Programación Funcional en Java

Círculo Siete Capacitación

Clase 3 de 12

1 Marzo 2025

Funcionalidad de *Function*, *Predicate*, *Consumer* y *Supplier* en Java

- Desde Java 8, se introdujeron las interfaces funcionales en el paquete ***java.util.function***, facilitando el uso de expresiones lambda y promoviendo la programación funcional en Java. Entre las más utilizadas están:
  - ***Function<T, R>*** → Transformación de datos
  - ***Predicate<T>*** → Evaluación de condiciones
  - ***Consumer<T>*** → Ejecución de acciones sin retorno
  - ***Supplier<T>*** → Provisión de datos sin entrada

Function<T, R> - Transformación de Datos

- Propósito:
  - Representa una función que recibe un parámetro de tipo ***T*** y devuelve un resultado de tipo ***R***.
- Ejemplo: Convertir una lista de palabras en mayúsculas.
- Uso común:
  - Transformación de datos con ***map()*** en ***Streams***.
  - Composición de funciones con ***andThen()*** y ***compose()***.

Ejemplo de Composición de Funciones

```
Function<Integer, Integer> doblar = x -> x * 2;
Function<Integer, Integer> incrementar = x -> x + 1;

Function<Integer, Integer> dobleYSumar = doblar.andThen(incrementar);

System.out.println(dobleYSumar.apply(3)); // (3 * 2) + 1 = 7
```

- ***andThen()*** aplica la segunda función después de la primera.
- ***compose()*** aplica la segunda función antes de la primera.

Predicate<T> - Evaluación de Condiciones

- Propósito:
  - Representa una función que recibe un parámetro de tipo T y devuelve un boolean, indicando si cumple una condición.
- Ejemplo: Filtrar números pares en una lista.
- Uso común:
  - Filtrar datos en filter() de Streams.
  - Composición de condiciones con and(), or(), negate().

Consumer<T> - Ejecución de Acciones sin Retorno

- Propósito:
  - Representa una función que recibe un parámetro T pero no devuelve nada. Se usa para realizar acciones como imprimir, modificar estructuras o registrar logs.
- Ejemplo: Imprimir nombres en una lista.
- Uso común:
  - Recorrer listas con forEach().
  - Registrar logs o depurar información.

Supplier<T> - Generación de Datos sin Parámetros

- Propósito:
  - Representa una función que no recibe parámetros pero devuelve un valor T. Se usa para generar datos bajo demanda.
- Ejemplo: Obtener un número aleatorio.
- Uso común:
  - Generar valores en la inicialización de datos.
  - Retrasar la ejecución de cálculos hasta que sean necesarios (Lazy Evaluation).

Comparación de Function, Predicate, Consumer y Supplier

Interfaz Funcional	Parámetro de Entrada	Valor de Salida	Uso Principal
Function<T, R>	Sí (T)	Sí (R)	Transformar datos
Predicate<T>	Sí (T)	Sí (boolean)	Evaluar condiciones
Consumer<T>	Sí (T)	No	Ejecutar acciones
Supplier<T>	No	Sí (T)	Generar valores

Ejemplo Combinado

- Aplicando todas las interfaces funcionales en un mismo flujo de datos.

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.*;

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

        // Predicate: Filtrar nombres que empiezan con "A"
        Predicate<String> empiezaConA = nombre -> nombre.startsWith("A");

        // Function: Convertir a mayúsculas
        Function<String, String> aMayusculas = String::toUpperCase;

        // Consumer: Imprimir nombre
        Consumer<String> imprimir = System.out::println;

        // Aplicar la transformación funcional
        nombres.stream()
               .filter(empiezaConA)       // Filtrar nombres con "A"
               .map(aMayusculas)          // Convertir a mayúsculas
               .forEach(imprimir);        // Imprimir resultado
    }
}
```

Conclusión

- Las interfaces funcionales Function, Predicate, Consumer y Supplier son esenciales para la programación funcional en Java, permitiendo:
  - Código más conciso y expresivo con lambdas.
  - Operaciones con Streams de forma declarativa.
  - Composición de funciones para mayor reutilización.
- Su uso adecuado mejora la legibilidad, reduce código innecesario y potencia la programación funcional en Java.

Definición y Uso de Interfaces Funcionales Personalizadas en Java

- Desde Java 8, el paradigma funcional en Java ha sido impulsado mediante interfaces funcionales, las cuales permiten definir funciones como objetos.
- Aunque Java proporciona interfaces funcionales predefinidas en el paquete java.util.function (Function, Predicate, Consumer, Supplier, etc.), también podemos crear nuestras propias interfaces funcionales personalizadas.

¿Qué es una Interfaz Funcional?

- Una interfaz funcional es una interfaz que tiene un único método abstracto.
- Se usa con expresiones lambda o referencias a métodos.
- Puede contener métodos default y static, pero solo un método abstracto.
- Se recomienda usar la anotación @FunctionalInterface para asegurar que sigue esta regla.

Creación de una Interfaz Funcional Personalizada

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

- Usamos @FunctionalInterface para asegurar que solo tenga un método abstracto.
- Solo tiene un método ejecutar(int, int), permitiendo su uso con lambdas.
- Puede tener métodos default o static, pero no más de un método abstracto.

Uso de la Interfaz Funcional con Expresiones Lambda

```java
public class Main {
    public static void main(String[] args) {
        // Implementación con lambda
        Operacion suma = (a, b) -> a + b;
        Operacion multiplicacion = (a, b) -> a * b;

        System.out.println(suma.ejecutar(5, 3)); // 8
        System.out.println(multiplicacion.ejecutar(5, 3)); // 15
    }
}
```

Agregando Métodos default y static en la Interfaz Funcional

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

    // Método default
    default void mostrarResultado(int a, int b) {
        System.out.println("Resultado: " + ejecutar(a, b));
    }

    // Método static
    static void imprimirMensaje() {
        System.out.println("Ejecutando una operación matemática...");
    }
}
```

Comparación con Interfaces Funcionales Predefinidas

Interfaz Funcional	Parámetro de Entrada	Retorno	Uso Principal
Function<T, R>	T	R	Transformación de datos
Predicate<T>	T	boolean	Evaluar condiciones
Consumer<T>	T	void	Ejecutar acciones
Supplier<T>	Ninguno	T	Generar valores
Interfaz Personalizada	Cualquier número de parámetros	Definido por el usuario	Lógica específica del negocio

Cuándo Usar Interfaces Funcionales Personalizadas

- Cuando se necesita una operación muy específica que no cubren las interfaces estándar.
- Cuando se requiere una función con múltiples parámetros.
- Cuando se desea mayor claridad en el código en lugar de usar Function, Predicate, etc.

Conclusión

- Las interfaces funcionales personalizadas permiten crear funciones reutilizables y expresivas, potenciando la programación funcional en Java.
- Beneficios:
  - Código más claro y modular.
  - Mayor flexibilidad en el diseño del código.
  - Mejor integración con lambdas y Streams.
- Cuando las interfaces funcionales predefinidas (Function, Predicate, etc.) no son suficientes, crear interfaces personalizadas permite adaptar el código a necesidades específicas.

¿Qué son las Funciones de Orden Superior en Java?

- Las Funciones de Orden Superior (Higher-Order Functions) son funciones que pueden recibir otras funciones como parámetros o devolver una función como resultado.
- Este concepto es clave en la programación funcional, ya que permite escribir código más modular, reutilizable y expresivo.

Definición de Función de Orden Superior

- Una función de orden superior es aquella que:
  - Recibe una función como argumento
  - Devuelve una función como resultado
- En Java, esto se implementa utilizando interfaces funcionales como ***Function<T, R>***, ***Predicate<T>***, ***Consumer<T>***, o creando interfaces personalizadas.

```java
import java.util.function.Function;

/**
 * Ejemplo de Función de Orden Superior en Java
 */
public class Lab01 {
    // Función de orden superior que aplica otra función sobre un número
    public static int operar(int numero, Function<Integer, Integer> operacion) {
        return operacion.apply(numero);
    }

    public static void main(String[] args) {
        // Expresión lambda para duplicar un número
        Function<Integer, Integer> duplicar = x -> x * 2;

        // Usamos la función de orden superior
        System.out.println(operar(5, duplicar)); // 10
    }
}
```

```java
import java.util.function.Function;

/**
 * Función que genera funciones matemáticas dinámicamente
 */
public class Lab02 {
    // Función de orden superior que devuelve otra función
    public static Function<Integer, Integer> crearMultiplicador(int factor) {
        return x -> x * factor;
    }

    public static void main(String[] args) {
        Function<Integer, Integer> porTres = crearMultiplicador(3);

        System.out.println(porTres.apply(4)); // 12
    }
}
```

Uso de Funciones de Orden Superior con Streams

- En la API de Streams, se usan funciones de orden superior en métodos como ***map(), filter(), reduce()***, etc.
  - ***map()*** toma una función y la aplica a cada elemento de la lista.
  - ***filter()*** recibe un ***Predicate***, que es una función de orden superior.

Ventajas de las Funciones de Orden Superior

- Reutilización de código
  - Podemos pasar funciones como parámetros sin duplicar lógica.
- Mayor flexibilidad
  - Permiten construir código más dinámico.
- Código más limpio y expresivo
  - Se eliminan bucles innecesarios.
- Mejor integración con Streams y APIs funcionales
  - Uso eficiente con ***map(), filter(), reduce()***, etc.

Comparación con Métodos Tradicionales

Enfoque Tradicional	Enfoque con Función de Orden Superior
Usa estructuras ***for*** repetitivas	Usa ***map(), filter(), reduce()***
Código extenso y difícil de modificar	Código más modular y reutilizable
Menos flexible	Permite personalizar lógica con funciones

```java
import java.util.Arrays;
import java.util.function.Function;
import java.util.function.Predicate;

public class Lab03 {
    public static void main(String[] args) {
        final var numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Definir funciones de orden superior
        Function<Integer, Integer> doblar = x -> x * 2;
        Predicate<Integer> esPar = x -> x % 2 == 0;

        // Aplicar transformación con `map()` y filtrar con `filter()`
        final var resultado = numeros.stream()
                .map(doblar)       // Multiplicar cada número por 2
                .filter(esPar)     // Filtrar solo los pares
                .toList();

        System.out.println(resultado); // [4, 8, 12, 16, 20]
    }
}
```

Conclusión

- Las funciones de orden superior permiten escribir código más flexible, modular y reutilizable en Java.
- Se usan en:
  - Transformaciones de datos con ***Function<T, R>***.
  - Filtrado de listas con ***Predicate<T>***.
  - Aplicaciones funcionales con Streams (***map(), filter()***, etc.).
- Adoptar funciones de orden superior en Java permite programar de forma más expresiva y eficiente, facilitando la transición a un paradigma funcional.

Uso de Funciones como Parámetros y Valores de Retorno en Java

- Desde Java 8, la programación funcional permite que las funciones sean ciudadanos de primera clase, es decir, pueden ser pasadas como parámetros y devueltas como resultado en otras funciones.

```java
import java.util.function.Function;

/**
 * Pasar una Función como Parámetro
 */
public class Lab04 {
    // Función de orden superior que recibe otra función como parámetro
    public static int aplicarOperacion(int numero, Function<Integer, Integer> operacion) {
        return operacion.apply(numero);
    }

    public static void main(String[] args) {
        // Definir funciones lambda
        Function<Integer, Integer> cuadrado = x -> x * x;
        Function<Integer, Integer> triple = x -> x * 3;

        // Pasar funciones como parámetros
        System.out.println(aplicarOperacion(5, cuadrado)); // 25
        System.out.println(aplicarOperacion(4, triple));   // 12
    }
}
```

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

/**
 * Usamos Predicate<T> para recibir una función que evalúa una condición.
 */
public class Lab05 {
    // Función de orden superior que filtra una lista según una condición
    public static List<Integer> filtrar(
            List<Integer> numeros,
            Predicate<Integer> condicion
    ) {
        return numeros.stream()
                .filter(condicion)
                .toList();
    }

    public static void main(String[] args) {
        final var numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

        // Pasamos diferentes funciones como parámetros
        System.out.println(filtrar(numeros, n -> n % 2 == 0));  // [2, 4, 6, 8]
        System.out.println(filtrar(numeros, n -> n > 5));       // [6, 7, 8, 9]
    }
}
```

```java
import java.util.function.Function;

/**
 * Retornar una Función Personalizada
 */
public class Lab06 {
    // Función de orden superior que devuelve una función multiplicadora
    public static Function<Integer, Integer> crearMultiplicador(int factor) {
        return x -> x * factor;
    }

    public static void main(String[] args) {
        // Obtenemos una función multiplicadora
        final var porCinco = crearMultiplicador(5);

        // Aplicamos la función
        System.out.println(porCinco.apply(2)); // 10
        System.out.println(porCinco.apply(3)); // 15
    }
}
```

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

/**
 * Retornar una Función con Parámetro Personalizado.
 * Generamos funciones que comparan números de forma flexible.
 */
public class Lab07 {
    // Función que devuelve una función que compara con un umbral
    public static Predicate<Integer> mayorQue(int umbral) {
        return n -> n > umbral;
    }

    public static void main(String[] args) {
        final var mayorQueDiez = mayorQue(10);

        System.out.println(mayorQueDiez.test(5));  // false
        System.out.println(mayorQueDiez.test(15)); // true
    }
}
```

```java
import java.util.Arrays;
import java.util.function.Function;
import java.util.function.Predicate;

/**
 * Combinación de Funciones en un Flujo Completo
 */
public class Lab08 {
    public static void main(String[] args) {
        final var numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Definir funciones
        Function<Integer, Integer> doblar = x -> x * 2;
        Predicate<Integer> esMayorQueDiez = x -> x > 10;

        // Aplicar transformaciones con map() y filter()
        final var resultado = numeros.stream()
                .map(doblar)            // Multiplica por 2
                .filter(esMayorQueDiez)  // Filtra mayores a 10
                .toList();

        System.out.println(resultado); // [12, 14, 16, 18, 20]
    }
}
```

Beneficios de Usar Funciones como Parámetros y Retorno

- Código más reutilizable
  - Podemos pasar funciones sin modificar la función principal.
- Mayor flexibilidad
  - Se pueden definir comportamientos dinámicos en tiempo de ejecución.
- Código más declarativo
  - En lugar de escribir bucles y condicionales, usamos map(), filter(), etc.
- Composición de funciones
  - Podemos combinar funciones para construir flujos de datos más expresivos.

Conclusión

- Pasar funciones como parámetros y retornarlas permite un código más flexible y expresivo en Java.
- Se usa en:
  - Transformación de datos (Function<T, R>)
  - Filtrado de colecciones (Predicate<T>)
  - Ejecución de acciones (Consumer<T>)
  - Generación de valores (Supplier<T>)
- Adoptar esta técnica facilita la transición a la programación funcional, permitiendo construir aplicaciones más dinámicas y modulares.

Composición de Funciones y su Uso Práctico en Java

- La composición de funciones es un concepto fundamental en programación funcional, donde múltiples funciones se combinan para crear una nueva función.
- En Java, esto se logra mediante las interfaces funcionales de ***java.util.function***, como ***Function<T, R> y BiFunction<T, U, R>***, utilizando métodos como ***andThen()*** y ***compose()***.

¿Qué es la Composición de Funciones?

- Definición:
  - La composición de funciones consiste en combinar dos o más funciones para producir un resultado sin modificar su implementación interna.
- Dos métodos clave para la composición en Java:
  - ***andThen()*** → Ejecuta la primera función y pasa el resultado a la segunda.
  - ***compose()*** → Ejecuta la segunda función y pasa el resultado a la primera.
- Ejemplo Conceptual, si tenemos las funciones:
  - ***f(x) = x + 2***
  - ***g(x) = x * 3***
- Entonces:
  - ***f.andThen(g).apply(5) → g(f(5))*** = (5 + 2) * 3 = 21
  - ***f.compose(g).apply(5) → f(g(5))*** = (5 * 3) + 2 = 17

```java
import java.util.function.Function;

/**
 * Composición de Funciones con andThen()
 * Convertir un número a String y calcular su longitud
 */
public class Lab09 {
    public static void main(String[] args) {
        Function<Integer, String> convertirAString = "Número: %d"::formatted;
        Function<String, Integer> calcularLongitud = String::length;

        // Composición con andThen()
        final var longitudDelNumero = convertirAString.andThen(calcularLongitud);

        System.out.println(longitudDelNumero.apply(100)); // 11
    }
}
```

```java
import java.util.function.Function;

/**
 * Composición de Funciones con compose()
 * Convertir un número a su doble y luego restarle 5
 */
public class Lab10 {
    public static void main(String[] args) {
        Function<Integer, Integer> doblar = x -> x * 2;
        Function<Integer, Integer> restarCinco = x -> x - 5;

        // Composición con compose()
        final var operacion = restarCinco.compose(doblar);

        System.out.println(operacion.apply(6)); // (6 * 2) - 5 = 7
    }
}
```

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;

/**
 * Composición de Funciones en Flujos de Datos con map()
 * Convertir nombres a mayúsculas y contar sus caracteres
 */
public class Lab11 {
    public static void main(String[] args) {
        List<String> nombres = Arrays.asList("ana", "pedro", "luis");

        Function<String, String> aMayusculas = String::toUpperCase;
        Function<String, Integer> contarCaracteres = String::length;

        // Composición con Streams
        final var resultado = nombres.stream()
                .map(aMayusculas.andThen(contarCaracteres))
                .toList();

        System.out.println(resultado); // [3, 5, 4]
    }
}
```

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

/**
 * Composición de Predicados (Predicate<T>)
 * Filtrar números pares y mayores a 10
 */
public class Lab12 {
    public static void main(String[] args) {
        final var numeros = Arrays.asList(5, 8, 12, 15, 20);

        Predicate<Integer> esPar = x -> x % 2 == 0;
        Predicate<Integer> mayorQueDiez = x -> x > 10;

        // Composición con and()
        final var resultado = numeros.stream()
                .filter(esPar.and(mayorQueDiez))
                .toList();

        System.out.println(resultado); // [12, 20]
    }
}
```

```java
import java.util.function.BiFunction;
import java.util.function.Function;


/**
 * Uso de BiFunction<T, U, R> en Composición
 * Función que suma dos números y luego los multiplica por un factor
 */
public class Lab13 {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> sumar = Integer::sum;
        Function<Integer, Integer> multiplicarPorDos = x -> x * 2;

        // Composición con andThen()
        final var operacion = sumar.andThen(multiplicarPorDos);

        System.out.println(operacion.apply(3, 5)); // (3 + 5) * 2 = 16
    }
}
```

Beneficios de la Composición de Funciones

- Código más modular y reutilizable
  - Se pueden combinar pequeñas funciones en flujos de trabajo más grandes.
- Mayor legibilidad y expresividad
  - En lugar de escribir múltiples transformaciones manualmente, las funciones se encadenan elegantemente.
- Evita la repetición de código
  - No es necesario escribir lógica repetitiva, ya que cada función se puede reutilizar y combinar.
- Integración con Streams y API funcionales
  - Se usa en map(), filter(), reduce(), etc.

API de Streams en Java

- La API de Streams en Java se introdujo en Java 8 como parte del paquete ***java.util.stream***.
- Permite procesar colecciones de datos de forma declarativa y funcional, eliminando la necesidad de bucles ***for*** y mejorando la eficiencia del código.

¿Qué es un Stream en Java?

- Un Stream es una secuencia de elementos que permite realizar operaciones como filtrado, transformación, agregación y reducción de datos de manera eficiente y expresiva.
- Características clave:
  - Inmutable: No modifica la colección original.
  - Pipelines: Se compone de una secuencia de operaciones encadenadas.
  - Lazy Evaluation: Se ejecuta solo cuando es necesario (evaluación perezosa).
  - Soporta procesamiento paralelo: Puede ejecutarse en múltiples hilos (***parallelStream()***).

```java
import java.util.Arrays;
import java.util.stream.Stream;

/**
 * Tipos de Streams en Java
 */
public class Lab14 {
    public static void main(String[] args) {
        // Streams de Colecciones
        final var lista = Arrays.asList("A", "B", "C");
        lista.stream().forEach(System.out::println);
        // alternativa
        lista.forEach(System.out::println);

        // Streams de Arrays
        int[] numeros = {1, 2, 3, 4, 5};
        Arrays.stream(numeros).forEach(System.out::println);

        // Streams con Stream.of()
        final var stream = Stream.of("Java", "Python", "Go");
        stream.forEach(System.out::println);

        // Streams Infinitos (generate() y iterate())
        final var aleatorios = Stream
                .generate(Math::random)
                .limit(5);
        aleatorios.forEach(System.out::println);

        final var secuencia = Stream
                .iterate(1, n -> n + 2)
                .limit(5);
        secuencia.forEach(System.out::println); // 1, 3, 5, 7, 9
    }
}
```

Operaciones en Streams

- Las operaciones en Streams se dividen en:
  - Intermedias: Devuelven otro Stream (lazy evaluation).
  - Finales: Ejecutan la operación y devuelven un resultado.

Operaciones Intermedias (Transforman el Stream)

- map() – Transformación de elementos
- filter() – Filtrar elementos
- sorted() – Ordenar elementos
- distinct() – Eliminar duplicados
- limit() y skip() – Control de elementos
- peek() - Ojeada de elementos

Operaciones Finales (Ejecutan el Stream)

- forEach() – Iterar sobre los elementos
- collect() – Convertir a lista, set o mapa
- reduce() – Reducción de valores
- count() – Contar elementos
- anyMatch(), allMatch(), noneMatch() – Evaluaciones
- findFirst() y findAny() – Obtener elementos

Procesamiento Paralelo con parallelStream()

- Los Streams pueden ejecutarse en paralelo, dividiendo el procesamiento en múltiples núcleos.
- Cuándo usar parallelStream():
  - Cuando se procesan grandes volúmenes de datos.
  - Cuando se realizan operaciones independientes sin dependencia de orden.

Funcionamiento Interno de Streams Paralelos en Java

- La API de Streams en Java permite el procesamiento paralelo utilizando parallelStream() o parallel(), dividiendo el trabajo en múltiples hilos para mejorar el rendimiento en grandes volúmenes de datos.

¿Cómo Funciona Internamente un Stream Paralelo?

- Cuando un Stream se convierte en paralelo (***parallelStream()***), Java utiliza un mecanismo basado en Fork/Join Framework (ForkJoinPool), el cual:
  - Divide la colección en partes más pequeñas (fork).
  - Procesa cada parte en un hilo separado en un pool de hilos (ForkJoinPool).
  - Une los resultados después de procesar cada sublista (join).

Ejemplo Visual de Ejecución

```
Lista Original:     [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

División:
  Hilo 1 → [1, 2, 3, 4, 5]
  Hilo 2 → [6, 7, 8, 9, 10]

Procesamiento:
  Hilo 1 → reduce([1, 2, 3, 4, 5]) → 15
  Hilo 2 → reduce([6, 7, 8, 9, 10]) → 40

Fusión:
  15 + 40 = 55
```

Fork/Join Framework y ForkJoinPool

- Internamente, parallelStream() usa un ForkJoinPool para manejar la concurrencia.
- ForkJoinPool es un pool de hilos basado en el algoritmo "Divide y vencerás"
  - Usa división recursiva (fork) de las tareas.
  - Luego combina (join) los resultados.
- Obtener el número de hilos en el ForkJoinPool predeterminado:
  - System.out.println(ForkJoinPool.commonPool().getParallelism());
- Por defecto, el número de hilos es N - 1, donde N es el número de núcleos del procesador.
- Ejemplo: Modificar el número de hilos (System.setProperty)
  - System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "4");
- Se recomienda modificar el número de hilos solo cuando se tiene un control preciso de la concurrencia.

Cuándo Usar parallelStream() y Cuándo Evitarlo

- Usar parallelStream() cuando:
  - Hay grandes volúmenes de datos (miles o millones de elementos).
  - La operación es independiente por cada elemento (ejemplo: suma, multiplicación, transformación).
  - Se requiere mejorar el rendimiento en múltiples núcleos.
  - La operación en cada elemento es costosa en CPU y se puede paralelizar.
- Evitar parallelStream() cuando:
  - Se trabaja con colecciones pequeñas (el costo de crear hilos es mayor que el beneficio).
  - La operación es dependiente del orden (forEachOrdered() es mejor en estos casos).
  - Se trabaja con estructuras mutables como ArrayList, Map o Set sin sincronización.
  - Se está en un entorno con alta concurrencia, donde los hilos pueden competir por recursos.

Conclusión

- parallelStream() mejora el rendimiento en grandes volúmenes de datos, pero no siempre es la mejor opción.
- Resumen:
  - Usa ForkJoinPool para dividir y procesar datos en múltiples hilos.
  - Es ideal para operaciones independientes y colecciones grandes.
  - Puede reducir el rendimiento en colecciones pequeñas o en operaciones dependientes del orden.
  - Evitar en estructuras mutables sin sincronización.
- Recomendación: Probar siempre con stream() y luego con parallelStream() para medir la mejora real en cada caso.
- Sobrecarga de gestión de hilos: Si el trabajo es ligero o la cantidad de datos es pequeña, el costo de crear y gestionar los hilos puede ser mayor que el beneficio de procesarlo en paralelo.