

# JAVA PARA TONTOS

Escrito por: Ale M.

Iniciado: Septiembre 2025

# Table of Contents

INTRODUCCIÓN.....	6
.....	6
TEMA 1. JAVA: DE LA HISTORIA A LA PRACTICA, TU PRIMER PASO EN LA PROGRAMACIÓN.....	7
1.1 Java y su relevancia en la actualidad.....	7
1.2 Historia de Java.....	8
1.3 Como funciona Java.....	8
TEMA 2. CONSTRUYENDO CON BLOQUES: VARIABLES Y TIPOS DE DATOS EN JAVA, LA BASE DE TODA PROGRAMACIÓN.....	10
2.1 Variables en Java.....	10
2.2 Tipos de datos en Java.....	10
2.3 Buenas prácticas en el uso de variables.....	13
2.4 Alcance de una variable.....	14
2.5 Conversión de variables.....	15
EJERCICIOS TEMA 2.....	17
TEMA 3. JAVA EN ACCIÓN: EL PODER DE LOS OPERADORES.....	21
3.1 Introducción a los operadores.....	21
3.2 Clasificación de los operadores.....	21
3.3 Precedencia y asociatividad de los operadores.....	23
EJERCICIOS TEMA 3.....	24
TEMA 4. DECIDIENDO EL CAMINO: DOMINA LAS CONDICIONALES EN JAVA.....	26
4.1 Definición de condicional.....	26
4.2 Condicional if.....	26
4.3 Condicional if-else.....	27
4.4 Condicional else if.....	28
4.5 Condicional switch.....	29
EJERCICIOS TEMA 4.....	31
TEMA 5. CICLOS QUE DAN VIDA AL CÓDIGO: EXPLORANDO LOS BUCLES EN JAVA...34	34
5.1 Concepto de bucle.....	34
5.2 Bucle for.....	34
5.3 Bucle for-each.....	35
5.4 Bucle while.....	36
5.5 Bucle do-while.....	37
EJERCICIOS TEMA 5.....	38
TEMA 6. ARREGLOS EN JAVA: LA BASE DEL ORDEN Y LA EFICIENCIA EN LA PROGRAMACIÓN.....	40
6.1 Definición de arreglo.....	40
6.2 Arreglos unidimensionales.....	40
6.3 Arreglos bidimensionales.....	41
6.4 Arreglos multidimensionales.....	42
EJERCICIOS TEMA 6.....	43
TEMA 7. MÉTODOS EN JAVA: LA ARQUITECTURA INVISIBLE QUE DA VIDA AL CÓDIGO.....	46
7.1 Definición de método.....	46
7.2 Tipos de métodos.....	46
7.2.1 Métodos de instancia.....	46
7.2.2 Métodos estáticos.....	47
7.2.3 Métodos sobrecargados.....	47
TEMA 8. JAVA Y EL MUNDO DE LOS OBJETOS: FUNDAMENTOS DE LA POO (PARTE 1) 48	48
8.1 Concepto de Programación Orientada a Objetos.....	48
8.2 Clases y objetos.....	48

8.2.1 Clase.....	48
8.2.2 Objeto.....	49
8.3 Modificadores de acceso.....	49
8.4 Static vs. Final.....	51
8.4.1 Palabra clave static.....	51
8.4.2 Palabra clave final.....	52
8.5 Clases anidadas.....	52
<b>TEMA 9. JAVA Y EL MUNDO DE LOS OBJETOS: FUNDAMENTOS DE LA POO (PARTE 2)</b>	<b>55</b>
9.1 Ciclo de vida de un objeto.....	55
9.2 Herencia.....	56
9.3 Abstracción.....	57
9.4 Encapsulamiento.....	58
9.5 Interfaces.....	59
9.6 Polimorfismo.....	60
9.6.1 Sobre carga de métodos.....	60
9.6.2 Sobreescritura de métodos.....	60
9.7 Enlace estático y dinámico.....	61
9.8 Paso por valor y por referencia en POO.....	62
9.9 Encadenamiento de métodos.....	63
9.10 Enums, records e Initializer Blocks.....	64
9.10.1 Enums.....	64
9.10.2 Records.....	65
9.10.3 Initializer Blocks.....	65
EJERCICIOS.....	66
<b>TEMA 10. DOMANDO ERRORES: ESTRATEGIAS Y BUENAS PRÁCTICAS EN EL MANEJO DE EXCEPCIONES EN JAVA</b>	<b>69</b>
10.1 Manejo de excepciones.....	69
10.2 Jerarquía de excepciones.....	69
10.3 Uso de try, catch y finally.....	70
10.4 Declaración y lanzamiento de excepciones con throw y throws.....	71
10.5 Creación de excepciones personalizadas.....	71
10.6 Manejo de múltiples excepciones y uso de multi-catch.....	72
10.7 Registro de errores (logging) y depuración.....	73
EJERCICIOS TEMA 10.....	73
<b>TEMA 11. JAVA EN UNA SOLA LINEA: EL PODER OCULTO DE LAS EXPRESIONES LAMBDA</b>	<b>76</b>
11.1 De la programación imperativa a la funcional en Java.....	76
11.2 Sintaxis y fundamentos de las expresiones lambda.....	77
11.3 Interfaces funcionales.....	77
11.4 Aplicaciones prácticas de las lambdas.....	78
EJERCICIOS TEMA 11.....	79
<b>TEMA 12. JAVA AL MÁXIMO: DOMINA ANOTACIONES, MÓDULOS Y OPCIONALES</b>	<b>82</b>
12.1 Definición y concepto de Anotaciones.....	82
12.1.1 Anotaciones predefinidas.....	83
12.1.2 Anotaciones personalizadas.....	83
12.1.3 Elementos importantes de las anotaciones.....	83
12.2 Módulos.....	83
12.2.1 Estructura básica de un modulo.....	84
12.2.2 Declaración de dependencias entre módulos.....	84
12.2.3 Exportación de paquetes y encapsulamiento.....	85
12.3 Uso de Optional.....	85
12.4 Métodos principales de Optional.....	86

EJERCICIOS TEMA 12.....	86
TEMA 13. CONECTANDO PIEZAS: LA MAGIA DE LA INYECCIÓN DE DEPENDENCIAS EN JAVA.....	89
13.1 Introducción al concepto de dependencias en programación.....	89
13.2 Acoplamiento y desacoplamiento.....	89
13.3 Principios SOLID relacionados con la Inyección de Dependencias.....	90
13.4 Concepto de Inyección de dependencias (DI).....	90
13.5 Tipos de inyección de Dependencias.....	91
TEMA 14. ENTRE BYTES Y ARCHIVOS: DOMINANDO LA ENTRADA Y SALIDA EN JAVA .....	93
14.1 Introducción al concepto de Entrada y Salida (I/O).....	93
14.2 Concepto de flujo (stream).....	93
14.2.1 Flujos de bytes (Byte Streams).....	94
14.2.2 Flujos de caracteres (Character Streams).....	94
14.3 Clases principales del paquete java.io.....	94
14.3.1 InputStream y OutputStream.....	94
14.3.2 Reader y Writer.....	94
14.4 Clases derivadas.....	95
14.5 Buffering y rendimiento.....	95
14.6 Manejo de excepciones en I/O.....	96
14.7 Introducción a Java NIO.....	96
14.8 Conceptos fundamentales de NIO.....	96
14.8.1 Buffers.....	96
14.8.2 Canales.....	97
14.9 Clases clave de java.nio.file.....	98
14.9.1 Clase Paths.....	98
14.9.2 Clase Files.....	98
14.10 Operaciones avanzadas con NIO.....	99
14.10.1 Transferencia directa entre canales.....	99
14.10.2 Mapear archivos en memoria.....	99
14.11 Manejo de excepciones y try-with-resources.....	99
14.12 Integración con operaciones no bloqueantes y redes.....	100
EJERCICIOS TEMA 14.....	100
TEMA 15. ENTRE HILOS Y MEMORIA: DOMINANDO LA CONCURRENCIA MODERNA EN JAVA.....	102
15.1 Introducción a la concurrencia.....	102
15.2 Concepto de threads.....	103
15.3 Ventajas de la concurrencia en Java.....	103
15.4 Creación y control de hilos en Java.....	103
15.4.1 Extender la clase Thread.....	103
15.4.2 Implementar la interfaz Runnable.....	104
15.4.3 Usar ExecutorService.....	104
15.5 Ciclo de vida de un hilo.....	105
15.6 Sincronización y condiciones de carrera.....	105
15.7 Clases y librerías clave en Java para concurrencia.....	105
15.8 Problemas comunes de la concurrencia.....	106
15.9 Hilos virtuales.....	106
15.9.1 Motivación y ventajas.....	106
15.9.2 Como funcionan internamente los hilos virtuales.....	107
15.9.3 Creación y uso de los hilos virtuales.....	107
15.9.4 Casos de uso típicos.....	108
15.9.5 Limitaciones y consideraciones.....	108

15.10 Java Memory Model (JMM).....	108
15.10.1 ¿Por qué se necesita un modelo de memoria?.....	108
15.10.2 Conceptos clave de JMM.....	109
15.10.2.1 Memoria principal y memoria de hilos.....	109
15.10.2.2 Operaciones atómicas.....	109
15.10.2.3 Reordenamiento de instrucciones.....	109
15.10.3 Problemas que el JMM resuelve.....	109
15.10.4 Principio de Happens-Before.....	109
15.10.5 Sincronización y visibilidad.....	110
15.10.6 Recomendaciones para usar el JMM correctamente.....	111
TEMA 16. JAVA EN ACCIÓN: EL PODER DETRÁS DE LO COTIDIANO.....	112
16.1 Criptografía en Java.....	112
16.1.1 Tipos de criptografía.....	112
16.1.2.1 Simétrica.....	112
16.1.2.2 Asimétrica.....	112
16.1.2.3 Hash o de resumen.....	112
16.2 Manejo de fechas y tiempos en Java.....	113
16.2.1 Clases principales del paquete java.time.....	113
16.3 Networking en Java.....	113
16.3.1 Clases principales del paquete java.net.....	114
16.3.1.1 InetAddress.....	114
16.3.1.2 Socket y ServerSocket.....	114
16.3.1.3 URL Y URLConnection.....	114
16.3.2 Arquitectura cliente-servidor.....	114
16.3.3 Aplicaciones prácticas del networking en Java.....	114
16.4 Expresiones regulares en Java.....	114
16.4.1 Paquete y clases principales.....	115
16.4.2 Sintaxis básica de las expresiones regulares.....	115
16.4.3 Aplicaciones comunes de las expresiones regulares.....	115
TEMA 17. INTEGRACIÓN PRÁCTICA DE CONCEPTOS EN UNA APLICACIÓN REAL.....	116
17.1 Ejercicio final.....	116
EPILOGO.....	120

# INTRODUCCIÓN

Si estás leyendo esto, probablemente hayas abierto este libro con miedo, curiosidad... o simple desesperación. Tal vez te dijeron que “Java es fácil”, o que “solo es cuestión de práctica”. Mentira. Java no es solo un lenguaje de programación: es una jungla donde los puntos y comas son depredadores, los errores de compilación acechan detrás de cada línea, y los NullPointerException son más persistentes que tu ex.

Pero tranquilo, estás en buenas manos. Este libro digital, Java para tontos, no pretende convertirte en un genio de Silicon Valley (aunque, oye, no estaría mal), sino en alguien que pueda mirar el código sin que le tiemble el ojo derecho.

Aprenderás qué es una clase (no, no la de la escuela), cómo funcionan los objetos (no los del universo, sino los del código), y por qué todos los programadores parecen adictos al café. Paso a paso, línea a línea, irás entendiendo por qué Java es tan poderoso... y tan quisquilloso.

Este libro digital está pensado para los que no saben nada, los que creen saber algo, y los que ya se rindieron una vez pero quieren volver a intentarlo. Porque aquí, hasta los errores son bienvenidos; de hecho, son parte del proceso (y a veces del chiste).

Así que, si alguna vez pensaste que la programación no era para ti, permíteme decirte algo:

Si, tal vez no sea fácil, pero tampoco es magia. Es paciencia, práctica y, sobre todo, actitud. Y con este libro, esa actitud empieza con una sonrisa (y una buena taza de café).

¡Preparate para adentrarte en el mundo de Java, donde los “sabelotodos” solo lo son hasta que logran ejecutar su primer Hello World sin errores!

# **TEMA 1. JAVA: DE LA HISTORIA A LA PRACTICA, TU PRIMER PASO EN LA PROGRAMACIÓN**

Objetivo general: Comprender qué es Java, conocer su historia, funcionamiento, proceso de instalación y aplicar los fundamentos de su sintaxis básica para iniciar en la programación.

Objetivos específicos:

- Explicar qué es Java y su relevancia en el mundo de la programación.
- Describir la historia y evolución de Java desde su creación hasta la actualidad.
- Analizar cómo funciona Java, incluyendo la máquina virtual (JVM) y la portabilidad.
- Guiar en el proceso de instalación de Java en diferentes sistemas operativos.
- Introducir la sintaxis básica de Java mediante ejemplos prácticos y sencillos.

## **1.1 Java y su relevancia en la actualidad**

En el mundo actual, la programación se ha convertido en una herramienta esencial para desarrollo de tecnologías que usamos día a día: aplicaciones móviles, páginas web, sistemas bancarios, inteligencia artificial y más. Dentro de este panorama, Java se mantiene como uno de los lenguajes de programación más utilizados, gracias a su estabilidad, portabilidad y seguridad. Comprender qué es Java y por qué sigue siendo tan importante es el primer paso para adentrarse en el mundo del desarrollo de software.

Las características que ha puesto Java serían las siguientes:

- Lenguaje orientado a objetos (POO): todo en Java se organiza en torno a clases y objetos, lo que facilita la reutilización del código, la modularidad y el mantenimiento de programas grandes.
- Multiplataforma: un programa escrito en Java, puede ejecutarse en diferentes sistemas operativos sin modificaciones, ya que se ejecuta sobre la Maquina Virtual Java (JVM).
- Seguro: Java evita accesos directos a la memoria y cuenta con un sistema de verificación de código. Esto reduce riesgos de virus, corrupción de datos y accesos no autorizados.
- Robusto: tiene un manejo eficiente de errores y excepciones. Además, cuenta con recolección automática de basura (garbage collector) que gestiona la memoria de forma automática.
- Sencillo: Java se diseñó con una sintaxis clara, inspirada en C y C++, pero eliminando características complejas. Esto lo hace más fácil de aprender.
- Distribuido: Java permite trabajar con aplicaciones en red de manera sencilla, integrando librerías que facilitan la comunicación entre sistemas a través de Internet.
- Alto rendimiento: Java logra buen rendimiento gracias a la compilación intermedia (bytecode) y a la tecnología JIT (Just In Time Compiler), que optimiza la ejecución.
- Multithreading: permite ejecutar varias tareas de manera simultánea dentro de un mismo programa, algo esencial en videojuegos, servidores y aplicaciones interactivas.

- Dinámico y extensible: Java se adapta a entornos en evolución, puede cargar nuevas clases en tiempo de ejecución y trabajar con librerías externas sin necesidad de modificar el núcleo del programa.

## 1.2 Historia de Java

El lenguaje Java fue creado en 1991 por James Gosling y un equipo de ingenieros en Sun Microsystems, dentro de un proyecto conocido como Green Project. Su objetivo inicial era desarrollar un lenguaje para controlar dispositivos electrónicos y electrodomésticos inteligentes. En esa primera etapa, el lenguaje fue llamado Oak, pero en 1995 cambió a Java, inspirándose en el café de la isla de Java en Indonesia.

En 1995, Java se presentó oficialmente junto con el eslogan “Write Once, Run Anywhere”, destacando su portabilidad gracias a la Máquina Virtual de Java (JVM). Esto le permitió diferenciarse de otros lenguajes que dependían directamente del sistema operativo.

Con el auge de internet en los 90s, Java se popularizó rápidamente gracias a los applets, pequeños programas que se ejecutaban en navegadores web. Más adelante, Java se consolidó en el mundo empresarial con la introducción de Java 2 (J2SE, J2EE, J2ME) a finales de los 90s e inicios de los 2000s, dividiendo el lenguaje en ediciones para escritorio, empresas y dispositivos móviles.

En 2009, Sun Microsystems fue adquirida por Oracle Corporation, que tomó el control del desarrollo de Java. Desde entonces, el lenguaje ha seguido evolucionando con versiones regulares que incluyen mejoras en rendimiento, seguridad, sintaxis y herramientas modernas.

Hoy en día, Java sigue siendo uno de los lenguajes más usados en el mundo, especialmente en:

- Aplicaciones móviles Android.
- Sistemas empresariales
- Frameworks web (Spring)
- Big Data y nube.

## 1.3 Como funciona Java

El proceso de funcionamiento de Java se basa en un concepto fundamental: la compilación a bytecode. A diferencia de otros lenguajes como C o C++, que se compilan directamente al lenguaje máquina de un sistema operativo específico, Java sigue este procedimiento:

- Escritura del código fuente: El programador escribe el programa en un archivo con extensión .java
- Compilación: El compilador de Java (.javac) traduce el código fuente en bytecode, que se almacena en archivos con extensión .class
- Ejecución en JVM: La Máquina Virtual de Java (JVM) interpreta o compila en tiempo real ese bytecode para convertirlo en instrucciones que el sistema operativo pueda ejecutar.

Este proceso es lo que hace posible la portabilidad de Java: un mismo programa puede ejecutarse en Windows, Linux, macOS, o incluso en dispositivos móviles, siempre y cuando exista una JVM instalada en el sistema.

Además, la JVM no solo garantiza la portabilidad, sino también:

- Seguridad: mediante la verificación del bytecode antes de su ejecución.
- Gestión de memoria: con el garbage collector, libera memoria automáticamente.
- Optimización: con el compilador JIT, convierte secciones del bytecode en código nativo para mejorar el rendimiento.

# TEMA 2. CONSTRUYENDO CON BLOQUES: VARIABLES Y TIPOS DE DATOS EN JAVA, LA BASE DE TODA PROGRAMACIÓN.

Objetivo general: Explicar el concepto de variables y los distintos tipos de datos en Java, destacando su importancia como fundamentos esenciales en la programación para el correcto manejo y almacenamiento de información.

Objetivos específicos:

- Definir qué es una variable en Java y cuál es su función dentro de un programa.
- Clasificar los tipos de datos en Java en primitivos y de referencia, identificando sus características principales.
- Ejemplificar el uso de variables con diferentes tipos de datos en programas sencillos.
- Analizar la importancia de elegir correctamente el tipo de dato según la información que se desea almacenar.
- Promover buenas prácticas en la declaración y uso de variables para escribir código claro, eficiente y legible.

## 2.1 Variables en Java

En Java, una variable es un espacio de memoria identificado por un nombre, en el cual se almacena un valor que puede ser utilizado y modificado durante la ejecución de un programa. Las variables permiten que el programador manipule datos de forma dinámica, ya que funcionan como “contenedores” que guardan información temporal. Para utilizarlas correctamente, es necesario declararlas, indicando el tipo de dato que contendrán y su nombre.

```
public class tema1 {  
    int i = 25;
```

En este caso, i es la variable, int define que solo puede guardar números enteros, y 25 es el valor inicial. El correcto uso de variables facilita la legibilidad, reutilización y eficiencia del código, ya que permiten representar y manipular datos reales en un programa.

## 2.2 Tipos de datos en Java

En programación con Java, los tipos de datos son la base que define cómo se representa, almacena y manipula la información dentro de un programa. Cada dato que se utiliza (ya sea un número, carácter, una cadena de texto, o un objeto más complejo) debe estar asociado a un tipo de dato específico que indica tanto su naturaleza como el espacio de memoria que ocupa. Comprender esta clasificación no solo es un requisito técnico, sino también una habilidad esencial para escribir programas eficientes, organizados y escalables.

En un lenguaje fuertemente tipado como Java, el tipo de dato debe ser declarado explícitamente al momento de crear una variable. Esto significa que el compilador necesita saber con precisión qué clase de información contendrá dicha variable, lo cual ofrece múltiples ventajas:

- Seguridad en el código: evita errores al impedir operaciones incompatibles, como sumar un texto con un número sin conversión previa.
- Eficiencia en memoria: cada tipo de dato tiene un tamaño fijo que permite al sistema optimizar los recursos.
- Claridad y mantenibilidad: el uso correcto de tipos hace que el código sea más legible y comprensible para otros programadores.

En Java, los tipos de datos se dividen en dos grandes categorías:

Datos primitivos	Datos de referencia
<ul style="list-style-type: none"><li>• Son los más básicos y fundamentales</li><li>• Se utilizan para representar valores simples, como números, caracteres, etc.</li><li>• No son objetos; se almacenan directamente en memoria</li></ul>	<ul style="list-style-type: none"><li>• Incluyen objetos y arreglos</li><li>• No almacenan directamente el valor, sino una referencia al lugar donde está guardado el dato real.</li><li>• Son más complejos, permiten manipular estructuras grandes y flexibles de información.</li></ul>

Java cuenta con ocho tipos de datos primitivos. Cada uno tiene un tamaño definido y un rango específico, lo que garantiza la portabilidad entre diferentes sistemas.

Tipo de dato	Descripción
<b>byte</b>	<ul style="list-style-type: none"><li>• Tamaño: 8 bits</li><li>• Rango: -128 a 127</li><li>• Uso: ideal cuando se requiere ahorrar memoria en arreglos grandes de números pequeños.</li></ul>
<b>short</b>	<ul style="list-style-type: none"><li>• Tamaño: 16 bits</li><li>• Rango: -32,768 a 32,767</li><li>• Uso: poco común en la práctica, pero útil en entornos con restricciones de memoria.</li></ul>
<b>int</b>	<ul style="list-style-type: none"><li>• Tamaño: 32 bits</li><li>• Rango: muy amplio</li><li>• Uso: se utiliza cuando los valores numéricos exceden la capacidad de int.</li></ul>
<b>long</b>	<ul style="list-style-type: none"><li>• Tamaño: 64 bits</li><li>• Rango: muy amplio</li><li>• Uso: se utiliza cuando los valores numéricos exceden la capacidad de int.</li></ul>
<b>float</b>	<ul style="list-style-type: none"><li>• Tamaño: 32 bits</li><li>• Precisión: hasta 7 decimales aproximados.</li><li>• Uso: se emplea para representar números con decimales, pero no se recomienda para cálculos que requieran alta precisión.</li></ul>
<b>double</b>	<ul style="list-style-type: none"><li>• Tamaño: 64 bits</li><li>• Precisión: hasta 15 decimales aproximados.</li><li>• Uso: es el tipo más común para trabajar con números decimales en Java.</li></ul>
<b>char</b>	<ul style="list-style-type: none"><li>• Tamaño: 16 bits</li></ul>

	<ul style="list-style-type: none"> <li>Precisión: hasta 15 decimales aproximados.</li> <li>Uso: es el tipo más común para trabajar con números decimales en Java.</li> </ul>
<b>boolean</b>	<ul style="list-style-type: none"> <li>Solo puede tomar dos valores: true o false</li> <li>Uso: se emplea en condiciones lógicas y estructuras de control.</li> </ul>

Un aspecto importante es que estos tipos no son objetos, por lo que no poseen métodos asociados. Sin embargo, Java ofrece las clases envolventes (wrapper classes) como Integer, Double o Boolean, que permiten tratarlos como objetos cuando es necesario.

A diferencia de los primitivos, los tipos de referencia permiten trabajar con estructuras de datos mucho más flexibles y poderosas. Estos tipos incluyen:

Tipo de referencia	Descripción
<b>Clases</b>	<ul style="list-style-type: none"> <li>Una clase en Java define un modelo a partir del cual se crean objetos.</li> <li>Cada objeto es una instancia que contiene variables (atributos) y métodos (comportamientos).</li> </ul>
<b>Interfaces</b>	<ul style="list-style-type: none"> <li>Son contratos que definen un conjunto de métodos abstractos.</li> <li>No almacenan datos directamente, pero son consideradas un tipo de referencia porque se usan para manipular objetos que implementan esas interfaces.</li> </ul>
<b>Arreglos</b>	<ul style="list-style-type: none"> <li>Una estructura de datos que almacena múltiples elementos del mismo tipo.</li> <li>Los arreglos en Java son objetos, incluso si almacenan tipos primitivos.</li> </ul>
<b>Objetos creados por el programador</b>	<ul style="list-style-type: none"> <li>A través de clases personalizadas, el programador puede definir nuevos tipos de referencia adaptados a sus necesidades.</li> </ul>

Los tipos de referencia siempre apuntan a una dirección de memoria, es decir, no guardan directamente el valor como los primitivos. Si se asigna un objeto a otra variable, ambos apuntarán a la misma referencia en memoria.

La elección del tipo de dato no es un detalle menor, sino una decisión que impacta directamente en la eficiencia, claridad y precisión del programa. Por ejemplo:

- Si se necesita contar objetos, lo más adecuado es usar un int.
- Si se manejan valores monetarios con alta precisión, un double puede ser insuficiente y conviene usar la clase BigDecimal.
- Para almacenar cadenas de texto, es imprescindible usar String, ya que los tipos primitivos no ofrecen esta capacidad.

Un mal uso de tipos puede generar problemas como pérdida de precisión, errores de conversión o incluso ineficiencia en el consumo de memoria.

## 2.3 Buenas prácticas en el uso de variables

El uso adecuado de variables en Java no solo consiste en declararlas y asignarles valores, sino también en aplicarlas de manera que el código sea claro, eficiente y fácil de mantener. Adoptar buenas prácticas en la programación mejora la calidad del software, reduce errores y facilita la colaboración entre desarrolladores.

Las reglas al escribir variables en Java son:

Descripción	
<b>Nombres descriptivos</b>	<ul style="list-style-type: none"><li>• Usa nombres que indiquen claramente el propósito de la variable.</li><li>• Ejemplo correcto: edadUsuario, precioProducto</li><li>• Evita nombres como x, y, temp, a menos que sean temporales o muy obvias.</li></ul>
<b>Convenciones de nomenclatura</b>	<ul style="list-style-type: none"><li>• Usa camelCase para variables: la primera palabra en minúscula y las siguientes con mayúscula. Ej. numerosIntentos, totalVentas</li><li>• Las constantes se escriben en mayúsculas con guiones bajos. Ej. MAX_INTENTOS, PI.</li></ul>
<b>Declaración clara</b>	<ul style="list-style-type: none"><li>• Siempre declara la variable con su tipo antes de usarla. Ej. int edad = 26;</li></ul>
<b>Inicialización adecuada</b>	<ul style="list-style-type: none"><li>• Asigna un valor inicial si es posible especialmente para tipos primitivos, para evitar errores. EJ. double precio = 0.0;</li></ul>
<b>Alcance limitado (scope)</b>	<ul style="list-style-type: none"><li>• Declara la variable en el bloque más cercano donde se utilizara (dentro de for, if, while)</li><li>• Evita declarar variables globales si no son necesarias en todo el programa.</li></ul>
<b>Consistencia en tipos de datos</b>	<ul style="list-style-type: none"><li>• Mantén coherencia en las operaciones; no mezcles tipos sin conversión explícita.</li><li>• Ejemplo correcto: int total = (int) precioDouble;</li></ul>
<b>Comentarios útiles</b>	<ul style="list-style-type: none"><li>• Si la variable tiene un propósito complejo; agrega comentarios explicativos.</li><li>• EJ. int totalUsuarios = 0; // Contador de usuarios registrados.</li></ul>

Las reglas en no escribir las variables (o prohibirlas de por vida) en Java son:

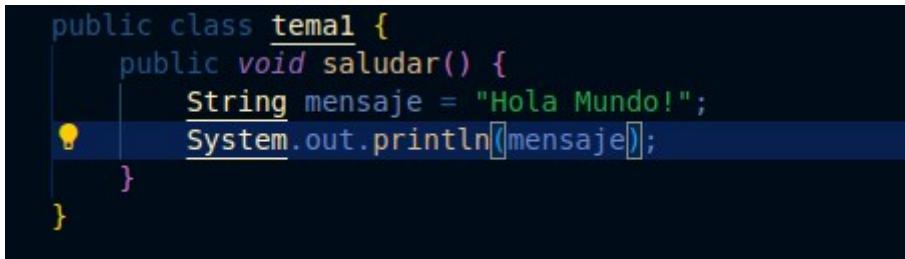
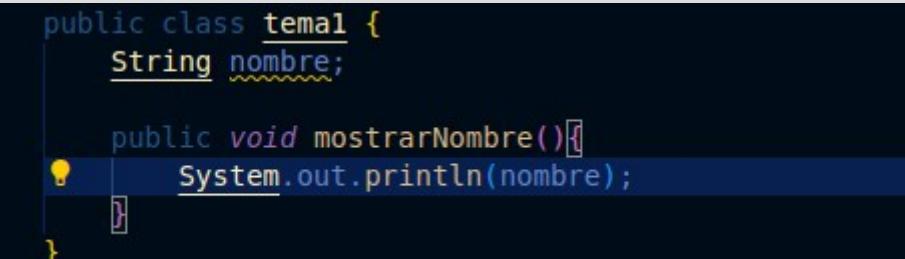
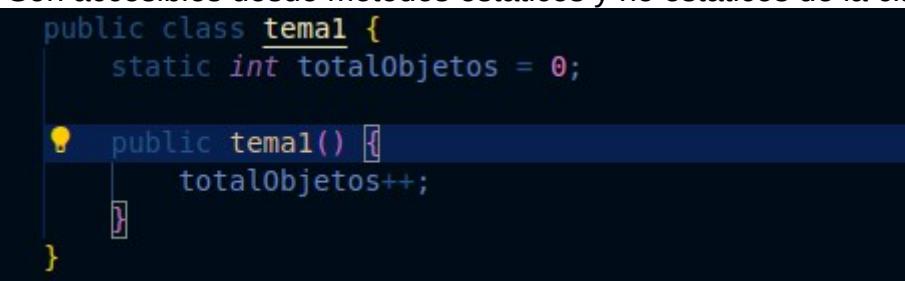
Descripción	
<b>No usar nombres genéricos o confusos</b>	Evita a, b, data, temp, si no es evidente su propósito.
<b>No iniciar nombres con números o caracteres especiales</b>	Incorrecto: 1edad, @precio. Correcto: edad1, precioTotal
<b>No usar palabras reservadas de Java</b>	Palabras como int, class, for, if no pueden ser nombres de variables.
<b>No declarar variables innecesarias</b>	Evita variables que nunca se usan o duplican información
<b>No mezclar estilos de nomenclatura</b>	Evita escribir algunas variables en mayúsculas, otras en minúsculas y otras en snake_case de forma inconsistente

<b>No dejar variables sin inicializar</b>	Especialmente los tipos primitivos; puede generar errores de compilación
<b>No cambiar el tipo de dato de una variable existente</b>	Una vez declarada, una variable no puede cambiar su tipo.

## 2.4 Alcance de una variable

El alcance de una variable, conocido como scope, define el área del programa en la que la variable puede ser utilizada o referenciada. Comprender este concepto es fundamental en Java, ya que permite organizar correctamente el código, evitar errores de acceso y optimizar el uso de memoria. El alcance depende del lugar donde se declare la variable y determina su visibilidad y tiempo de vida dentro del programa.

En Java, las variables pueden clasificarse según su alcance en varias categorías:

Tipo	Descripción
<b>Locales</b>	<ul style="list-style-type: none"> <li>Se declaran dentro de un método, constructor o bloque de código.</li> <li>Solo existen mientras se ejecuta ese bloque y no son accesibles fuera de él</li> </ul>  <pre> public class temal {     public void saludar() {         String mensaje = "Hola Mundo!";         System.out.println(mensaje);     } } </pre>
<b>De instancia</b>	<ul style="list-style-type: none"> <li>Se declaran dentro de una clase, pero fuera de cualquier método.</li> <li>Cada objeto de la clase tiene su propia copia.</li> <li>Son accesibles desde todos los métodos de la clase, usando this si es necesario.</li> </ul>  <pre> public class temal {     String nombre;      public void mostrarNombre(){         System.out.println(nombre);     } } </pre>
<b>De clase</b>	<ul style="list-style-type: none"> <li>Se declaran con la palabra clave static dentro de una clase.</li> <li>Comparten el mismo valor entre todas las instancias de la clase.</li> <li>Son accesibles desde métodos estáticos y no estáticos de la clase.</li> </ul>  <pre> public class temal {     static int totalObjetos = 0;      public temal() {         totalObjetos++;     } } </pre>

### De bloque

- Se declaran dentro de bloques de código como for, if o while
- Solo existen dentro del bloque donde fueron creadas.

```
public class ejemplo1 {  
    for(int i = 0; i < 5; i++) {  
        int cuadrado = i * i;  
        System.out.println(cuadrado);  
    }  
}
```

## 2.5 Conversión de variables

La conversión de variables, también conocida como type casting, es el proceso mediante el cual se cambia un valor de un tipo de dato a otro. En Java, este mecanismo es fundamental porque el lenguaje es fuertemente tipado, lo que significa que cada variable debe pertenecer a un tipo específico y no se puede cambiar directamente a otro sin una conversión. Comprender cómo funcionan estas conversiones es esencial para evitar errores de compilación, pérdidas de información y asegurar la eficiencia del programa.

En Java, existen dos formas principales de conversión: implícita y explícita. Además, cuando se trabaja con objetos, aparece el concepto de casting entre clases y el uso de clases envolventes (wrappers) para convertir entre primitivos y objetos.

Tipo	Descripción
<b>Conversión implícita</b>	Ocurre automáticamente cuando un valor de un tipo de dato más pequeño se asigna a un tipo de dato más grande. Este proceso es seguro porque no existe riesgo de pérdida de información significativa.

```
ejemploJava.java  
1 public class ejemploJava {  
2     int numero = 100;  
3     double decimal = numero; // Conversión implícita  
4     System.out.println(decimal);  
5 }
```

### Conversión explícita

Se utiliza cuando se necesita almacenar un valor de un tipo de dato más grande en uno más pequeño. Este proceso no es automático porque puede ocasionar pérdida de información o truncamiento de valores decimales. Para realizarlo, se coloca entre paréntesis el tipo al cual se quiere convertir.

```
public class EjemploJava {  
    public static void main(String[] args) {  
        double decimal = 9.68;  
        int numero = (int) decimal;  
        System.out.println(numero);  
    }  
}
```

**Conversión entre Java** incluye las clases envolventes (wrappers) que permiten tratar los

## primitivos y objetos

tipos primitivos como objetos.

- Autoboxing: conversión automática de un tipo primitivo a su clase envolvente.

```
public class EjemploJava {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int numero = 4;  
        Integer numeroObjeto = numero;  
    }  
}
```

- Unboxing: conversión automática de un objeto envolvente a su tipo primitivo.

```
public class EjemploJava {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        Integer numeroObjeto = 10;  
        int numero = numeroObjeto;  
    }  
}
```

## Conversión entre clases

En la POO (programación orientada a objetos), también se pueden realizar conversiones entre clases relacionadas por herencia.

- Upcasting: convertir un objeto de una clase hija a su clase padre.
- Downcasting: convertir un objeto de la clase padre a clase hija.

## Conversión de cadenas a tipos primitivos

Muchas veces es necesario convertir datos introducidos como texto en consola a tipos numéricos o booleanos. Java proporciona métodos en las clases envolventes.

```
public class EjemploJava {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        String texto = "24";  
        int numero = Integer.parseInt(texto);  
        System.out.println(numero + 5);  
    }  
}
```

## Conversión de tipos primitivos a cadenas

También se puede convertir cualquier tipo primitivo en texto con el método `String.valueOf()`:

```
public class EjemploJava {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int edad = 30;  
        String texto = String.valueOf(edad);  
        System.out.println("Edad: " + texto);  
    }  
}
```

## EJERCICIOS TEMA 2.

EJERCICIO 1. Crear variables de diferentes tipos y muéstralas en consola.

### SOLUCIÓN

```
public class Ejercicio1 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int edad = 27;
        double altura = 1.75;
        char inicial = 'M';
        boolean estudiante = false;

        System.out.println("Edad: " +edad);
        System.out.println("Altura: " +altura);
        System.out.println("Inicial: " +inicial);
        System.out.println("¿Es estudiante?: " +estudiante);
    }
}
```

EJERCICIO 2. Declara dos enteros y realiza operaciones básicas.

### SOLUCIÓN

```
public class Ejercicio2 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int a = 10;
        int b = 3;

        System.out.println("Suma: " + (a + b));
        System.out.println("Resta: " + (a - b));
        System.out.println("Multiplicación: " + (a * b));
        System.out.println("División: " + (a / b));
        System.out.println("Resto: " + (a % b));
    }
}
```

### EJERCICIO 3. Combina variables con un mensaje de texto

#### SOLUCIÓN

```
Ejercicio3.java / ...
1  public class Ejercicio3 {
2      Run main | Debug main | Run | Debug
3      public static void main(String[] args) {
4          String nombre = "Marco";
5          int edad = 27;
6          System.out.println("Hola, me llamo " +nombre+ "y tengo"
7                           +edad+ " años.");
8      }
9 }
```

### EJERCICIO 4. Asignar un int a un double

#### SOLUCIÓN

```
Ejercicio4.java / ...
1  public class Ejercicio4 {
2      Run main | Debug main | Run | Debug
3      public static void main(String[] args) {
4          int numero = 15;
5          double decimal = numero;
6          System.out.println("Numero entero: " + numero);
7          System.out.println("Convertido a decimal: " + decimal);
8      }
9 }
```

### EJERCICIO 5. Convertir un número decimal en entero

#### SOLUCIÓN

```
Ejercicio5.java / ...
1  public class Ejercicio5 {
2      Run main | Debug main | Run | Debug
3      public static void main(String[] args) {
4          double decimal = 9.68;
5          int entero = (int) decimal;
6          System.out.println("Decimal original: " + decimal);
7          System.out.println("Convertido a entero: " + entero);
8      }
9 }
```

EJERCICIO 6. Declara una constante y usala en un calculo.

SOLUCIÓN

```
1  public class Ejercicio6 {
2      Run main | Debug main | Run | Debug
3      public static void main(String[] args) {
4          final double PI = 3.1416;
5          double radio = 5.0;
6          double area = PI * radio * radio;
7          System.out.println("Área del círculo: " + area);
8      }
9 }
```

EJERCICIO 7. Convierte un valor en texto a un número entero.

SOLUCIÓN

```
public class Ejercicio7 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        String texto = "123";
        int numero = Integer.parseInt(texto);
        System.out.println("Texto convertido a entero: " + numero);
        System.out.println("Suma con 10: " + (numero + 10));
    }
}
```

EJERCICIO 8. Solicita datos del usuario desde teclado

SOLUCIÓN

```
J Ejercicio8.java > ...
2  import java.util.Scanner;
3
4  public class Ejercicio8 {
5      Run main | Debug main | Run | Debug
6      public static void main(String[] args) {
7          Scanner sc = new Scanner(System.in);
8
9          System.out.println("Ingresa tu nombre: ");
10         String nombre = sc.nextLine();
11
12         System.out.println("Ingresa tu edad: ");
13         int edad = sc.nextInt();
14
15         System.out.println("Hola: " + nombre + ", tienes "
16                     + edad + " años.");
17     }
18 }
```

## EJERCICIO 9. Trabaja con Integer y Int

### SOLUCIÓN

```
public class Ejercicio9 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int numeroPrimitivo = 50;
        Integer numeroObjeto = numeroPrimitivo;

        int convertido = numeroObjeto;

        System.out.println("Número primitivo " + numeroPrimitivo);
        System.out.println("Número objeto: " + numeroObjeto);
        System.out.println("Convertido de nuevo: " + convertido);
    }
}
```

## EJERCICIO 10. Muestra el uso de variables locales, de instancia y estáticas.

### SOLUCIÓN

```
class Ejemplo {
    int variableInstancia = 10;
    static int variableEstatica = 20;

    public void mostrar() {
        int variableLocal = 30;
        System.out.println("Variable local: " + variableLocal);
        System.out.println("Variable de instancia: " + variableInstancia);
        System.out.println("Variable estática: " +variableEstatica);
    }
}

public class Ejercicio10 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Ejemplo obj1 = new Ejemplo();
        obj1.mostrar();
    }
}
```

# TEMA 3. JAVA EN ACCIÓN: EL PODER DE LOS OPERADORES

Objetivo general: Explicar que son los operadores en Java, su clasificación y utilidad, para comprender como permiten manipular datos y realizar operaciones dentro de un programa.

Objetivos específicos:

- Definir qué son los operadores en Java y su papel en la programación
- Identificar y describir los distintos tipos de operadores en Java
- Explicar con ejemplos prácticos cómo se utilizan los operadores en expresiones y sentencias.
- Analizar la precedencia y asociatividad de los operadores para evitar errores comunes en el código.
- Fomentar el uso correcto y eficiente de los operadores en la resolución de problemas de programación.

## 3.1 Introducción a los operadores

En Java, los operadores son símbolos especiales que permiten realizar operaciones sobre uno o más operandos, es decir, valores o variables. Su función es fundamental, ya que constituyen la base para manipular datos, tomar decisiones y controlar el flujo de un programa. Gracias a los operadores, un desarrollador puede combinar variables, constantes y expresiones para obtener resultados, evaluar condiciones o modificar estados en memoria.

Un operando es el elemento sobre el cual actúa un operador. Por ejemplo, en la expresión  $a + b$ , los operandos son  $a$  y  $b$ , mientras que el operador es el signo  $+$ . Esta relación entre operadores y operandos es lo que da vida a las expresiones en Java.

La importancia de los operadores radica en que permiten expresar de forma sencilla operaciones complejas. Desde simples cálculos matemáticos, hasta evaluaciones lógicas o manipulaciones avanzadas de datos, los operadores hacen que el código sea más legible y eficiente. Además, facilitan la interacción entre distintos tipos de datos, como números enteros, decimales, cadenas de texto y valores booleanos.

## 3.2 Clasificación de los operadores

En Java, los operadores se dividen en varias categorías según la función que desempeñan en una expresión. Esta clasificación permite comprender cómo se construyen las instrucciones y de qué manera interactúan con los datos dentro de un programa. Cada grupo de operadores cumplen un propósito específico, desde operaciones matemáticas básicas hasta evaluaciones lógicas complejas o manipulaciones a nivel de bits.

Operador	Descripción
Aritméticos	Son los encargados de realizar operaciones matemáticas entre valores numéricos <ul style="list-style-type: none"><li>• Suma: <math>(a + b)</math></li><li>• Resta: <math>(a - b)</math></li></ul>

	<ul style="list-style-type: none"> <li>• Multiplicación: (a * b)</li> <li>• División: (a / b)</li> <li>• Residuo: (a % b)</li> </ul>
De asignación	Permite almacenar valores en variables. El más simple es el signo igual (=), que asigna el valor del operando derecho al izquierdo. Además, existen operadores de asignación compuesta, que simplifican expresiones combinando operaciones aritméticas con la asignación.
	<pre>public class Ejemplo1 {     Run main   Debug main   Run   Debug     public static void main(String[] args) {         int x = 5;         x+= 3;     } }</pre>
Relacionales	<p>Se utilizan para comparar valores, devolviendo siempre un resultado booleano (true o false)</p> <ul style="list-style-type: none"> <li>• Igualdad: ==</li> <li>• Desigualdad: !=</li> <li>• Mayor que: &gt;</li> <li>• Menor que: &lt;</li> <li>• Mayor o igual: &gt;=</li> <li>• Menor o igual: &lt;=</li> </ul> <p>Estos operadores son ampliamente utilizados en estructuras condicionales.</p>
Logicos	Permiten combinar o negar condiciones booleanas. Son esenciales para tomar decisiones múltiples.
	<ul style="list-style-type: none"> <li>• AND (&amp;&amp;): devuelve verdadero si ambas condiciones son verdaderas.</li> <li>• OR (  ): devuelve verdadero si al menos una condición es verdadera.</li> <li>• NOT (!): invierte el valor de verdad de la condición.</li> </ul>
Unarios	Actúan sobre un solo operando. Los más importantes son:
	<ul style="list-style-type: none"> <li>• Incremento (++): aumenta en 1 el valor de la variable.</li> <li>• Decremento (--): disminuye en 1 el valor de la variable.</li> <li>• Posfijo (x++) y prefijo (++x) determinan el momento de la evaluación.</li> </ul>
Ternario	Es una forma abreviada de condicional if-else. Su sintaxis es: condición ? ValorSiTrue : valorSiFalse
Bit a bit	Trabajan a nivel de bits, lo cual es útil en programación de sistemas y optimización.
	<pre>public class Ejemplo1 {     Run main   Debug main   Run   Debug     public static void main(String[] args) {         int a = 5, b = 3;         System.out.println(a &amp; b); // 1 (0101 &amp; 0011 = 0001)     } }</pre>
Otros	<ul style="list-style-type: none"> <li>• Concatenación (+ en cadenas): permite unir texto y variables</li> </ul>

### 3.3 Precedencia y asociatividad de los operadores

En Java, los operadores pueden incluir varios operadores al mismo tiempo, lo que plantea la necesidad de establecer un orden de evaluación. Para ello, el lenguaje define dos reglas fundamentales: la precedencia y la asociatividad de los operadores. Comprender estos conceptos es esencial para evitar errores lógicos y garantizar que el programa produzca los resultados esperados.

La precedencia de operadores determina cuál de ellos se evalúa primero cuando existen varios en una misma expresión. Por ejemplo, en la operación  $3 + 4 * 2$ , el resultado es 11 y no 14, porque el operador de multiplicación (\*) tiene mayor precedencia que la suma (+). De esta manera, Java ejecuta primero  $4 * 2$  y luego realiza la suma.

Por otro lado, la asociatividad de operadores establece la dirección en que se evalúan cuando dos o más tienen la misma precedencia. Generalmente, la evaluación es de izquierda a derecha, como ocurre con los operadores aritméticos y relacionales. Sin embargo, algunos operadores, como la asignación (=) y el ternario (? :), tienen asociatividad de derecha a izquierda, lo que significa que se resuelven desde el último hacia el primero.

Un aspecto clave es que una mala interpretación de estas reglas puede conducir a resultados inesperados, incluso si el programa no presenta errores de sintaxis. Por ejemplo, en la expresión  $a = b = 5$ , gracias a la asociatividad de derecha a izquierda, primero se asigna el valor 5 a  $b$  y luego ese mismo valor a  $a$ .

## EJERCICIOS TEMA 3.

EJERCICIO 1. Un cliente compra 3 refrescos a \$15 cada uno y 2 bolsas de papas a \$10. Calcula el total a pagar considerando que primero debe multiplicarse la cantidad por el precio antes de sumar.

### SOLUCIÓN

```
public class Ejercicio1 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int refrescos = 3 * 15;  
        int papas = 2 * 10;  
        int total = refrescos + papas;  
  
        System.out.println("Total a pagar: " +total);  
    }  
}
```

EJERCICIO 2. Determina si una persona puede votar (edad  $\geq 18$ ) y si esta en la primera votación de su vida (edad == 18).

### SOLUCIÓN

```
public class Ejercicio2 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int edad = 18;  
        boolean puedeVotar = edad >= 18 && edad <= 100;  
        boolean primeraVez = (edad == 18);  
  
        System.out.println("¿Puede votar?: " +puedeVotar);  
        System.out.println("¿Es tu primera votación?: " +primeraVez);  
    }  
}
```

EJERCICIO 3. Si una compra supera \$1000, se aplica un 10% de descuento, si no, paga el precio normal.

### SOLUCIÓN

```
public class Ejercicio3 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        double compra = 1200;  
        double total = (compra > 1000) ? compra * 0.9 : compra;  
        System.out.println("Total con posible descuento: $" +total);  
    }  
}
```

EJERCICIO 4. Calcula el promedio de tres calificaciones y determina si el estudiante aprobo ( $\geq 6$ )

### SOLUCIÓN

```
public class Ejercicio4 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int c1 = 7;  
        int c2 = 8;  
        int c3 = 6;  
        double promedio = (c1 + c2 + c3)/3.0;  
        boolean aprobado = promedio >= 6;  
  
        System.out.println("Promedio: " +promedio);  
        System.out.println("¿Aprobado?: " +aprobado);  
    }  
}
```

EJERCICIO 5. Un maestro asigna la misma calificación inicial a tres estudiantes a la vez.

### SOLUCIÓN

```
public class Ejercicio5 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int a, b, c;  
        a = b = c = 10;  
        System.out.println("a=" + a + ", b=" + b + ", c=" +c);  
    }  
}
```

# TEMA 4. DECIDIENDO EL CAMINO: DOMINA LAS CONDICIONALES EN JAVA

Objetivo general: Comprender y aplicar correctamente las estructuras condicionales en Java para controlar el flujo de ejecución de los programas según diferentes condiciones y situaciones.

Objetivos específicos:

- Identificar los tipos de condicionales en Java, incluyendo if, if-else, else if y switch.
- Analizar y evaluar expresiones booleanas que determinan la ejecución de bloques de código.

## 4.1 Definición de condicional

En programación, una condicional es una estructura que permite al programar tomar decisiones basadas en ciertas condiciones. Es decir, dependiendo de si una condición es verdadera o falsa, el programa puede ejecutar un bloque de código u otro. Las condicionales son fundamentales para controlar el flujo de ejecución y hacer que los programas sean dinámicos y adaptables a distintas situaciones.

Java, como lenguaje de programación orientado a objetos, ofrece varias formas de condicionales, entre las cuales más comunes son: if, if-else, else if y switch. Estas estructuras permiten evaluar expresiones booleanas (que resultan en true o false) y ejecutar código dependiendo del resultado de esa evaluación.

## 4.2 Condicional if

La condicional if es la estructura más básica y utilizada para la toma de decisiones en Java. Su sintaxis general es la siguiente:

```
public class Ejemplo1 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        if(condicion) {  
            // Bloque de código  
        }  
    }  
}
```

Funciona en:

1. Primero, se evalúa la condición entre paréntesis. Esta condición debe ser una expresión que devuelva un valor booleano (true o false)
2. Si la condición es verdadera, el bloque de código dentro de las llaves {} se ejecuta.
3. Si la condición es falsa, el bloque de código se ignora y el programa continúa con la siguiente instrucción después del if.

```
public class Ejemplo1 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int edad = 18;

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

En este ejemplo, la condición `edad >= 18` se evalúa. Como `edad` es 18, la condición es verdadera y se imprime el mensaje en pantalla. Si `edad` fuera menor a 18, el bloque dentro del `if` no se ejecutaría.

Consideraciones importantes:

- Siempre que se use `if`, la condición debe producir un valor booleano.
- Se puede anidar varios `if` dentro de otros `if` para evaluar múltiples condiciones, aunque para casos complejos es recomendable usar `else if` o `switch` para mayor claridad.
- El uso correcto de las llaves `{}` garantiza que el bloque de código asociado se ejecute de forma precisa y evita errores lógicos.

### 4.3 Condicional `if-else`

Mientras que el `if` por sí solo permite ejecutar un bloque de código únicamente cuando la condición es verdadera, la estructura `if-else` añade una alternativa: ejecutar un bloque diferente cuando la condición es falsa.

```
public class Ejemplo2 {
    if(condicion) {
        // Bloque
    } else {
        // Bloque
    }
}
```

Funciona en:

1. El programa evalúa la condición dentro del `if`
2. Si la condición es verdadera, se ejecuta el primer bloque de código
3. Si la condición es falsa, se ejecuta el bloque de código dentro del `else`.
4. En ambos casos, una vez que se ha ejecutado uno de los bloques, el programa continúa con la siguiente instrucción después del `if-else`.

Ejemplo:

```
public class Ejemplo2 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int edad = 17;  
  
        if(edad >= 18) {  
            System.out.println(x:"Eres mayor de edad");  
        } else {  
            System.out.println(x:"Eres menor de edad");  
        }  
    }  
}
```

En este ejemplo, la condición `edad >= 18` se evalúa. Como el valor de `edad` es 17, la condición resulta falsa, por lo que se ejecuta el bloque dentro de `else`, mostrando en pantalla “Eres menor de edad”.

## 4.4 Condicional else if

La estructura `else if` se utiliza cuando existen varias condiciones distintas y el programa debe ejecutar un bloque de código diferente para cada una de ellas.

```
public class Ejemplo2 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        if(condicion1) {  
            // Código  
        } else if (condicion2) {  
            // Código  
        } else if (condicion3) {  
            // Código  
        } else {  
            // Código  
        }  
    }  
}
```

El funcionamiento paso a paso es:

1. El programa evalúa la primera condición (`if`)
2. Si la primera condición es falsa, pasa a evaluar la siguiente (`else if`)
3. El proceso continúa hasta que una de las condiciones resulte verdadera.
4. En cuanto una condición se cumpla, se ejecuta únicamente el bloque de código asociado y se ignoran las demás.
5. Si ninguna condición es verdadera, se ejecuta el bloque dentro de `else` (si existe)

Mira este ejemplo:

```
1 public class Ejemplo2 {  
2     Run main | Debug main | Run | Debug  
3     public static void main(String[] args) {  
4         int nota = 85;  
5  
6         if(nota >= 90) {  
7             System.out.println("Excelente");  
8         } else if (nota >= 75) {  
9             System.out.println("Bueno");  
10        } else if (nota >= 60) {  
11            System.out.println("Suficiente");  
12        } else {  
13            System.out.println("Reprobado");  
14        }  
15    }  
16}
```

En este caso:

- Si la nota es 90 o más, se imprime “Excelente”.
- Si no, pero la nota es 75 o más, se imprime “Bueno”.
- Si tampoco, pero la nota es 60 o más, se imprime “Suficiente”.
- Si ninguna de las condiciones se cumple, se ejecuta el bloque else y imprime “Reprobado”.

## 4.5 Condicional switch

La estructura switch evalúa el valor de una expresión (generalmente una variable) y lo compara con una lista de posibles casos (case). Cuando encuentra una coincidencia, ejecuta el bloque de código correspondiente.

```
Ejemplo3.java > Language Support for Java(TM) by Red Hat > Ejemplo3  
public class Ejemplo3 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        switch (expresion) {  
            case valor1:  
                // Código  
                break;  
            case valor2:  
                // Código  
                break;  
            case valor3:  
                // Código  
                break;  
            default:  
                // Código  
        }  
    }  
}
```

El funcionamiento paso a paso es:

1. El switch evalúa la expresión dentro de los parentesis
2. Busca un case que coincida con el valor de esa expresión
3. Cuando encuentra el caso correspondiente, ejecuta las instrucciones dentro de ese bloque.
4. La instrucción break evita que el programa continúe ejecutando los casos siguientes de forma accidental (comportamiento llamado fall-through)
5. Si ningun case coincide, se ejecuta el bloque default, que es opcional pero recomendable.

```
import java.util.*;  
public class Ejemplo3 {  
    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("Miercoles");  
                break;  
            case 4:  
                System.out.println("Jueves");  
                break;  
            case 5:  
                System.out.println("Viernes");  
                break;  
            default:  
                System.out.println("Fin de semana");  
        }  
    }  
}
```

En este ejemplo, como dia vale 3, el programa imprime Miercoles.

## EJERCICIOS TEMA 4

EJERCICIO 1. Pide la edad y muestra si la persona es mayor o menor de edad.

### SOLUCIÓN

```
Ejercicio1.java > ...
import java.util.Scanner;

public class Ejercicio1 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print(s:"Ingresa tu edad: ");
        int edad = sc.nextInt();

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

EJERCICIO 2. Evalúa la nota y muestra si es Excelente, Bueno, Suficiente o Reprobado.

### SOLUCIÓN

```
import java.util.Scanner;

public class Ejercicio2 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println(x:"Ingresa tu calificación (0 - 100): ");
        int nota = sc.nextInt();

        if(nota >= 90) {
            System.out.println(x:"Excelente");
        } else if (nota >= 75) {
            System.out.println(x:"Bueno");
        } else if (nota >= 60) {
            System.out.println(x:"Suficiente");
        } else {
            System.out.println(x:"Reprobado");
        }
    }
}
```

### EJERCICIO 3. Usa un número del 1 al 7 para mostrar al día

#### SOLUCIÓN

```
3
4  public class Ejercicio3 {
5      Run | Debug | Run main | Debug main
6      public static void main(String[] args) {
7          Scanner sc = new Scanner(System.in);
8
9          System.out.print(x:"Ingresa un número (1-7): ");
10         int dia = sc.nextInt();
11
12         switch (dia) {
13             case 1:
14                 System.out.println(x:"Lunes");
15                 break;
16             case 2:
17                 System.out.println(x:"Martes");
18                 break;
19             case 3:
20                 System.out.println(x:"Miercoles");
21                 break;
22             case 4:
23                 System.out.println(x:"Jueves");
24                 break;
25             case 5:
26                 System.out.println(x:"Viernes");
27                 break;
28             case 6:
29                 System.out.println(x:"Sabado");
30                 break;
31             case 7:
32                 System.out.println(x:"Domingo");
33             default:
34                 System.out.println(x:"Número invalido");
35         }
36     }
37 }
```

### EJERCICIO 4. Si la compra es mayor a 500 pesos, aplicar 10% de descuento.

#### SOLUCIÓN

```
2  import java.util.Scanner;
3
4  public class Ejercicio4 {
5      Run | Debug | Run main | Debug main
6      public static void main(String[] args) {
7          Scanner sc = new Scanner(System.in);
8
9          System.out.println(x:"Ingresa el monto de la compra: ");
10         double monto = sc.nextDouble();
11
12         if(monto > 500) {
13             double descuento = monto * 0.10;
14             double total = monto - descuento;
15             System.out.println(x:"Aplica un 10% de descuento: ");
16             System.out.println("Total a pagar: $" + total);
17         } else {
18             System.out.println("No aplica descuento. Total a pagar: " + monto);
19         }
20     }
21 }
```

## EJERCICIO 5. Muestra un menú y realiza una operación según la opción elegida.

### SOLUCIÓN

```
public class Ejercicio5 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Menú de opciones: ");
        System.out.println("1. Sumar");
        System.out.println("2. Restar");
        System.out.println("3. Multiplicar");
        System.out.println("4. Dividir");

        System.out.println("Elige una opción: ");
        int opcion = sc.nextInt();

        switch (opcion) {
            case 1:
                System.out.println("Seleccionaste: Sumar");
                break;
            case 2:
                System.out.println("Seleccionaste: Restar");
                break;
            case 3:
                System.out.println("Seleccionaste: Multiplicar");
                break;
            case 4:
                System.out.println("Seleccionaste: Dividir");
                break;
            default:
                System.out.println("Opción invalida");
        }
    }
}
```

# TEMA 5. CICLOS QUE DAN VIDA AL CÓDIGO: EXPLORANDO LOS BUCLES EN JAVA

Objetivo general: Comprender el funcionamiento, la importancia y la aplicación de los bucles en Java, para utilizarlos de manera eficiente en la resolución de problemas de programación repetitivos y estructurados.

Objetivos específicos:

- Identificar los diferentes tipos de bucles en Java y sus características principales.
- Explicar la sintaxis, funcionamiento y diferencias de cada tipo de bucle.
- Aplicar bucles en la resolución de problemas prácticos mediante ejemplos de código.
- Analizar cuándo es más conveniente utilizar un bucle específico según el contexto del programa.
- Desarrollar algoritmos que integren bucles para optimizar procesos repetitivos en aplicaciones Java.

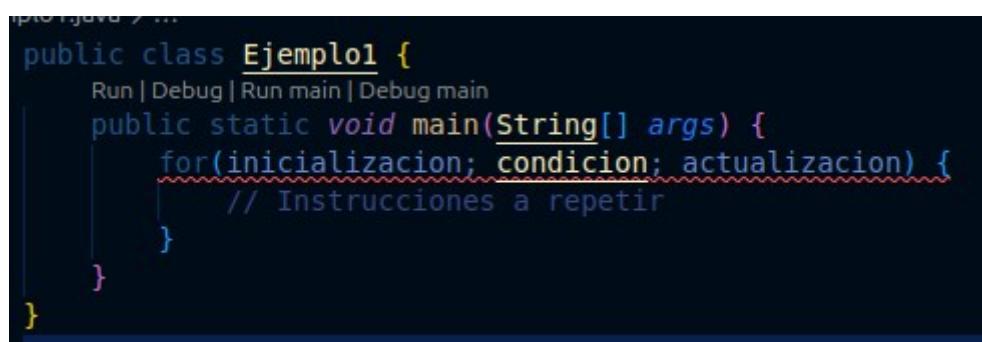
## 5.1 Concepto de bucle

En programación, un bucle es una estructura de control que permite ejecutar de manera repetida un bloque de instrucciones mientras se cumpla una condición. Su finalidad principal es automatizar tareas repetitivas, reducir la redundancia en el código y facilitar la resolución de problemas que requieren realizar una misma acción varias veces, ya sea con diferentes datos o hasta que se cumpla un criterio definido.

En el lenguaje Java, los bucles se forman parte de las estructuras de control de flujo, las cuales permiten modificar la secuencia normal de ejecución de un programa. Existen distintos tipos de bucles: while, do-while, for y for-each, cada uno con características que los hacen útiles en contextos específicos. Gracias a ellos, es posible desde recorrer listas de datos hasta implementar algoritmos complejos que dependen de repeticiones controladas.

## 5.2 Bucle for

El bucle for es uno de los más utilizados en Java, especialmente cuando se conoce de antemano la cantidad de repeticiones necesarias. Su sintaxis está diseñada para integrar en una sola línea tres elementos fundamentales: la inicialización de una variable de control, la condición lógica que debe cumplirse para continuar el ciclo y la actualización de la variable de control en cada iteración. Su forma general es:



```
public class Ejemplo1 {
    public static void main(String[] args) {
        for(inicializacion; condicion; actualizacion) {
            // Instrucciones a repetir
        }
    }
}
```

The image shows a screenshot of a Java code editor. The code is displayed in a dark-themed interface. The code itself is a simple for loop within a class named 'Ejemplo1'. The loop variables are 'inicializacion', 'condicion', and 'actualizacion'. A comment 'Instrucciones a repetir' is placed inside the loop body. At the top of the code editor, there is a toolbar with icons for 'Run', 'Debug', 'Run main', and 'Debug main'. The file name 'Ejemplo1.java' is visible at the top left.

Por ejemplo, el siguiente bucle imprime los números del 1 a 5.

```
src / ...  
public class Ejemplo1 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        for(int i = 1; i <= 5; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

En este caso, la variable `i` se inicializa en 1, la condición establece que el ciclo continuará mientras `i` sea menor o igual a 5, y en cada iteración `i` se incrementa en una unidad. Esto permite recorrer un rango de valores de manera sencilla y controlada.

### 5.3 Bucle for-each

En Java, el bucle for-each (también conocido como enhanced for loop) es una estructura de control diseñada específicamente para recorrer colecciones de datos como arreglos (arrays), listas, conjuntos u otras estructuras que implementan la interfaz Iterable. A diferencia del bucle for tradicional, el for-each simplifica el proceso al eliminar la necesidad de manejar explícitamente una variable de control o un índice, proporcionando así un código más legible, conciso y menos propenso a errores.

La sintaxis general del bucle for-each es la siguiente:

```
src / ...  
public class Ejemplo2 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        for(tipo variable : colección) {  
            // Instrucciones  
        }  
    }  
}
```

En esta estructura:

- tipo corresponde al tipo de dato de los elementos de la colección.
- Variable es una referencia temporal que adopta, en cada iteración, el valor de un elemento de la colección.
- Colección representa el arreglo, lista o estructura de datos que se desea recorrer.

Por ejemplo, el siguiente código imprime cada elemento de un arreglo de enteros:

```
Ejemplo2.java > ...
public class Ejemplo2 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int[] numeros = {10, 20, 30, 40, 50};
        for(int n : numeros) {
            System.out.println(n);
        }
    }
}
```

En este caso, la variable n toma secuencialmente los valores almacenados en el arreglo numeros sin necesidad de especificar un índice ni manejar manualmente el incremento de este. De esta manera, el código resulta más claro y directo.

## 5.4 Bucle while

Uno de los bucles más utilizados es el bucle while, cuya característica principal es que evalúa la condición antes de ejecutar el bloque de instrucciones, lo que lo convierte en un bucle de control previo.

La sintaxis general del bucle while es:

```
Ejemplo3.java > ...
public class Ejemplo3 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        while(condicion) {
            // INSTRUCCIONES
        }
    }
}
```

En esta estructura:

- condicion es una expresión lógica que debe evaluarse como verdadera (true) para que el ciclo continue.
- El bloque de instrucciones dentro de las llaves se ejecutará repetidamente mientras la condición sea verdadera.
- Cuando la condición se evalúe como false, el ciclo se detendrá y la ejecución del programa continuará con la siguiente instrucción después del bucle.

Por ejemplo, el siguiente código imprime los números del 1 a 5 utilizando un bucle.

```
Ejemplo3.java / ...
public class Ejemplo3 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int i = 1;
        while(i <= 5) {
            System.out.println(i);
            i++;
        }
    }
}
```

En este caso, la variable `i` se inicializa en 1. La condición `i <= 5` se evalúa antes de cada iteración. Mientras sea verdadera, se imprime el valor de `i` y luego se incrementa en una unidad. Una vez que `i` supera el valor 5, la condición resulta falsa y el bucle finaliza.

No obstante, es importante tener preocupación al usar `while`. Si la condición nunca llega a ser falsa, el programa puede caer un bucle infinito, lo que provoca que se quede ejecutando indefinidamente. Para evitarlo, se debe garantizar que dentro del bloque de instrucciones exista alguna operación que modifique el estado de la condición.

## 5.5 Bucle do-while

En Java, el bucle `do-while` es una estructura de control que permite ejecutar de manera repetida un bloque de instrucciones, con la particularidad de que siempre se ejecuta al menos una vez, independientemente si la condición se cumple o no. Esto se debe a que, a diferencia del bucle `while`, el `do-while` evalúa la condición después de haber ejecutado el bloque de instrucciones, razón por la cual se denomina un bucle de control posterior.

La sintaxis general del `do-while` es la siguiente:

```
Ejemplo4.java / ...
public class Ejemplo4 {
    public static void main(String[] args) {
        do {
            // Instrucciones
        } while (condicion);
    }
}
```

En esta estructura:

- El bloque dentro de `do { ... }` se ejecuta primero
- Luego, la condición se evalúa.
- Si la condición es verdadera (`true`), el ciclo se repite.
- Si la condición es falsa (`false`), el bucle termina y la ejecución continúa con la siguiente instrucción del programa.

Por ejemplo, el siguiente código imprime los números del 1 al 5:

```
src> Ejemplo4.java
public class Ejemplo4 {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println(i);
            i++;
        } while (i <= 5);
    }
}
```

En este caso, el bloque de instrucciones se ejecuta, imprime el valor de *i* y lo incrementa en cada iteración. Posteriormente, se evalúa la condición *i*  $\leq$  5. Mientras esta sea verdadera, el ciclo continuara. Una vez que *i* supera el valor 5, la condición resulta falsa y el bucle se detiene.

Sin embargo, se debe tener cuidado con la lógica de la condición, ya que un diseño inadecuado puede provocar bucles infinitos, en los que la condición nunca se vuelve falsa.

## EJERCICIOS TEMA 5.

EJERCICIO 1. Escribe un programa que use un bucle para imprimir todos los números pares de 1 al 20.

### SOLUCIÓN

```
src> Ejercicio1.java
public class Ejercicio1 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        for(int i = 2; i <= 20; i += 2) {
            System.out.println(i);
        }
    }
}
```

EJERCICIO 2. Crea un arreglo con los nombres de 5 frutas. Usa un bucle para imprimir cada uno de los elementos.

### SOLUCIÓN

```
src> Ejercicio2.java
public class Ejercicio2 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        String[] frutas = {"Manzana", "Platano", "Naranja",
                           "Fresa", "Uva"};
        for(String fruta : frutas) {
            System.out.println(fruta);
        }
    }
}
```

EJERCICIO 3. Desarrolla un programa que utilice un bucle para mostrar una cuenta regresiva desde 10 hasta 1, y al final imprima la palabra ¡Despegue!

#### SOLUCIÓN

```
Ejercicio3.java > ...
public class Ejercicio3 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int i = 10;
        while (i >= 1) {
            System.out.println(i);
            i--;
        }
        System.out.println("¡Despegue!");
    }
}
```

EJERCICIO 4. Escribe un programa que pida al usuario ingresar una contraseña. El programa debe seguir pidiendo la contraseña usando un bucle hasta que el usuario escriba la palabra “java123”.

#### SOLUCIÓN

```
import java.util.Scanner;

public class Ejercicio4 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String contraseña;

        do {
            System.out.println("Ingrese la contraseña: ");
            contraseña = sc.nextLine();
        } while (!contraseña.equals("java123"));

        System.out.println("¡Acceso concedido!");
        sc.close();
    }
}
```

EJERCICIO 5. Escribe un programa que muestre las tablas de multiplicar del 1 al 5 utilizando dos bucles anidados.

#### SOLUCIÓN

```
public class Ejercicio5 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        for(int i = 1; i <= 5; i++) {
            System.out.println("Tabla del " + i + ":");
            for(int j = 1; j <= 10; j++) {
                System.out.println(i + " x " + j + " = " + (i * j));
            }
            System.out.println();
        }
    }
}
```

# TEMA 6. ARREGLOS EN JAVA: LA BASE DEL ORDEN Y LA EFICIENCIA EN LA PROGRAMACIÓN

Objetivo general: Analizar y aplicar el uso de arreglos en Java como una estructura de datos fundamental para organizar, almacenar y manipular información de manera eficiente en la programación.

Objetivos específicos:

- Explicar el concepto de arreglos y su importancia dentro de la programación en Java
- Identificar los tipos de arreglos y sus características principales.

## 6.1 Definición de arreglo

En Java, los arreglos (arrays) constituyen una de las estructuras de datos más utilizadas y fundamentales. Un arreglo puede definirse como una colección ordenada de elementos del mismo tipo de datos, almacenados en posiciones consecutivas de memoria y accesibles mediante un índice numérico. Este índice comienza en cero, lo que significa que el primer elemento ocupa la posición 0, el segundo la posición 1, y así sucesivamente.

Los arreglos en Java permiten organizar y manipular conjuntos de información de forma eficiente. A diferencia de las variables individuales, que almacenan un único valor, un arreglo posibilita trabajar con múltiples valores bajo un mismo nombre, facilitando el acceso, recorrido y procesamiento de datos. Por ejemplo, en lugar de declarar diez variables para almacenar calificaciones, se puede declarar un solo arreglo de diez posiciones.

Existen varios tipos de arreglos en Java, que son los siguientes en resumir:

## 6.2 Arreglos unidimensionales

Un arreglo unidimensional es una estructura de datos lineal que permite almacenar y manipular un conjunto de elementos del mismo tipo de dato bajo un único identificador. Los elementos se encuentran organizados en memoria de manera contigua y son accesibles mediante un índice numérico que inicia en cero. Esto significa que el primer elemento se ubica en la posición 0, el segundo en la posición 1, y así sucesivamente hasta n-1, donde n corresponde al tamaño del arreglo.

La declaración de un arreglo unidimensional en Java requiere definir el tipo de dato y el tamaño fijo que contendrá. Por ejemplo:

```
mplo1.java > ...
  public class Ejemplo1 {
    public static void main(String[] args) {
      int[] numeros = new int[5];
    }
  }
```

En este caso, se crea un arreglo de enteros con cinco posiciones, donde cada elemento se inicializa automáticamente con el valor por defecto del tipo correspondiente (para enteros, el valor es 0). También es posible inicializar un arreglo de manera explícita con valores definidos:

```
public class Ejemplo1 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int[] numeros = {10, 20, 30, 40, 50};  
    }  
}
```

El recorrido de un arreglo unidimensional suele realizarse mediante estructuras repetitivas, como el ciclo for, lo cual permite acceder a cada elemento utilizando su índice. Asimismo, el ciclo for-each facilita recorrer el arreglo sin necesidad de manejar explícitamente los índices.

Los arreglos unidimensionales ofrecen múltiples ventajas: permiten almacenar grandes volúmenes de datos homogéneos, facilitan la implementación de algoritmos de búsqueda y ordenamiento, y proporcionan un acceso rápido a los elementos gracias a su indexación. Sin embargo, también presentan limitaciones, entre ellas: el tamaño debe definirse al momento de la creación y no puede modificarse durante la ejecución del programa, lo que reduce su flexibilidad frente a otras estructuras dinámicas como ArrayList.

### 6.3 Arreglos bidimensionales

Un arreglo bidimensional es una estructura de datos que extiende el concepto de los arreglos unidimensionales, permitiendo organizar información en forma de filas y columnas, similar a una tabla o matriz. Cada elemento dentro de este arreglo se identifica mediante dos índices: el primero indica la fila y el segundo la columna, ambos comenzando desde cero.

La declaración de un arreglo bidimensional requiere especificar el tipo de dato y las dimensiones que lo conforman. Por ejemplo:

```
ript01.java > ...  
public class Ejemplo1 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int[][] matriz = new int[3][4];  
    }  
}
```

En este caso, se crea una matriz de enteros con 3 filas y 4 columnas, generando un total de 12 elementos. De manera predeterminada, cada posición se inicializa con el valor por defecto correspondiente al tipo de dato (en enteros, el valor es 0). También es posible inicializar el arreglo de forma directa.

```
ript01.java > ...  
public class Ejemplo1 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int[][] matriz = {  
            {1, 2, 3},  
            {4, 5, 6},  
            {7, 8, 9}  
        };  
    }  
}
```

El acceso a los elementos se realiza utilizando dos índices, por ejemplo: matriz[0][2] devolverá el valor contenido en la primera fila y tercera columna. El recorrido de un arreglo bidimensional se logra generalmente mediante ciclos anidados (for dentro de otro for), aunque también es posible emplear el ciclo for-each para simplificar la lectura.

Los arreglos bidimensionales resultan útiles para representar datos estructurados en dos dimensiones, como tablas de multiplicar, tableros de juegos, matrices matemáticas o registros en formato de cuadrícula. Entre sus principales ventajas destacan la facilidad para organizar información compleja y el acceso rápido a cada posición gracias a la indexación. No obstante, presentan limitaciones como el tamaño fijo, que debe definirse en el momento de creación y no puede modificarse durante la ejecución, lo que reduce la flexibilidad frente a colecciones dinámicas como ArrayList o HashMap.

## 6.4 Arreglos multidimensionales

Un arreglo multidimensional es una extensión de los arreglos unidimensionales y bidimensionales que permite representar y manipular datos en más de dos dimensiones. Se puede entender como un arreglo que contiene otros arreglos, donde cada nivel de índice agrega una nueva dimensión. Gracias a esta característica, los arreglos multidimensionales son útiles para modelar estructuras complejas de información, como cubos de datos, matrices de mayor orden o representaciones tridimensionales.

La declaración de un arreglo multidimensional en Java sigue la misma sintaxis que los arreglos de menor dimensión, pero con más corchetes. Por ejemplo:

```
import java ...  
public class Ejemplo1 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int[][][] cubo = new int[3][3][3];  
    }  
}
```

En este caso, se crea un arreglo tridimensional con 3 filas, 3 columnas y 3 niveles de profundidad, totalizando 27 elementos. De forma predeterminada, todos los valores se inicializan con el valor por defecto del tipo de datos (para enteros, 0). También es posible inicializar directamente los valores:

```
import java ...  
public class Ejemplo1 {  
    Run main | Debug main | Run | Debug  
    public static void main(String[] args) {  
        int[][][] cubo = {  
            { {1, 2}, {3, 4} },  
            { {5, 6}, {7, 8} }  
        };  
    }  
}
```

El acceso a los elementos requiere tantos índices como dimensiones tenga el arreglo. Por ejemplo, cubo [1][0][1] accedería a la segunda fila, primera columna y segunda posición de profundidad. Para recorrer un arreglo multidimensional se utilizan ciclos anidados, incrementando el nivel de bucle por cada dimensión que se desee procesar.

Los arreglos multidimensionales son muy utilizados en aplicaciones que requieren manejar grandes volúmenes de información estructurada en varias dimensiones, como simulaciones en 3D, procesamiento de imágenes, análisis de datos y programación científica. Sus principales ventajas son la organización de datos de forma jerárquica y el acceso directo a los elementos mediante índices. Sin embargo, presentan limitaciones, entre ellas el tamaño fijo y el consumo de memoria que puede crecer de manera considerable con cada dimensión adicional.

## EJERCICIOS TEMA 6.

EJERCICIO 1. Crea un programa que almacene un arreglo los números {5, 8, 12, 20, 7} y muestre la suma de todos sus elementos.

### SOLUCIÓN

```
EJERCICIO1.java > ...
public class Ejercicio1 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int[] numeros = {5, 8, 12, 20, 7};
        int suma = 0;

        for(int i = 0; i < numeros.length; i++) {
            suma+= numeros[i];
        }

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

EJERCICIO 2. Dado un arreglo de enteros {3, 9, 15, 20, 30}, solicita un número al usuario y verifica si está en el arreglo.

### SOLUCIÓN

```
import java.util.Scanner;

public class Ejercicio2 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int[] numeros = {3, 9, 15, 20, 30};
        Scanner sc = new Scanner(System.in);
        boolean encontrado = false;

        System.out.print("Ingrese un número: ");
        int buscado = sc.nextInt();
    }
}
```

## SOLUCIÓN PT. 2

```
        for(int num : numeros) {
            if(num == buscado) {
                encontrado = true;
                break;
            }
        }

        if(encontrado) {
            System.out.println(x:"Esta encontrado!");
        } else {
            System.out.println(x:"No esta encontrado. Lo siento.");
        }
    }
```

EJERCICIO 3. Guarda 6 calificaciones en un arreglo y calcula el promedio de los valores.

## SOLUCIÓN

```
public class Ejercicio3 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        double[] calificaciones = {8.5, 9.0, 7.5, 10.0, 6.8, 9.2};
        double suma = 0;

        for (double c : calificaciones) {
            suma+= c;
        }

        double promedio = suma/calificaciones.length;
        System.out.println("El promedio es: " +promedio);
    }
}
```

EJERCICIO 4. Genera una matriz de 5x5 que almacene la tabla de multiplicar del 1 a 5 y muéstralala en pantalla.

## SOLUCIÓN

```
public class Ejercicio4 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int[][] tabla = new int[5][5];

        for(int i=0; i<5; i++){
            for(int j=0; j<5; j++){
                tabla[i][j] = (i + j) * (j + 1);
                System.out.println(tabla[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

EJERCICIO 5. Declara dos matrices de 2x2 y muestra la matriz resultante de su suma.

### SOLUCIÓN

```
public class Ejercicio5 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int[][] matrizA = {{1, 2}, {3, 4}};
        int[][] matrizB = {{5, 6}, {7, 8}};
        int[][] suma = new int[2][2];

        for(int i = 0; i < 2; i++) {
            for(int j = 0; j < 2; j++) {
                suma[i][j] = matrizA[i][j] + matrizB[i][j];
                System.out.println(suma[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

EJERCICIO 6. Crea un arreglo tridimensional de tamaño 2x2x2 y almacena en él los números del 1 al 8. Luego, muéstralos en pantalla.

### SOLUCIÓN

```
public class Ejercicio6 {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        int[][][] cubo = new int[2][2][2];
        int valor = 1;

        for(int i=0; i<2; i++) {
            for(int j=0; j<2; j++){
                for(int k=0; k<2; k++){
                    cubo[i][j][k] = valor++;
                    System.out.println("cubo[" + i + "][" + j + "][" + k + "] = " + cubo[1][j][k]);
                }
            }
        }
    }
}
```

# TEMA 7. MÉTODOS EN JAVA: LA ARQUITECTURA INVISIBLE QUE DA VIDA AL CÓDIGO.

Objetivo general: Explicar la importancia, funcionamiento y aplicación de los métodos en Java, destacando cómo favorecen la modularidad, reutilización y eficiencia en el desarrollo de programas.

Objetivos específicos:

- Definir qué son los métodos en Java y su papel dentro de la programación orientada a objetos.
- Clasificar los distintos tipos de métodos y sus características.
- Demostrar cómo se declaran, llaman y utilizan los métodos en la construcción de programas.

## 7.1 Definición de método

En Java, un método es un bloque de código que realiza una tarea específica y que puede ser invocado en cualquier parte de un programa. Los métodos permiten organizar el código en unidades lógicas más pequeñas, favoreciendo la modularidad, la reutilización y la legibilidad del software. En términos simples, un método representa una acción o un comportamiento asociado a una clase u objeto.

Un método en Java está compuesto por cuatro elementos principales:

1. Modificador de acceso (public, private, protected): que determina el nivel de visibilidad del método.
2. Tipo de retorno: indica el valor que devolverá el método (int, String, double). Si no devuelve nada, se usa la palabra clave void.
3. Nombre del método: debe ser único dentro de la clase y seguir las convenciones de nomenclatura en Java (camelCase)
4. Parámetros: valores que el método puede recibir como entrada para procesar.

La sintaxis general es la siguiente:

```
modificador tipoRetorno nombreMetodo(parámetros) {  
    // Cuerpo del método  
    return valor; // opcional si no es void  
}
```

Los métodos se pueden clasificar en varias categorías:

## 7.2 Tipos de métodos

### 7.2.1 Métodos de instancia

Los métodos de instancia son aquellos que pertenecen directamente a los objetos creados a partir de una clase. Para invocarlos, es necesario primero crear una instancia de la clase, ya que estos métodos operan sobre los atributos individuales de cada objeto. Por ejemplo, si se tiene una clase

Persona, el método mostrarNombre() accederá a la información específica de cada objeto creado, como el nombre de María o el de Saúl, sin que el comportamiento general cambie.

Estos métodos se declaran de manera común, sin la palabra clave static, y su principal característica es que pueden manipular tanto los atributos de instancia como los atributos estáticos. Además, permiten la interacción directa con el estado interno de los objetos, lo que fomenta la aplicación del concepto de encapsulación.

### 7.2.2 Métodos estáticos

En contraste, los métodos estáticos no requieren de un objeto para ser ejecutados, ya que pertenecen a la clase en sí misma. Se definen utilizando la clave static y se invocan directamente con el nombre de la clase. Un ejemplo clásico se encuentra en la clase Math de Java, que contiene métodos como Math.sqrt() o Math.random().

Los métodos estáticos se utilizan cuando el comportamiento no depende de atributos particulares de los objetos, sino que responde a una operación general. Al no depender de una instancia, estos métodos no pueden acceder directamente a los atributos de instancia de la clase, sino únicamente a otros elementos estáticos.

Su uso favorece la eficiencia y practicidad, ya que resultan ideales para implementar funciones utilitarias o cálculos comunes. Sin embargo, abusar de los métodos estáticos puede llevar a diseños menos flexibles, pues limitan la capacidad de aprovechar el polimorfismo y el dinamismo de los objetos.

### 7.2.3 Métodos sobrecargados

La sobrecarga de métodos ocurre cuando se definen varios métodos con el mismo nombre dentro de una clase, pero con diferentes listas de parámetros (ya sea en número, tipo o ambos). Este mecanismo no debe confundirse con la sobrescritura (override), que consiste en redefinir un método heredado en una subclase.

La sobrecarga se utiliza para ofrecer diferentes formas de realizar una misma acción, adaptándose a distintos contextos. Por ejemplo, en una clase Calculadora, se podría definir un método sumar(int a, int b) y otro sumar(double a, double b). Ambos comparten el mismo nombre, pero permiten trabajar con distintos tipos de datos.

Este recurso incrementa la legibilidad y la flexibilidad, ya que el programador no necesita recordar múltiples nombres de métodos para una misma operación. Además, mejora la consistencia del diseño, evitando la redundancia en la nomenclatura.

En Java, la sobrecarga también se aplica a los constructores, lo que permite crear objetos con diferentes configuraciones iniciales, de acuerdo con los parámetros recibidos.

# TEMA 8. JAVA Y EL MUNDO DE LOS OBJETOS: FUNDAMENTOS DE LA POO (PARTE 1)

Objetivo general: Comprender los fundamentos de la Programación Orientada a Objetos (POO) en Java para desarrollar programas estructurados, modulares y reutilizables que faciliten la resolución de problemas del mundo real.

Objetivos específicos:

- Identificar los principios básicos de la POO: clases, objetos, atributos y métodos.
- Diseñar clases sencillas en Java que representen entidades del mundo real.

## 8.1 Concepto de Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es un paradigma que organiza el software en torno a entidades llamadas objetos, los cuales representan elementos del mundo real o abstracto. Estos objetos poseen atributos (características) y métodos (comportamientos), y se crean a partir de clases, que actúan como moldes o planos. En Java, uno de los más utilizados para la enseñanza y desarrollo de aplicaciones, la comprensión de las clases y objetos es la base fundamental para avanzar en la creación de programas modulares, escalables y reutilizables.

## 8.2 Clases y objetos

### 8.2.1 Clase

Una clase en Java es una plantilla que define la estructura y el comportamiento de un conjunto de objetos. En términos sencillos, puede considerarse como un plano o diseño que describe cómo serán los objetos que se crean a partir de ella.

- Atributos: representan las propiedades o datos que un objeto puede tener. Se suelen declarar como variables dentro de la clase.
- Métodos: son las acciones o comportamientos asociados a los objetos, definidos como funciones dentro de la clase.

```
public class Ejemplo1 {  
  
    String nombre;  
    int edad;  
  
    void saludar() {  
        System.out.println("Hola, soy " +nombre);  
    }  
}
```

En este ejemplo, la clase Persona describe que cada persona tendrá un nombre, una edad y la capacidad de ejecutar la acción de saludar.

## 8.2.2 Objeto

Un objeto es la instancia concreta de una clase. Mientras la clase es una abstracción, el objeto es la materialización de esa abstracción en memoria. Cada objeto tiene sus propios valores en los atributos, aunque todos comparten la estructura definida por la clase.

```
public static void main(String[] args) {
    Ejemplo1 personal = new Ejemplo1();

    personal.nombre = "Twilight";
    personal.edad = 27;

    personal.saludar();
}
```

Aquí, `personal1` es un objeto distinto creado a partir de la clase `Persona`. Cada uno posee valores diferentes para sus atributos (`nombre` y `edad`), pero comparten el mismo método `saludar`.

La relación entre clase y objeto se puede comprender mediante la analogía del diseño y la construcción:

- La clase es el plano de una casa.
- El objeto es la casa construida a partir de ese plano.

Esto implica que una misma clase puede dar origen a múltiples objetos, cada uno con características propias, pero todos con la misma estructura.

## 8.3 Modificadores de acceso

En Java, los modificadores de acceso son palabras clave que controlan el nivel de visibilidad de clases, atributos, métodos y constructores dentro de un programa. Su finalidad principal es establecer encapsulamiento, uno de los pilares de la POO. Gracias a estos modificadores, un programador puede proteger la información sensible de un objeto, restringir el acceso a ciertas partes del código o, en contraste, permitir la reutilización de componentes en distintos paquetes o proyectos.

Java cuenta con cuatro niveles de acceso principales:

Tipo	Descripción
<code>public</code>	<ul style="list-style-type: none"><li>• Significa que el elemento puede ser accedido desde cualquier clase y desde cualquier paquete.</li><li>• Se utiliza cuando se desea que un método o clase esté disponible globalmente.</li></ul> <pre>public class Estudiante {     public String nombre;      public void mostrarNombre() {         System.out.println("Nombre: " + nombre);     } }</pre>

<b>private</b>	<ul style="list-style-type: none"> <li>• Indica que el miembro de la clase solo puede ser accedido dentro de la misma clase.</li> <li>• Se utiliza para proteger la información sensible de un objeto.</li> <li>• Promueve el encapsulamiento al obligar el uso de métodos getters y setters.</li> </ul> <pre>public class Cuenta {     private double saldo;      public void depositar(double cantidad) {         if (cantidad &gt; 0) {             saldo += cantidad;         }     }      public double getSaldo() {         return saldo;     } }</pre>
<b>protected</b>	<ul style="list-style-type: none"> <li>• Permite el acceso al miembro de una clase desde: la misma clase, clases del mismo paquete y clases hijas (subclases), aunque estén en paquetes diferentes.</li> <li>• Es muy útil cuando se trabaja con herencia.</li> </ul> <pre>public class Animal {     protected String especie; }  public class Perro extends Animal {     public void mostrarEspecie() {         System.out.println("Soy un " + especie);     } }</pre>
<b>default</b>	<ul style="list-style-type: none"> <li>• Si no se coloca ningún modificador de acceso, se aplica el nivel por defecto.</li> <li>• El acceso queda limitado a las clases del mismo paquete.</li> <li>• También se le conoce como package-private.</li> </ul> <pre>class Libro {     String titulo; // acceso por defecto      void mostrarTitulo() {         System.out.println("Título: " + titulo);     } }</pre>

El empleo correcto de modificadores de acceso proporciona varias ventajas:

1. Seguridad: protege datos sensibles de accesos no autorizados.
2. Encapsulamiento: obliga a utilizar métodos controlados para modificar atributos.
3. Claridad: permite al programador identificar qué métodos son públicos y deben usarse, y cuáles son internos de la clase.

- Herencia controlada: con protected, se puede dar acceso a subclases sin exponer los datos al resto del programa.
- Organización modular: ayuda a separar responsabilidades entre diferentes paquetes y componentes.

## 8.4 Static vs. Final

En Java, las palabras clave static y final son fundamentales para el control de atributos, métodos y clases. Ambas cumplen funciones diferentes, pero se relacionan con el alcance, la memoria y las restricciones en la POO. Comprenderlas es esencial para escribir código más eficiente, seguro y estructurado.

### 8.4.1 Palabra clave static

La palabra clave static indica que un miembro de la clase (atributo, método o bloque) pertenece a la clase y no a los objetos. Esto significa que se comparte entre todas las instancias y se carga en memoria solo una vez.

Tipo	Descripción
Atributos	<ul style="list-style-type: none"> <li>Son compartidos por todos los objetos creados de la clase.</li> <li>No dependen de cada instancia individual.</li> </ul> <pre>public class Contador {     static int total = 0; // variable compartida      Contador() {         total++;     } }</pre>
Métodos	<ul style="list-style-type: none"> <li>Pueden llamarse sin crear un objeto de la clase.</li> <li>Solo pueden acceder a <u>miembros estáticos</u>.</li> </ul> <pre>public class Matematicas {     static int sumar(int a, int b) {         return a + b;     } }  public class Main {     public static void main(String[] args) {         System.out.println(Matematicas.sumar(5, 3));     } }</pre>
Bloques	<ul style="list-style-type: none"> <li>Se ejecutan una sola vez, al cargar la clase en memoria.</li> <li>Útiles para inicializar datos estáticos.</li> </ul> <pre>public class Ejemplo {     static {         System.out.println("Bloque estático ejecutado");     } }</pre>

## 8.4.2 Palabra clave final

La palabra clave final establece restricciones. Se utiliza para evitar cambios en variables, métodos o clases.

Tipo	Descripción
Variables	<ul style="list-style-type: none"><li>Son constantes: su valor no puede modificarse después de la inicialización.</li></ul> <pre>public class Constantes {     final double PI = 3.1416; }</pre>
Métodos	<ul style="list-style-type: none"><li>No pueden ser sobreescritos (override) en clases hijas.</li></ul> <pre>class Animal {     final void sonido() {         System.out.println("Sonido genérico");     } }  class Perro extends Animal {     // Error: no se puede sobreescibir sonido() }</pre>
Clases	<ul style="list-style-type: none"><li>No pueden heredarse</li></ul> <pre>final class Utilidades {     void imprimir() {         System.out.println("Clase final");     } }</pre>

## 8.5 Clases anidadas

Una clase anidada es simplemente una clase declarada dentro de otra clase. La clase externa actúa como contenedor, y la clase interna puede acceder a los miembros de la externa (incluyendo atributos privados).

Se usan principalmente para:

- Agrupar clases que solo tienen sentido en conjunto.
- Encapsular detalles de implementación.
- Hacer el código más legible y modular.

Java permite cuatro tipos de clases anidadas:

Tipo	Descripción
Clase interna	<ul style="list-style-type: none"> <li>• Declarada dentro de otra clase sin la palabra clave static.</li> <li>• Puede acceder a todos los miembros de la clase externa (incluso privados).</li> <li>• Requiere una instancia de la clase externa para ser utilizada.</li> </ul> <pre>public class Externa {     private String mensaje = "Hola desde la clase externa";      class Interna {         void mostrar() {             System.out.println(mensaje); // accede a miembro privado         }     }      public static void main(String[] args) {         Externa externa = new Externa();         Externa.Interna interna = externa.new Interna();         interna.mostrar();     } }</pre>
Clase interna estática	<ul style="list-style-type: none"> <li>• Declarada con static</li> <li>• No puede acceder directamente a miembros no estáticos de la clase externa.</li> <li>• Se instancia sin necesidad de un objeto de la clase externa.</li> </ul> <pre>public class Externa {     static String saludo = "Hola";      static class Interna {         void mostrar() {             System.out.println("Mensaje: " + saludo);         }     }      public static void main(String[] args) {         Externa.Interna interna = new Externa.Interna();         interna.mostrar();     } }</pre>
Clase local	<ul style="list-style-type: none"> <li>• Se declaran dentro de un método, constructor o bloque.</li> <li>• Solo son visibles dentro de ese ámbito.</li> <li>• Útiles para lógica auxiliar temporal.</li> </ul> <pre>public class Externa {     void metodo() {         class Local {             void imprimir() {                 System.out.println("Soy una clase local");             }         }         Local local = new Local();         local.imprimir();     }      public static void main(String[] args) {         new Externa().metodo();     } }</pre>
Clase anónima	<ul style="list-style-type: none"> <li>• No tienen nombre</li> <li>• Se crean para implementar rápidamente una clase (interfaces o clases abstractas).</li> </ul>

- Muy usadas en callbacks o con interfaces funcionales.

```
public class Ejemplo {  
    public static void main(String[] args) {  
        Runnable r = new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Ejecución en clase anónima");  
            }  
        };  
        new Thread(r).start();  
    }  
}
```

Las tres ventajas de las clases anidadas serian las siguientes:

- 1) Organización: agrupan clases que tienen sentido lógico juntas.
- 2) Encapsulamiento: ocultan detalles de implementación que no deben ser accesibles desde fuera.
- 3) Claridad: reducen el desorden cuando una clase solo se usa en un lugar.
- 4) Flexibilidad: con clases anónimas se crean implementaciones rápidas sin declarar clases extra.

# **TEMA 9. JAVA Y EL MUNDO DE LOS OBJETOS: FUNDAMENTOS DE LA POO (PARTE 2)**

Objetivo general: Comprender y aplicar los principios avanzados de la POO en Java para el diseño y desarrollo de aplicaciones modulares, reutilizables y mantenibles.

Objetivos específicos:

- Explicar y aplicar el mecanismo de herencia en Java para reutilizar código y crear jerarquías de clases que representen relaciones del mundo real.
- Implementar clases y métodos abstractos para definir plantillas de comportamiento que promuevan el diseño flexible y escalable de software.
- Aplicar modificadores de acceso para proteger los datos internos de una clase, garantizando seguridad e integridad.
- Demostrar el uso de polimorfismo estático y dinámico para lograr la flexibilidad en la reutilización de métodos y clases.
- Integrar los pilares de la POO en el desarrollo de proyectos en Java, fomentando la legibilidad, mantenimiento y evolución del software.

## **9.1 Ciclo de vida de un objeto**

En la POO, los objetos representan entidades del mundo real y constituyen la base del diseño y la ejecución de programas. En Java, el ciclo de vida de un objeto describe las etapas por las que pasa un objeto desde su creación hasta su destrucción, involucrando procesos como la instanciación, el uso, la inaccesibilidad y la recolección de memoria.

### 1. Creación:

El ciclo inicia cuando un objeto es creado a partir de una clase. Para ello, Java utiliza la palabra clave `new`, que reserva memoria en el heap e invoca al constructor de la clase.

### 2. Inicialización:

Después de la instanciación, se ejecuta el constructor y, opcionalmente, bloques de inicialización. Estos asignan valores a los atributos del objeto, asegurando que comience su vida en un estado válido.

### 3. Uso:

En esta fase, el objeto se encuentra accesible a través de su referencia. El programador puede invocar métodos, modificar atributos y participar en interacciones con otros objetos. Es la etapa más extensa, pues el objeto “vive” mientras sea necesario en el programa.

### 4. Inaccesibilidad:

Un objeto se vuelve inaccesible cuando no existen referencias que lo apunten. Aunque siga existiendo físicamente en la memoria, ya no puede ser utilizado por el programa. Este estado ocurre, por ejemplo, cuando una variable que contenía la referencia cambia a otra, o cuando la referencia sale de su alcance.

5. Recolección de basura:
- Java cuenta con un mecanismo automático llamado Garbage Collector, que identifica los objetos inaccesibles y libera la memoria que ocupaban. Esto evita fugas de memoria y asegura un uso eficiente de los recursos. A diferencia de otros lenguajes, en Java el programador no destruye explícitamente los objetos, aunque puede sugerir la ejecución del recolector con System.gc();

## 9.2 Herencia

La herencia es uno de los pilares fundamentales de la POO y consiste en la capacidad de una clase de adquirir atributos y comportamientos (métodos) de otra clase. Este mecanismo permite reutilizar código, establecer jerarquías entre clases y modelar relaciones naturales del mundo real, favoreciendo la extensibilidad y el mantenimiento de los programas.

En Java, la herencia se establece mediante la palabra clave extends, que indica que una clase (subclase o clase hija) hereda las características de otra (superclase o clase padre). De este modo, la subclase puede utilizar los atributos y métodos de la superclase como si fueran propios, además de definir sus propias características particulares.

```
class Animal {  
    void comer() {  
        System.out.println("El animal está comiendo");  
    }  
}  
  
class Perro extends Animal {  
    void ladurar() {  
        System.out.println("El perro está ladando");  
    }  
}
```

En este ejemplo, Perro hereda de Animal, por lo que puede usar el método comer() además de su propio método ladurar().

Los tipos de herencia en Java son:

- Herencia simple: Java solo permite herencia simple entre clases (una clase solo puede heredar de una clase padre). Esto evita problemas de ambigüedad presentes en otros lenguajes.
- Herencia multinivel: una clase hija puede ser a su vez padre de otra clase.
- Herencia múltiple a través de interfaces: aunque no es posible heredar de múltiples clases, sí se pueden implementar múltiples interfaces para obtener un efecto similar.

La herencia promueve la reutilización de código, pero también permite modificar el comportamiento heredado. Esto se logra con la sobrescritura de métodos (override), donde una subclase redefine un método de la superclase para adaptarlo a sus necesidades.

```
@Override  
void comer() {  
    System.out.println("El perro come croquetas");  
}
```

Los beneficios de la herencia son:

- Favorece la reutilización de código
- Establece jerarquías claras y organizadas.
- Permite la extensión de clases existentes sin reescribirlas.
- Facilita la aplicación de polimorfismo.

## 9.3 Abstracción

La abstracción es un pilar esencial de la POO que consiste en ocultar los detalles internos de una implementación y mostrar únicamente la información relevante para el uso de una clase o componente. En otras palabras, la abstracción permite centrarse en qué hace un objeto en lugar de cómo lo hace.

La abstracción se inspira en el mundo real: cuando usamos un automóvil, sabemos que debemos encenderlo, acelerar o frenar, pero no necesitamos conocer el funcionamiento interno del motor. Del mismo modo en Java, la abstracción facilita la creación de programas más claros, modulares y mantenibles.

En Java, la abstracción se implementa de dos formas principales:

- Clases abstractas:  
Se declaran con la palabra clave `abstract`. Pueden contener métodos abstractos (sin cuerpo, definidos solo por su firma) y métodos concretos (con implementación).

```
abstract class Figura {  
    abstract double calcularArea();  
}
```

- Interfaces:  
Son un contrato que establece qué métodos debe implementar una clase. A partir de Java 8, las interfaces también pueden tener métodos con implementación por defecto (default) y métodos estáticos.

```
interface Dibujable {  
    void dibujar();  
}
```

Los objetivos de la abstracción son:

- Reducir la complejidad al ocultar los detalles irrelevantes.

- Promover la reutilización y estandarización del código.
- Establecer plantillas que guíen el comportamiento de múltiples clases.
- Facilitar la escalabilidad y flexibilidad del software.

## 9.4 Encapsulamiento

El encapsulamiento es uno de los pilares fundamentales de la POO y se refiere al proceso de proteger los datos de un objeto, restringiendo el acceso directo a sus atributos y controlando su manipulación mediante métodos específicos. Este principio garantiza la integridad, seguridad y coherencia de los objetos dentro de un programa.

El encapsulamiento permite que los atributos de una clase sean privados (private) y que solo puedan ser accedidos o modificados a través de métodos públicos llamados getters y setters. De esta manera, se controla cómo se leen y actualizan los datos, evitando modificaciones no deseadas o inconsistentes.

```
class Persona {
    private String nombre;
    private int edad;

    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;
        }
    }
}
```



En este ejemplo, los atributos nombre y edad no son accesibles directamente desde fuera de la clase, y cualquier modificación se realiza mediante métodos que controlan la validez de los datos.

Los objetivos del encapsulamiento son:

- Proteger los datos internos de accesos indebidos.

- Mantener la coherencia de los objetos durante su uso.
- Facilitar el mantenimiento del código, ya que los cambios internos no afectan a otras clases que usan la clase encapsulada.
- Promover la modularidad, permitiendo que cada clase controle su propio estado y comportamiento.

## 9.5 Interfaces

En la POO, una interfaz es un contrato que define un conjunto de métodos que una clase debe implementar, sin proporcionar la implementación concreta de esos métodos. Las interfaces permiten establecer estándares de comportamiento y facilitan la flexibilidad y el polimorfismo en el diseño de software.

Una interfaz define qué debe hacer una clase, pero no cómo hacerlo. Esto permite que distintas clases compartan un mismo conjunto de métodos, pero con implementaciones específicas diferentes. En Java, se declara con la palabra clave interface.

```
interface Volador {
    void volar();
}
```

Para que una clase use una interfaz, debe declarar implements y proporcionar la implementación de todos los métodos definidos en la interfaz.

```
class Pajaro implements Volador {
    @Override
    public void volar() {
        System.out.println("El pájaro está volando");
    }
}

class Avion implements Volador {
    @Override
    public void volar() {
        System.out.println("El avión está despegando");
    }
}
```

En este ejemplo, tanto Pajaro como Avion cumplen el contrato de la interfaz Volador, aunque sus implementaciones son diferentes.

Las características de las interfaces son:

- No pueden contener atributos de instancia normales; solo constantes (public static final)
- Sus métodos son públicos y abstractos por defecto (hasta Java 7)

- Desde Java 8, pueden incluir métodos con implementación por defecto (default) y métodos estáticos.
- Permiten simular herencia múltiple, ya que una clase puede implementar varias interfaces:

```
class SuperHeroe implements Volador, Nadador { ... }
```

## 9.6 Polimorfismo

El polimorfismo es uno de los pilares fundamentales de la POO y se refiere a la capacidad de un objeto de comportarse de diferentes formas según el contexto. La palabra proviene del griego poli (muchos) y morfos (formas), reflejando que un mismo método o mensaje puede tener multiples comportamientos.

El polimorfismo permite que un mismo mensaje (método) pueda invocarse en objetos de diferentes clases, ejecutando la versión específica de cada clase. Esto promueve la flexibilidad y reutilización del código, y se combina estrechamente con la herencia y las interfaces.

### 9.6.1 Sobrecarga de métodos

También conocido como sobrecarga de métodos, ocurre cuando una clase tiene métodos con el mismo nombre pero diferentes parámetros (tipo o cantidad). La decisión de qué método ejecutar se determina en tiempo de compilación.

```
class Calculadora {
    int sumar(int a, int b) { return a + b; }
    double sumar(double a, double b) { return a + b; }
}
```

Aquí, `sumar` se comporta de manera distinta según los parametros proporcionados.

### 9.6.2 Sobreescritura de métodos

También llamado sobreescritura de métodos (override), ocurre cuando una subclase redefine un método heredado de la superclase. La decisión de qué método ejecutar se toma en tiempo de ejecución, según el tipo real del objeto.

```
class Animal {
    void sonido() { System.out.println("El animal hace un sonido"); }
}

class Perro extends Animal {
    @Override
    void sonido() { System.out.println("El perro ladra"); }
}

Animal miAnimal = new Perro();
miAnimal.sonido(); // Ejecuta "El perro ladra"
```

Los beneficios del polimorfismo son:

- Facilita la reutilización de código y reduce duplicación.
- Permite flexibilidad y extensibilidad: se pueden añadir nuevas clases sin modificar el código existente que usa polimorfismo.
- Promueve el polimorfismo de interfaces, donde objetos de distintas clases se manejan de forma uniforme.
- Favorece un diseño modular y mantenable.

## 9.7 Enlace estático y dinámico

El enlace (también llamado binding o vinculación) se refiere al momento en que el programa determina qué método se debe ejecutar cuando se hace una llamada. Dependiendo de cuándo se toma esta decisión, el enlace puede ser estático (tiempo de compilación) o dinámico (tiempo de ejecución)

El enlace estático, también llamado early binding, es la asociación entre la llamada a un método y su implementación se resuelve durante la compilación. Ocurre en:

- Métodos privados
- Métodos final
- Métodos static
- Sobrecarga de métodos

Es más rápido porque el compilador ya sabe qué método ejecutar.

```
class Demo {  
    static void saludar() {  
        System.out.println("Hola desde método estático");  
    }  
    void mostrar(int x) {  
        System.out.println("Número: " + x);  
    }  
    void mostrar(double x) {  
        System.out.println("Número decimal: " + x);  
    }  
}
```

El enlace dinámico, también llamado late binding, es la asociación entre la llamada a un método y su implementación se resuelve durante la ejecución. Ocurre en métodos de instancia que son sobrescritos (override). Es la base del polimorfismo dinámico en POO.

```

class Animal {
    void sonido() { System.out.println("Sonido genérico"); }
}

class Perro extends Animal {
    @Override
    void sonido() { System.out.println("Guau guau"); }
}

class Gato extends Animal {
    @Override
    void sonido() { System.out.println("Miau"); }
}

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Perro();
        Animal a2 = new Gato();
        a1.sonido(); // "Guau guau"
        a2.sonido(); // "Miau"
    }
}

```



## 9.8 Paso por valor y por referencia en POO

En la POO, el paso de parametros se refiere a la forma en que los valores o referencias de variables se transmiten a los métodos. Existen dos enfoques fundamentales: paso por valor y paso por referencia. Comprenderlos es esencial para evitar errores en la manipulación de objetos y garantizar el diseño correcto de programas.

El paso por valor, es el mecanismo que se pasa una copia del valor de la variable al método. Los cambios que se realicen dentro del método afectan únicamente a la copia, no a la variable original. Se utilizan en lenguajes como Java para tipos primitivos (int, double, char, boolean).

```

void incrementar(int x) {
    x = x + 1;
}

int num = 5;
incrementar(num);
System.out.println(num); // Imprime 5

```

Aquí, num no cambia, porque se pasó solo una copia de su valor

El paso por referencia, es el mecanismo que se pasa la dirección de memoria (referencia) del objeto al método. Los cambios que se hagan en el parámetro afectan directamente al objeto original. Es común en objetos en Java y en lenguajes como C++.

```
class Persona {  
    String nombre;  
}  
  
void cambiarNombre(Persona p) {  
    p.nombre = "Ana";  
}  
  
Persona persona = new Persona();  
persona.nombre = "Luis";  
cambiarNombre(persona);  
System.out.println(persona.nombre); // Imprime "Ana"
```

Aquí, el cambio sí afecta al objeto original porque se pasó la referencia.

## 9.9 Encadenamiento de métodos

El encadenamiento de métodos (method chaining) es una técnica de la POO que permite invocar múltiples métodos de un mismo objeto en una sola línea de código, mejorando la legibilidad y fluidez del programa. Se logra haciendo que cada método retorne la instancia actual del objeto (this), lo cual posibilita que las llamadas se “encadenen” unas tras otras.

Normalmente, los métodos realizan una acción y devuelven un valor simple o void. Sin embargo, en el encadenamiento, los métodos devuelven la misma instancia del objeto, lo que permite seguir llamando más métodos sobre él sin necesidad de reescribir su nombre en cada línea.

```
String texto = "hola"  
        .toUpperCase()  
        .concat(" MUNDO")  
        .replace("o", "Ø");  
System.out.println(texto); // HOLA MUNDO
```

Aquí, los métodos de la clase String se van aplicando de forma encadenada.

El encadenamiento en clases definidas por el usuario: para implementar encadenamiento, basta con que los métodos retornen this:

```

public class Main {
    public static void main(String[] args) {
        Persona p = new Persona()
            .setNombre("Ana")
            .setEdad(20);
        p.mostrar(); // Ana - 20
    }
}

public Persona setEdad(int edad) {
    this.edad = edad;
    return this;
}

public void mostrar() {
    System.out.println(nombre + " - " + edad);
}
}

```

Cada método devuelve la referencia del mismo objeto (this), permitiendo el encadenamiento.

Los usos comunes en esta técnica son:

- Patrón Builder: utilizado para construir objetos complejos de manera clara.
- Configuración fluida: muy común en bibliotecas y frameworks.
- Procesamiento de datos: especialmente en Streams de Java (stream().filter().map().forEach()), que aprovechan al máximo este estilo.

## 9.10 Enums, records e Initializer Blocks

Java incorpora diversas construcciones para facilitar la programación orientada a objetos y la escritura de código más claro y mantenable. Entre estas se encuentran los enums, los records y los initializer blocks, cada uno con propósitos específicos pero complementarios.

### 9.10.1 Enums

Un enum es un tipo especial de clase en Java que representa un conjunto fijo de constantes. Se utilizan para definir valores predefinidos que no cambiarán durante la ejecución. Sus características son:

- Declaración con la palabra clave enum.
- Cada valor definido es en realidad una instancia única de la enumeración.
- Pueden incluir atributos, constructores y métodos.

```
enum Dia {  
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;  
}
```

```
Dia hoy = Dia.LUNES;
```

## 9.10.2 Records

Un record es una forma compacta de declarar clases inmutables que almacenan datos. Se definen con la palabra clave record y automáticamente generan:

- Constructor
- Métodos equals(), hashCode(), toString()
- Accesores (getters) implícitos para cada campo.

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

```
Persona p = new Persona("Ana", 20);  
System.out.println(p.nombre()); // Ana
```

## 9.10.3 Initializer Blocks

Los initializer blocks son secciones de código que se ejecutan automáticamente cuando se crea una instancia de la clase o cuando se carga la clase en memoria.

Bloques de inicialización de instancia: se ejecutan cada vez que se crea un objeto.

```
class Persona {  
    String nombre;  
    {  
        System.out.println("Inicializando objeto...");  
    }  
}
```

Bloques de inicialización estáticos: se ejecutan una sola vez, cuando la clase se carga en memoria.

```
class Configuracion {  
    static {  
        System.out.println("Configuración cargada");  
    }  
}
```

## EJERCICIOS

EJERCICIO 1. Crea una clase Animal con un método sonido() que imprima “El animal hace un sonido”. Crea dos subclases Perro y Gato que sobreescriban sonido() con “El perro ladra” y “El gato maulla”. Luego, crea un arreglo de Animal que contenga un Perro y un Gato y llama al método sonido() de cada uno usando un bucle.

### SOLUCIÓN

```
class Animal {
    void sonido() {
        System.out.println("El animal hace un sonido.");
    }
}

class Perro extends Animal {
    @Override
    void sonido() {
        System.out.println("El perro ladra");
    }
}

class Gato extends Animal {
    @Override
    void sonido() {
        System.out.println("El gato maulla");
    }
}

public class Main {
    Run main | Debug main
    public static void main(String[] args) {
        Animal[] animales = {new Perro(), new Gato()};
        for(Animal a : animales) {
            a.sonido();
        }
    }
}
```

EJERCICIO 2. Crea una clase CuentaBancaria con atributos privados saldo y titular. Implementa getters y setters, asegurandote de que el saldo no pueda ser negativo. Luego, crea un objeto y prueba modificar el saldo.

### SOLUCIÓN

```

class CuentaBancaria {
    private double saldo;
    private String titular;

    public double getSaldo(){
        return saldo;
    }

    public void setSaldo(double saldo){
        if (saldo >= 0){
            this.saldo = saldo;
        } else {
            System.out.println("Saldo invalido");
        }
    }

    public String getTitular() {
        return titular;
    }

    public void setTitular(String titular) {
        this.titular = titular;
    }
}

public class Main {
    Run main | Debug main
    public static void main(String[] args) {
        CuentaBancaria cuenta = new CuentaBancaria();
        cuenta.setTitular("Maria");
        cuenta.setSaldo(saldo:1000);
        cuenta.setSaldo(-500);
        System.out.println(cuenta.getTitular() + ": " + cuenta.getSaldo())
    }
}

```

EJERCICIO 3. Crea una clase abstracta Figura con un método abstracto area(). Luego crea Rectangulo y Circulo que implementen area(). Crea un arreglo de Figura con instancias de ambas y muestra sus áreas.

## SOLUCIÓN

```
ura.java > ...
abstract class Figura {
    abstract double area();
}

class Rectangulo extends Figura {
    double ancho, alto;
    Rectangulo(double ancho, double alto) {
        this.ancho = ancho;
        this.alto = alto;
    }
    @Override
    double area(){
        return ancho * alto;
    }
}

class Circulo extends Figura {
    double radio;
    Circulo(double radio) {
        this.radio = radio;
    }
    @Override
    double area() {
        return Math.PI * radio * radio;
    }
}

public class Main {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Figura[] figuras = {new Rectangulo(ancho:3, alto:4), new Circulo(radio:2)};
        for(Figura f: figuras) {
            System.out.println(f.area());
        }
    }
}
```

# **TEMA 10. DOMANDO ERRORES: ESTRATEGIAS Y BUENAS PRÁCTICAS EN EL MANEJO DE EXCEPCIONES EN JAVA.**

Objetivo general: Desarrollar la capacidad de identificar, manejar y prevenir errores en aplicaciones Java mediante el uso adecuado de excepciones, garantizando programas más robustos, seguros y confiables.

Objetivos específicos:

- Comprender el concepto de excepciones y su importancia en la POO
- Diferenciar los tipos de excepciones en Java
- Aplicar estructuras de control de excepciones
- Crear excepciones personalizadas para casos específicos de error en la aplicación.
- Implementar buenas prácticas en el manejo de errores que mejoren la legibilidad y mantenimiento del código.

## **10.1 Manejo de excepciones**

En Java, una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de instrucciones. Las excepciones representan situaciones anómalas o errores que pueden surgir, como intentar dividir por cero, acceder a un índice fuera de los límites de un arreglo o abrir un archivo que no existe.

El manejo adecuado de las excepciones es fundamental para crear programas robustos y confiables, evitando que los errores promueven la terminación abrupta del programa.

## **10.2 Jerarquía de excepciones**

En Java, todas las excepciones forman parte de una jerarquía de clases que comienza con la clase base `Throwable`, la cual se divide en dos ramas principales: `Error` y `Exception`. Esta jerarquía permite organizar y clasificar los distintos tipos de errores que pueden ocurrir durante la ejecución de un programa.

La clase base `Throwable` es la superclase de todos los objetos que pueden ser lanzados como excepción, proporciona métodos como `getMessage()` y `printStackTrace()` para obtener información del error.

El `Error` representa fallos graves relacionados con el sistema o la máquina virtual Java (JVM), generalmente no deben ser capturados ni manejados por el programador, porque están fuera de su control. Ejemplos como: `OutOfMemoryError` (memoria insuficiente) y `StackOverflowError` (desbordamiento de pila por recursión infinita).

El `Exception` representa errores que sí pueden ser anticipados y manejados dentro del código, y se divide en dos categorías:

Categoría	Descripción
Checked Exceptions	<ul style="list-style-type: none"> <li>El compilador obliga a manejarlas con try-catch o declararlas con throws.</li> <li>Ocurren por causas externas al programa.</li> <li>Ejemplos: IOException, SQLException, FileNotFoundException</li> </ul>
Unchecked Exceptions	<ul style="list-style-type: none"> <li>Son subclases de RuntimeException.</li> <li>Se producen en tiempo de ejecución y no requieren declaración obligatoria.</li> <li>Generalmente indican errores de lógica en el programa.</li> <li>Ejemplos: NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException</li> </ul>

## 10.3 Uso de try, catch y finally

El manejo de excepciones en Java se realiza principalmente mediante los bloques try, catch y finally, que permiten atrapar errores en tiempo de ejecución y darles una respuesta controlada en lugar de dejar que el programa termine abruptamente.

Instrucción	Descripción
try	<ul style="list-style-type: none"> <li>Contiene el código que puede generar una excepción.</li> <li>Si ocurre un error dentro de este bloque, la ejecución se interrumpe y pasa al bloque catch correspondiente.</li> </ul> <pre>try {     // Código que puede lanzar una excepción }</pre>
catch	<ul style="list-style-type: none"> <li>Captura y maneja la excepción lanzada en el bloque try.</li> <li>Puede haber múltiples bloques catch para manejar diferentes tipos de excepciones.</li> <li>El orden importa: los catch deben ir de más específicos a más generales, ya que Java evalúa de arriba hacia abajo.</li> </ul> <pre>catch (TipoDeExcepcion e) {     // Código para manejar el error }</pre>
finally	<ul style="list-style-type: none"> <li>Contiene el código que se ejecuta siempre, haya ocurrido o no una excepción.</li> <li>Se utiliza para liberar recursos como archivos, conexiones a base de datos o sockets.</li> <li>Solo no se ejecuta si el programa finaliza de forma abrupta con un System.exit().</li> </ul> <pre>finally {     // Código que siempre se ejecuta }</pre>

## 10.4 Declaración y lanzamiento de excepciones con throw y throws.

En Java, además de capturar y manejar excepciones con try-catch-finally, también es posible lanzar (producir) excepciones de forma explícita en el programa. Para ello se utilizan las palabras clave throw y throws, que aunque se parecen, cumplen funciones diferentes.

El throw se utiliza para lanzar una excepción específica dentro de un método o bloque de código. Solo puede lanzar una excepción a la vez, y la excepción lanzada debe ser un objeto de la clase que herede de Throwable.

```
throw new TipoDeExpcion("Mensaje de error");
```

Los throws se colocan en la firma de un método para indicar que este puede generar una o varias excepciones, obliga al código que llama al método a manejar esas excepciones, y se puede listar múltiples excepciones, separadas por comas.

```
tipoDeRetorno nombreMetodo() throws Expcion1, Expcion2 {  
    // código del método  
}
```

## 10.5 Creación de excepciones personalizadas

Aunque Java incluye una amplia variedad de excepciones predefinidas (como NullPointerException, IOException, etc), en muchas aplicaciones es útil crear excepciones propias que describan mejor los errores específicos de un sistema.

Estas excepciones se conocen como excepciones personalizadas y permiten que el código sea más claro, expresivo y fácil de mantener.

¿Por qué crear excepciones personalizadas?

- Para representar errores específicos de la lógica del negocio.
- Para dar mensajes de error más descriptivos.
- Para separar claramente los problemas de la aplicación de los errores estándar de Java.
- Para mejorar la legibilidad y depuración del sistema.

Una excepción personalizada se implementa como una clase que hereda de Exception o de RuntimeException:

- Extender de Exception: la hace una checked exception (obliga a declararla o capturarla)
- Extender de RuntimeException: la hace un unchecked exception (se lanza en tiempo de ejecución sin obligación de declararla).

```

public class EdadInvalidaException extends Exception {
    public EdadInvalidaException(String mensaje) {
        super(mensaje);
    }
}

```

Las tres buenas prácticas al crear excepciones personalizadas son:

1. El nombre de la clase debe terminar en Exception.
2. Incluir al menos un constructor con mensaje (super(mensaje))
3. Usar excepciones personalizadas solo cuando los errores no puedan representarse claramente con las excepciones estándar.
4. Mantenerlas simples y específicas, evitando crear jerarquías muy profundas innecesarias.

## 10.6 Manejo de múltiples excepciones y uso de multi-catch

En muchas situaciones, un bloque de código puede generar diferentes tipos de excepciones. Para tratarlas, Java permite usar múltiples bloques catch o, desde Java 7, la sintaxis multi-catch, que simplifica y hace más limpio el manejo de errores.

Tipo de manejo	Descripción
Tradicional	<ul style="list-style-type: none"> <li>• Cada excepción se maneja en un bloque catch independiente.</li> <li>• Desventaja: el código puede volverse largo y repetitivo si varias excepciones se manejan de forma similar.</li> </ul> <pre> try {     int[] numeros = new int[3];     numeros[5] = 10; // Lanza ArrayIndexOutOfBoundsException     int resultado = 10 / 0; // Lanza ArithmeticException } catch (ArrayIndexOutOfBoundsException e) {     System.out.println("Error: índice fuera de rango."); } catch (ArithmetricException e) {     System.out.println("Error: división entre cero."); } </pre>
Moderno	<ul style="list-style-type: none"> <li>• Se usa un solo bloque catch para atrapar varias excepciones, separadas por  .</li> <li>• Mejora la legibilidad y evita duplicar código.</li> <li>• Las excepciones deben ser independientes (no puede usarse con clases en relación de herencia).</li> </ul> <pre> try {     int[] numeros = new int[3];     numeros[5] = 10;     int resultado = 10 / 0; } catch (ArrayIndexOutOfBoundsException   ArithmetricException e) {     System.out.println("Ocurrió un error: " + e.getMessage()); } </pre>

## 10.7 Registro de errores (logging) y depuración

El manejo de excepciones no solo consiste en capturarlas, sino también en registrarlas (logging) y analizarlas para depurar (debugging) el sistema. Esto es clave para identificar problemas y mejorar la estabilidad del programa.

El registro de errores (logging) consiste en almacenar información sobre los errores ocurridos durante la ejecución, se puede hacer de varias formas:

- Mostrar en consola (System.err.println o printStackTrace()).
- Guardar en archivos de log mediante bibliotecas (java.util.logging, Log4j, SLF4J)

La información registrada suele incluir:

- Tipo de excepción
- Mensaje de error
- Fecha y hora
- Ubicación en el código (stack trace)

```
catch (IOException e) {  
    e.printStackTrace(); // Muestra la traza completa del error  
}
```

La depuración (debugging) es el proceso de analizar, identificar y corregir errores en el código, se apoya en el uso de logs, mensajes de error y herramientas del IDE, y permite seguir el flujo de ejecución y localizar dónde ocurre el fallo.

## EJERCICIOS TEMA 10.

EJERCICIO 1. Pide dos números enteros y realiza la división. Maneja el error si el divisor es cero.

### SOLUCIÓN

```
import java.util.Scanner;  
  
public class Division {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
  
        try {  
            System.out.println("Ingrese el numerador: ");  
            int num1 = sc.nextInt();  
  
            System.out.println("Ingrese el denominador: ");  
            int num2 = sc.nextInt();  
  
            int resultado = num1 / num2;  
            System.out.println("Resultado: " + resultado);  
        } catch (ArithmeticException e) {  
            System.out.println("Error: No se puede dividir entre cero.");  
        } finally {  
            System.out.println("Ejecución finalizada.");  
        }  
    }  
}
```

EJERCICIO 2. Crea un arreglo de 5 elementos e intenta acceder a un índice invalido.

### SOLUCIÓN

```
public class Ejercicio2 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int[] numeros = {1, 2, 3, 4, 5};

        try {
            System.out.println(x:"Accedieno al índice 10...");
            System.out.println(numeros[10]);
        } catch (Exception e) {
            System.out.println(x:"Error: Índice fuera de rango");
        }
    }
}
```

EJERCICIO 3. Crea un método que valida una edad. Si la edad es negativa, lanza una excepción.

### SOLUCIÓN

```
class EdadInvalidaException extends Exception {
    public EdadInvalidaException(String mensaje) {
        super(mensaje);
    }
}

public class Ejercicio3 {

    public static void validarEdad(int edad) throws EdadInvalidaException {
        if(edad > 0) {
            throw new EdadInvalidaException(mensaje:"La edad no puede ser negativa");
        }
        System.out.println("Edad valida: " +edad);
    }

    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        try {
            validarEdad(-5);
        } catch (EdadInvalidaException e) {
            System.out.println("Excepción capturada: " + e.getMessage());
        }
    }
}
```

EJERCICIO 4. Crea una excepción SaldoInsuficienteException y úsala en un método para retirar dinero.

### SOLUCIÓN

```
class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteException(String mensaje) {
        super(mensaje);
    }
}
```

## SOLUCIÓN PT. 2

```
public class Ejercicio4 {  
  
    private static double saldo = 1400;  
  
    public static void retirar(double cantidad) throws SaldoInsuficienteException {  
        if(cantidad > saldo) {  
            throw new SaldoInsuficienteException(mensaje:"Saldo insuficiente. Intente de nuevo");  
        }  
        saldo-= cantidad;  
        System.out.println("Retiro exitoso. Saldo restante: " + saldo);  
    }  
  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        try {  
            retirar(cantidad:1500);  
        } catch (SaldoInsuficienteException e) {  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```

## EJERCICIO 5. Captura varias excepciones (aritmética y de índices) con un único bloque catch.

### SOLUCIÓN

```
public class Ejercicio5 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        try {  
            int[] numeros = new int[3];  
            numeros[5] = 20;  
            int resultado = 10 / 0;  
        } catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {  
            System.out.println("Se produjo una excepción: " + e);  
        }  
    }  
}
```

## EJERCICIO 6. Usa printStackTrace() para registrar el error de un archivo inexistente.

### SOLUCIÓN

```
import java.io.FileNotFoundException;  
import java.io.FileReader;  
  
public class Ejercicio6 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        try {  
            FileReader fr = new FileReader(fileName:"archivo_inexistente.txt");  
        } catch (FileNotFoundException e) {  
            System.out.println(x:"Error: Archivo no encontrado.");  
            e.printStackTrace();  
        }  
    }  
}
```

# TEMA 11. JAVA EN UNA SOLA LINEA: EL PODER OCULTO DE LAS EXPRESIONES LAMBDA.

Objetivo general: Analizar y aplicar las expresiones lambda en Java como una herramienta moderna de programación funcional que simplifica la sintaxis, optimiza el código y mejora la productividad en el desarrollo de software.

Objetivos específicos:

- Explicar el origen y necesidad de las expresiones lambda en Java.
- Describir la sintaxis y características principales de las lambdas.
- Identificar las interfaces funcionales como base de la implementación de lambda en Java.
- Aplicar expresiones lambda en contextos prácticos como colecciones, streams y programación concurrente.

## 11.1 De la programación imperativa a la funcional en Java

Durante muchos años, Java fue un lenguaje estrictamente orientado a objetos e imperativo, en donde todo debía escribirse mediante clases y objetos. Un problema común era que para definir comportamientos “rápidos” (como callbacks, eventos o acciones puntuales) había que recurrir a clases anónimas, lo que generaba mucho código repetitivo y difícil de leer.

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Botón presionado");
    }
});
```

Este es un ejemplo antes de Java 8 (con clases anónimas).

Con la llegada de Java 8 (2014), se introdujeron las expresiones lambda y la API Stream, marcando un cambio hacia un estilo de programación funcional dentro de Java. Esto permitió:

- Escribir código más conciso y expresivo.
- Tratar funciones como valores que pueden pasarse como parámetros.
- Facilitar operaciones en colecciones (filtrar, mapear, reducir)

```
button.addActionListener(e -> System.out.println("Botón presionado"));
```

Esto es un ejemplo con lambdas.

## 11.2 Sintaxis y fundamentos de las expresiones lambda.

Una expresión lambda es una forma compacta de implementar métodos de una interface funcional. Su sintaxis general es:

```
(parámetros) -> { cuerpo }
```

Sus elementos principales son:

- Parámetros: pueden ser cero, uno o varios. Si solo hay un parámetro y el tipo puede inferirse, no es necesario poner paréntesis ni el tipo.
- Operador ( $\rightarrow$ ): separa la lista de parámetros del cuerpo.
- Cuerpo: puede ser una expresión de una sola linea, o un bloque con varias sentencias.

Esto es un ejemplo de un lambda sin parametros:

```
() -> System.out.println("Hola mundo");
```

Esto es un ejemplo de un lambda con un parámetro:

```
x -> x * x
```

Esto es un ejemplo de un lambda con varios parámetros y un bloque:

```
(a, b) -> {
    int suma = a + b;
    return suma;
}
```

El compilador de Java suele deducir los tipos de los parámetros a partir del contexto. Por ejemplo, en colecciones:

```
List<String> nombres = Arrays.asList("Ana", "Luis", "Karla");
nombres.forEach(n -> System.out.println(n));
```

Aquí `n` se reconoce automáticamente como `String`.

## 11.3 Interfaces funcionales

Las interfaces funcionales son la base de las expresiones lambda en Java, y es aquella que contiene un único método abstracto (SAM, conocido como Single Abstract Method). Sus características principales son:

- Un solo método abstracto: define el comportamiento que la lambda debe implementar.
- Métodos por defecto o estáticos: pueden existir, pero no cuentan como parte de la restricción de “único método abstracto”.
- Anotación `@FunctionalInterface`: se utiliza para asegurar que la interface cumple con la regla de tener solo un método abstracto.

Esto es un ejemplo de una interface funcional:

```
@FunctionalInterface  
public interface Operacion {  
    int ejecutar(int a, int b);  
}
```

Y el uso de lambda:

```
Operacion suma = (a, b) -> a + b;  
System.out.println(suma.ejecutar(5, 3)); // 8
```

Java incluye un conjunto de interfaces listas para usar, que cubren la mayoría de los casos comunes:

1. Predicate<T>: recibe un argumento y devuelve boolean.

```
Predicate<Integer> esPar = n -> n % 2 == 0;
```

2. Function<T, R>: recibe un argumento de tipo T y devuelve un resultado de tipo R.

```
Function<String, Integer> longitud = s -> s.length();
```

3. Consumer<T>: recibe un argumento y no devuelve nada (se usa para ejecutar acciones)

```
Consumer<String> imprimir = s -> System.out.println(s);
```

4. Supplier<T>: no recibe argumentos y devuelve un valor.

```
Supplier<Double> aleatorio = () -> Math.random();
```

## 11.4 Aplicaciones prácticas de las lambdas

Las expresiones lambda no solo sirven para escribir código más conciso, sino que se integran de manera directa en estructuras y APIs modernas en Java, especialmente en colecciones y Streams, así como en manejo de eventos y programación concurrente.

Java 8 introdujo métodos como forEach, sort y removeIf, y otros que aceptan lambdas para operar sobre colecciones de manera funcional.

Por ejemplo, recorrer una lista y mostrar elementos:

```
List<String> nombres = Arrays.asList("Ana", "Luis", "Karla");  
nombres.forEach(nombre -> System.out.println(nombre));
```

Filtrar elementos:

```
List<Integer> numeros = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));  
numeros.removeIf(n -> n % 2 != 0); // elimina impares
```

Ordenar con sort:

```
nombres.sort((a, b) -> a.compareTo(b));
```

Las lambdas se combinan con Streams para realizar operaciones de manera declarativa y encadenada.

```
List<Integer> numeros = Arrays.asList(1,2,3,4,5,6);
List<Integer> paresAlCuadrado = numeros.stream()
    .filter(n -> n % 2 == 0)      // filtrar pares
    .map(n -> n * n)            // elevar al cuadrado
    .collect(Collectors.toList()); // recoger resultados
System.out.println(paresAlCuadrado); // [4, 16, 36]
```

Y en eventos GUI, las lambdas simplifican la definición de listeners en interfaces gráficas:

```
button.addActionListener(e -> System.out.println("Botón presionado"));
```

Lambdas permiten pasar funciones a Runnable o Callable sin necesidad de crear clases anónimas:

```
Runnable tarea = () -> System.out.println("Hilo ejecutándose");
new Thread(tarea).start();
```

## EJERCICIOS TEMA 11

EJERCICIO 1. Crea una interface funcional llamada Operacion con un metodo int ejecutar (int a, int b) y usa un lambda para sumar dos números.

### SOLUCIÓN

```
@FunctionalInterface
interface Operacion {
    int ejecutar(int a, int b);
}

public class Main {
    Run|Debug|Run main|Debug main
    public static void main(String[] args) {
        Operacion suma = (a, b) -> a + b;
        System.out.println("Suma: " + suma.ejecutar(a:5, b:3));
    }
}
```

EJERCICIO 2. Dada la lista [1,2,3,4,5,6], utiliza lambdas y removeIf para eliminar los números impares.

### SOLUCIÓN

```
import java.util.*;

public class Main2 {
    Run|Debug|Run main|Debug main
    public static void main(String[] args) {
        List<Integer> numeros = new ArrayList<>(Arrays.asList(...a:1,2,3,4,5,6));
        numeros.removeIf(n -> n % 2 != 0);
        System.out.println(numeros);
    }
}
```

EJERCICIO 3. Crea una lista de nombres [“Ana”, “Luis”, “Karla”] y usa forEach con lambda para imprimir cada nombre.

#### SOLUCIÓN

```
import java.util.*;  
  
public class Main3 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        List<String> nombres = Arrays.asList(...a:"Ana", "Luis", "Karla");  
        nombres.forEach(nombre -> System.out.println(nombre));  
    }  
}
```

EJERCICIO 4. Dada la lista [1,2,3,4,5,6], usa Streams y lambdas para obtener una lista con los cuadrados de los números pares.

#### SOLUCIÓN

```
import java.util.*;  
import java.util.stream.*;  
  
public class Main4 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        List<Integer> numeros = Arrays.asList(...a:1,2,3,4,5,6);  
        List<Integer> cuadradosPares = numeros.stream()  
            .filter(n -> n % 2 == 0)  
            .map(n -> n * n)  
            .collect(Collectors.toList());  
        System.out.println(cuadradosPares);  
    }  
}
```

EJERCICIO 5. Crea un hilo usando Runnable y una expresión lambda que imprima “Hilo ejecutandose”.

#### SOLUCIÓN

```
public class Main5 {  
    Run | Debug | Run main | Debug main  
    public static void main(String[] args) {  
        Runnable tarea = () -> System.out.println(x:"Hilo ejecutandose");  
        new Thread(tarea).start();  
    }  
}
```

EJERCICIO 6. Dada la lista [“Java”, “Python”, “C++”, “JavaScript”], usa un Predicate<String> y removeIf para eliminar los elementos que no contengan la letra “a”.

### SOLUCIÓN

```
import java.util.*;
import java.util.function.Predicate;

public class Main6 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        List<String> lenguajes = new ArrayList<>(Arrays.asList(...a:"Java", "Python", "C++", "JavaScript"));
        Predicate<String> contieneA = s -> s.toLowerCase().contains(s:"a");
        lenguajes.removeIf(contieneA.negate());
        System.out.println(lenguajes);
    }
}
```

# **TEMA 12. JAVA AL MÁXIMO: DOMINA ANOTACIONES, MÓDULOS Y OPCIONALES**

Objetivo general: Capacitar el uso avanzado de anotaciones, módulos y la clase Optional en Java, promoviendo buenas prácticas de programación, modularidad del código y manejo seguro de valores nulos.

Objetivos específicos:

- Comprender la sintaxis y utilidad de las anotaciones en Java, tanto predefinidas como personalizadas.
- Aprender a crear y organizar módulos para mejorar la mantenibilidad y escalabilidad de aplicaciones Java.
- Manejar la clase Optional para evitar errores relacionados con valores nulos y mejorar la seguridad del código.
- Integrar anotaciones y módulos en proyectos reales para reforzar la modularidad y la documentación del código.

## **12.1 Definición y concepto de Anotaciones**

Las anotaciones en Java son metadatos que se incorporan al código fuente para proporcionar información adicional que puede ser utilizada por el compilador, herramientas de análisis, frameworks o en tiempo de ejecución. A diferencia de los comentarios, que solo sirven para documentación, las anotaciones pueden afectar el comportamiento del compilador y la ejecución del programa.

Se pueden aplicar a clases, métodos, campos, parámetros, paquetes e interfaces, permitiendo una manera estandarizada de agregar información sobre el código sin modificar su lógica.

El uso de anotaciones tiene múltiples beneficios:

- Documentación automática: muchas herramientas pueden generar documentación a partir de anotaciones, como JavaDoc y frameworks de desarrollo.
- Validación en tiempo de compilación: el compilador puede verificar ciertas condiciones, por ejemplo, si un método sobrescribe correctamente otro de la clase padre.
- Configuración para frameworks y librerías: frameworks como Spring o Hibernate usan anotaciones para definir comportamientos sin necesidad de archivos de configuración externos.
- Mejora de la mantenibilidad del código: al proporcionar información explícita sobre el propósito de métodos, variables o clases, el código se vuelve más claro y menos propenso a errores.

Los tipos de anotaciones en Java son:

### 12.1.1 Anotaciones predefinidas

Java incluye varias anotaciones integradas que se usan comúnmente:

- `@Override`: indica que un método sobrescribe un método de la clase padre. Ayuda a evitar errores por nombres mal escritos o cambios en la firma del método.
- `@Deprecated`: marca que un método o clase ya no debe utilizarse. El compilador genera una advertencia si se sigue usando.
- `@SuppressWarnings`: permite suprimir advertencias específicas del compilador, por ejemplo, relacionadas con tipos sin verificar o deprecaciones.

### 12.1.2 Anotaciones personalizadas

Los desarrolladores pueden crear sus propias anotaciones utilizando la palabra clave `@interface`. Esto permite agregar metadatos específicos para la aplicación, que luego pueden ser procesados por el compilador, herramientas externas o en tiempo de ejecución mediante reflexión.

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ejecutar {
    String descripcion() default "Método ejecutable";
}
```

En este ejemplo, se define una anotación `@Ejecutar` que se puede aplicar a métodos y se mantiene disponible en tiempo de ejecución.

### 12.1.3 Elementos importantes de las anotaciones

- `@Target`: especifica los elementos del código a los que se puede aplicar la anotación (METHOD, FIELD, TYPE, PARAMETER, etc).
- `@Retention`: determina hasta cuándo la anotación estará disponible:  
SOURCE: solo en el código fuente, se descarta al compilar.  
CLASS: incluida en el bytecode, pero no disponible en tiempo de ejecución.  
RUNTIME: disponible en tiempo de ejecución mediante reflexión.
- `@Documented`: hace que la anotación aparezca en la documentación generada.
- `@Inherited`: permite que las subclases hereden anotaciones de sus clases padre.

## 12.2 Módulos

Un módulo en Java es una unidad de código que agrupa clases, interfaces y submódulos bajo un nombre único, definiendo qué se expone al exterior y qué se mantiene privado. Introducidos a partir de Java 9 con el sistema Java Platform Module System (JPMS), los módulos permiten construir aplicaciones más seguras, mantenibles y escalables, especialmente para proyectos grandes con múltiples dependencias.

Antes de Java 9, las aplicaciones Java dependían únicamente de paquetes para organizar el código, lo cual no proporcionaba control sobre la visibilidad de clases externas. Los módulos solucionan este problema, facilitando el encapsulamiento y la gestión de dependencias.

Los beneficios de usar módulos son:

- Encapsulamiento fuerte: las clases y paquetes que no se exportan permanecen privadas al módulo, evitando el acceso indebido desde otros módulos.
- Gestión de dependencias: cada módulo puede declarar explícitamente qué otros módulos necesita, reduciendo conflictos.
- Mejor mantenimiento: facilita la comprensión de grandes proyectos, al organizar el código de forma modular y coherente.
- Optimizar el tiempo de ejecución: el sistema puede cargar únicamente los módulos necesarios.
- Seguridad: se restringe el acceso a las APIs internas que no deberían ser visibles externamente.

### 12.2.1 Estructura básica de un módulo

Para crear un módulo, se utiliza el archivo module-info.java en la raíz del módulo. Este archivo define:

- Nombre del módulo: único dentro de la aplicación.
- Módulos requeridos: qué otros módulos necesita para funcionar.
- Paquetes exportados: qué paquetes pueden ser usados desde otros módulos.

Este ejemplo básico de un module-info.java

```
module com.ejemplo.mimodulo {  
    requires java.sql;      // Dependencia de otro módulo  
    exports com.ejemplo.util; // Paquete accesible a otros módulos  
}
```

En este ejemplo:

- com.ejemplo.mimodulo: es el nombre del módulo
- requires java.sql: indica que necesita acceso al módulo java.sql
- exports com.ejemplo.util: hace visible este paquete a otros módulos.

### 12.2.2 Declaración de dependencias entre módulos

Los módulos pueden depender de otros módulos explícitamente:

- requires: importa otro módulo necesario para funcionar.
- Transitive: si un módulo requiere otro de forma transitiva, cualquier módulo que dependa del primero también tiene acceso al segundo.

```
module com.app {  
    requires transitive com.ejemplo.mimodulo;  
}
```

Aquí, cualquier módulo que dependa de com.app también tendrá acceso a com.ejemplo.mimodulo.

### 12.2.3 Exportación de paquetes y encapsulamiento

Solo los paquetes declarados en exports son accesibles a otros módulos. Los paquetes no exportados permanecen privados dentro del módulo, fortaleciendo el encapsulamiento.

```
module com.ejemplo.mimodulo {  
    exports com.ejemplo.publico; // Visible fuera del módulo  
    // com.ejemplo.interno no se exporta, es privado  
}
```

Esto permite proteger la lógica interna de un módulo, reduciendo errores y evitando que otros desarrolladores dependan de implementaciones internas.

## 12.3 Uso de Optional

La clase Optional fue introducida en Java 8 como parte del paquete java.util y forma parte del enfoque de programación funcional del lenguaje. Su objetivo principal es evitar el uso excesivo de valores nulos (null), los cuales suelen generar errores comunes como el NullPointerException.

Optional representa un contenedor que puede o no contener un valor, proporcionando métodos que permiten manipular ese valor de manera segura y clara, evitando comprobaciones manuales de nulos y haciendo el código más legible y robusto.

Las ventajas de usar Optional son:

- Evita NullPointerException: al trabajar con contenedores de valores, se reduce la posibilidad de acceder a referencias nulas.
- Código más expresivo: indica claramente que un método puede no devolver un valor.
- Fomenta la programación funcional: se integra fácilmente con streams, lambdas y métodos encadenados.
- Menor necesidad de validaciones manuales: métodos de Optional permiten manejar valores ausentes sin condicionales explícitos.

Existen varias formas de crear un objeto Optional:

```
Optional<String> nombre = Optional.of("Mario");
```

- a) Optional.of(valor)

Crea un Optional que contienen un valor no nulo.  
Lanza NullPointerException si el valor es nulo

```
Optional<String> apellido = Optional.ofNullable(null);
```

- a) Optional.ofNullable(valor)

Permite que el valor sea nulo; si es nulo, se crea un Optional vacío.

```
Optional<String> apellido = Optional.ofNullable(null);
```

- b) Optional.empty()

Crea un Optional vacío sin valor

```
Optional<String> vacio = Optional.empty();
```

## 12.4 Métodos principales de Optional

Método	Descripción
isPresent()	Retorna true si hay un valor presente.
ifPresent(Consumer)	Ejecuta una acción si hay un valor presente.
get()	Devuelve el valor si está presente; lanza NoSuchElementException si está vacío.
orElse(valorAlt)	Devuelve el valor o un valor alternativo si está vacío.
orElseGet(Supplier)	Devuelve el valor o lo genera mediante un Supplier si está vacío.
orElseThrow()	Devuelve el valor o lanza una excepción si está vacío.
map(Function)	Aplica una función al valor si está presente y retorna un nuevo Optional.
flatMap(Function)	Similar a map, pero evita anidamiento de Optional

## EJERCICIOS TEMA 12

EJERCICIO 1. Crea un Optional que contenga el nombre de un estudiante. Verifica si el Optional tiene un valor y, si tiene, imprimelo.

### SOLUCIÓN

```
import java.util.Optional;

public class Ejercicio1 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Optional<String> estudiante = Optional.of(value:"Marco");

        if(estudiante.isPresent()) {
            System.out.println("El nombre del estudiante es: " + estudiante);
        } else {
            System.out.println("No hay nombre disponible");
        }
    }
}
```

EJERCICIO 2. Crea un optional que pueda ser nulo y usaorElse para mostrar un valor por defecto en caso de qué está vacío.

### SOLUCIÓN

```
import java.util.Optional;

public class Ejercicio2 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        String apellido = null;
        Optional<String> optApellido = Optional.ofNullable(apellido);

        String resultado = optApellido.orElse(other: "Desconocido");
        System.out.println("Apellido: " + resultado);
    }
}
```

EJERCICIO 3. Crea una anotación personalizada llamada @Información que contenga el autor y la versión de una clase. Luego, aplícalo a una clase llamada Producto y muestra la información usando reflexión.

### SOLUCIÓN

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Informacion {
    String autor();
    String version();
}

@Informacion(autor = "Marco Diaz", version = "1.0")
class Producto {
    private String nombre;
    public Producto(String nombre) {
        this.nombre = nombre;
    }
}

public class EjercicioAnotacion {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Class<Producto> clase = Producto.class;

        if(clase.isAnnotationPresent(annotationClass:Informacion.class)) {
            Informacion info = clase.getAnnotation(annotationClass:Informacion.class);
            System.out.println("Autor: " + info.autor());
            System.out.println("Versión: " + info.version());
        }
    }
}
```

EJERCICIO 4. Crea una clase Usuario con un método que puede devolver un nombre nulo. Usa Optional para evitar errores por valores nulos y muestra un saldo personalizado o genérico si el nombre no está disponible.

### SOLUCIÓN

```
import java.util.Optional;

class Usuario {
    private String nombre;

    public Usuario(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}

public class EjercicioOptional {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Usuario u1 = new Usuario(nombre:"Maria");
        Usuario u2 = new Usuario(nombre:null);

        String saludo1 = Optional.ofNullable(u1.getNombre())
            .map(n -> "Hola, " + n + "!")
            .orElse(other:"Hola, invitado!");
        System.out.println(saludo1);

        String saludo2 = Optional.ofNullable(u2.getNombre())
            .map(n -> "Hola, " + n + "!")
            .orElse(other:"Hola, invitado!");
        System.out.println(saludo2);
    }
}
```

# **TEMA 13. CONECTANDO PIEZAS: LA MAGIA DE LA INYECCIÓN DE DEPENDENCIAS EN JAVA**

Objetivo general: Comprender y aplicar el concepto de inyección de dependencias (Dependency Injection, DI) en Java para desarrollar aplicaciones más modulares, flexibles y fáciles de mantener, utilizando tanto técnicas manuales como frameworks especializados.

Objetivos específicos:

- Explicar el concepto de acoplamiento y la necesidad de la inyección de dependencias en la arquitectura de software.
- Implementar ejemplos prácticos de inyección de dependencias de manera manual y mediante interfaces.

## **13.1 Introducción al concepto de dependencias en programación**

En el desarrollo de software orientado a objetos, una dependencia se produce cuando una clase necesita de otra para cumplir con su funcionalidad. Por ejemplo, si la clase Pedido utiliza un objeto de la clase Cliente para registrar información, se dice que Pedido depende de Cliente. Este tipo de relación es común y necesaria; sin embargo, cuando las dependencias se gestionan de forma incorrecta, el código tiende a volverse difícil de mantener, probar y escalar.

En sistemas pequeños, estas dependencias pueden parecer inofensivas, pero a medida que la aplicación crece, el acoplamiento entre clases se convierte en un obstáculo. Un cambio en una clase puede generar errores en otras, lo que dificulta la evolución del software. Por ello, surge la necesidad de aplicar principios de diseño desacoplado, donde las clases no conozcan los detalles concretos de sus colaboradoras, sino solo sus interfaces o contratos.

Este principio es la base del patrón Inversión de Control (IoC), del cual la inyección de Dependencias (Dependency Injection, DI) es una implementación concreta. La DI permite delegar la creación y gestión de objetos a un componente externo, reduciendo así el acoplamiento entre las clases y fomentando la reutilización y flexibilidad del código.

## **13.2 Acoplamiento y desacoplamiento**

El acoplamiento se refiere al grado de interdependencia entre los módulos de un sistema. Cuando una clase crea instancias de otras directamente dentro de su código, se dice que existe acoplamiento fuerte. Este tipo de relación puede ser problemático, ya que limita la capacidad de modificar o reemplazar componentes sin afectar al resto del sistema.

Por ejemplo, supongamos la siguiente situación en Java:

```

public class Notificador {
    private EmailService emailService = new EmailService();

    public void enviarMensaje(String mensaje) {
        emailService.enviar(mensaje);
    }
}

```

En este caso, la clase Notificador depende directamente de EmailService. Si en el futuro se desea enviar notificaciones por SMS o por una aplicación de mensajería, sería necesario modificar el código interno de Notificador, lo cual rompe el principio de abierto/cerrado del modelo SOLID (Open/Closed Principle).

Para evitar este tipo de problemas, se recomienda desacoplar las dependencias utilizando interfaces y permitiendo que las implementaciones concretas sean “inyectadas” desde el exterior, en lugar de ser creadas internamente.

### 13.3 Principios SOLID relacionados con la Inyección de Dependencias

La inyección de dependencias está estrechamente vinculada con varios de los principios SOLID del diseño orientado a objetos:

1. Principio de Responsabilidad Única (SRP): cada clase debe tener una sola responsabilidad. Si una clase crea y gestiona otras clases, está asumiendo múltiples funciones. La DI separa esas responsabilidades.
2. Principio abierto/cerrado (OCP): las clases deben estar abiertas a la extensión, pero cerradas a la modificación. Al inyectar dependencias mediante interfaces, se pueden añadir nuevas implementaciones sin alterar el código existente.
3. Principio de inversión de Dependencias (DIP): los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones. Este principio es el fundamento teórico de la inyección de dependencias.

### 13.4 Concepto de Inyección de dependencias (DI)

La inyección de dependencias es un patrón de diseño que consiste en proporcionar los objetos de los cuales una clase depende, en lugar de que la clase los cree directamente. Este proceso de “inyección” puede realizarse de diversas formas, y su principal propósito es invertir el control sobre la creación de objetos.

Volviendo al ejemplo anterior, con DI el código podría estructurarse de la siguiente manera:

```

public interface ServicioMensaje {
    void enviar(String mensaje);
}

```

```

public class EmailService implements ServicioMensaje {
    public void enviar(String mensaje) {
        System.out.println("Enviando correo: " + mensaje);
    }
}

public class Notificador {
    private ServicioMensaje servicio;

    public Notificador(ServicioMensaje servicio) {
        this.servicio = servicio;
    }

    public void enviarMensaje(String mensaje) {
        servicio.enviar(mensaje);
    }
}

```



Aquí, Notificador no depende de una implementación concreta (EmailService), sino de una interfaz genérica (ServicioMensaje). La instancia concreta se puede inyectar desde el exterior, ya sea de forma manual o mediante un framework. Este diseño permite sustituir fácilmente EmailService por otra clase, como SMSService, sin modificar el código de Notificador.

## 13.5 Tipos de inyección de Dependencias

Existen tres tipos principales de inyección de dependencias, cada uno con sus ventajas y aplicaciones:

### 1. Inyección

por

constructor:

Las dependencias se pasan como parámetros en el constructor de la clase. Es la forma más recomendada, ya que garantiza que las dependencias estén disponibles desde el momento de la creación del objeto.

```

public class Notificador {
    private ServicioMensaje servicio;
    public Notificador(ServicioMensaje servicio) {
        this.servicio = servicio;
    }
}

```

### 2. Inyección

por

métodos

setter:

Las dependencias se establecen mediante métodos modificadores (setters). Este enfoque es útil cuando la dependencia es opcional o puede cambiar en tiempo de ejecución.

```

public void setServicio(ServicioMensaje servicio) {
    this.servicio = servicio;
}

```

### 3. Inyección

por

interfaz:

Se utiliza cuando una clase implementa una interfaz diseñada específicamente para recibir dependencias. Aunque menos común, es útil en entornos donde el control de las dependencias se delega completamente a un contenedor IoC.

El siguiente ejemplo muestra una pequeña aplicación con inyección manual:

```
public interface ServicioMensaje {
    void enviar(String mensaje);
}

public class EmailService implements ServicioMensaje {
    public void enviar(String mensaje) {
        System.out.println("Enviando email: " + mensaje);
    }
}

public class SMSservice implements ServicioMensaje {
    public void enviar(String mensaje) {
        System.out.println("Enviando SMS: " + mensaje);
    }
}

public class Notificador {
    private ServicioMensaje servicio;

    public Notificador(ServicioMensaje servicio) {
        this.servicio = servicio;
    }

    public void notificar(String mensaje) {
        servicio.enviar(mensaje);
    }
}
```

```
public class Main {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        ServicioMensaje servicio = new EmailService(); // o new SMSservice()
        Notificador notificador = new Notificador(servicio);
        notificador.notificar(mensaje:"¡Bienvenido al sistema!");
    }
}
```

# TEMA 14. ENTRE BYTES Y ARCHIVOS: DOMINANDO LA ENTRADA Y SALIDA EN JAVA

Objetivo general: Comprender y aplicar los principios de entrada y salida (I/O) en Java, utilizando flujos de datos y operaciones con archivos para desarrollar programas capaces de leer, escribir y manipular información de manera eficiente y segura.

Objetivos específicos:

- Identificar los diferentes tipos de flujos (streams) en Java y su función dentro del manejo de datos.
- Implementar operaciones de lectura y escritura en archivos utilizando las clases del paquete `java.io` y `java.nio`
- Aplicar técnicas de manejo de excepciones y buenas prácticas para optimizar la gestión de recursos durante las operaciones I/O.

## 14.1 Introducción al concepto de Entrada y Salida (I/O)

En el desarrollo de aplicaciones, es común la necesidad de intercambiar información entre el programa y el exterior. Este proceso de comunicación se denomina Entrada/Salida (Input/Output o I/O). En Java, la entrada se refiere a obtener datos desde una fuente externa (como teclado, archivo o una red), mientras que la salida consiste en enviar información a un destino (como una consola, archivo o un socket).

El sistema de I/O de Java está diseñado bajo el principio de la abstracción, lo que significa que las operaciones de lectura y escritura se realizan sin importar el tipo de dispositivo físico involucrado. De esta forma, un programa puede leer datos desde un archivo o desde el teclado utilizando mecanismos similares, gracias a la estructura unificada que ofrece el lenguaje.

Java proporciona un conjunto de clases y paquetes especializados para manejar estas operaciones, principalmente dentro del paquete `java.io`, que permite trabajar con flujos (streams) de datos. A partir de Java 1.4, se introdujo el paquete `java.nio` (New Input/Output), que ofrece una alternativa más moderna y eficiente, especialmente para operaciones de alto rendimiento o en sistemas que manejan grandes volúmenes de información.

## 14.2 Concepto de flujo (stream)

El elemento central del sistema I/O en Java es el flujo o stream. Un flujo representa una secuencia ordenada de datos que se transfiere entre un origen y un destino. Los flujos permiten a un programa manejar datos de manera secuencial, independientemente del tipo de medio (archivo, red, memoria, etc).

Podemos imaginar un flujo como un “canal” por el que circulan los bytes o caracteres. Por ejemplo, cuando un programa lee un archivo, los datos viajan desde el archivo hacia el programa a través de un flujo de entrada (input stream). De manera inversa, cuando el programa guarda información en un archivo, los datos viajan hacia el archivo mediante un flujo de salida (output stream).

Existen dos grandes categorías de flujos en Java:

## 14.2.1 Flujos de bytes (Byte Streams)

Trabajan con datos binarios. Se utilizan para leer y escribir información en su forma más básica, byte por byte. Son ideales para manejar imágenes, archivos binarios, audio o cualquier tipo de dato no textual.

- Clases principales: InputStream y OutputStream.
- Subclases comunes: FileInputStream, FileOutputStream, BufferedInputStream y BufferedOutputStream.

## 14.2.2 Flujos de caracteres (Character Streams)

Están diseñados para trabajar con texto (caracteres Unicode). Estos flujos convierten automáticamente los bytes en caracteres según la codificación utilizada, lo que facilita el trabajo con archivos de texto.

- Clases principales: Reader y Writer.
- Subclases comunes: FileReader, FileWriter, BufferedReader y BufferedWriter.

## 14.3 Clases principales del paquete java.io

El paquete java.io contiene una amplia jerarquía de clases que permiten realizar operaciones de entrada y salida de manera flexible. A continuación se presentan las clases fundamentales:

### 14.3.1 InputStream y OutputStream

Son clases abstractas que representan flujos de entrada y salida de bytes, respectivamente.

- InputStream: permite leer datos desde una fuente mediante métodos como:  
int read() → lee un byte.  
int read(byte[] b) → lee varios bytes y los guarda en un arreglo.  
void close() → cierra el flujo.
- OutputStream: permite escribir datos con métodos como:  
void write(int b) → escribe un byte  
void write(byte[] b) → escribe varios bytes  
void flush() → vacía el búfer de salida y fuerza la escritura de los datos pendientes.

### 14.3.2 Reader y Writer

De manera análoga, Reader y Writer son clases abstractas que manejan flujos de caracteres.

- Reader: se utiliza para leer texto. Sus métodos principales son:  
int read() → lee un carácter.  
Int read(char[] cbuf) → lee varios caracteres.  
Void close() → cierra el flujo.
- Writer: permite escribir texto.  
Void write(int c) → escribe un carácter.  
Void write(String str) → escribe una cadena completa  
void flush() y void close() → vacía el flujo y liberan recursos.

## 14.4 Clases derivadas

Java ofrece una amplia gama de clases derivadas de estas abstracciones para cubrir necesidades específicas:

- FileInputStream y FileOutputStream: permiten leer y escribir archivos en formato binario.
- FileReader y FileWriter: realizan las mismas operaciones, pero en formato de texto.
- BufferedInputStream y BufferedOutputStream: mejoran el rendimiento utilizando un búfer intermedio para reducir la cantidad de accesos directos al sistema de archivos.
- BufferedReader y BufferedWriter: realizan la misma función, pero para flujos de texto; además, BufferedReader permite leer líneas completas mediante el método readLine().
- PrintWriter: facilita la escritura de texto de manera formateada (por ejemplo, para imprimir cadenas y valores numéricos con salto de línea)

El uso de clases con búfer es una práctica recomendada, ya que disminuye la carga del sistema al reducir el número de operaciones de E/S físicas.

## 14.5 Buffering y rendimiento

El buffering consiste en utilizar una memoria intermedia (búfer) que acumula datos antes de procesarlos o enviarlos al destino final. Sin buffering, cada lectura o escritura implicaría una llamada directa al sistema operativo, lo que ralentizaría el programa.

En Java, el buffering se implementa mediante clases como BufferedReader, BufferedWriter, BufferedInputStream y BufferedOutputStream. Estas clases envuelven a los flujos originales, ofreciendo una capa adicional de rendimiento.

```
BufferedReader reader = new BufferedReader(new FileReader("datos.txt"))
String linea = reader.readLine();
reader.close();
```

En este caso, el BufferedReader permite leer texto línea por línea, lo que resulta más eficiente y fácil de manejar.

## 14.6 Manejo de excepciones en I/O.

Las operaciones de entrada y salida son propensas a errores, como la falta de un archivo, permisos insuficientes o problemas de hardware. Por ello, todas las clases de java.io lanzan excepciones del tipo IOException.

El manejo de excepciones es esencial para evitar fallos en el programa. Un ejemplo básico seria:

```
try {
    FileReader fr = new FileReader("archivo.txt");
    int c;
    while ((c = fr.read()) != -1) {
        System.out.print((char) c);
    }
    fr.close();
} catch (IOException e) {
    System.out.println("Error al leer el archivo: " + e.getMessage());
}
```

Desde Java 7, se recomienda el uso de la estructura try-with-resources, que cierra automáticamente los flujos al finalizar el bloque:

```
Writer writer = new OutputStreamWriter(new FileOutputStream("salida.txt"), "UTF-8");
writer.write("Hola, mundo");
writer.close();
```

## 14.7 Introducción a Java NIO

Con el crecimiento de las aplicaciones modernas y la necesidad de procesar grandes volúmenes de datos o trabajar con redes de alta velocidad, el paquete tradicional java.io presento algunas limitaciones en cuanto a rendimiento y flexibilidad. Para abordar estos problemas, Java introdujo en la versión 1.4 el paquete java.nio (New Input/Output)

NIO ofrece un modelo basado en buffers y canales, en contraste con el modelo de flujos de java.io. Mientras que los flujos se procesan de manera secuencial y bloqueante, NIO permite operaciones más eficientes y flexibles, incluyendo:

- Lectura y escritura no bloqueante (non-blocking I/O)
- Manipulación de grandes archivos mediante buffers.
- Acceso directo a archivos y memoria.
- Integración avanzada con redes (sockets, selectores)

## 14.8 Conceptos fundamentales de NIO

### 14.8.1 Buffers

En NIO, los buffers son estructuras de datos que almacenan información temporalmente durante la lectura o escritura. Todos los datos deben pasar primero por un buffer antes de ser procesados. Esto

permite gestionar bloques de datos de manera eficiente y reducir las operaciones directas sobre el sistema de archivos.

Existen diferentes tipos de buffers según el tipo de datos:

- ByteBuffer → bytes
- CharBuffer → caracteres
- IntBuffer, DoubleBuffer, etc. → tipos primitivos específicos.

Un buffer tiene tres propiedades esenciales:

1. Capacity: número máximo de elementos que puede contener.
2. Position: índice donde se leerá o escribirá el próximo elemento.
3. Limit: indica el final de los datos válidos en el buffer.

```
ByteBuffer buffer = ByteBuffer.allocate(1024); // Buffer de 1 KB
buffer.put((byte)65); // Añade un byte al buffer
buffer.flip();       // Prepara el buffer para lectura
byte b = buffer.get(); // Lee el byte
buffer.clear();      // Limpia el buffer para reutilizarlo
```

## 14.8.2 Canales

Un canal (Channel) representa un punto de conexión hacia una fuente o destino de datos, como archivos, sockets o memoria. A diferencia de los flujos, los canales permiten:

- Lectura/escritura bidireccional
- Transferencia directa de grandes bloques de datos
- Operaciones no bloqueantes en combinación con selectores.

Las clases importantes de canales son:

- FileChannel → para archivos
- SocketChannel → para archivos sockets TCP
- ServerSocketChannel → para servidores TCP
- DatagramChannel → para UDP

Esto es un ejemplo de lectura desde un archivo usando FileChannel:

```

FileInputStream fis = new FileInputStream("datos.txt");
FileChannel channel = fis.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(1024);
int bytesRead = channel.read(buffer);
channel.close();
fis.close();

```

En este ejemplo, los datos se leen en bloques de hasta 1024 bytes, lo que mejora la eficiencia frente a la lectura byte a byte de java.io.

## 14.9 Clases clave de java.nio.file

Java 7 introdujo al paquete java.nio.file, que simplifica la manipulación de archivos y directorios mediante la API Files & Paths. Entre sus principales ventajas están:

- Creación, lectura y escritura de archivos de manera más intuitiva.
- Operaciones de copia, movimiento y eliminación de archivos.
- Manejo seguro de excepciones y control de permisos.

### 14.9.1 Clase Paths

La clase Paths permite representar rutas de archivos y directorios de manera portátil, independiente del sistema operativo.

```
Path ruta = Paths.get("C:/datos/archivo.txt");
```

### 14.9.2 Clase Files

Files ofrece métodos estáticos para realizar operaciones comunes:

```

// Crear un archivo
Files.createFile(ruta);

// Escribir texto
Files.write(ruta, "Hola NIO".getBytes());

// Leer texto
List<String> lineas = Files.readAllLines(ruta);

// Copiar archivo
Files.copy(ruta, Paths.get("C:/datos/copia.txt"));

// Mover archivo
Files.move(ruta, Paths.get("C:/datos/movido.txt"));

// Eliminar archivo
Files.delete(Paths.get("C:/datos/movido.txt"));

```

Estos métodos reemplazan y simplifican muchas operaciones que anteriormente requerían varias líneas de código con java.io

## 14.10 Operaciones avanzadas con NIO

### 14.10.1 Transferencia directa entre canales

NIO permite copiar archivos de manera más eficiente usando transferTo o transferFrom:

```
FileChannel origen = new FileInputStream("origen.txt").getChannel();
FileChannel destino = new FileOutputStream("destino.txt").getChannel();
origen.transferTo(0, origen.size(), destino);
origen.close();
destino.close();
```

Esto es más rápido que leer y escribir manualmente mediante buffers, porque permite que la transferencia se haga directamente en memoria del sistema operativo.

### 14.10.2 Mapear archivos en memoria

Con FileChannel.map(), un archivo puede ser mapeado directamente a memoria, lo que permite acceso aleatorio sin leer todo el archivo:

```
RandomAccessFile raf = new RandomAccessFile("archivo.txt", "rw");
FileChannel fc = raf.getChannel();
MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ_WRITE, 0, fc.size());
byte b = mbb.get(0); // Leer primer byte
mbb.put(0, (byte) 'H'); // Modificar primer byte
fc.close();
raf.close();
```

Esta técnica es útil para archivos grandes, bases de datos locales o caches de alto rendimiento.

## 14.11 Manejo de excepciones y try-with-resources

Al igual que en java.io, todas las operaciones NIO pueden lanzar excepciones (IOException o subclases). Se recomienda siempre usar try-with-resources para garantizar que los canales y flujos se cierren correctamente.

```
try (FileChannel fc = FileChannel.open(Paths.get("archivo.txt"), StandardOpenOption.READ)) {
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    fc.read(buffer);
} catch (IOException e) {
    e.printStackTrace();
}
```

Esto evita fugas de memoria y asegura que los recursos del sistema se liberen automáticamente.

## 14.12 Integración con operaciones no bloqueantes y redes

Uno de los mayores avances de NIO es la capacidad de realizar I/O no bloqueante, especialmente útil en servidores y aplicaciones concurrentes. En combinación con Selector y canales de sockets (SocketChannel y ServerSocketChannel), se pueden gestionar múltiples conexiones simultáneas sin necesidad de crear un hilo por cliente.

- Selector: observa varios canales para detectar cuándo están listos para leer o escribir.
- Registro de canales: cada canal se registra con un selector y un interés específico.
- No bloqueo: la llamada a lectura/escritura devuelve inmediatamente, incluso si no hay datos disponibles, evitando que el hilo se detenga.

Esto permite aplicaciones altamente escalables, como servidores web de alto rendimiento o sistemas de chat en tiempo real.

## EJERCICIOS TEMA 14.

EJERCICIO 1. Crea un programa que lea el contenido de un archivo llamado “texto.txt” y lo muestre línea por línea en la consola.

### SOLUCIÓN

```
import java.io.*;

public class Ejercicio1 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader(fileName:"texto.txt"))) {
            String linea;
            while((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```

EJERCICIO 2. Crea un programa que pida al usuario una frase y lo guarde en un archivo llamado “salida.txt”.

### SOLUCIÓN

```
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Ejercicio2 {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println(x:"Escribe una frase: ");
        String texto = sc.nextLine();

        try (BufferedWriter bw = new BufferedWriter(new FileWriter(fileName:"salida.txt", append:true))) {
            bw.write(texto);
            bw.newLine();
            System.out.println(x:"Frase guardada en salida.txt");
        } catch (IOException e) {
            System.out.println("Error al escribir el archivo: " + e.getMessage());
        }
    }
}
```

EJERCICIO 3. Crea un programa que lea “texto.txt” y muestre: numero total de lineas, de palabras, y de caracteres.

## SOLUCIÓN

```
import java.io.*;

public class Ejercicio3 {
    public static void main(String[] args) {
        int lineas = 0, palabras = 0, caracteres = 0;

        try (BufferedReader br = new BufferedReader(new FileReader(fileName:"texto.txt"))){
            String linea;
            while ((linea = br.readLine()) != null) {
                lineas++;
                caracteres += linea.length();
                palabras += linea.split(regex:"\\s+").length;
            }

            System.out.println("Lineas: " + lineas);
            System.out.println("Palabras: " + palabras);
            System.out.println("Caracteres: " + caracteres);
        } catch (IOException e) {
            System.out.println("Error al procesar el archivo: " + e.getMessage());
        }
    }
}
```

EJERCICIO 4. Crea un programa que copie el contenido de “origen.txt” y “copia.txt”.

## SOLUCIÓN

```
import java.io.*;

public class Ejercicio4 {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("origen.txt"));
            BufferedWriter bw = new BufferedWriter(new FileWriter("copia.txt"))
        } {
            String linea;
            while ((linea = br.readLine()) != null) {
                bw.write(linea);
                bw.newLine();
            }
            System.out.println("Archivo copiado correctamente.");
        } catch (IOException e) {
            System.out.println("Error al copiar: " + e.getMessage());
        }
    }
}
```

EJERCICIO 5. Crea un programa que lea el contenido de “entrada.txt”, lo convierta a mayúsculas y lo escriba en “salida.txt”.

## SOLUCIÓN

```
import java.nio.file.*;
import java.io.IOException;
import java.util.List;
import java.util.stream.Collectors;

public class Ejercicio5 {
    public static void main(String[] args) {
        Path entrada = Paths.get("entrada.txt");
        Path salida = Paths.get("salida.txt");

        try {
            List<String> lineas = Files.readAllLines(entrada);
            List<String> mayusculas = lineas.stream()
                .map(String::toUpperCase)
                .collect(Collectors.toList());
            Files.write(salida, mayusculas);
            System.out.println("Archivo convertido a mayúsculas exitosamente.");
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

# TEMA 15. ENTRE HILOS Y MEMORIA: DOMINANDO LA CONCURRENCIA MODERNA EN JAVA

Objetivo general: Comprender y aplicar los principios de la concurrencia en Java, utilizando hilos tradicionales, hilos virtuales y el modelo de memoria de Java para desarrollar aplicaciones eficientes, seguras y escalables.

Objetivos específicos:

- Analizar los fundamentos de la concurrencia y el paralelismo en Java.
- Implementar programas que utilicen hilos tradicionales, comprendiendo su ciclo de vida y control.
- Explorar el funcionamiento y ventajas de los hilos virtuales introducidos en las versiones modernas de Java.
- Comprender el Java Memory Model (JMM) y su impacto en la visibilidad y consistencia de los datos compartidos.
- Aplicar correctamente la palabra clave volatile y otros mecanismos de sincronización para evitar condiciones de carrera.

## 15.1 Introducción a la concurrencia

En el mundo de la programación moderna, la concurrencia es un concepto esencial para aprovechar al máximo los recursos del hardware actual. La mayoría de los procesadores modernos cuentan con varios núcleos o “cores”, lo que permite ejecutar múltiples tareas al mismo tiempo. La concurrencia busca que un programa pueda realizar varias operaciones de manera simultánea o solapada, mejorando su rendimiento, capacidad de respuesta y eficiencia.

En Java, la concurrencia ha sido una característica presente desde sus primeras versiones, y con el tiempo ha evolucionado para ofrecer herramientas cada vez más potentes y seguras. Comprender cómo funciona la concurrencia es fundamental para desarrollar aplicaciones que manejen múltiples procesos, por ejemplo: servidores web, videojuegos, programas de análisis de datos o sistemas de tiempo real, etc., sin perder estabilidad ni consistencia.

La concurrencia no debe confundirse con el paralelismo. Aunque ambos términos están relacionados, no son exactamente lo mismo.

- Concurrencia: significa que varias tareas progresan al mismo tiempo, compartiendo recursos y tiempo de CPU.
- Paralelismo, en cambio, implica que múltiples tareas se ejecutan realmente al mismo tiempo en diferentes núcleos del procesador.

En muchos casos, un programa concurrente puede parecer paralelo, aunque en realidad los hilos se alternen rápidamente en la CPU. Java proporciona mecanismos para ambos enfoques, dependiendo del tipo de aplicación.

## 15.2 Concepto de threads

El hilo o thread es la unidad básica de ejecución dentro de un programa concurrente. Un programa Java siempre tiene al menos un hilo principal: el que ejecuta el método main(). Sin embargo, es posible crear hilos adicionales para realizar tareas secundarias mientras el programa principal continúa con otras operaciones.

Por ejemplo, en una aplicación que descarga archivos de Internet, un hilo podría encargarse de la descarga, otro de mostrar el progreso y otro de responder a las acciones del usuario. Así, el programa se mantiene fluido y no “se congela” mientras realiza tareas pesadas.

Cada hilo tiene su propio contador de programa, su pila de ejecución y sus variables locales, pero comparte el mismo espacio de memoria con los demás hilos del proceso. Esta característica permite la comunicación y el intercambio de datos entre ellos, pero también introduce riesgos: si dos hilos acceden a la misma variable al mismo tiempo, puede producirse errores de sincronización, conocido como condiciones de carrera (race conditions).

## 15.3 Ventajas de la concurrencia en Java

El uso de múltiples hilos ofrece varios beneficios:

- Mejor aprovechamiento del hardware: en equipos con varios nucleos, los hilos pueden ejecutarse en paralelo, aumentando la velocidad total del programa.
- Mayor capacidad de respuesta: las tareas intensivas se pueden realizar en segundo plano sin bloquear la interfaz del usuario.
- Modularidad y escalabilidad: al dividir un problema en tareas más pequeñas (hilos), el código se vuelve más flexible y fácil de mantener.
- Simulación de tareas simultáneas: incluso en sistemas con un solo procesador, los hilos permiten ejecutar operaciones de forma alternada, dando la impresión de simultaneidad.

Sin embargo, la concurrencia también introduce complejidad, ya que los hilos deben coordinarse para evitar conflictos en el acceso a los recursos compartidos. Por eso, Java proporciona un conjunto sólido de herramientas para el control y la sincronización.

## 15.4 Creación y control de hilos en Java

Existen varias formas de crear hilos en Java, las más comunes son:

### 15.4.1 Extender la clase Thread

Se puede crear una clase que extienda Thread y sobrescriba el método run(), que contiene el código que se ejecutará en el nuevo hilo.

```

class MiHilo extends Thread {
    public void run() {
        System.out.println("Ejecutando hilo: " + getName());
    }
}

public class Ejemplo {
    public static void main(String[] args) {
        MiHilo hilo = new MiHilo();
        hilo.start(); // inicia el nuevo hilo
    }
}

```

El método start() crea un nuevo hilo de ejecución y llama internamente a run(). Es importante no invocar directamente run(), ya que eso no crea un hilo nuevo, sino que ejecuta el código en el mismo hilo actual.

### 15.4.2 Implementar la interfaz Runnable

Otra forma es implementar la interfaz Runnable, que también define el método run(). Este enfoque es más flexible, ya que permite que la clase extienda otra clase al mismo tiempo.

```

class MiTarea implements Runnable {
    public void run() {
        System.out.println("Tarea ejecutándose...");
    }
}

public class Ejemplo {
    public static void main(String[] args) {
        Thread hilo = new Thread(new MiTarea());
        hilo.start();
    }
}

```

### 15.4.3 Usar ExecutorService

En versiones más recientes de Java, se recomienda usar pools de hilos mediante la API java.util.concurrent. Esto evita crear y destruir hilos continuamente, lo cual es costoso en rendimiento.

```

ExecutorService ejecutor = Executors.newFixedThreadPool(3);
ejecutor.submit(() -> System.out.println("Ejecutando tarea en pool"));
ejecutor.shutdown();

```

## 15.5 Ciclo de vida de un hilo

Un hilo en Java pasa por varios estados durante su vida:

1. New: cuando el objeto Thread ha sido creado, pero aún no se ha iniciado.
2. Runnable: cuando se llama a start(), el hilo está listo para ejecutarse.
3. Running: cuando la CPU asigna tiempo al hilo y ejecuta su código.
4. Blocked/Waiting: cuando el hilo espera un recurso o una condición.
5. Terminated: cuando finaliza el método run() o se produce una excepción no controlada.

El control de ciclo de vida es esencial para evitar errores como bloqueos (deadlocks) o esperas indefinidas.

## 15.6 Sincronización y condiciones de carrera

Una de las mayores dificultades de la programación concurrente es evitar que varios hilos modifiquen los mismos datos al mismo tiempo.

Por ejemplo, si dos hilos intentan incrementar la misma variable compartida simultáneamente, el resultado puede ser incorrecto debido a que ambas operaciones leen y escriben en la memoria sin coordinación.

Para resolverlo, Java proporciona mecanismos de sincronización, como la palabra clave synchronized, que garantiza que solo un hilo pueda ejecutar una sección crítica de código a la vez.

```
public synchronized void incrementar() {  
    contador++;  
}
```

Además, existen métodos como wait(), notify() y notifyAll() que permiten la comunicación entre hilos, especialmente cuando uno debe esperar a que otro termine una tarea antes de continuar.

## 15.7 Clases y librerías clave en Java para concurrencia

El paquete java.util.concurrent introducido en Java 5 simplificó enormemente el trabajo con hilos. Algunas clases importantes son:

- ExecutorService: administra pools de hilos
- Future y Callable: permiten ejecutar tareas que devuelven resultados.
- ReentrantLock: ofrece un control de bloque más avanzado que synchronized.
- Semaphore y CountDownLatch: permite coordinar la ejecución de múltiples hilos.
- ConcurrentHashMap y CopyOnWriteArrayList: colecciones seguras para el acceso concurrente.

Estas herramientas permiten escribir código concurrente de forma más estructurada, reduciendo el riesgo de errores y mejorando la legibilidad del programa.

## 15.8 Problemas comunes de la concurrencia

Aunque poderosa, la concurrencia trae consigo varios retos:

- Condiciones de carrera: cuando dos o más hilos acceden simultáneamente a un recurso compartido sin la debida sincronización.
- Interbloqueo (deadlock): cuando dos hilos se quedan esperando indefinidamente por un recurso que el otro posee.
- Inanición (starvation): cuando un hilo nunca obtiene acceso a los recursos que necesita para ejecutarse.
- Inconsistencia de datos: cuando los cambios de un hilo no son visibles inmediatamente para otros debido al modelo de memoria de Java.

Reconocer estos problemas es el primer paso para diseñar programas concurrentes confiables.

## 15.9 Hilos virtuales

Los hilos virtuales son una implementación ligera de los hilos de Java, diseñados para manejar millones de tareas concurrentes de manera eficiente. A diferencia de los hilos tradicionales, que dependen de los recursos del sistema operativo, los hilos virtuales son gestionados por la JVM.

Podemos imaginar los hilos virtuales como “hilos de usuario”, que se ejecutan sobre un número reducido de hilos del sistema, llamados carrier threads. Estos hilos “transportadores” se encargan de ejecutar múltiples hilos virtuales en diferentes momentos, suspendiéndolos y reanudándolos según sea necesario.

Gracias a esta arquitectura, Java puede crear miles o incluso millones de hilos virtuales sin agotar la memoria ni los recursos del sistema operativo, algo impensable con los hilos tradicionales.

### 15.9.1 Motivación y ventajas

La principal motivación detrás de los hilos virtuales es simplificar la escritura del código concurrente sin sacrificar rendimiento. En modelos anteriores, para manejar miles de conexiones simultáneas (como en un servidor HTTP), los desarrolladores tenían que usar técnicas complejas como programación asíncrona, callbacks o futuros.

Con los hilos virtuales, ya no es necesario recurrir a estos modelos complicados. Ahora, cada tarea puede ejecutarse en su propio hilo, utilizando el mismo código bloqueante y secuencial que los programadores de Java ya conocen.

Las ventajas principales son:

- Ligereza: los hilos virtuales son extremadamente livianos; crear uno cuesta apenas unos pocos kilobytes de memoria, en contraste con los megabytes de un hilo tradicional.
- Alta escalabilidad: una aplicación puede manejar millones de conexiones concurrentes sin degradar significativamente el rendimiento.
- Simplicidad del código: se puede escribir código concurrente utilizando el estilo clásico, sin necesidad de frameworks o patrones complicados.

- Compatibilidad total: los hilos virtuales son totalmente compatibles con las APIs y bibliotecas existentes de Java.
- Gestión automática: la JVM se encarga de suspender y reanudar hilos virtuales de manera eficiente, especialmente durante operaciones de entrada/salida.

### 15.9.2 Como funcionan internamente los hilos virtuales

Para entender el poder de los hilos virtuales, es importante conocer su funcionamiento interno.

Cuando un hilo virtual ejecuta una tarea que no implica bloqueo, se comporta igual que un hilo tradicional. Pero si encuentra una operación bloqueante, como leer un archivo, esperar una respuesta de red o acceder a una base de datos, la JVM suspende el hilo virtual y libera el hilo portador para que ejecute otro hilo virtual.

Una vez que la operación bloqueante finaliza, el hilo virtual se reanuda en cualquier otro hilo portador disponible. Este mecanismo se conoce como desacoplamiento (decoupling) entre hilos virtuales y del sistema operativo.

### 15.9.3 Creación y uso de los hilos virtuales

El uso de hilos virtuales es muy sencillo y familiar para quienes ya trabajan con hilos en Java. A partir de Java 21, los hilos virtuales forman parte de la API estándar del lenguaje.

```
public class EjemploVirtualThread {
    public static void main(String[] args) throws InterruptedException {
        Thread hiloVirtual = Thread.ofVirtual().start(() -> {
            System.out.println("Ejecutando hilo virtual: " + Thread.currentThread());
        });

        hiloVirtual.join();
    }
}
```

También es posible crear grupos de hilos virtuales mediante un ExecutorService especial:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 10000; i++) {
        executor.submit(() -> {
            // Simulación de tarea concurrente
            Thread.sleep(1000);
            return "Tarea completada";
        });
    }
}
```

En este ejemplo, se puede lanzar miles de tareas concurrentes sin sobrecargar la memoria ni el CPU. Cada tarea se ejecuta en un hilo virtual independiente, y la JVM los administra automáticamente.

### **15.9.4 Casos de uso típicos**

Los hilos virtuales están especialmente diseñados para aplicaciones altamente concurrentes pero no intensivas en CPU. Algunos ejemplos son:

- Servidores web o REST APIs: cada petición puede manejarse en su propio hilo virtual.
- Sistemas de mensajería o colas de eventos.
- Conectores de base de datos y microservicios.
- Simulaciones o motores de juego que dependen de múltiples tareas pequeñas y paralelas.

En todos estos escenarios, los hilos virtuales permiten mantener la simplicidad del código bloqueante sin perder rendimiento.

### **15.9.5 Limitaciones y consideraciones**

Aunque los hilos virtuales representan un gran avance, no son una solución mágica. Algunas limitaciones actuales son:

- Tareas de CPU intensivas: si un hilo virtual realiza cálculos pesados durante mucho tiempo, bloqueara el hilo portador, reduciendo la eficiencia general.
- Llamadas nativas o JNI: las operaciones que usan código nativo pueden no ser suspendibles, lo que limita los beneficios.
- Depuración y monitoreo: aunque se ha mejorado, el seguimiento de millones de hilos puede ser más complejo.
- Compatibilidad con frameworks antiguos: algunas bibliotecas que asumen un modelo fijo de hilos pueden requerir ajustes.

Por estas razones, es importante evaluar el tipo de carga de trabajo antes de decidir usar hilos virtuales.

## **15.10 Java Memory Model (JMM)**

La razón se encuentra el Java Memory Model (JMM) o Modelo de Memoria de Java. Este modelo define cómo interactúan los hilos con la memoria, y qué garantías ofrece el lenguaje sobre la visibilidad, el orden y la coherencia de los datos. En otras palabras, el JMM especifica cuándo y cómo los cambios realizados por un hilo pueden ser vistos por otros hilos.

### **15.10.1 ¿Por qué se necesita un modelo de memoria?**

En los sistemas modernos, el rendimiento del procesador dependen en gran parte de optimizaciones internas como el reordenamiento de instrucciones, cachés locales y múltiples núcleos de ejecución. Estas optimizaciones permiten ejecutar el código más rápido, pero también pueden provocar que el orden aparente de las operaciones no coincida con el orden real en memoria.

Por ejemplo, si un hilo escribe en una variable y otro la lee inmediatamente, el segundo hilo podría no ver el valor actualizado, ya que el cambio podría seguir almacenado en una caché local del primer hilo.

Sin un modelo de memoria bien definido, sería casi imposible garantizar la coherencia entre los hilos. Por eso, el JMM establece un conjunto de reglas de visibilidad y sincronización que aseguran que todos los hilos vean una versión consistente de la memoria cuando se utilizan correctamente las herramientas del lenguaje.

## 15.10.2 Conceptos clave de JMM

Para entender como funciona el JMM, es necesario conocer algunos conceptos fundamentales:

### 15.10.2.1 Memoria principal y memoria de hilos

- Memoria principal: área donde se almacenan las variables compartidas por todos los hilos.
- Memoria local (de hilo): cada hilo mantiene una copia temporal de las variables en su propio espacio de trabajo.

Cuando un hilo modifica una variable, ese cambio no se refleja automáticamente en la memoria principal. Para que otros hilos vean el nuevo valor, es necesario que el hilo escriba (flush) la variable de vuelta en la memoria principal. De igual manera, un hilo debe leer (reload) una variable desde la memoria principal para obtener su valor actualizado.

### 15.10.2.2 Operaciones atómicas

Algunas operaciones, como la lectura y escritura de variables primitivas (ejemplo, int, boolean), son atomicas, es decir, indivisibles. Sin embargo, eso no garantiza la visibilidad entre hilos; una variable puede cambiar en un hilo y otro hilo no lo notara si no hay sincronización adecuada.

### 15.10.2.3 Reordenamiento de instrucciones

El compilador y la CPU pueden reordenar las instrucciones del programa para optimizar la ejecución. Esto no afecta el resultado en un programa de un solo hilo, pero en un entorno concurrente puede generar resultados inesperados si un hilo depende del orden de ejecución de otro.

## 15.10.3 Problemas que el JMM resuelve.

Sin el JMM, los hilos podrían comportarse de manera errática, leyendo datos obsoletos o inconsistentes. Los problemas más comunes son:

1. Visibilidad: un hilo no ve los cambios hechos por otro.
2. Reordenamiento: el CPU o compilador cambian el orden de ejecución, rompiendo las dependencias lógicas.
3. Atomicidad parcial: una operación aparentemente simple puede dividirse en varias instrucciones no atómicas.

El JMM proporciona una solución al definir happens-before, una relación fundamental para la sincronización en Java.

## 15.10.4 Principio de Happens-Before

El concepto de happens-before es la base del JMM. Este principio define cuándo los efectos de una operación son visibles para otra.

Si una operación A happens-before una operación B, entonces todos los efectos de A (lecturas y escrituras en memoria) serán visibles para B. Si no existe esta relación, el comportamiento no está garantizando y puede variar entre ejecuciones.

Algunas de las reglas más importantes del happens-before son:

- Orden del programa: dentro de un mismo hilo, las instrucciones se ejecutan en el orden en que aparecen en el código.
- Bloques sincronizados: una salida de un bloque synchronized happens-before cualquier entrada a ese mismo bloque por otro hilo.
- Variables volatile: una escritura en una variable volatile happens-before cualquier lectura posterior de esa misma variable.
- Inicio y finalización de hilos: La llamada a Thread.start() happens-before cualquier acción dentro del nuevo hilo. La finalización de un hilo happens-before el retorno de Thread.join() en otro hilo.

### 15.10.5 Sincronización y visibilidad

El JMM está estrechamente relacionado con las herramientas de sincronización en Java. Cuando se usan correctamente, permiten que los hilos compartan información sin conflictos.

- Palabra clave synchronized: asegura que solo un hilo puede ejecutar una sección crítica de código a la vez. Además, al entrar y salir del bloque sincronizado, se actualizan las variables desde y hacia la memoria principal, garantizando la visibilidad.
- Palabra clave volatile: asegura que las lecturas y escrituras en una variable sean directamente desde la memoria principal, evitando que los hilos trabajen con copias desactualizadas.
- Clases de concurrencia: implementan mecanismos internos que respetan las reglas del JMM, ofreciendo operaciones seguras sin que el programador tenga que manejar la sincronización manualmente.

```
class Ejemplo {
    private static boolean listo = false;
    private static int valor = 0;

    public static void main(String[] args) {
        new Thread(() -> {
            valor = 42;
            listo = true;
        }).start();

        new Thread(() -> {
            if (listo) {
                System.out.println(valor);
            }
        }).start();
    }
}
```

A simple vista, podríamos esperar que el programa imprima 42. Sin embargo, en algunas ejecuciones no imprime nada o incluso imprime 0.

Esto ocurre porque no hay sincronización entre los hilos. El segundo hilo podría leer `listo` como `false` o leer valor antes de que se haya actualizado en la memoria principal. Si se declara `listo` como `volatile`, o si se usaran bloques `synchronized`, el JMM garantizaría que los cambios fueran visibles para ambos hilos.

### **15.10.6 Recomendaciones para usar el JMM correctamente**

Para aprovechar las garantías del JMM y evitar errores de concurrencia, se recomienda:

1. Evitar la comunicación sin sincronización entre hilos.
2. Usar `volatile` cuando una variable sea leída por múltiples hilos y escrita por uno solo.
3. Preferir las clases del paquete `java.util.concurrent`, ya que están diseñadas para respetar las reglas del JMM.
4. Reducir el uso de variables compartidas y, en lo posible, mantener la inmutabilidad de los datos.
5. Probar con cuidado: los errores de memoria pueden ser intermitentes y difíciles de reproducir.

# TEMA 16. JAVA EN ACCIÓN: EL PODER DETRÁS DE LO COTIDIANO

Objetivo general: explorar y aplicar las funcionalidades de Java que permiten desarrollar soluciones prácticas para la vida diaria, enfocándose en el uso de criptografía, manejo de hora y fecha, networking y expresiones regulares.

Objetivos específicos:

- Comprender los principios básicos y las bibliotecas de Java relacionadas con la criptografía para garantizar la seguridad de los datos.
- Implementar programas que gestionen correctamente la hora y fecha utilizando las clases del paquete `java.time`
- Analizar las herramientas de Java para establecer conexiones de red y transferir información entre sistemas.
- Aplicar expresiones regulares para la validación y filtrado de datos en distintos contextos cotidianos.

## 16.1 Criptografía en Java

La criptografía es el conjunto de técnicas destinadas a proteger la información mediante su transformación en un formato ilegible para quienes no poseen la clave adecuada. Su propósito principal es garantizar los principios de confidencialidad, integridad, autenticidad y no repudio.

En Java, la criptografía se implementa a través de la Java Cryptography Architecture (JCA) y la Java Cryptography Extension (JCE), las cuales proporcionan un marco unificado para realizar operaciones criptográficas de manera segura y estandarizada.

Existen tres tipos principales de criptografía utilizados en aplicaciones Java:

### 16.1.1 Tipos de criptografía

#### 16.1.2.1 Simétrica

Utiliza la misma clave para cifrar y descifrar la información. Es eficiente para grandes volúmenes de datos. Ejemplos como algoritmos AES o DES. En Java se puede implementar con la clase `Cipher` del paquete `javax.crypto`, especificando el algoritmo deseado.

#### 16.1.2.2 Asimétrica

Emplea un par de claves: una pública y una privada. Es más segura pero también más lenta. Se usa comúnmente en el intercambio de claves y firmas digitales. Ejemplo: RSA. Java permite generar pares de claves mediante la clase `KeyPairGenerator`.

#### 16.1.2.3 Hash o de resumen

Convierte la información en una cadena de longitud fija (hash), imposible de revertir. Se usa para verificar integridad o almacenar contraseñas. Ejemplo: SHA-256 o MD5. Java proporciona la clase `MessageDigest` para crear y comparar valores hash.

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] hash = md.digest("contraseña123".getBytes(StandardCharsets.UTF_8));
String hashHex = DatatypeConverter.printHexBinary(hash);
System.out.println("Hash: " + hashHex);
```

Un ejemplo simple de aplicación criptográfica en Java es la generación de un hash de contraseña. Esté código transforma una cadena en un hash seguro que puede almacenarse en bases de datos sin revelar la contraseña original.

## 16.2 Manejo de fechas y tiempos en Java

En versiones antiguas de Java, el manejo de fechas y horas se realizaba con las clases Date y Calendar, las cuales presentaban limitaciones en cuanto a legibilidad y precisión. A partir de Java 8, se introdujo el paquete java.time, inspirado en la librería Jada-Time, que ofrece una API moderna, clara y precisa para trabajar con tiempo y zonas horarias.

### 16.2.1 Clases principales del paquete java.time

- LocalDate: representa una fecha sin hora. Ideal para registrar cumpleaños, fechas de vencimiento o eventos diarios.
- LocalTime: representa una hora sin fecha. Se usa en alarmas, cronómetros o tareas programadas.
- LocalDateTime: combina fecha y hora en una sola entidad.
- ZonedDateTime: agrega información sobre zona horaria, útil para aplicaciones globales o distribuidas.
- Duration y Period: permiten calcular diferencias entre instantes o fechas.

```
LocalDateTime ahora = LocalDateTime.now();
DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
System.out.println("Fecha y hora actual: " + ahora.format(formato));
```

En muchos casos, los mecanismos de seguridad y control temporal se combinan para crear sistemas más robustos. Por ejemplo, al generar un token de acceso, se usa un algoritmo criptográfico que incluye un sello de tiempo, garantizando que el código tenga una vigencia limitada.

## 16.3 Networking en Java

El término networking se refiere a la capacidad de una aplicación para comunicarse con otras máquinas a través de una red, ya sea local o global (Internet). Java incluye un conjunto de clases y paquetes que permiten implementar la comunicación entre dispositivos mediante protocolos estandarizados, como TCP/IP o HTTP.

El paquete principal encargado de esta funcionalidad es java.net, que ofrece las herramientas necesarias para crear aplicaciones cliente-servidor, enviar y recibir datos, o conectarse a servicios web.

## 16.3.1 Clases principales del paquete java.net

### 16.3.1.1 InetAddress

Representa una dirección IP, permitiendo obtener información sobre hosts, como nombre y dirección numérica.

```
InetAddress host = InetAddress.getLocalHost();
System.out.println("Dirección IP: " + host.getHostAddress());
```

### 16.3.1.2 Socket y ServerSocket

Se utilizan para establecer una comunicación entre cliente y servidor mediante el protocolo TCP.

- ServerSocket: escucha peticiones en un puerto determinado
- Socket: permite al cliente conectarse al servidor y transmitir información.

### 16.3.1.3 URL Y URLConnection

Permiten interactuar con recursos en Internet (páginas web, APIs, etc).

```
URL url = new URL("https://www.example.com");
URLConnection conexion = url.openConnection();
InputStreamReader entrada = new InputStreamReader(conexion.getInputStream());
```

Este código permite leer el contenido de una página web directamente desde Java.

## 16.3.2 Arquitectura cliente-servidor

En una arquitectura cliente-servidor, una máquina (servidor) ofrece recursos o servicios, mientras que otra (cliente) los solicita.

- El servidor permanece a la escucha de solicitudes en un puerto específico mediante ServerSocket.
- El cliente se conecta al servidor a través de un Socket y puede enviar o recibir información.

## 16.3.3 Aplicaciones prácticas del networking en Java

El networking en Java permite desarrollar funcionalidades presentes en la vida cotidiana como:

- Transferencia de archivos entre dispositivos.
- Aplicaciones de mensajería instantánea.
- Clientes y servidores web
- Programas que consumen APIs o servicios REST
- Sistemas distribuidos que comparten información en tiempo real.

## 16.4 Expresiones regulares en Java

Las expresiones regulares son patrones que permiten identificar, buscar y manipular cadenas de texto de manera eficiente. En Java, se utilizan ampliamente para validar datos de entrada, filtrar información o procesar textos complejos. Por ejemplo, una aplicación puede usar expresiones

regulares para verificar si un correo electrónico tiene el formato correcto o si una contraseña cumple con los requisitos de seguridad.

### 16.4.1 Paquete y clases principales

Java incluye el paquete `java.util.regex`, que contiene dos clases fundamentales:

- `Pattern`: representa una expresión regular compilada.
- `Matcher`: permite comparar una cadena con el patrón y realizar operaciones como búsqueda, coincidencia o reemplazo.

```
Pattern patron = Pattern.compile("[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,3}");
Matcher coincidencia = patron.matcher("usuario@email.com");
if (coincidencia.matches()) {
    System.out.println("Correo válido");
}
```

Este fragmento valida si una cadena tiene el formato de un correo electrónico.

### 16.4.2 Sintaxis básica de las expresiones regulares

`.` → cualquier carácter

`*` → repetición de cero o más veces

`+` → repetición de una o más veces

`?` → cero o una ocurrencia.

`[ ]` → conjunto de caracteres permitidos

`^` → inicio de cadena

`$` → fin de cadena

Por ejemplo:

`^[A-Z][a-z]+$` → valida que una palabra comience con mayúscula y continúe con minúsculas.

`\d{4}-\d{2}-\d{2}` → valida una fecha con formato “aaaa-mm-dd”

### 16.4.3 Aplicaciones comunes de las expresiones regulares

Las expresiones regulares se aplican en:

- Validación de formularios (correo, contraseña, números de teléfono)
- Procesamiento de archivos de texto o logs.
- Extracción de datos en aplicaciones web.
- Filtrado de información en motores de búsqueda o bases de datos.

# TEMA 17. INTEGRACIÓN PRÁCTICA DE CONCEPTOS EN UNA APLICACIÓN REAL.

Objetivo principal: Desarrollar una aplicación en Java que integre los principales temas aprendidos, con el fin de demostrar la aplicación práctica de la POO en la solución de problemas cotidianos.

## 17.1 Ejercicio final

Crea una aplicación de consola llamada TaskManager, que permita al usuario:

- Agregar tareas con una descripción y prioridad.
- Listar tareas pendientes.
- Marcar tareas como completadas
- Guardar y cargar tareas desde un archivo.

```
import java.io.Serializable;

@interface Important {}

class Task implements Serializable {
    private String description;
    private int priority;
    private boolean completed;

    public Task(String description, int priority) {
        this.description = description;
        this.priority = priority;
        this.completed = completed;
    }
}
```

Podemos ver que en esta parte, tenemos la anotación personalizada para marcar métodos importantes, la clase Task como modelo en POO.

```
public String getDescription() {
    return description;
}

public int getPriority() {
    return priority;
}

public boolean isCompleted() {
    return completed;
}

public void setCompleted(boolean completed) {
    this.completed = completed;
}
```

Aquí ponemos los getters y setters.

```
    @Override
    public String toString() {
        return (completed ? "[✓]" : "[ ]") + " " + description + " (Prioridad: " + priority + ")";
    }

interface TaskRepository {
    void saveTasks(List<Task> tasks) throws IOException;
    List<Task> loadTasks() throws IOException, ClassNotFoundException;
}
```

```
class FileTaskRepository implements TaskRepository {
    private final String fileName;

    public FileTaskRepository(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void saveTasks(List<Task> tasks) throws IOException {
        try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fileName))) {
            oos.writeObject(tasks);
        }
    }
}
```

```
    @Override
    public List<Task> loadTasks() throws IOException, ClassNotFoundException {
        File file = new File(fileName);
        if(!file.exists()) return new ArrayList<>();
        try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file))) {
            return (List<Task>) ois.readObject();
        }
    }
}
```

```
public class TaskManager {
    private List<Task> tasks;
    private final TaskRepository repository;
    private final Scanner scanner = new Scanner(System.in);

    public TaskManager(TaskRepository repository) {
        this.repository = repository;
        try {
            this.tasks = repository.loadTasks();
        } catch (Exception e) {
            System.out.println("⚠ No se pudieron cargar las tareas: " + e.getMessage());
            this.tasks = new ArrayList<>();
        }
    }
}
```

```

public static void main(String[] args) {
    TaskRepository repo = new FileTaskRepository("tareas.dat");
    TaskManager manager = new TaskManager(repo);
    manager.run();
}

public void run() {
    int option;
    do {
        System.out.println("\n==== GESTOR DE TAREAS ====");
        System.out.println("1. Agregar tarea");
        System.out.println("2. Listar tareas");
        System.out.println("3. Marcar como completada");
        System.out.println("4. Guardar y salir");
        System.out.println("Elige una opción: ");

        option = getIntInput();

        switch(option) {
            case 1 -> addTask();
            case 2 -> listTasks();
            case 3 -> completeTask();
            case 4 -> exitProgram();
            default -> System.out.println("X Opción no válida");
        }
    } while (option != 4);
}

```

```

@Important
private void addTask() {
    System.out.println("Descripción: ");
    String desc = scanner.nextLine();

    System.out.println("Prioridad (1-5): ");
    int priority = getIntInput();

    tasks.add(new Task(desc, priority));
    System.out.println("✓ Tarea agregada con éxito.");
}

private void listTasks() {
    if(tasks.isEmpty()) {
        System.out.println("⚠ No hay tareas registradas.");
        return;
    }
}

```

```
private void listTasks() {
    if(tasks.isEmpty()) {
        System.out.println("⚠️ No hay tareas registradas.");
        return;
    }

    List<Task> sorted = tasks.stream()
        .sorted(Comparator.comparingInt(Task::getPriority))
        .collect(Collectors.toList());

    System.out.println("\n--- Lista de tareas ---");
    sorted.forEach(System.out::println);
}
```

```
private void completeTask() {
    listTasks();
    System.out.println("Número de tarea a completar: ");
    int index = getIntInput();

    Optional<Task> optTask = getTaskByIndex(index - 1);
    optTask.ifPresentOrElse(
        t -> {
            t.setCompleted(completed:true);
            System.out.println("✅ Tarea marcada como completada.");
        }, () -> System.out.println("⚠️ Tarea no encontrada.");
)
```

```
private Optional<Task> getTaskByIndex(int index) {
    if(index >= 0 && index < tasks.size()) {
        return Optional.of(tasks.get(index));
    }
    return Optional.empty();
}
```

```
private void exitProgram(){
    try {
        repository.saveTasks(tasks);
        System.out.println("💾 Tareas guardadas correctamente. ¡Hasta luego!");
    } catch (IOException e) {
        System.out.println("❌ Error al guardar: " + e.getMessage());
    }
}
```

```
private int getIntInput() {
    try {
        return Integer.parseInt(scanner.nextLine());
    } catch (NumberFormatException e) {
        System.out.println("⚠️ Entrada no válida. Intenta de nuevo.");
        return getIntInput();
    }
}
```

## EPILOGO

Si llegaste hasta aquí, felicidades: sobreviviste a las clases, los bulces, los arrays y los sustos del compilador. Puede que aún no seas un maestro Jedi del código, pero ya no eres el mismo que abrió este libro digital preguntándose qué demonios era un “boolean”. Has pasado del caos al código, del miedo al main(), y del “¿por qué no corre esto?” al “¡funciona!”.

Y eso, querido lector, es el primer paso para dejar de ser un tonto de Java y empezar a ser alguien que entiende el poder de transformar ideas en programas.

Sin embargo, este no es el final del camino. Java es solo la puerta de entrada a un universo enorme. Desde aquí, podrás explorar bases de datos para almacenar información como un profesional, frameworks como Spring Boot para construir aplicaciones modernas, e incluso interfaces gráficas que hagan que tus programas cobren vida.

Recuerda: los grandes programadores también empezaron rompiendo el código, maldiciendo al teclado y buscando respuestas en foros a las tres de la mañana. No es un defecto, es parte del ritual. Así que no dejes de experimentar, equivocarte y reírte de tus propios errores.

Cada error de compilación es una pequeña lección disfrazada y cada programa que logres hacer funcionar será una victoria que solo tú entenderás.

Ahora cierra este libro, abre tu Visual Studio Code o editor de código favorito y escribe. No porque tengas que hacerlo, sino porque puedes hacerlo. Y cuando alguien te diga: “¿Java? Eso es difícil”, tu sonrie, toma tu café, y responde con orgullo:

TAL VEZ, PERO YO LO APRENDÍ... CON JAVA PARA TONTOS