



# JAVA BACK-END MEDIO- AVANZADO

Programación Avanzada en Java y Principios  
de Back-End

# Presentación

Ingeniera Civil Industrial de la Universidad de Chile con más de 20 años de experiencia profesional. Actualmente dedicada a la consultoría en transformación digital y gestión de proyectos. Mi expertis son las soluciones basadas en datos para optimizar la toma de decisiones estratégicas.

He liderado la implementación de proyectos de Business Intelligence (BI) y he participado activamente en el desarrollo de soluciones tecnológicas como desarrolladora full stack, integrando tanto el diseño de interfaces como la lógica de negocio y el acceso a datos.

He trabajado en diversas industrias como la bancaria, financiera, educativa y productiva.

Mi enfoque está en apoyar a las empresas en su proceso de transformación digital, brindando soluciones que les permitan avanzar hacia su siguiente nivel de competitividad.

Actualmente, estoy cursando el segundo año de un Magíster en Data Science en la Universidad del Desarrollo, lo que me permite integrar conocimientos avanzados en ciencia de datos y machine learning para generar modelos predictivos de alto impacto.



Ingrid Solís

DataLover | Data Science | Specialized in  
Business Intelligence and Digital Transform...



# Conociéndonos

Persona

Laboral

Curso

¿Cuál es tu  
nombre?

¿A qué te dedicas?

¿Cuál es tu expectativa  
para este curso?

# Objetivo del curso

Desarrollar en los participantes las competencias técnicas necesarias para diseñar e implementar soluciones back-end utilizando arquitectura de microservicios, estándares modernos de desarrollo en Java, y frameworks como Spring y Spring Boot, con el fin de crear aplicaciones robustas, eficientes y escalables.





# JavaScript vs. Java: ¿son lo mismo?

Aunque comparten parte del nombre, **JavaScript** y **Java** son como el agua y el aceite: diferentes en origen, propósito y uso.

 Misma década, mismo nombre... pero caminos totalmente distintos.





# JavaScript vs. Java: ¿son lo mismo?



## JavaScript

- Nació en 1995 en Netscape para hacer páginas web interactivas.
- Se ejecuta principalmente en el navegador (aunque también en servidores con Node.js).
- Tiene tipado **dinámico** y es más flexible.
- Es el alma del desarrollo web frontend: React, Vue, Angular, ¡todo usa JavaScript!





# JavaScript vs. Java: ¿son lo mismo?

## Java

- También nació en 1995, pero en Sun Microsystems.
- Se ejecuta en la **Máquina Virtual de Java (JVM)**.
- Tiene tipado **estático**, más robusto y estricto.
- Es ampliamente usado en sistemas empresariales, bancos, Android, y backend corporativo con frameworks como Spring Boot.





## Módulo 1.- Programación Avanzada en Java

- Manejo avanzado de colecciones y streams.
- Programación funcional en Java (lambdas, optional, etc.).
- Excepciones personalizadas y buenas prácticas.
- Manejo eficiente de memoria y recursos.

# Módulo 1.- Programación Avanzada en Java

## Guía para Configurar Visual Studio Code para Java

### 1. Instala Java JDK

Descarga e instala **Java 17** (recomendado para compatibilidad con herramientas modernas):

 [Adoptium Temurin JDK 17](#)

### 2. Instala Visual Studio Code

 [Descargar VS Code](#)



# Módulo 1.- Programación Avanzada en Java

## 3. Instala el **Java Extension Pack**

Desde VS Code:

- Ve a la sección de **extensiones**
- Busca: Java Extension Pack
- Instala el paquete completo

Incluye:

- Language Support for Java™ by Red Hat
- Debugger for Java
- Maven for Java
- Java Test Runner
- IntelliCode for Java



# Módulo 1.- Programación Avanzada en Java

Otra forma de instalar Java

Para usuarios de Mac

1. Si tienes [Homebrew](#) instalado, solo abre Terminal y ejecuta:

```
brew install openjdk@21
```

Después de la instalación, debes **agregarlo a tu PATH**. Ejecuta lo siguiente:

```
echo 'export PATH="/usr/local/opt/openjdk@21/bin:$PATH"' >> ~/.zprofile
```

```
source ~/.zprofile
```

Y luego verifica:

```
java -version
```



# Módulo 1.- Programación Avanzada en Java

Otra forma de instalar Java

Para usuarios de Mac

2. Si prefieres una instalación tradicional:

- Ve a  <https://www.oracle.com/java/technologies/javase-downloads.html>
- Elige la última versión LTS (por ejemplo, Java 21 o 17).
- Descarga el instalador .dmg para macOS.
- Instálalo como cualquier otra app.
- Abre de nuevo Terminal y ejecuta `java -version`.



# Módulo 1.- Programación Avanzada en Java

## 4. Crea un proyecto Java simple

### Opción A: Manual

- Crea una carpeta vacía (por ejemplo: CursoJavaAvanzado)
- Dentro, crea src/Main.java
- Estructura base:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("¡Hola desde Java!");  
    }  
}
```

- Y luego en la terminal:

```
javac HolaMundo.java
```

```
java HolaMundo
```



# Módulo 1.- Programación Avanzada en Java

## Opción B: Desde VS Code (Java Project)

The screenshot shows the VS Code interface with a Java project. The code editor displays `Main.java` containing the following code:

```
src > J Main.java > ...
1 public class Main {
2     Run | Debug
3     public static void main(String[] args) {
4         System.out.println("¡Hola desde Java en VS Code!");
5     }
6
7 }
```

The terminal below shows the execution of the code:

```
● (base) desarollo@MacBook-Pro-de-Ingrid CursoJavaAvanzado % cd /Users/desarollo/Relatorias/CursoJavaAvanzado ; /usr/bin/env /usr/local/Cellar/openjdk@21/21.0.7/libexec/openjdk.jdk/Contents/Home/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/desarollo/Library/Application\ Support/Code/User/workspaceStorage/813da219765e3d0f2bb361d6f5c97cef/redhat.java/jdt_ws/CursoJavaAvanzado_a1b435b9/bin Main
¡Hola desde Java en VS Code!
○ (base) desarollo@MacBook-Pro-de-Ingrid CursoJavaAvanzado %
```



# Módulo 1.- Programación Avanzada en Java

## 5. Ejecuta tu código

- Abre Main.java
- Haz clic en el botón de **Run** o usa Ctrl+F5
- La consola integrada mostrará la salida



Welcome Main.java Extension: Extension Pack for Java



# Extension Pack for Java

Microsoft [microsoft.com](https://microsoft.com) | ⚡ 36,516,111 | ★★★★☆(82)

Popular extensions for Java development that provides Java IntelliSense, debugging, test...

[Disable](#) [Uninstall](#) [Switch to Pre-Release Version](#)  Auto Update 

[DETAILS](#) [FEATURES](#) [CHANGELOG](#)



Screenshot of a GitHub repository page for 'Relatorias' showing the 'Code' tab. The repository path is 'Relatorias / CursoJavaAvanzado /'. A commit by 'Joselota' titled 'Agrega nuevos programas' is shown, dated 61031db · 1 minute ago. The commit message is 'Más modificaciones y archivos añadidos al repositorio.' The commit details a file named 'launch.json' with the message 'Agrega nuevos programas' and a date of 3 days ago.

Name	Last commit message	Last commit date
..		
.vscode	Más modificaciones y archivos añadidos al repositorio.	3 days ago
bin	Agrega nuevos programas	1 minute ago
src	Agrega nuevos programas	1 minute ago
ejemplo1		
ejemplo10		
ejemplo12		
ejemplo13		
launch.json	Agrega nuevos programas	1 minute ago

Los próximos ejemplos están disponibles en esta ruta:

<https://github.com/Joselota/Relatorias/tree/main/CursoJavaAvanzado>



# Módulo 1.- Programación Avanzada en Java

## Starter Pack

```
import java.util.*;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> nombres = Arrays.asList("Ingrid", "Pedro", "Camila", "Iván");

        List<String> resultado = nombres.stream()
            .filter(n -> n.startsWith("I"))
            .map(String::toUpperCase)
            .sorted()
            .collect(Collectors.toList());

        System.out.println("Filtrados: " + resultado);
    }
}
```



# Módulo 1.- Programación Avanzada en Java

## ◆ filter, map, sorted, collect

```
.filter(n -> n.startsWith("I"))
.map(String::toUpperCase)
.sorted()
.collect(Collectors.toList());
```

### 📌 ¿Qué hacen estas funciones?

- `.stream()` convierte la lista en un flujo de datos.
- `.filter(n -> n.startsWith("I"))` selecciona solo los que comienzan con "I".
- `.map(String::toUpperCase)` transforma a mayúsculas.
- `.sorted()` los ordena alfabéticamente.
- `.collect(Collectors.toList())` junta el resultado en una lista nueva.



### Analogía completa:

- Tomas la lista de nombres.
- Te quedas solo con los que empiezan con "I".
- Los conviertes a mayúscula.
- Los ordenas.
- Los guardas en una nueva lista.



# Módulo 1.- Programación Avanzada en Java

💡 Resultado del código completo

```
List<String> nombres = Arrays.asList("Ingrid", "Pedro", "Camila", "Iván");

List<String> resultado = nombres.stream()
    .filter(n -> n.startsWith("I"))
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.toList());

System.out.println("Filtrados: " + resultado);
```

💻 **Salida esperada:** Filtrados: [INGRID, IVÁN]



# Módulo 1.- Programación Avanzada en Java

## Colecciones y Streams

**Objetivo Específico:** Comprender el uso avanzado de colecciones en Java y aplicar operaciones comunes de la Stream API para transformar y filtrar datos de forma declarativa.



# Módulo 1.- Programación Avanzada en Java

## Colecciones y Streams

### Ejercicio 1: List y duplicados

**Objetivo:** Comprender cómo List mantiene el orden y permite duplicados.

```
List<String> frutas = new ArrayList<>();
frutas.add("Manzana");
frutas.add("Banana");
frutas.add("Manzana");
System.out.println(frutas); // ¿Qué ves? ¿Qué pasa con los duplicados?
System.out.println(frutas.get(1)); // Accede al segundo elemento
```

**Extensión:**

- Reemplaza ArrayList por LinkedList y mide la diferencia en comportamiento (ideal para disminuir rendimiento).



- ✓ Código completo para ejecutar:

```
import java.util.*;  
  
public class Main {  
    public static void main(String[] args) {  
        List<String> frutas = new ArrayList<>();  
        frutas.add("Manzana");  
        frutas.add("Banana");  
        frutas.add("Manzana");  
  
        System.out.println(frutas); // Imprime toda la lista  
        System.out.println(frutas.get(1)); // Accede al segundo elemento (índice 1)  
    }  
}
```



# Módulo 1.- Programación Avanzada en Java

## Colecciones y Streams

### Ejercicio 2: Set y unicidad

**Objetivo:** Observar cómo se eliminan duplicados automáticamente.

```
Set<String> paises = new HashSet<>();
paises.add("Chile");
paises.add("Argentina");
paises.add("Chile");
```

**Extensión:**

- Reemplaza HashSet por TreeSet y observa el orden.
- Usa LinkedHashSet para mantener el orden de inserción.



- ✓ Código completo para ejecutar:

```
import java.util.*;  
  
public class Main {  
    public static void main(String[] args) {  
        Set<String> paises = new HashSet<>();  
        paises.add("Chile");  
        paises.add("Argentina");  
        paises.add("Chile");  
  
        System.out.println(paises);  
    }  
}
```



# Módulo 1.- Programación Avanzada en Java

## Colecciones y Streams

### Ejercicio 3: Map y clave-valor

**Objetivo:** Familiarizarse con estructuras tipo diccionario.

```
Map<String, Integer> stock = new HashMap<>();
stock.put("Lápiz", 10);
stock.put("Cuaderno", 5);
stock.put("Lápiz", 12); // ¿Qué pasa aquí?

System.out.println(stock);
System.out.println("Stock de lápices: " + stock.get("Lápiz"));
```

**Extensión:**

- Cambia `HashMap` por `TreeMap` para ver el orden alfabético.
- Usa `LinkedHashMap` para preservar orden de inserción.



- ✓ Código completo para ejecutar:

```
import java.util.*;  
  
public class Main {  
    public static void main(String[] args) {  
        Map<String, Integer> stock = new HashMap<>();  
        stock.put("Lápiz", 10);  
        stock.put("Cuaderno", 5);  
        stock.put("Lápiz", 12); // Sobrescribe el valor anterior  
  
        System.out.println(stock);  
        System.out.println("Stock de lápices: " + stock.get("Lápiz"));  
    }  
}
```



# Módulo 1.- Programación Avanzada en Java

## Colecciones y Streams

### Ejercicio 4: Diferencia entre interfaz e implementación

**Objetivo:** Entender el polimorfismo a través de colecciones.

```
List<String> lista = new LinkedList<>();
Set<String> conjunto = new TreeSet<>();
Map<String, Integer> mapa = new LinkedHashMap<>();

// ¿Qué métodos comunes puedes usar en cada uno?
```

### Discusión:

- ¿Por qué usamos List como tipo y no ArrayList directamente?
- ¿Qué nos da esa flexibilidad a largo plazo?



 Código completo para ejecutar:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> lista = new LinkedList<>();
        Set<String> conjunto = new TreeSet<>();
        Map<String, Integer> mapa = new LinkedHashMap<>();

        // Ejemplos de métodos comunes
        lista.add("uno");
        lista.add("dos");
        System.out.println("Lista: " + lista);
        System.out.println("Contiene 'uno': " + lista.contains("uno"));
        System.out.println("Tamaño de lista: " + lista.size());

        conjunto.add("beta");
        conjunto.add("alfa");
        conjunto.add("beta"); // Ignorado, ya que TreeSet no permite duplicados
        System.out.println("Set ordenado: " + conjunto);
        System.out.println("Set contiene 'alfa': " + conjunto.contains("alfa"));

        mapa.put("clave1", 100);
        mapa.put("clave2", 200);
        mapa.put("clave1", 300); // Sobrescribe
        System.out.println("Mapa: " + mapa);
        System.out.println("Valor de 'clave1': " + mapa.get("clave1"));
    }
}
```



🧠 ¿Qué representa cada estructura?

1. `List<String> lista = new LinkedList<>();`

📌 Lista **ordenada**, permite duplicados, acceso por índice.

🔧 Métodos comunes:

- `add(element)`, `get(index)`, `remove(index)`
- `contains(element)`, `size()`, `clear()`

2. `Set<String> conjunto = new TreeSet<>();`

📌 Conjunto **ordenado alfabéticamente**, no permite duplicados.

🔧 Métodos comunes:

- `add(element)`, `remove(element)`
- `contains(element)`
- `isEmpty()`, `size()`, `clear()`

🧠 **Nota:** TreeSet ordena automáticamente los elementos al insertarlos.

3. `Map<String, Integer> mapa = new LinkedHashMap<>();`

📌 Mapa de **clave → valor**, mantiene el **orden de inserción**.

🔧 Métodos comunes:

- `put(key, value)`, `get(key)`, `remove(key)`
- `containsKey(key)`, `containsValue(value)`
- `keySet()`, `values()`, `entrySet()`, `size()`

🧠 **Nota:** LinkedHashMap recuerda el orden en que insertaste las claves.



# Módulo 1.- Programación Avanzada en Java

## Colecciones y Streams

### Ejercicio 5: Mutable vs Inmutable

Objetivo: Comparar comportamiento de colecciones mutables vs inmutables.

```
List<String> mutable = new ArrayList<>();
mutable.add("Hola");
mutable.set(0, "Chao");

List<String> inmutable = List.of("Hola");

// inmutable.set(0, "Chao"); // ¿Qué pasa si descomentas esto?
```

Extensión:

Usa Collections.unmodifiableList(mutable) y luego intenta modificarla.

Discute ventajas y casos de uso de colecciones inmutables (p. ej., configuración de app, constantes



- ✓ Código completo para ejecutar:

```
import java.util.*;  
  
public class Main {  
    public static void main(String[] args) {  
        // Lista mutable  
        List<String> mutable = new ArrayList<>();  
        mutable.add("Hola");  
        mutable.set(0, "Chao"); // ✓ Modificación permitida  
        System.out.println("Mutable: " + mutable);  
  
        // Lista inmutable  
        List<String> inmutable = List.of("Hola");  
        // inmutable.set(0, "Chao"); // ✗ Esto lanzaría una excepción si se descomenta  
  
        System.out.println("Inmutable: " + inmutable);  
    }  
}
```



🧠 ¿Qué está pasando?

- ◆ `ArrayList` (mutable)

```
List<String> mutable = new ArrayList<>();
```

- Puedes agregar (`add`), modificar (`set`), eliminar (`remove`) elementos.
- Es **mutable**: su contenido puede cambiar después de su creación.

Resultado:

```
Mutable: [Chao]
```

- ◆ `List.of(...)` (inmutable)

```
List<String> immutable = List.of("Hola");
```

- Crea una **lista inmutable**: no puedes modificar su contenido.
- Si descomentas esta línea:

```
immutable.set(0, "Chao");
```

Lanzará una excepción en tiempo de ejecución:

```
java.lang.UnsupportedOperationException
```

Porque la lista fue creada como **no modificable**.

🔒 ¿Por qué usar listas inmutables?

✓ **Ventajas:**

- Más seguras en entornos concurrentes.
  - Evitan efectos secundarios inesperados.
  - Ideales para constantes o configuraciones.
- 🛠 Se usan mucho en programación funcional, APIs, y diseño robusto.



💡 BONUS: ¿Cómo crear una copia mutable de una lista inmutable?

```
List<String> copiaMutable = new ArrayList<>(inmutable);  
copiaMutable.set(0, "Nuevo valor");
```

¿Quieres que hagamos un ejercicio en el que compares listas inmutables creadas con  
`Collections.unmodifiableList(...)` vs `List.of(...)`?



# Módulo 1.- Programación Avanzada en Java

## Colecciones y Streams

Opcional: Mini reto de combinación

Crea una función que reciba una lista con nombres repetidos y retorne un Map<String, Integer> con la cantidad de veces que aparece cada nombre.

```
[ "Ingrid", "Pedro", "Ingrid", "Camila", "Pedro", "Pedro"]
```

```
// → {Ingrid=2, Pedro=3, Camila=1}
```



 Solución: función para contar nombres repetidos

```
import java.util.*;  
  
public class Main {  
    public static void main(String[] args) {  
        List<String> nombres = Arrays.asList("Ingrid", "Pedro", "Ingrid", "Camila", "Pedro", "Pedro");  
        Map<String, Integer> conteo = contarNombres(nombres);  
  
        System.out.println(conteo); // → {Ingrid=2, Pedro=3, Camila=1}  
    }  
  
    public static Map<String, Integer> contarNombres(List<String> lista){  
        Map<String, Integer> mapa = new HashMap<>();  
  
        for (String nombre : lista) {  
            mapa.put(nombre, mapa.getOrDefault(nombre, 0) + 1);  
        }  
        return mapa;  
    }  
}
```



## ¿Qué hace este código?

- ◆ `mapa.getOrDefault(nombre, 0)`
- Si el nombre ya está en el mapa, obtiene su valor actual.
- Si no está, devuelve 0.
- Luego le suma 1 y lo actualiza.
- ◆ **Resultado esperado:**

{Ingrid=2, Pedro=3, Camila=1}

→ El orden no está garantizado porque se usa HashMap.

### ¿Opcional con orden?

Si quieres mantener el **orden de aparición**, puedes usar LinkedHashMap en lugar de HashMap.

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

¿Quieres que también prepare una versión usando Streams y Collectors.groupingBy(...)? Es un poco más avanzada y funcional.



# Módulo 1.- Programación Avanzada en Java

## ○ Programación funcional en Java



Objetivo:

Aplicar conceptos de programación funcional para escribir código más declarativo, claro y reutilizable.



Contenidos:

- Lambdas (( ) -> {}), referencias a métodos (String::toUpperCase)
- Optional: evitación de null, métodos .isPresent(), .ifPresent(), .orElse()
- Interfaces funcionales: Predicate, Function, Consumer, Supplier
- Uso de var para inferencia de tipos (Java 10+)



# Módulo 1.- Programación Avanzada en Java

## Programación funcional en Java

- ◆ Ejercicio 1: Reescribir un for como expresión lambda

Objetivo: Aplicar forEach con expresión lambda y referencias a métodos.

```
List<String> nombres = List.of("Ingrid", "Pedro", "Camila");

// FOR tradicional
for (String nombre : nombres) {
    System.out.println(nombre.toUpperCase());
}

// ¿Cómo lo harías con una expresión funcional?
```

**Reto extra:**

- Usar Consumer<String> para encapsular el comportamiento de impresión.



Código original (forma tradicional):

```
List<String> nombres = List.of("Ingrid", "Pedro", "Camila");

for (String nombre : nombres){
    System.out.println(nombre.toUpperCase());
}
```





¿Cómo hacerlo de forma funcional?

Opción 1: **Lambda con forEach**

```
nombres.forEach(nombre -> System.out.println(nombre.toUpperCase()));
```

Explicación:

forEach(...) recorre la lista.

nombre -> ... es una expresión lambda que define lo que se hace por cada elemento.

Opción 2: **Stream + map + referencia a método**

```
nombres.stream()  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

Explicación:

- `.stream()` crea un flujo de datos.
- `.map(String::toUpperCase)` transforma cada nombre a mayúsculas.
- `.forEach(System.out::println)` imprime cada elemento usando una **referencia directa a un método**.



## Comparación rápida

Forma	Ventaja principal
Lambda directa	Más simple y explícita
Stream + map	Más expresiva y escalable (transformaciones encadenadas)
Tradicional for	Fácil de leer para principiantes



Reto extra: Usar Consumer<String> para encapsular el comportamiento de impresión.

¿Qué es un Consumer<String>?

Un Consumer<String> es una **función que recibe un String y no devuelve nada** (solo **consume** el dato).

 Solución usando Consumer<String>

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

public class Main {
    public static void main(String[] args) {
        List<String> nombres = List.of("Ingrid", "Pedro", "Camila");

        // Definimos un Consumer que imprime en mayúsculas
        Consumer<String> imprimirEnMayusculas = nombre ->
            System.out.println(nombre.toUpperCase());

        // Lo usamos en un forEach
        nombres.forEach(imprimirEnMayusculas);
    }
}
```



🧠 ¿Qué pasa aquí?

- Se declara un Consumer<String> llamado `imprimirEnMayusculas`.
- Este Consumer usa una lambda para definir su comportamiento: imprimir en mayúsculas.
- Luego lo pasamos a `forEach()` como argumento.

✓ Resultado:

INGRID

PEDRO

CAMILA



# Módulo 1.- Programación Avanzada en Java

## Programación funcional en Java

- ◆ Ejercicio 2: Evitar NullPointerException usando Optional

Objetivo: Evitar retornos null y usar Optional de forma segura.

```
public static Optional obtenerNombreUsuario(boolean existe) {  
    return existe ? Optional.of("Ingrid") : Optional.empty();  
}
```

// ¿Cómo usarías .isPresent(), .ifPresent() y .orElse() para trabajar con este método?

**Extensión:**

Intenta llamar a **.toUpperCase()** directamente sobre el resultado con **.map(...)**.



Función base:

```
public static Optional<String> obtenerNombreUsuario(boolean existe) {  
    return existe ? Optional.of("Ingrid") : Optional.empty();  
}
```

💡 Esta función simula que, si el usuario existe, retorna su nombre dentro de un `Optional`; si no, retorna un `Optional.empty()`.





¿Cómo trabajar con este Optional?

✓ 1. Usando `.isPresent()`

```
Optional<String> resultado = obtenerNombreUsuario(true);

if (resultado.isPresent()) {
    System.out.println("Usuario: " + resultado.get());
} else {
    System.out.println("Usuario no encontrado.");
}
```

🔍 `.isPresent()` verifica si hay un valor, y `.get()` lo extrae.

⚠ **Advertencia:** usar `.get()` es seguro solo si ya validaste `.isPresent()`.



## 2. Usando .ifPresent()

```
obtenerNombreUsuario(true)
    .ifPresent(nombre -> System.out.println("Hola, " + nombre));
```

 .ifPresent() ejecuta el bloque **solo si el valor está presente**. Es más elegante que .isPresent() + .get().

## 3. Usando .orElse()

```
String nombre = obtenerNombreUsuario(false).orElse("Invitado");
System.out.println("Bienvenido: " + nombre);
```

 .orElse(valorPorDefecto) retorna el valor si existe; si no, retorna uno por defecto.



## Comparación rápida:

Método	Uso recomendado
.isPresent()	Para condiciones simples (aunque algo verboso)
.ifPresent()	Para ejecutar una acción solo si hay valor
.orElse()	Para definir un valor por defecto si está vacío



# Módulo 1.- Programación Avanzada en Java

## Programación funcional en Java

- ◆ Ejercicio 3: Uso de Predicate para filtros condicionales

Objetivo: Usar predicados para separar lógica condicional.

```
List palabras = List.of("mesa", "monitor", "mouse", "móvil");
```

```
Predicate empiezaConM = s -> s.startsWith("m");
```

```
// ¿Cómo usariás este predicado con stream().filter(...)?
```

Reto adicional:

Crear un predicado que combine condiciones (`length > 4 && startsWith("m")`).



Código base con Predicate:

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

public class Main {
    public static void main(String[] args) {
        List<String> palabras = List.of("mesa", "monitor", "mouse", "móvil");

        Predicate<String> empiezaConM = s -> s.startsWith("m");

        List<String> resultado = palabras.stream()
            .filter(empiezaConM)
            .collect(Collectors.toList());

        System.out.println("Palabras que comienzan con 'm': " + resultado);
    }
}
```



## ¿Qué hace este código?

### 1. **Predicate<String> empiezaConM**

Define una función que devuelve true si la palabra empieza con "m".

### 2. **.stream().filter(empiezaConM)**

Usa el predicado para **filtrar** los elementos de la lista.

### 3. **.collect(Collectors.toList())**

Recolecta los elementos filtrados en una nueva lista.



# Módulo 1.- Programación Avanzada en Java

## Programación funcional en Java

- ◆ Ejercicio 4: Implementar una función que reciba otra función

Objetivo: Usar Function<T, R> como parámetro de entrada.

```
public static List<Integer> aplicarFuncion(List<Integer> numeros, Function<Integer, Integer> f) {  
    return numeros.stream()  
        .map(f)  
        .collect(Collectors.toList());  
  
    // ¿Cómo pasarías una lambda como parámetro?
```

**Prueba con:**

```
List<Integer> cuadrados = aplicarFuncion(List.of(2, 3, 4), x -> x * x);
```



 Versión genérica y tipada correctamente:

Primero, aquí está una versión **tipada** de tu función:

```
import java.util.*;
import java.util.function.Function;
import java.util.stream.Collectors;

public class Main {
    public static List<Integer> aplicarFuncion(List<Integer> numeros, Function<Integer, Integer> f) {
        return numeros.stream()
            .map(f)
            .collect(Collectors.toList());
    }

    public static void main(String[] args) {
        List<Integer> numeros = List.of(1, 2, 3, 4, 5);

        // Pasando una lambda: multiplicar por 2
        List<Integer> resultado = aplicarFuncion(numeros, n -> n * 2);

        System.out.println("Resultado: " + resultado); // → [2, 4, 6, 8, 10]
    }
}
```



 ¿Qué ocurre aquí?

- ◆ Function<Integer, Integer> f
- Recibe un Integer como entrada
- Devuelve un Integer como resultado
- ◆  $n \rightarrow n * 2$
- Es una **lambda** pasada como argumento que representa esa Function

 Otros ejemplos de uso: Cuadrados:

```
aplicarFuncion(numeros, n -> n * n);
```

 Convertir a negativos:

```
aplicarFuncion(numeros, n -> -n);
```

 Más elaborado (usar una función definida aparte):

```
Function<Integer, Integer> triple = x -> x * 3;  
aplicarFuncion(numeros, triple);
```



# Módulo 1.- Programación Avanzada en Java

## ○ Programación funcional en Java

- ◆ Ejercicio 5: Supplier para generación de valores

Objetivo: Entender Supplier<T> como proveedor sin argumentos.

```
Supplier<String> saludo = () -> "¡Hola desde Supplier!";  
System.out.println(saludo.get());
```

### Reto extra:

Generar una lista de 5 valores aleatorios (Supplier<Double>) y mostrarlos con Stream.generate(...).



 Código completo:

```
import java.util.function.Supplier;

public class Main {
    public static void main(String[] args) {
        Supplier<String> saludo = () -> "¡Hola desde Supplier!";
        System.out.println(saludo.get());
    }
}
```

 ¿Qué hace este código?

- `Supplier<String>`: define un proveedor de objetos `String`.
- `() -> "¡Hola desde Supplier!"`: es una **lambda sin argumentos** que retorna esa cadena.
- `saludo.get()`: ejecuta la función y obtiene el valor generado.

 Salida esperada:

¡Hola desde Supplier!



# Módulo 1.- Programación Avanzada en Java

## ○ Programación funcional en Java

- ◆ Ejercicio 6: Uso de var (Java 10+)

Objetivo: Simplificar la declaración de tipos sin perder claridad.

```
var lista = List.of("A", "B", "C"); for (var elemento : lista) {  
    System.out.println(elemento); }
```

Discusión:

¿Qué ventajas y riesgos tiene el uso de var?

¿Cuándo conviene y cuándo no?



Código con var:

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        var lista = List.of("A", "B", "C");

        for (var elemento : lista) {
            System.out.println(elemento);
        }
    }
}
```

🧠 ¿Qué es var?

- var permite declarar una variable **sin escribir el tipo explícito**, pero el compilador **deduce** el tipo en tiempo de compilación.
- En el ejemplo, var lista = List.of("A", "B", "C"); es equivalente a:
- List<String> lista = List.of("A", "B", "C");



# Módulo 1.- Programación Avanzada en Java

## Excepciones personalizadas y manejo eficiente de recursos

🎯 Objetivo: Diseñar excepciones propias con buenas prácticas y gestionar correctamente los recursos del sistema.

### Contenidos:

- Tipos de excepciones: checked vs unchecked
- Crear excepciones personalizadas (extends RuntimeException)
- try-with-resources para liberar recursos automáticamente
- Reglas de propagación (throws, throw)
- Buenas prácticas: mensajes claros, jerarquía, documentación



# Módulo 1.- Programación Avanzada en Java

## Excepciones personalizadas y manejo eficiente de recursos

### Ejercicio 1: Crear una excepción personalizada

Objetivo: Aprender a definir y lanzar una excepción propia.

```
// SaldoInsuficienteException.java
public class SaldoInsuficienteException extends RuntimeException {
    public SaldoInsuficienteException(String mensaje) {
        super(mensaje);
    }
}
```

Discusión:

¿Por qué hereda de RuntimeException?

¿Cómo se ve diferente de una checked exception?



 Paso 1: Crear la clase de excepción

```
public class SaldoInsuficienteException extends RuntimeException {  
  
    public SaldoInsuficienteException(String mensaje) {  
        super(mensaje); // pasa el mensaje al constructor de RuntimeException  
    }  
}
```

💡 Esta clase:

- Extiende `RuntimeException`, lo que la convierte en una **excepción no verificada** (`unchecked`).
- Puedes lanzarla sin necesidad de usar `throws` en la firma del método.
- Incluye un constructor que permite pasar un mensaje personalizado al momento de lanzar la excepción.



- ✓ Paso 2: Usarla en un método (por ejemplo, retirar dinero)

```
public class CuentaBancaria {  
    private double saldo;  
  
    public CuentaBancaria(double saldoInicial) {  
        this.saldo = saldoInicial;  
    }  
  
    public void retirar(double monto) {  
        if (monto > saldo) {  
            throw new SaldoInsuficienteException("Saldo insuficiente. Intentaste retirar " + monto +  
", pero solo hay " + saldo);  
        }  
        saldo -= monto;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
}
```



### Paso 3: Probarla en el método main

```
public class Main {  
    public static void main(String[] args) {  
        CuentaBancaria cuenta = new CuentaBancaria(100);  
  
        try {  
            cuenta.retirar(150);  
        } catch (SaldoInsuficienteException e) {  
            System.out.println("✖ Error: " + e.getMessage());  
        }  
  
        System.out.println("Saldo actual: " + cuenta.getSaldo());  
    }  
}
```

#### ¿Cuándo conviene usar excepciones personalizadas?

- Para representar **errores de lógica específicos** del negocio.
- Para que otros desarrolladores (o APIs REST) **puedan manejarlas de forma diferenciada**.
- Para lanzar errores más claros que un simple IllegalArgumentException o NullPointerException.



# Módulo 1.- Programación Avanzada en Java

## Excepciones personalizadas y manejo eficiente de recursos

### Ejercicio 2: Simular transferencia bancaria

Objetivo: Aplicar la excepción personalizada en una lógica de negocio simple.

```
public class Cuenta {  
    private double saldo;  
    public Cuenta(double saldoInicial) {  
        this.saldo = saldoInicial;  
    }  
  
    public void transferir(double monto) {  
        if (monto > saldo) {  
            throw new SaldoInsuficienteException("Saldo insuficiente para transferir $" + monto);  
        }  
        saldo -= monto;  
        System.out.println("Transferencia exitosa. Nuevo saldo: $" + saldo);  
    }  
}
```

```
Cuenta cuenta = new Cuenta(10000);  
cuenta.transferir(12000); // Debería lanzar la excepción
```

Discusión:

¿Dónde conviene capturar la excepción? ¿En el método o donde se llama?



- Versión funcional y completa del código

1 Clase SaldoInsuficienteException.java:

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



2 Clase Cuenta.java:

```
public class Cuenta {  
    private double saldo;  
  
    public Cuenta(double saldoInicial) {  
        this.saldo = saldoInicial;  
    }  
  
    public void transferir(double monto) {  
        if (monto > saldo) {  
            throw new SaldoInsuficienteException("Saldo insuficiente para transferir $" +  
                monto);  
        }  
        saldo -= monto;  
        System.out.println("Transferencia exitosa. Nuevo saldo: $" + saldo);  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
}
```



3 Código para probarlo (Main.java):

```
public class Main {  
    public static void main(String[] args) {  
        Cuenta cuenta = new Cuenta(10000);  
  
        try {  
            cuenta.transferir(12000); // Debería lanzar la excepción  
        } catch (SaldoInsuficienteException e) {  
            System.out.println("X ERROR: " + e.getMessage());  
        }  
  
        System.out.println("Saldo final: $" + cuenta.getSaldo());  
    }  
}
```



# Módulo 1.- Programación Avanzada en Java

## Excepciones personalizadas y manejo eficiente de recursos

### Ejercicio 3: Lectura de archivo con try-with-resources

Objetivo: Usar correctamente try-with-resources para evitar fugas.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class LectorArchivo {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("archivo.txt"))) {
            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (IOException e) {
            System.err.println("Error leyendo el archivo: " + e.getMessage());
        }
    }
}
```

Extensión:

Usa un archivo que no existe para forzar un FileNotFoundException.

Agrega una segunda excepción a capturar.



## ¿Qué hace este código?

```
try (BufferedReader br = new BufferedReader(new FileReader("archivo.txt"))) {
```

- ◆ Usa **try-with-resources**, que **cierra automáticamente** el archivo al final, incluso si ocurre una excepción.
- ◆ Intenta abrir el archivo "archivo.txt" ubicado en el **directorio raíz del proyecto** o donde estés ejecutando el programa.

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

- ◆ Lee cada línea del archivo, una por una, y la imprime en consola.
- ◆ El bucle termina cuando readLine() devuelve null, es decir, cuando llega al **final del archivo (EOF)**.

```
} catch (IOException e) { System.err.println("Error leyendo el archivo: " + e.getMessage()); }
```

- ◆ Captura y muestra errores si:
  - El archivo no existe.
  - Hay problemas de permisos.
  - O cualquier otro error de entrada/salida.



# Módulo 1.- Programación Avanzada en Java

## Excepciones personalizadas y manejo eficiente de recursos

### Ejercicio 4: try-catch múltiple y jerarquía

Objetivo: Comprender el orden y prioridad de catch al capturar errores.

```
public class DemoErrores {  
    public static void main(String[] args) {  
        try {  
            String valor = null;  
            System.out.println(valor.length());  
        } catch (NullPointerException e) {  
            System.err.println("Error: objeto nulo");  
        } catch (Exception e) {  
            System.err.println("Error general: " + e.getMessage());  
        }  
    }  
}
```

### Discusión:

- ¿Qué ocurre si se invierte el orden de los catch?
- ¿Qué pasa si se elimina el NullPointerException y ocurre?



¿Qué hace este código?

```
String valor = null;  
System.out.println(valor.length());
```

- ◆ Aquí estás declarando una variable `valor` con valor `null`.
- ◆ Luego intentas acceder a `valor.length()`, lo que provoca una excepción en tiempo de ejecución:  
 `NullPointerException`

Porque estás intentando acceder a un método sobre un objeto que **no ha sido instanciado**.

 Manejo de la excepción:

```
} catch (NullPointerException e) {  
    System.err.println("Error: objeto nulo");  
}
```

- ◆ Este bloque **atrapa específicamente** la excepción `NullPointerException` y muestra un mensaje claro para el usuario.

```
} catch (Exception e) {  
    System.err.println("Error general: " + e.getMessage());  
}
```

- ◆ Este segundo bloque es una "**red de seguridad**" para atrapar cualquier otra excepción no anticipada.



# Módulo 1.- Programación Avanzada en Java

## Excepciones personalizadas y manejo eficiente de recursos

💡 Ejercicio extra: Propagación con throws

**Objetivo:** Aplicar throws para delegar la gestión de errores.

```
public static void validarEdad(int edad) throws IllegalArgumentException {  
    if (edad < 18) throw new IllegalArgumentException("Debes ser mayor de edad.");  
}
```

```
public static void main(String[] args) {  
    validarEdad(16);  
}
```

### Discusión:

- ¿Es mejor manejar la excepción aquí o propagarla más arriba?
- ¿Qué diferencia habría si usamos una excepción personalizada?



## ¿Qué hace este código?

```
public static void validarEdad(int edad) throws IllegalArgumentException { if (edad < 18) throw new  
IllegalArgumentException("Debes ser mayor de edad."); }
```

- ◆ Esta función valida si una persona tiene al menos 18 años.
- ◆ Si no cumple la condición, lanza una excepción de tipo `IllegalArgumentException` con un mensaje personalizado.

`IllegalArgumentException` es una excepción **no verificada** (*unchecked*), por eso no necesitas capturarla obligatoriamente.

```
public static void main(String[] args) { validarEdad(16); }
```

- ◆ Aquí llamas a `validarEdad(16)`, por lo tanto se cumple la condición de error ( $16 < 18$ ), y la excepción se lanza.



💡 ¿Qué sucede al ejecutar?

💥 Se lanza una excepción no controlada, y verás algo como:

```
Exception in thread "main" java.lang.IllegalArgumentException: Debes ser mayor de edad.  
    at Demo.validarEdad(Demo.java:2)  
    at Demo.main(Demo.java:6)
```

💡 ¿Cómo manejárla con try-catch?

Si quieras manejárla y evitar que detenga el programa:

```
public static void main(String[] args) {  
    try {  
        validarEdad(16);  
    } catch (IllegalArgumentException e) {  
        System.err.println("⚠️ Validación fallida: " + e.getMessage());  
    }  
}
```

💻 Salida:

⚠️ Validación fallida: Debes ser mayor de edad.



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

### 1. Evitar mutabilidad innecesaria

¿Por qué?

**La mutabilidad descontrolada puede causar efectos secundarios inesperados, especialmente en sistemas concurrentes o con múltiples capas.**

¿Qué hacer?

- Declara tus clases como **inmutables** siempre que sea posible.
- Evita exponer referencias modificables (`getLista()` que retorna una lista modificable).
- Usa constructores para establecer valores definitivos.



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

```
public final class Persona {  
    private final String nombre;  
    private final int edad;  
  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    public String getNombre() { return nombre; }  
    public int getEdad() { return edad; }  
}
```



¿Qué hace esta clase?

```
public final class Persona {
```

- ◆ La palabra clave **final** significa que **la clase no se puede extender** (no se pueden crear subclases). Esto ayuda a mantener su comportamiento seguro e inmutable.

```
private final String nombre;  
private final int edad;
```

- ◆ Los atributos son **private** y **final**, lo que significa:
  - **private**: solo se accede desde dentro de la clase.
  - **final**: solo pueden ser asignados una vez (en el constructor).

Esto hace que los objetos de esta clase sean **inmutables**: su estado no puede cambiar después de ser creados.

```
public Persona(String nombre, int edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
}
```



- ◆ El constructor requiere los dos valores y los asigna una única vez.

```
public String getNombre() { return nombre; }  
public int getEdad() { return edad; }
```

- ◆ Métodos **getters** públicos para acceder a los atributos, **sin permitir modificación**.

💡 ¿Cómo usar esta clase?

```
Persona p = new Persona("Ingrid", 30);  
System.out.println(p.getNombre()); // Ingrid  
System.out.println(p.getEdad()); // 30
```

❓ ¿Por qué es útil una clase inmutable?

✓ Ventajas:

- **Seguridad** en entornos concurrentes (threads).
- Más fácil de razonar, probar y mantener.
- Evita efectos secundarios inesperados.



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

### 2. Uso de final y colecciones inmutables

#### **final en variables y métodos:**

- Evita reasignaciones accidentales.
- Mejora la comprensión y legibilidad del código.
- Ayuda al compilador y a otras personas a entender tu intención.



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

```
final List<String> nombres = List.of("Ingrid", "Pedro");
// nombres.add("Juan"); // ERROR: la lista es inmutable
```

### Colecciones inmutables:

- Desde Java 9 puedes usar `List.of()`, `Set.of()`, `Map.of()` para crear colecciones inmutables directamente.
- También puedes usar librerías como Guava o Lombok con `@Value`.



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

### 3. Minimizar el uso de null

¿Por qué evitarlo?

- El NullPointerException es una de las fuentes más comunes de bugs en Java.

Alternativas:

- Usa Optional<T> cuando un valor puede estar presente o no, en vez de devolver null.

```
public Optional<String> buscarPorId(int id) {  
    return Optional.ofNullable(db.get(id));  
}
```

Siempre valida si el valor está presente:

```
buscarPorId(5).ifPresent(nombre -> System.out.println("Hola " + nombre));
```



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

### 4. Prefiere expresiones sobre instrucciones imperativas

**Imperativo (tradicional):** Tú le dices al programa **cómo** hacer algo paso a paso (instrucciones).

```
List<String> mayusculas = new ArrayList<>();  
for (String nombre : nombres) {  
    if (nombre.startsWith("I")) {  
        mayusculas.add(nombre.toUpperCase());  
    }  
}  
Collections.sort(mayusculas);
```



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

**Declarativo (con Streams):** Tú le dices **qué** hacer, no **cómo**.

```
List<String> mayusculas = nombres.stream()  
    .filter(n -> n.startsWith("I"))  
    .map(String::toUpperCase)  
    .sorted()  
    .collect(Collectors.toList());
```

### Ventajas:

- Más legible
- Menos propenso a errores
- Más fácil de testear y mantener
- Se alinea con principios funcionales



 ¿Por qué preferir expresiones?

 **1. Más legible**

- Se enfoca en *qué se quiere lograr*, no en *cómo hacerlo* paso a paso.

 **2. Menos código repetido y más conciso**

- Evita inicialización de listas, bucles, condiciones, etc.

 **3. Más seguro (sin efectos secundarios)**

- Al evitar modificar estructuras externas (`resultado.add(...)`), se reduce el riesgo de errores y efectos colaterales.

 **4. Encadenamiento fluido**

- Puedes combinar filtros, transformaciones y agrupamientos en una sola línea clara.

 ¿Cuándo **no** usarlo?

- Si la lógica es **demasiado compleja o con múltiples efectos secundarios** (como escribir archivos, interactuar con APIs, etc.).
- Si necesitas **máximo rendimiento**, a veces el estilo imperativo es más eficiente (en casos muy puntuales).



# Módulo 1.- Programación Avanzada en Java

## Buenas prácticas generales

**Consejo final: Código limpio ≠ código complejo**

Adoptar estas buenas prácticas **no es para que el código sea “sofisticado”**, sino para que sea **más fácil de entender y mantener para ti y tu equipo en el futuro.**

