



Programación Funcional en Java

Círculo Siete Capacitación

Clase 2 de 12
26 Febrero 2025

Sintaxis Básica de Expresiones Lambda en Java

- Desde Java 8, las expresiones lambda permiten escribir código más conciso y funcional.
- Son una forma de definir funciones anónimas, eliminando la necesidad de crear clases o métodos adicionales para implementar interfaces funcionales.

Estructura de una Expresión Lambda

- La sintaxis general de una expresión lambda en Java es la siguiente:
- Si la expresión es más compleja, se pueden usar llaves **`{}`** para definir un bloque de código:



```
(parametros) -> expresión
```

```
(parametros) -> {  
    // Cuerpo de la función  
}
```

Ejemplo Básico: Lambda con un Solo Parámetro

```
// Antes de Java 8 (Clase Anónima)
Function<Integer, Integer> cuadrado = new Function<Integer, Integer>() {
    @Override
    public Integer apply(Integer x) {
        return x * x;
    }
};

// Después de Java 8 (Expresión Lambda)
Function<Integer, Integer> cuadrado = x -> x * x;
System.out.println(cuadrado.apply(5)); // 25
```

- Diferencias:
 -  Se elimina la necesidad de definir la interfaz manualmente.
 -  Se reduce la sintaxis, haciéndola más legible.

Variantes de Sintaxis Lambda

```
//Lab01
// Lambda con Múltiples Parámetros
BiFunction<Integer, Integer, Integer> suma = (a, b) -> a + b;
System.out.println(suma.apply(10, 5)); // 15

// Lambda con un Solo Parámetro (Sin Paréntesis)
// Si hay un solo parámetro, los paréntesis () pueden omitirse
Consumer<String> imprimir = mensaje -> System.out.println(mensaje);
imprimir.accept("Hola, Java!"); // Hola, Java!

// Lambda con Bloque de Código
Function<Integer, Integer> factorial = n -> {
    int resultado = 1;
    for (int i = 1; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
};
System.out.println(factorial.apply(5)); // 120

// Lambda sin Parámetros
Supplier<Double> aleatorio = () -> Math.random();
System.out.println(aleatorio.get()); // Un número aleatorio

// Referencias a Métodos (::)
// Si una lambda solo llama a un método existente, se puede usar una referencia a método (::)
Function<String, Integer> longitud = String::length;
System.out.println(longitud.apply("Lambda")); // 6
```

Uso de Lambdas con Interfaces Funcionales

- Una interfaz funcional es una interfaz que tiene un solo método abstracto.

```
// Lab02
@FunctionalInterface
interface Operacion {
    int ejecutar(int a, int b);
}

// Creación de la implementación
Operacion multiplicacion = (x, y) -> x * y;
System.out.println(multiplicacion.ejecutar(3, 4)); // 12
```

Ventajas de las Expresiones Lambda

- Menos código: Eliminan la necesidad de clases anónimas.
- Mayor legibilidad: Expresan la intención de manera concisa.
- Uso con Streams y APIs funcionales: Permiten manipular colecciones de manera declarativa.
- Facilitan la programación concurrente: Son inmutables y seguras para la concurrencia.

Conclusión Sintaxis

- Las expresiones lambda permiten escribir código más expresivo y funcional en Java.
- Son esenciales para trabajar con Streams, funciones de orden superior y programación concurrente, mejorando la legibilidad y reduciendo la complejidad del código.

Funciones Anónimas vs Expresiones Lambda en Java

- Las funciones anónimas y las expresiones lambda permiten definir comportamientos sin necesidad de crear clases explícitas.
- Sin embargo, las expresiones lambda son más concisas y mejoran la legibilidad del código. A continuación, se comparan ambos enfoques en Java.

¿Qué es una Función Anónima?

- Una función anónima en Java se implementa a través de clases anónimas.
- Estas clases no tienen un nombre y se utilizan principalmente para definir implementaciones rápidas de interfaces.





Ejemplo: Función Anónima con una Interfaz Funcional

```
// Lab03
// Definición de una interfaz funcional
interface Operacion {
    int ejecutar(int a, int b);
}

// Implementación usando una clase anónima
public class Main {
    public static void main(String[] args) {
        Operacion suma = new Operacion() {
            @Override
            public int ejecutar(int a, int b) {
                return a + b;
            }
        };

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

Características de una Función Anónima





-  Se puede definir dentro del mismo método sin necesidad de una clase concreta.
-  Permite acceder a variables del ámbito exterior.
-  Código más extenso debido a la necesidad de definir métodos y clases.
-  Menos legible, especialmente en métodos pequeños.

¿Qué es una Expresión Lambda?

- Una expresión lambda es una forma más concisa de escribir una función anónima cuando se usa con una interfaz funcional (interfaz con un solo método abstracto).
- Ejemplo: Expresión Lambda Equivalente:

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

Características de una Expresión Lambda

-  Menos código: Reduce la sintaxis innecesaria.
-  Mayor legibilidad: Expresa el propósito de manera clara y concisa.
-  Más flexible: Compatible con APIs funcionales como Streams.
-  Más difícil de entender si no se está familiarizado con el paradigma funcional.

Comparación entre Funciones Anónimas y Expresiones Lambda

Característica	Funciones Anónimas	Expresiones Lambda
Sintaxis	Extensa	Concisa
Legibilidad	Menos legible	Más clara
Requiere new y @Override ?	Sí	No
Uso con Interfaces Funcionales	Sí	Sí
Uso con Streams y Programación Funcional	Menos eficiente	Más eficiente

Ejemplo Comparativo

- Diferencias:
 - La versión con función anónima usa ***new***, ***@Override*** y código más extenso.
 - La versión con lambda es más clara y directa.

```
// Con Función Anónima
Predicate<Integer> esPar = new Predicate<Integer>() {
    @Override
    public boolean test(Integer num) {
        return num % 2 == 0;
    }
};

System.out.println(esPar.test(4)); // true

// Con Expresión Lambda
Predicate<Integer> esPar = num -> num % 2 == 0;
System.out.println(esPar.test(4)); // true
```


Casos de Uso

- Funciones anónimas:
 - Cuando se necesita definir una implementación rápida que solo se usará una vez.
 - Si se requiere acceder a variables locales no finales.
- Expresiones Lambda:
 - Cuando se trabaja con interfaces funcionales (Function, Predicate, Supplier, etc.).
 - Para hacer el código más legible y conciso en APIs modernas como Streams.

Conclusión Funciones anónimas vs lambdas

- Las expresiones lambda son una alternativa más moderna y funcional a las funciones anónimas en Java.
- Reducen el código innecesario y facilitan la escritura de código limpio y mantenible.

¿Por qué una Expresión Lambda es más Eficiente que una Función Anónima en Java?

- Las expresiones lambda son más eficientes que las funciones anónimas debido a cómo el compilador y la JVM las manejan internamente.
- A continuación las principales razones.

Menor Sobrecarga de Objetos

- Funciones Anónimas: Crean Clases Internas Adicionales
 - Cuando se usa una función anónima, Java genera una clase interna adicional en tiempo de ejecución.
 - Cada vez que se crea una función anónima, se genera una nueva instancia de una clase anónima.
 - Esto consume memoria y aumenta la sobrecarga de ejecución.

Ejemplo de Función Anónima: ¿Qué pasa internamente?

```
Runnable tarea = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Ejecutando tarea...");  
    }  
};  
new Thread(tarea).start();
```

- Java crea una nueva clase anónima que implementa Runnable.
- Se genera un archivo .class extra en la compilación.
- Se necesita más memoria para instanciar la clase.

Expresiones Lambda:

Uso de Invocación de Métodos




- Cuando se usa una expresión lambda, el compilador optimiza su representación utilizando invocación de métodos estáticos (***invokedynamic***) en lugar de generar una clase adicional.

Ejemplo con Lambda: ¿Qué pasa internamente?

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

- No se genera una clase anónima nueva.
- Se usa ***invokedynamic***, que optimiza la invocación en tiempo de ejecución.
- Menos memoria y mejor rendimiento.

Uso de *invokedynamic* en Lambdas

- Desde Java 8, la JVM usa la instrucción *invokedynamic* para procesar expresiones lambda. Esto permite:
 -  Menos clases anónimas generadas.
 -  Mayor reutilización de código en tiempo de ejecución.
 -  Mejor rendimiento en comparación con clases anónimas tradicionales.

Ejemplo: Código Bytecode Generado

```
// Para la función anónima:  
ClassName$1.class // Clase anónima extra generada  
  
//Para la expresión lambda:  
// No se genera un archivo de clase extra. JVM usa `invokedynamic`
```

- Conclusión: Las lambdas son procesadas en tiempo de ejecución de manera más eficiente, sin necesidad de crear múltiples clases anónimas.

Mejor Uso de la Memoria (Menos Objetos en Heap)

- Las funciones anónimas generan una nueva instancia cada vez que se usan, lo que aumenta el consumo de memoria.

Qué ocurre internamente?

```
// Ejemplo con Función Anónima (Nueva Instancia Cada Vez)
Supplier<String> saludo1 = new Supplier<String>() {
    @Override
    public String get() {
        return "Hola!";
    }
};

Supplier<String> saludo2 = new Supplier<String>() {
    @Override
    public String get() {
        return "Hola!";
    }
};

// saludo1 y saludo2 son dos objetos distintos en memoria.

// Ejemplo con Lambda (Misma Referencia, No Nuevas Instancias)
Supplier<String> saludo = () -> "Hola!";
```

- La lambda no crea múltiples instancias si se reutiliza la misma referencia.
- Mejor uso de la memoria Heap.

Mejor Soporte para Programación Concurrente

- Las expresiones lambda son más seguras para concurrencia, ya que:
 - Usan inmutabilidad por defecto.
 - No dependen de clases anónimas que podrían generar estados mutables.
 - Son más fáciles de paralelizar con ***Streams*** y ***CompletableFuture***.

Ejemplo con Streams (Declarativo y Seguro)

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);  
numeros.parallelStream()  
    .map(n -> n * 2)  
    .forEach(System.out::println);
```

- ¿Por qué es más eficiente?
 - Java puede optimizar el paralelismo con Streams y Lambdas.
 - En comparación, las funciones anónimas dependen de instancias mutables, lo que dificulta la concurrencia.

Conclusión

- Las Expresiones Lambda son más eficientes que las Funciones Anónimas porque:
 - No generan clases adicionales (Menos consumo de memoria).
 - Usan ***invokedynamic***, optimizando la ejecución en tiempo de ejecución.
 - Evitan la creación innecesaria de objetos (Reutilización de código).
 - Mejoran el rendimiento en programación concurrente.
- ¿Cuándo seguir usando funciones anónimas?
 - Si se necesita acceso a múltiples métodos en la misma implementación.
 - En casos donde se usa código legado que no soporta lambdas.
- Para todo lo demás, es mejor usar lambdas para obtener mejor rendimiento y código más limpio.

Uso de Expresiones Lambda con Colecciones en Java

- Las expresiones lambda en Java permiten manipular colecciones de manera más declarativa y concisa.
- Gracias a ***Streams, forEach, map, filter*** y otras operaciones, podemos escribir código más eficiente y legible sin necesidad de bucles tradicionales.

Iteración con forEach

- En lugar de usar un bucle for, podemos utilizar el método `forEach` con una expresión lambda para recorrer una colección.

```
// Iterar sobre una Lista
List<String> nombres = Arrays.asList("Ana", "Pedro", "Luis");
nombres.forEach(nombre -> System.out.println(nombre));
```

```
// Más legible y menos código que un for.
// También podemos usar una referencia a método (::)
// si solo llamamos un método existente:
nombres.forEach(System.out::println);
```


Transformación de Datos con *map()*

- El método *map()* permite transformar los elementos de una colección en otros valores. Se usa con Streams.

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

List<String> nombresMayusculas = nombres.stream()
                                          .map(String::toUpperCase)
                                          .toList();

System.out.println(nombresMayusculas); // [ANA, PEDRO, LUIS]

// Evita bucles y estructuras auxiliares como listas temporales.
```

Filtrado de Datos con *filter()*

- El método `filter()` permite seleccionar elementos que cumplan una condición.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
```

```
List<Integer> pares = numeros.stream()  
    .filter(n -> n % 2 == 0)  
    .toList();
```

```
System.out.println(pares); // [2, 4, 6]  
// Más declarativo que un for con if.
```

Ordenamiento con *sorted()*

- Podemos ordenar una colección usando `sorted()` con un comparador lambda.

```
List<Integer> numeros = Arrays.asList(5, 2, 8, 1, 3);

List<Integer> ordenados = numeros.stream()
    .sorted()
    .toList();

System.out.println(ordenados); // [1, 2, 3, 5, 8]

// Para ordenar de forma descendente, podemos usar Comparator.reverseOrder():
List<Integer> descendente = numeros.stream()
    .sorted(Comparator.reverseOrder())
    .toList();

System.out.println(descendente); // [8, 5, 3, 2, 1]
```

Reducción de Datos con *reduce()*

- El método *reduce()* combina los elementos de una colección en un único resultado.

```
// Lab04
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

int suma = numeros.stream()
    .reduce(0, (a, b) -> a + b);

System.out.println(suma); // 15

// Alternativa declarativa a un for acumulador.
int suma = numeros.stream().reduce(0, Integer::sum);
```

Agrupamiento de Datos con ***Collectors.groupingBy()***

- Podemos agrupar elementos con ***Collectors.groupingBy()***, útil para clasificar datos.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);  
  
Map<Boolean, List<Integer>> paresEImpares = numeros.stream()  
    .collect(Collectors.groupingBy(n -> n % 2 == 0));  
  
System.out.println(paresEImpares);  
// {false=[1, 3, 5], true=[2, 4, 6]}
```

Eliminación de Duplicados con *distinct()*

- El método *distinct()* elimina valores repetidos.

```
List<Integer> numeros = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
```

```
List<Integer> unicos = numeros.stream()  
                                .distinct()  
                                .toList();
```

```
System.out.println(unicos); // [1, 2, 3, 4, 5]
```

Combinación de Múltiples Operaciones

- Podemos encadenar varias operaciones en una misma secuencia.

```
List<String> nombres = Arrays
    .asList("Ana", "Pedro", "Luis", "Andrea", "Pablo");

List<String> resultado = nombres
    .stream()
    .filter(n -> n.startsWith("A")) // Filtrar nombres que inician con "A"
    .map(String::toUpperCase)       // Convertir a mayúsculas
    .sorted()                       // Ordenar alfabéticamente
    .toList();

System.out.println(resultado); // [ANA, ANDREA]
```


Conclusión

- Las expresiones lambda en Java facilitan la manipulación de colecciones al hacer el código:
 - Más legible y sin bucles innecesarios.
 - Más eficiente gracias a la ejecución optimizada con ***Streams***.
 - Más expresivo, permitiendo trabajar de manera declarativa con los datos.
- Las lambdas y Streams han revolucionado la forma de trabajar con colecciones en Java, haciéndolo más funcional y moderno.

¿Por qué se llama *identity* a algunos parámetros en las lambdas en Java?

- En Java, el término ***identity*** (identidad) se usa en algunas funciones de la API de ***Streams*** y colecciones funcionales para representar un valor neutral que no afecta la operación en la que se usa.
- Esto proviene del concepto matemático de elemento neutro, que es un valor que no cambia el resultado cuando se aplica en una operación.
- En ***reduce()***, evita que el resultado sea ***null*** en caso de listas vacías.
- En ***Function.identity()***, permite usar funciones sin modificar los datos.
- Mejora la legibilidad y mantiene la seguridad del código.
- Si ves ***identity*** en una lambda, significa que el parámetro representa un valor que no afecta el resultado final.

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.

```
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

```
@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

```
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

```
@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.