación**: Acceso rápido a los elementos mediante índices (comienzan desde 0).
* **Limitaciones**:
  + Su tamaño no puede cambiar después de ser declarado.
  + No tiene métodos integrados para operaciones como agregar o eliminar elementos.



**Declaración**

Ya que los arrays pueden guardar múltiples elementos, la convención es escribir los nombres de las variables en plural.

int[] numeros; // Recomendado

int numeros[]; // También válido

**Creación de un arreglo**

numeros = new int[5]; // Crea un arreglo de tamaño 5, inicializado con ceros.

**Declaracion e inicializacion en una línea**

int[] numeros = {1, 2, 3, 4, 5};

String[] nombres = {"Ana", "Luis", "Pedro"};

**Acceso a sus elementos internos**

1. **Índices**

Los índices son variables simples que nos ayudan a identificar las posiciones en un arreglo. Estas variables siempre guardan números, comienzan en 0 e incrementan de abajo a arriba y de izquierda a derecha a medida que guardamos más elementos en nuestros arrays.

Para guardar un valor en alguna posición de nuestro array solo debemos usar el índice de la siguiente forma:

nombreVariable[indice] = valor;

Para identificar la posición de un arreglo o índice es necesario hacer el cálculo de n-1 ya que los índices de los arreglos inician en 0.

numeros[0] = 10; // Asigna 10 al primer elemento.

System.out.println(numeros[0]); // Muestra: 10

1. **Recorrido**

* **Usando un bucle for:**

for (int i = 0; i < numeros.length; i++) {

System.out.println(numeros[i]);

}

* **Usando un bucle for-each:**

for (int numero : numeros) {

System.out.println(numero);

}

### **Multidimensionales**

Una propiedad de los arrays es que se pueden anidar dependiendo de la necesidad creando más dimensiones.

**Declaración e inicialización**

Un arreglo bidimensional puede considerarse una "matriz":

int[][] matriz = {

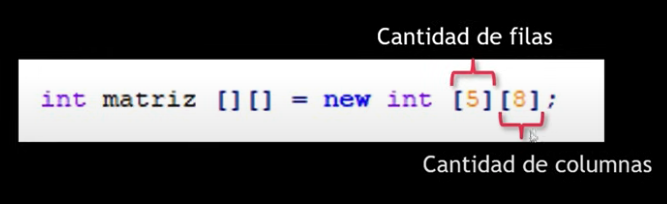
{1, 2, 3},

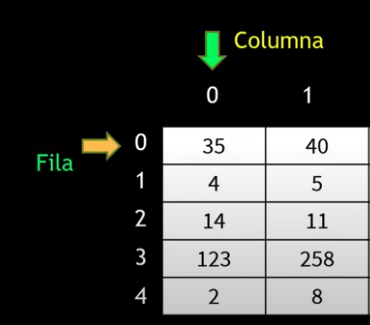
{4, 5, 6},

{7, 8, 9}

};

Como las matrices tienen dos dimensiones, hay que especificar la longitud de ambas. Por ejemplo, supongamos una matriz de 8 columnas y 5 filas.





**Acceso a elementos**

System.out.println(matriz[0][0]); // Primer elemento: 1

**Uso de bucles anidados**

Para poder recorrer los datos de estos arrays multidimensionales se hace uso de bucles anidados.

for (int i = 0; i < matriz.length; i++) {

for (int j = 0; j < matriz[i].length; j++) {

System.out.print(matriz[i][j] + " ");

}

System.out.println();

}

**Ejemplo de uso de array multidimensional:**

//Array multidimensional

String[][] cities = new String[4][2]; // 4\*2 = 8

/\*

\* país | ciudad

\* México | CDMX

\* México | GDL

\* Col | Bogotá

\* Col | Medellín

\*/

cities[0][0] = 'Colombia';

cities[0][1] = 'Medellin';

cities[1][0] = 'Colombia';

cities[1][1] = 'Bogota';

cities[2][0] = 'Mexico';

cities[2][1] = 'GDL';

cities[3][0] = 'Mexico';

cities[3][1] = 'CDMX';

String[][][][] animals = new String[2][3][2][2];

animals[1][0][0][1] = "Monkey";

System.out.println(animals[1][0][0][1]);

**Buenas prácticas usando arrays**

**Usa bucles for-each siempre que sea posible.**

* Más legible y menos propenso a errores:

for (int numero : numeros) {

System.out.println(numero);

}

**Inicializa los arreglos al declararlos si conoces los valores.**

int[] valores = {10, 20, 30};

**Válida siempre los índices.**

* Nunca asumas que un índice existe; podrías obtener una excepción ArrayIndexOutOfBoundsException.

**Usa librerías como Arrays para operaciones comunes.**

* Ordenar:

import java.util.Arrays;

Arrays.sort(numeros); // Ordena el arreglo en orden ascendente.

* Buscar:

int indice = Arrays.binarySearch(numeros, 20);

System.out.println("Índice de 20: " + indice);

**Considera estructuras de datos más avanzadas si el tamaño no es fijo.**

* Usa ArrayList si necesitas una colección de tamaño dinámico.

**Evita usar arreglos bidimensionales si no es necesario.**

* En su lugar, considera listas de listas (List<List<T>>).

**Documenta claramente el propósito del arreglo.**

* Especifica si es para almacenar datos temporales, constantes, etc.

**Si quieres copiar un arreglo se recomienda usar System.arraycoppy**

int[] copia = new int[numeros.length];

System.arraycopy(numeros, 0, copia, 0, numeros.length);

### 

### **Clase Arrays**

La clase Arrays en Java, que forma parte del paquete java.util, proporciona métodos estáticos para manipular arreglos de forma eficiente. Es una herramienta muy útil para tareas como ordenar, buscar, comparar, rellenar y transformar arreglos en cadenas.

**Principales métodos de la clase Arrays:**

#### Conversión a String

Convierte un arreglo en una representación legible en forma de cadena:

int[] numeros = {1, 2, 3, 4, 5};

System.out.println(Arrays.toString(numeros));

// Salida: [1, 2, 3, 4, 5]

#### Ordenar (Sort)

Ordena un arreglo en orden ascendente:

int[] numeros = {5, 3, 8, 1, 2};

Arrays.sort(numeros);

System.out.println(Arrays.toString(numeros));

// Salida: [1, 2, 3, 5, 8]

También soporta rangos:

Arrays.sort(numeros, 1, 4); // Solo ordena de índices 1 a 3.

**Para objetos:** Puedes usar un Comparator para ordenar con criterios personalizados:

String[] nombres = {"Ana", "Luis", "Pedro"};

Arrays.sort(nombres, Comparator.reverseOrder()); // Orden descendente

System.out.println(Arrays.toString(nombres));

// Salida: [Pedro, Luis, Ana]

#### Buscar (binarySearch)

Busca un elemento en un arreglo previamente ordenado. Devuelve el índice o un valor negativo si no se encuentra:

int[] numeros = {1, 2, 3, 4, 5};

int indice = Arrays.*binarySearch*(numeros, 3);

System.out.println(indice); // Salida: 2

#### Comparar Arreglos (equals)

Compara si dos arreglos son iguales (mismo tamaño y elementos en el mismo orden):

int[] a = {1, 2, 3};

int[] b = {1, 2, 3};

System.out.println(Arrays.equals(a, b)); // Salida: true

#### Comparar Arreglos anidados (deepEquals)

Para arreglos multidimensionales:

int[][] a = {{1, 2}, {3, 4}};

int[][] b = {{1, 2}, {3, 4}};

System.out.println(Arrays.deepEquals(a, b)); // Salida: true

#### Rellenar Arreglos (fill)

Rellena un arreglo con un valor específico:

int[] numeros = new int[5];

Arrays.fill(numeros, 7);

System.out.println(Arrays.toString(numeros));

// Salida: [7, 7, 7, 7, 7]

Rango específico:

Arrays.fill(numeros, 1, 4, 3); // Rellena de índice 1 a 3 con el valor 3.

System.out.println(Arrays.toString(numeros));

// Salida: [7, 3, 3, 3, 7]

#### Crear copias (copyOf & copyRange)

Copia Completa:

int[] copia = Arrays.*copyOf*(numeros, numeros.length);

System.out.println(Arrays.toString(copia));

Copia Parcial:

int[] parcial = Arrays.*copyOfRange*(numeros, 1, 4); // Índices 1 a 3

System.out.println(Arrays.toString(parcial));

#### 

#### Generar códigos Hash (hashCode)

Devuelve un código hash para el arreglo:

int[] numeros = {1, 2, 3};

System.out.println(Arrays.hashCode(numeros)); // Salida: Código hash del arreglo

Para arreglos multidimensionales:

int[][] matriz = {{1, 2}, {3, 4}};

System.out.println(Arrays.deepHashCode(matriz));

#### Desordenar y aleatorizar (paralellSort)

**Ordenación en paralelo:** Usa múltiples hilos para ordenar grandes arreglos de manera más eficiente:

int[] numeros = {5, 2, 8, 1, 3};

Arrays.parallelSort(numeros);

System.out.println(Arrays.toString(numeros));

// Salida: [1, 2, 3, 5, 8]

#### Convertir a stream

Convierte un arreglo en un Stream para aprovechar las operaciones funcionales:

int[] numeros = {1, 2, 3, 4, 5};

Arrays.stream(numeros).forEach(System.out::println);

Ejemplo completo:

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

int[] numeros = {5, 3, 8, 1, 2};

// Mostrar arreglo

System.*out*.println("Arreglo original: " + Arrays.*toString*(numeros));

// Ordenar

Arrays.*sort*(numeros);

System.*out*.println("Ordenado: " + Arrays.*toString*(numeros));

// Buscar

int indice = Arrays.*binarySearch*(numeros, 3);

System.*out*.println("Índice de 3: " + indice);

// Rellenar

Arrays.*fill*(numeros, 0, 2, 7);

System.*out*.println("Rellenado: " + Arrays.*toString*(numeros));

// Copiar

int[] copia = Arrays.*copyOf*(numeros, numeros.length);

System.*out*.println("Copia: " + Arrays.*toString*(copia));

// Stream

Arrays.*stream*(copia).map(n -> n \* 2).forEach(System.*out*::println);

}

}

**Buenas prácticas usando la clase Arrays**

**a) Usa Arrays.toString para depuración.**Evita imprimir directamente el arreglo, ya que imprimirá su referencia en memoria.

**b) Recuerda ordenar antes de usar binarySearch.**Si no lo haces, los resultados serán impredecibles.

**c) Prefiere Arrays.copyOf en lugar de copiar manualmente.**Es más limpio y evita errores.

**d) Usa parallelSort para grandes arreglos.**Es más rápido que sort para datos voluminosos.

**e) Para arreglos multidimensionales, usa deepEquals y deepToString.**Esto asegura comparaciones y representaciones correctas.

## ***Factorial***

# 6 - Manejo de excepciones

## ***Excepciones***

En Java, una excepción es un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de las instrucciones. Las excepciones se utilizan para manejar errores y situaciones inesperadas de manera controlada.

**Jerarquía de excepciones**

En Java, todas las excepciones son subclases de la clase Throwable. La jerarquía es la siguiente:

Throwable

├── Error (errores graves, no se deben capturar)

└── Exception

├── RuntimeException (excepciones no verificadas)

└── Checked Exception (excepciones verificadas)

* **Errores (Error)**: Representan problemas graves que no deberían ser manejados por la aplicación (por ejemplo, OutOfMemoryError).
* **Excepciones verificadas (Checked Exception)**: Son excepciones que el compilador obliga a manejar (por ejemplo, IOException, SQLException) controlado por un try - catch -finally.
* **Excepciones no verificadas (RuntimeException)**: No están obligadas a ser manejadas (por ejemplo, NullPointerException, ArithmeticException).

**Manejo de excepciones**

Java proporciona tres bloques clave para manejar excepciones:

1. **try**: Bloque donde se coloca el código que puede lanzar una excepción.
2. **catch**: Bloque que captura y maneja la excepción.
3. **finally**: Bloque opcional que se ejecuta siempre, haya o no excepción.

try {

int resultado = 10 / 0; // Esto lanza una ArithmeticException

} catch (ArithmeticException e) {

System.*out*.println("Error: División por cero.");

} finally {

System.*out*.println("Este bloque siempre se ejecuta.");

}

**Tipo de excepciones**

a) **Excepciones verificadas (Checked Exceptions)**

Las excepciones verificadas son aquellas que el compilador obliga a manejar. Estas excepciones suelen estar relacionadas con errores externos al programa, como operaciones de E/S o problemas de red.

**Características**

* Obligatorias: Debes manejar estas excepciones usando try-catch o declarándolas con throws.
* Comunes en operaciones de E/S: Por ejemplo, IOException, FileNotFoundException, SQLException.

Ejemplo:

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class Main {

public static void main(String[] args) {

try {

FileInputStream file = new FileInputStream("archivo.txt");

} catch (FileNotFoundException e) {

System.*out*.println("Archivo no encontrado: " + e.getMessage());

}

}

}

**¿Cuándo usarlas?**

* Cuando el código puede fallar debido a factores externos (por ejemplo, archivos no encontrados, conexiones de red fallidas).
* Cuándo quieres obligar al llamador a manejar el error.

b) **Excepciones no verificadas (Unchecked Exceptions)**

Son excepciones que no están obligadas a ser manejadas. Suelen ser errores de lógica del programa.

**Características**

* No obligatorias: No es necesario manejarlas, aunque es recomendable hacerlo.
* Comunes en errores de programación: Por ejemplo, NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

Ejemplo:

public class Main {

public static void main(String[] args) {

int[] numeros = {1, 2, 3};

try {

System.*out*.println(numeros[5]); // Esto lanza una ArrayIndexOutOfBoundsException

} catch (ArrayIndexOutOfBoundsException e) {

System.*out*.println("Índice fuera de los límites del arreglo: " + e.getMessage());

}

}

}

**¿Cuándo usarlas?**

* Cuando el error es causado por un problema en la lógica del programa (por ejemplo, división por cero, acceso a un índice inválido).
* Cuando no quieres obligar al llamador a manejar el error.

**Diferencias clave entre Checked y Unchecked Exceptions**

| **Característica** | Checked Exceptions | Unchecked Exceptions |
| --- | --- | --- |
| **Obligatoriedad** | Deben ser manejadas o declaradas. | No es obligatorio manejarlas. |
| **Tipo de errores** | Errores externos (E/S, red, etc.). | Errores de lógica del programa. |
| **Herencia** | Subclases de Exception. | Subclases de RuntimeException. |
| **Ejemplos comunes** | IOException, SQLException. | NullPointerException, ArithmeticException. |
| **Uso recomendado** | Para errores recuperables. | Para errores de programación. |

**c) Excepciones personalizadas**

Puedes crear tus propias excepciones extendiendo la clase Exception o RuntimeException.

Ejemplo:

class MiExcepcion extends Exception {

public MiExcepcion(String mensaje) {

super(mensaje);

}

}

public class Main {

public static void main(String[] args) {

try {

throw new MiExcepcion("Esta es una excepción personalizada.");

} catch (MiExcepcion e) {

System.*out*.println(e.getMessage());

}

}

}

**Buenas prácticas usando excepciones**

**a) Usa excepciones específicas**

Captura excepciones específicas en lugar de la clase genérica Exception. Esto mejora la claridad y el manejo de errores.

Ejemplo:

try{

// Código que puede lanzar excepciones

}catch(FileNotFoundException e){

System.out.println("Archivo no encontrado.");

}catch(IOException e){

System.out.println("Error de E/S.");

}

**b) No ignores las excepciones**

Evita dejar bloques catch vacíos. Si no manejas la excepción, al menos registra el error.

**Mal:**

try {

// Código

} catch (Exception e) {

// Ignorar la excepción

}

**Bien:**

try {

// Código

} catch (Exception e) {

System.err.println("Error: " + e.getMessage());

}

**c) Usa finally para liberar recursos**

El bloque finally es ideal para liberar recursos como archivos, conexiones de red, etc.

Ejemplo:

FileInputStream file = null;

try {

file = new FileInputStream("archivo.txt");

// Procesar archivo

} catch (FileNotFoundException e) {

System.out.println("Archivo no encontrado.");

} finally {

if (file != null) {

try {

file.close();

} catch (IOException e) {

System.out.println("Error al cerrar el archivo.");

}

}

}

**d) Evita el uso excesivo de excepciones**

No uses excepciones para controlar el flujo normal del programa. Las excepciones son costosas en términos de rendimiento.

**Mal:**

try {

for (int i = 0; i < 10; i++) {

if (i == 5) throw new Exception("Número 5 encontrado.");

}

} catch (Exception e) {

System.out.println(e.getMessage());

}

**Bien:**

for (int i = 0; i < 10; i++) {

if (i == 5) {

System.out.println("Número 5 encontrado.");

break;

}

}

**e) Documenta las excepciones**

Si tu método lanza una excepción verificada, documéntala con throws.

**Ejemplo:**

public void leerArchivo(String ruta) throws FileNotFoundException {

FileInputStream file = new FileInputStream(ruta);

// Procesar archivo

}

**Ejemplos prácticos**

**Ejemplo 1:** Manejo de excepciones en una calculadora

public class Calculadora {

public static int dividir(int a, int b) throws ArithmeticException {

if (b == 0) {

throw new ArithmeticException("División por cero.");

}

return a / b;

}

public static void main(String[] args) {

try {

int resultado = *dividir*(10, 0);

System.*out*.println("Resultado: " + resultado);

} catch (ArithmeticException e) {

System.*out*.println("Error: " + e.getMessage());

}

}

}

**Ejemplo 2:** Uso de finally para liberar recursos

public class Archivo {

public static void main(String[] args) {

FileWriter writer = null;

try {

writer = new FileWriter("archivo.txt");

writer.write("Hola, mundo!");

} catch (IOException e) {

System.*out*.println("Error de E/S: " + e.getMessage());

} finally {

if (writer != null) {

try {

writer.close();

} catch (IOException e) {

System.*out*.println("Error al cerrar el archivo.");

}

}

}

}

}

**Preguntas frecuentes**

**a) ¿Cuál es la diferencia entre throw y throws?**

* throw: Se usa para lanzar una excepción explícitamente.
* throws: Se usa en la firma de un método para indicar que puede lanzar una excepción.

**b) ¿Qué es la propagación de excepciones?**

Es el proceso por el cual una excepción se pasa de un método a otro hasta que se maneja o llega al método main.

**c) ¿Qué es un bloque try-with-resources?**

Es una forma simplificada de manejar recursos que implementan AutoCloseable. Los recursos se cierran automáticamente al final del bloque.

Ejemplo:

try (FileInputStream file = new FileInputStream("archivo.txt")) {

// Procesar archivo

} catch (IOException e) {

System.*out*.println("Error de E/S: " + e.getMessage());

}

**d) ¿Puedo convertir una excepción verificada en no verificada?**

Sí, puedes envolver una excepción verificada en una RuntimeException para convertirla en no verificada.

Ejemplo:

try {

FileInputStream file = new FileInputStream("archivo.txt");

} catch (FileNotFoundException e) {

throw new RuntimeException("Archivo no encontrado", e);

}

# 7 - Características avanzadas

## ***Enums***

Un enum (enumeración) en Java es un tipo de dato especial que permite definir un conjunto de constantes nombradas. Los enums son útiles para representar un grupo de valores fijos, como días de la semana, estados, colores, etc.

Los enums son ideales cuando:

* Tienes un conjunto fijo y predefinido de valores constantes.
* Los valores no cambian en tiempo de ejecución.
* Necesitas un tipo de dato seguro y legible.

**Sintaxis básica**

La sintaxis para definir un enum es similar a la de una clase, pero utiliza la palabra clave enum.

Ejemplo básico de enums:

public enum DiaSemana {

*LUNES*, *MARTES*, *MIERCOLES*, *JUEVES*, *VIERNES*, *SABADO*, *DOMINGO*

}

* Cada constante enum es una instancia del tipo enum.
* Por defecto, las constantes enum son public, static y final.

**Características**

**a) Atributos y métodos**

Los enums pueden tener atributos, métodos y constructores, al igual que las clases para los atributos tendremos que crear un método constructor con casa atributo previamente seteado a un lado de su Enum o ordinal.

Ejemplo:

public enum Planetas {

*MERCURIO*(3.303e+23, 2.4397e6),

*VENUS*(4.869e+24, 6.0518e6),

*TIERRA*(5.976e+24, 6.37814e6);

private final double masa; // en kilogramos

private final double radio; // en metros

// Constructor

Planetas(double masa, double radio) {

this.masa = masa;

this.radio = radio;

}

// Métodos

public double getMasa() {

return masa;

}

public double getRadio() {

return radio;

}

}

**b) Métodos abstractos**

Los enums pueden tener métodos abstractos, que deben ser implementados por cada constante.

Ejemplo:

public enum Operacion {

*SUMA* {

public double aplicar(double x, double y) {

return x + y;

}

},

*RESTA* {

public double aplicar(double x, double y) {

return x - y;

}

};

public abstract double aplicar(double x, double y);

}

**c) Implementación de interfaces**

Los enums pueden implementar interfaces, lo que les permite definir comportamientos adicionales.

Ejemplo:

interface Imprimible {

void imprimir();

}

public enum Color implements Imprimible {

*ROJO* {

public void imprimir() {

System.*out*.println("Color rojo");

}

},

*VERDE* {

public void imprimir() {

System.*out*.println("Color verde");

}

};

@Override

public abstract void imprimir();

}

**Metodos utiles**

**a) values()**

Devuelve un array con todas las constantes del enum.

Ejemplo:

DiaSemana[] dias = DiaSemana.values();

for (DiaSemana dia : dias) {

System.*out*.println(dia);

}

**b) valueOf(String name)**

Devuelve la constante enum con el nombre especificado.

Ejemplo:

DiaSemana dia = DiaSemana.valueOf("LUNES");

System.*out*.println(dia); // Imprime: LUNES

**c) ordinal()**

Devuelve la posición de la constante enum en su declaración (comenzando desde 0).

Ejemplo:

System.*out*.println(DiaSemana.LUNES.ordinal()); // Imprime: 0

System.*out*.println(DiaSemana.MARTES.ordinal()); // Imprime: 1

**Clases Vs Enums**

La decisión de usar un enum o una clase en Java depende del contexto y del problema que estés resolviendo. Aquí te doy una guía clara con ejemplos para ayudarte a decidir cuándo usar cada uno.

**¿Cuándo usar un enum?**

Los enums son ideales cuando:

* Tienes un conjunto fijo y predefinido de valores constantes.
* Los valores no cambian en tiempo de ejecución.
* Necesitas un tipo de dato seguro y legible.

Ejemplos de uso de enums:

a) Días de la semana

public enum DiaSemana {

*LUNES*, *MARTES*, *MIERCOLES*, *JUEVES*, *VIERNES*, *SABADO*, *DOMINGO*

}

b) Estados de un pedido

public enum EstadoPedido {

*PENDIENTE*, *EN\_PROCESO*, *ENTREGADO*, *CANCELADO*

}

c) Operaciones matemáticas

public enum Operacion {

*SUMA*, *RESTA*, *MULTIPLICACION*, *DIVISION*

}

d) Colores primarios

public enum Color {

*ROJO*, *VERDE*, *AZUL*

}

**¿Cuándo usar una clase?**

Las clases son más adecuadas cuando:

* Necesitas múltiples instancias con diferentes estados.
* El comportamiento o los atributos pueden variar dinámicamente.
* Requieres herencia, polimorfismo o relaciones complejas entre objetos.

Ejemplos de uso de clases:

a) Representación de un usuario

public class Usuario {

private String nombre;

private int edad;

public Usuario(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

// Getters y setters

}

b) Productos en una tienda

public class Producto {

private String nombre;

private double precio;

public Producto(String nombre, double precio) {

this.nombre = nombre;

this.precio = precio;

}

// Getters y setters

}

c) Vehículos

public class Vehiculo {

private String marca;

private String modelo;

public Vehiculo(String marca, String modelo) {

this.marca = marca;

this.modelo = modelo;

}

// Getters y setters

}

**Comparación directa: clases vs enum**

Ejemplo 1: Representación de colores

Con enum:

public enum Color {

*ROJO*, *VERDE*, *AZUL*

}

Con clases:

public class Color {

public static final Color *ROJO* = new Color();

public static final Color *VERDE* = new Color();

public static final Color *AZUL* = new Color();

private Color() {} // Constructor privado para evitar instancias adicionales

}

¿Por qué usar un enum?

* Es más simple y seguro.
* No permite la creación de instancias adicionales.
* Proporciona métodos útiles como values() y valueOf().

Ejemplo 2: Representación de operaciones matemáticas

Con enum:

public enum Operacion {

*SUMA* {

public double aplicar(double x, double y) {

return x + y;

}

},

*RESTA* {

public double aplicar(double x, double y) {

return x - y;

}

};

public abstract double aplicar(double x, double y);

}

Con clases:

public abstract class Operacion {

public abstract double aplicar(double x, double y);

public static final Operacion *SUMA* = new Operacion() {

@Override

public double aplicar(double x, double y) {

return x + y;

}

};

public static final Operacion *RESTA* = new Operacion() {

@Override

public double aplicar(double x, double y) {

return x - y;

}

};

}

¿Por qué usar un enum?

* Es más conciso y legible.
* No requiere instancias anónimas.
* Proporciona métodos útiles como values() y valueOf().

Ejemplo 3: Representación de estados de un pedido

Con enum:

public enum EstadoPedido {

*PENDIENTE*("El pedido está pendiente de procesar"),

*EN\_PROCESO*("El pedido está siendo procesado"),

*ENTREGADO*("El pedido ha sido entregado");

private final String descripcion;

EstadoPedido(String descripcion) {

this.descripcion = descripcion;

}

public String getDescripcion() {

return descripcion;

}

}

Con clases:

public class EstadoPedido {

private final String descripcion;

private EstadoPedido(String descripcion) {

this.descripcion = descripcion;

}

public static final EstadoPedido *PENDIENTE* = new EstadoPedido("El pedido está pendiente de procesar");

public static final EstadoPedido *EN\_PROCESO* = new EstadoPedido("El pedido está siendo procesado");

public static final EstadoPedido *ENTREGADO* = new EstadoPedido("El pedido ha sido entregado");

public String getDescripcion() {

return descripcion;

}

}

¿Por qué usar un enum?

* Es más seguro y evita la creación de instancias no deseadas.
* Proporciona métodos útiles como values() y valueOf().

**¿Cuándo usar una clase en lugar de un enum?**

**Usa una clase cuando:**

* Necesitas múltiples instancias con diferentes estados.
* Los valores no son fijos y pueden cambiar en tiempo de ejecución.
* Requieres herencia o polimorfismo.

Ejemplo: Representación de usuarios

public class Usuario {

private String nombre;

private int edad;

public Usuario(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

// Getters y setters

}

En este caso, un enum no sería adecuado porque cada usuario tiene un estado único (nombre y edad).

**Usa enums cuando:**

* Tienes un conjunto fijo y predefinido de valores constantes.
* Los valores no cambian en tiempo de ejecución.
* Necesitas un tipo de dato seguro y legible.

**Buenas prácticas usando Enums**

**a) Usa enums para conjuntos de valores fijos**

Los enums son ideales para representar conjuntos de valores constantes y predefinidos, como días de la semana, estados, colores, etc.

**b) Evita el uso de ordinales**

El método ordinal() depende del orden de declaración de las constantes, lo que puede ser propenso a errores. En su lugar, usa atributos personalizados.

Mal:

if (dia.ordinal() == 0) {

System.*out*.println("Es lunes");

}

Bien:

if (dia == DiaSemana.LUNES) {

System.*out*.println("Es lunes");

}

**c) Agrega métodos y atributos para enriquecer los enums**

Los enums pueden tener atributos y métodos, lo que los hace más poderosos y flexibles.

**d) Usa enums en lugar de constantes**

En lugar de definir constantes en una clase, usa enums para agrupar valores relacionados.

Mal:

public class Constantes {

public static final int *LUNES* = 0;

public static final int *MARTES* = 1;

}

Bien:

public enum DiaSemana {

*LUNES*, *MARTES*, *MIERCOLES*, *JUEVES*, *VIERNES*, *SABADO*, *DOMINGO*

}

**Ejemplos prácticos**

**Ejemplo 1:** Enum con atributos y métodos

public enum EstadoPedido {

*PENDIENTE*("El pedido está pendiente de procesar"),

*EN\_PROCESO*("El pedido está siendo procesado"),

*ENTREGADO*("El pedido ha sido entregado");

private final String descripcion;

EstadoPedido(String descripcion) {

this.descripcion = descripcion;

}

public String getDescripcion() {

return descripcion;

}

}

**Ejemplo 2:** Enum con métodos abstractos

public enum Operacion {

*SUMA* {

public double aplicar(double x, double y) {

return x + y;

}

},

*RESTA* {

public double aplicar(double x, double y) {

return x - y;

}

};

public abstract double aplicar(double x, double y);

}

**Ejemplo 3:** Enum que implementa una interfaz

interface Imprimible {

void imprimir();

}

public enum Color implements Imprimible {

*ROJO* {

public void imprimir() {

System.*out*.println("Color rojo");

}

},

*VERDE* {

public void imprimir() {

System.*out*.println("Color verde");

}

};

@Override

public abstract void imprimir();

}

**Preguntas frecuentes**

**a) ¿Puedo extender un enum?**

No, los enums en Java no pueden extenderse. Son implícitamente final.

**b) ¿Puedo agregar nuevas constantes a un enum en tiempo de ejecución?**

No, las constantes enum son fijas y no pueden modificarse en tiempo de ejecución.

**c) ¿Cuándo debo usar un enum en lugar de una clase?**

Usa un enum cuando tengas un conjunto fijo y predefinido de valores constantes. Para casos más dinámicos, usa clases.

# 8 - Patrones de diseño

# 9 - GUI con Java Swing

## ***Diagramación UML***

## ***Java Docs***

Los Javadocs son una herramienta usada por muchas otras herramientas y aplicaciones porque nos ayuda a documentar todo nuestro código usando comentarios. Además, nos permite visualizar la documentación en formato HTML.

Tipos de comentarios

// Comentarios de una sola línea

/\* Comentario

\* en múltiples

\* líneas \*/

/\*\*

\* Comentario para Java Docs

\* \*/

Vamos a documentar la función convertToDolar. Recuerda que esta función devuelve un número double y recibe dos argumentos: quantity (de tipo double) y currency (de tipo String):

/\*\*

\* Descripción: función que permite convertir divisas a dólares

\*

\* @param quantity cantidad de dinero

\* @param currency tipo de moneda

\*/

public static double converToDolar(double quantity, String currency) {

switch (currency) {

case "MXN":

quantity \*= 0.052;

break;

case "COP":

quantity \*= 0.0031;

break;

default:

break;

}

return quantity;

}



## 

## ***Colecciones de datos***

### ArrayList

El arraylist es un tipo de array que no se necesita especificar su longitud pues esta es dinámica y se irá expandiendo o reduciendo la necesidad de la app.

Hay que tener en cuenta el rendimiento, los array son asignados a una zona de memoria de acceso más rápido ya que el tamaño es fijo, mientras que el arraylist es enviado a una zona de acceso más lento por su naturaleza dinámica. Esa es la razón de ser de uno y otro, de no ser así, no tendría sentido mantener los dos. Si bien suele usar siempre los arraylist, no está de más tener en cuenta esta diferencia para los array. En un programa de escritorio no es problema la diferencia de rendimiento pero en un programa de servidor donde tenga que manejar millones de transacciones, se nota mucho.

Características

* **Definición**: Una clase que implementa la interfaz List. Es un array dinámico.
* **Crecimiento dinámico**: Su tamaño se ajusta automáticamente cuando se agregan o eliminan elementos.
* **Indexación**: Similar a los arrays, permite acceso por índice.
* **Orden**: Mantiene el orden de inserción de los elementos.



### HashMap

Características

* **Definición**: Una estructura de datos que almacena pares clave-valor.
* **Claves únicas**: No se permiten claves duplicadas; los valores pueden repetirse.
* **Orden**: No garantiza un orden específico de los elementos.
* **Velocidad**: Operaciones de búsqueda, inserción y eliminación rápidas (O(1) en promedio).



Buenas Prácticas

1. **Usar claves únicas**: Asegúrate de que las claves sean únicas para evitar sobreescrituras.
2. **Evitar modificaciones concurrentes**: Usa ConcurrentHashMap si necesitas manejar accesos concurrentes.
3. **Definir tipos genéricos**: Especifica el tipo de clave y valor.



1. **Manejar valores nulos**: Verifica si las claves o valores pueden ser nulos para evitar errores.



### LinkedList

Características

* **Definición**: Una lista doblemente enlazada que implementa la interfaz List.
* **Inserciones/Eliminaciones eficientes**: Ideal para operaciones frecuentes de agregar/eliminar elementos (O(1) en extremos).
* **Orden**: Mantiene el orden de inserción.
* **Velocidad**: Acceso más lento por índice (O(n)) comparado con ArrayList.



Buenas Prácticas

1. **Usar para operaciones frecuentes en extremos**: Ideal si necesitas agregar o eliminar elementos al principio o final de la lista.
2. **Evitar accesos por índice**: Usa ArrayList si necesitas acceder frecuentemente a elementos mediante índices.
3. **Cuidear el rendimiento en listas grandes**: El rendimiento disminuye con listas muy grandes debido al acceso secuencial.

**Comparación Rápida**

| **Característica** | **Array** | **ArrayList** | **HashMap** | **LinkedList** |
| --- | --- | --- | --- | --- |
| **Tamaño** | Fijo | Dinámico | Dinámico (clave-valor) | Dinámico |
| **Orden** | Mantiene | Mantiene | No garantizado | Mantiene |
| **Acceso por índice** | Muy rápido (O(1)) | Rápido (O(1)) | No aplica | Lento (O(n)) |
| **Inserción/Eliminación** | Lento (reestructuración) | Lento (reestructuración) | Rápido (O(1) promedio) | Rápido en extremos (O(1)) |
| **Duplicados** | Sí | Sí | Claves únicas | Sí |

# 

# 

# 

# 

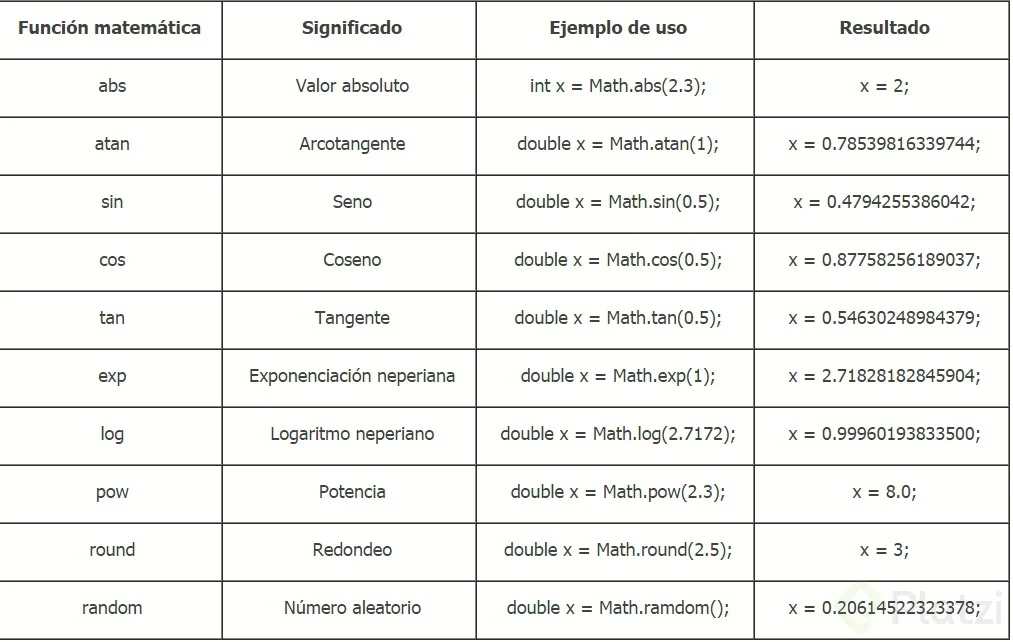
# 

# 

## ***Clases nativas útiles de Java***

### **Math**

Math es un api o conjunto de funciones o constantes nativas de Java que nos ofrece una gran cantidad de ecuaciones matemáticas., a continuación se muestran unas cuantas funcionalidades básicas del objeto math.



### **Scanner**

La clase Scanner permite a los usuarios contestar algunas preguntas para que nuestro programa actúe de una forma u otra. Para usarla solo debemos importar la clase Scanner de las APIs de desarrollo de Java:

import java.util.Scanner;

int response = 0;

Scanner sc = new Scanner(System.in);

response = Integer.valueOf(sc.nextLine());

# Patrones de diseño de software

## ***Patrones creacionales***

### **Factory Simple**

[Patrones de Diseño con Java. Curso completo: Capítulo 1: Factory / Factory Method](https://www.youtube.com/watch?v=D_YIHqw-p7s&list=PL4bT56Uw3S4x1_IhP0kQZ1mszuLvoNMJF&index=3)

El patrón Factory es un patrón creacional que proporciona una interfaz para crear objetos en una superclase, pero permite a las subclases alterar el tipo de objetos que se crearán.

**¿Cuándo usarlo?**

* Cuando tienes una jerarquía de clases y quieres centralizar la creación de objetos en un solo lugar.
* Cuando la lógica de creación de objetos es compleja o repetitiva.
* Cuándo quieres desacoplar la creación de objetos del código que los usa.
* Cuando el número de clases a generar son pocas ya que el uso de if ensucia el código.

Ejemplo:

public interface Enemy {

void attack();

}

public class Warrior implements Enemy {

@Override

public void attack() {

System.*out*.println("warrior attack");

}

}

public class Mage implements Enemy {

@Override

public void attack() {

System.*out*.println("Mage attack");

}

}

public class EnemyFactory {

public static Enemy createEnemy(String type) {

if (type.equalsIgnoreCase("warrior")) {

return new Warrior();

} else if (type.equalsIgnoreCase("mage")) {

return new Mage();

}

throw new IllegalArgumentException("Unknown product type");

}

}

public class Main {

public static void main(String[] args) {

EnemyFactory enemyFactory = new EnemyFactory();

Enemy mage = enemyFactory.*createEnemy*("mage");

mage.attack();

// o abreviado

Enemy warrior = EnemyFactory.*createEnemy*("warrior");

warrior.attack();

}

}

**Buenas Prácticas**

* Usa nombres descriptivos para los métodos de la fábrica.
* Evita lógica compleja dentro de la fábrica; si es necesario, dividirla en métodos más pequeños.
* Considera usar enums para los tipos de productos si son fijos.

### **Factory Method**

[Patrones de Diseño con Java. Curso completo: Capítulo 1: Factory / Factory Method](https://www.youtube.com/watch?v=D_YIHqw-p7s&list=PL4bT56Uw3S4x1_IhP0kQZ1mszuLvoNMJF&index=3)

El patrón Factory Method es un patrón creacional que define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar. Permite a una clase delegar la creación de objetos a sus subclases.

**¿Cuándo usarlo?**

* Cuando no sabes de antemano qué clases exactas necesitarás.
* Cuándo quieres extender la lógica de creación de objetos sin modificar el código existente.
* Cuando necesitas flexibilidad para agregar nuevos tipos de productos.
* Cuando el número de clases a instancias son muy numerosas ya que aumentaría el condicional (if-switch) provocando que el código se vea mal y aparte ahorrando pasar los parámetros de constructor.

Ejemplo:

public interface Enemy {

void attack();

}

public abstract class EnemyFactory {

public abstract Enemy createEnemy();

}

public class WarriorFactory extends EnemyFactory {

@Override

public Enemy createEnemy() {

return new Warrior();

}

}

public class MageFactory extends EnemyFactory {

@Override

public Enemy createEnemy() {

return new Mage();

}

}

public class Warrior implements Enemy {

@Override

public void attack() {

System.*out*.println("warrior attack");

}

}

public class Mage implements Enemy {

@Override

public void attack() {

System.*out*.println("mage attack");

}

}

public class Main {

public static void main(String[] args) {

Enemy warrior = new WarriorFactory().createEnemy();

Enemy mage = new MageFactory().createEnemy();

warrior.attack();

mage.attack();

}

}

**Buenas Prácticas**

* Usa este patrón cuando la creación de objetos varía según el contexto.
* Mantén la lógica de creación en las subclases para facilitar la extensión.
* Evita crear demasiadas subclases; si la lógica es simple, considera usar Factory en su lugar.

**Cuándo usar Factory vs Factory Method?**

* Usa Factory cuando:
  + La lógica de creación es simple y no cambia mucho.
  + Quieres centralizar la creación de objetos en un solo lugar.
* Usa Factory Method cuando:
  + La lógica de creación es compleja o variable.
  + Necesitas extender el sistema con nuevos tipos de productos sin modificar el código existente.

**Ejemplo de Uso en un Proyecto Real**

Factory:

* Crear diferentes tipos de conexiones a bases de datos (MySQL, PostgreSQL, etc.).
* Crear diferentes tipos de notificaciones (email, SMS, push).

Factory Method:

* Crear diferentes tipos de documentos (PDF, Excel, Word) en una aplicación de generación de informes.
* Crear diferentes tipos de vehículos (Coche, Moto, Camión) en un sistema de transporte.

# Modelo de capas

El modelo de capas es una arquitectura de software ampliamente utilizada en el desarrollo de aplicaciones de escritorio en Java. Este modelo organiza el código en capas separadas, cada una con una responsabilidad específica, lo que facilita el mantenimiento, la escalabilidad y la reutilización del código. A continuación, te explico todo lo que necesitas saber sobre el modelo de capas en Java, incluyendo buenas prácticas y ejemplos.

El modelo de capas divide una aplicación en capas lógicas, donde cada capa tiene una responsabilidad específica y se comunica con las demás de manera estructurada. Las capas más comunes son:

1. ***Capa de Presentación (UI):*** Responsable de la interacción con el usuario.
2. ***Capa de Negocio (Lógica de Negocio):*** Contiene la lógica de la aplicación.
3. ***Capa de Acceso a Datos (DAL):*** Gestiona el acceso a la base de datos o fuentes de datos externas.

**Beneficios**

* ***Separación de responsabilidades:*** Cada capa tiene una tarea específica, lo que facilita el desarrollo y el mantenimiento.
* ***Reutilización de código:*** La lógica de negocio y el acceso a datos pueden reutilizarse en diferentes partes de la aplicación.
* ***Escalabilidad:*** Es más fácil agregar nuevas funcionalidades o modificar las existentes.
* ***Facilidad de pruebas:*** Cada capa puede probarse de manera independiente.

**Capas en detalle**

**a) Capa de Presentación (UI)**

* ***Responsabilidad:*** Mostrar información al usuario y capturar sus interacciones.
* ***Tecnologías comunes:***Java Swing, JavaFX, AWT.
* Buenas prácticas:
  + Mantener la lógica de la interfaz de usuario separada de la lógica de negocio.
  + Usar patrones como MVC (Modelo-Vista-Controlador) para organizar la interfaz.

Ejemplo:

// Ejemplo de una interfaz de usuario simple con JavaFX

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class App extends Application {

@Override

public void start(Stage primaryStage) {

Button btn = new Button("Hola, Mundo!");

btn.setOnAction(e -> System.*out*.println("Botón presionado"));

StackPane root = new StackPane();

root.getChildren().add(btn);

Scene scene = new Scene(root, 300, 200);

primaryStage.setTitle("Mi Aplicación");

primaryStage.setScene(scene);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

**b) Capa de Negocio (Lógica de Negocio)**

* ***Responsabilidad:***Contiene las reglas y la lógica de la aplicación.
* Buenas prácticas:
  + Mantener la lógica de negocio independiente de la interfaz de usuario y el acceso a datos.
  + Usar patrones como Service o Manager para encapsular la lógica.

Ejemplo:

// Ejemplo de una clase de servicio en la capa de negocio

public class UsuarioService {

public boolean validarUsuario(String usuario, String contraseña) {

// Lógica de validación

return usuario.equals("admin") && contraseña.equals("1234");

}

}

**c) Capa de Acceso a Datos (DAL)**

* ***Responsabilidad:*** Gestionar el acceso a la base de datos o fuentes de datos externas.
* ***Tecnologías comunes:*** JDBC, Hibernate, JPA.
* Buenas prácticas:
  + Usar un patrón como DAO (Data Access Object) para encapsular el acceso a datos.
  + Evitar incluir lógica de negocio en esta capa.

Ejemplo:

// Ejemplo de un DAO para acceder a datos de usuarios

import java.sql.\*;

public class UsuarioDAO {

private Connection conexion;

public UsuarioDAO(Connection conexion) {

this.conexion = conexion;

}

public boolean validarUsuario(String usuario, String contraseña) {

String sql = "SELECT \* FROM usuarios WHERE usuario = ? AND contraseña = ?";

try (PreparedStatement stmt = conexion.prepareStatement(sql)) {

stmt.setString(1, usuario);

stmt.setString(2, contraseña);

ResultSet rs = stmt.executeQuery();

return rs.next();

} catch (SQLException e) {

e.printStackTrace();

return false;

}

}

}

**Comunicación entre capas**

Las capas se comunican de la siguiente manera:

1. La **capa de presentación** llama a la **capa de negocio** para realizar operaciones.
2. La **capa de negocio** llama a la **capa de acceso a datos** para obtener o almacenar información.

Ejemplo:

// Ejemplo de comunicación entre capas

public class ControladorUsuario {

private UsuarioService usuarioService;

public ControladorUsuario(UsuarioService usuarioService) {

this.usuarioService = usuarioService;

}

public boolean login(String usuario, String contraseña) {

return usuarioService.validarUsuario(usuario, contraseña);

}

}

**Buenas practicas con modelo de capas**

**a) Separación de responsabilidades**

* Cada capa debe tener una responsabilidad clara y no debe realizar tareas de otras capas.

**b) Inyección de dependencias**

* Usa inyección de dependencias para desacoplar las capas y facilitar las pruebas.

**c) Manejo de excepciones**

* Maneja las excepciones adecuadamente en cada capa. Por ejemplo, las excepciones de la capa de acceso a datos deben ser capturadas y manejadas en la capa de negocio.

**d) Pruebas unitarias**

* Prueba cada capa de manera independiente. Usa frameworks como JUnit y Mockito.

**e) Documentación**

* Documenta cada capa y sus componentes para facilitar el mantenimiento.

Ejemplo completo:

1. **Capa de preesentaciòn GUI**

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.TextField;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

public class App extends Application {

private ControladorUsuario controlador;

@Override

public void start(Stage primaryStage) {

controlador = new ControladorUsuario(new UsuarioService(new UsuarioDAO(getConexion())));

TextField txtUsuario = new TextField();

TextField txtContraseña = new TextField();

Button btnLogin = new Button("Login");

Label lblResultado = new Label();

btnLogin.setOnAction(e -> {

boolean resultado = controlador.login(txtUsuario.getText(), txtContraseña.getText());

lblResultado.setText(resultado ? "Login exitoso" : "Login fallido");

});

VBox root = new VBox(10, txtUsuario, txtContraseña, btnLogin, lblResultado);

Scene scene = new Scene(root, 300, 200);

primaryStage.setTitle("Login");

primaryStage.setScene(scene);

primaryStage.show();

}

private Connection getConexion() {

// Implementa la conexión a la base de datos

return null;

}

public static void main(String[] args) {

launch(args);

}

}

1. **Capa de negocio**

public class UsuarioService {

private UsuarioDAO usuarioDAO;

public UsuarioService(UsuarioDAO usuarioDAO) {

this.usuarioDAO = usuarioDAO;

}

public boolean validarUsuario(String usuario, String contraseña) {

return usuarioDAO.validarUsuario(usuario, contraseña);

}

}

1. **Capa de Acceso a datos**

import java.sql.\*;

public class UsuarioDAO {

private Connection conexion;

public UsuarioDAO(Connection conexion) {

this.conexion = conexion;

}

public boolean validarUsuario(String usuario, String contraseña) {

String sql = "SELECT \* FROM usuarios WHERE usuario = ? AND contraseña = ?";

try (PreparedStatement stmt = conexion.prepareStatement(sql)) {

stmt.setString(1, usuario);

stmt.setString(2, contraseña);

ResultSet rs = stmt.executeQuery();

return rs.next();

} catch (SQLException e) {

e.printStackTrace();

return false;

}

}

}

# Java Swing

[**Introducción a Java SE**](#_kepkkmoprkh1) **1**

[¿Qué es Java?](#_szx3qzdcahos) 1

[Versiones](#_61fbg23wu4d7) 1

[Arquitectura](#_ayr08tioknlh) 1

[¿Por qué Java es multiplataforma?](#_msjrruo3qvuw) 2

[Cambiar de versiones Java](#_18brxd6ox4zq) 3

[JShell](#_58dfa7np7bq9) 4

[Archivos JAR](#_m5h9lxr2jdb2) 4

[**Sintaxis básica**](#_803jtr7c692g) **5**

[Variables](#_2d480y76tpwz) 5

[Variables locales](#_vf95au3ef4uj) 5

[Variables de instancia](#_szahrvmusby4) 5

[Variables estáticas](#_q5b43idyi6zi) 5

[Constantes](#_rum0vgr34n1b) 7

[Constantes primitivas](#_rwp9nekhnjin) 8

[Constantes de objetos](#_jfwvl2yvvw51) 8

[Constantes estáticas](#_bsbne1j6p7u6) 8

[Tipos de datos primitivos](#_t2hnrkrgrl0h) 9

[Enteros](#_d5sqhi862lk4) 10

[Decimales](#_cmy1w9h0zrc3) 10

[Carácter](#_npz8apda81zp) 11

[Lógicos](#_if22zn7bthox) 11

[Técnicas de Naming](#_pta4znjm1zfn) 12

[Upper Camel Case](#_jvvecawv8vdq) 12

[Lower Camel Case](#_scjsbsqzyp18) 12

[Casteo de variables](#_1nqomfkzdswt) 13

[Operadores](#_9plvw4de89af) 16

[Aritméticos](#_767m14r4oli2) 16

[Lógicos](#_q01jom2ustr6) 16

[Asignación](#_ba7rxjqzsmt6) 17

[Unarios](#_w0zjkmmtij8v) 18

[Relacionales](#_80amg5fqz82v) 18

[Ternarios](#_2ws6w2ou0u4d) 19

[Especiales](#_7tquber1isri) 19

[Condicionales](#_auuh7le1jie0) 19

[If / else](#_mmtasaqv7m5w) 19

[Ternario](#_rj2f8knv3fzg) 22

[Switch](#_r5bn6zzdnn1o) 24

[Bucles](#_zcervx4xwbb3) 26

[While](#_sxy3gfavlar2) 27

[do-while](#_z1m3f47yecyo) 27

[For](#_k55vtrtqgx7i) 29

[Controladores de flujo](#_zbl6iegyxhhb) 31

[Break](#_lq0l34iy56hu) 31

[Continue](#_g92194731bxd) 32

[Return](#_rljxrmyst61e) 32

[Tags](#_ho4ykvnf7c5e) 33

[**Java SE con POO**](#_yhqp3fg1m6qc) **35**

[Clases](#_h0vnhqsvds26) 37

[Objetos](#_oekyfx1c4fr) 38

[Gestión de memoria](#_mluvzn8t0nlv) 40

[Memoria Stack](#_nowlu89nobnn) 40

[Memoria Heap](#_kd5b1gtd9rnj) 40

[Métodos](#_jjdx6990he14) 45

[Paso por valor y referencia](#_jyfvyb3vo2s8) 48

[Encapsulamiento](#_7ky46gqatu) 52

[Modificadores de acceso](#_l1lxmmsojk8u) 52

[Getters y Setters](#_87ggbuz22aqa) 55

[JavaBeans](#_qkdlrgypcfuc) 57

[Composicion](#_e1gfq6jn9nf8) 59

[Ámbito o scope de las variables](#_4gr20c7wwy0n) 59

[Tipos de dato Objeto](#_qbn065phpxud) 61

[Definidas por Java](#_8opjk5ve8h55) 62

[Definidas por el usuario](#_ve4du5m2121k) 62

[Arreglos / Arrays / Vectores](#_dkffbhf07ge6) 62

[Colecciones (List, Set, Map, etc)](#_x78t4n3odmv8) 63

[Envoltorios (Wrappers)](#_bsx3fuwbvwdo) 63

[Diagramación UML](#_8y862acpmssj) 64

[**Estructuras de datos**](#_cl4g5k8n2w7w) **64**

[Arrays](#_xycq8cx7m6ym) 64

[Multidimensionales](#_3i4fy2gqw9d8) 66

[Clase Arrays](#_gx5erg426s3f) 69

[Conversión a String](#_w7zju2m74mc0) 69

[Ordenar (Sort)](#_xsmn619ssz5p) 69

[Buscar (binarySearch)](#_cmkt8npq9iim) 70

[Comparar Arreglos (equals)](#_vfhmhywfzi2y) 70

[Comparar Arreglos anidados (deepEquals)](#_l3scc2qk8xl2) 70

[Rellenar Arreglos (fill)](#_t5z2v47debzu) 70

[Crear copias (copyOf & copyRange)](#_on79zjlp0e2c) 71

[Generar códigos Hash (hashCode)](#_x0qelr1t477u) 71

[Desordenar y aleatorizar (paralellSort)](#_qbaegsx4scjj) 71

[Convertir a stream](#_5b2c11k7rzk9) 71

[**Excepciones**](#_lmv0u3gkif25) **73**

[**Enums**](#_gjburxgk6dpp) **79**

[Java Docs](#_u2bl6x17ksq3) 89

[Colecciones de datos](#_xuewnrc5nlsg) 91

[ArrayList](#_liip5420wglf) 91

[HashMap](#_6obt442vo30a) 92

[LinkedList](#_xn24oq8103a) 93

[Clases nativas útiles de Java](#_l4mzzjhntdla) 95

[Math](#_g7kpjdgttcel) 95

[Scanner](#_4kc7sdvc1s16) 95

[**Patrones de diseño de software**](#_8j01pulwfrh6) **96**

[Patrones creacionales](#_xh0fzr1hvxh8) 96

[Factory Simple](#_zhz2olflbqq9) 96

[Factory Method](#_o3g6kdbkt9yx) 97

[**Modelo de capas**](#_bhyjpnhe59fr) **99**

[**Java Swing**](#_ehd9kwupougl) **104**