PROGRAMACIÓN FUNCIONAL EN JAVA

Contenido

1 Introducción

En la **versión 8** de Java se incluyó un API que facilitaba en gran medida el trabajo con colecciones. Este nos permite realizar operaciones sobre la colección, como por ejemplo buscar, filtrar, reordenar, reducir, etc…

Los streams de Java 8 no tienen nada que ver con los flujos de E/S de Java (por ejemplo, FileInputStream). Estos dos conceptos son diferentes y no están relacionados.

Stream es una **interfaz** de Java que se usa para procesar datos de forma eficiente.

Por otra parte, tenemos el concepto de **lambda, que se relaciona con la programación funcional y también se definió en Java 8** para tratar de dotar a Java de un paradigma funcional que difiere del paradigma de la programación imperativa.

Desde la versión 8 de Java podemos **utilizar cualquier clase que implemente la interfaz Collection como si fuese un Stream** con las ventajas que nos ofrece la Programación Funcional y las Expresiones Lambda.

En resumen, los streams en Java son una herramienta muy potente para manipular colecciones de datos de manera concisa y funcional, haciendo uso de las lambdas para ello. Son especialmente útiles para procesar grandes cantidades de datos de manera eficiente.

¿Cuáles son las diferencias entre Stream y Collection en Java 8?

Las colecciones en Java 8 son estructuras de datos que almacenan elementos, mientras que los Streams son secuencias de elementos que pueden ser manipulados y procesados de manera funcional.

2 Lambdas

2.1 Sintaxis

Las expresiones lambda son una **sintaxis para declarar funciones anónimas de forma concisa y expresiva.** Una función anónima es aquella que definimos sin un nombre específico. Es decir, a diferencia de las funciones normales, no se definen con una declaración completa y no requieren un identificador.

Su sintaxis básica es:

(parámetros) -> {cuerpo de la función}

1. El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.

2. **Parámetros**:

    - Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
    - Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

3. **Cuerpo de lambda**:

    - Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula return en el caso de que deban devolver valores.
    - Cuando el cuerpo de la expresión lambda tiene más de una línea, se hace necesario utilizar las llaves y es necesario incluir la cláusula return en el caso de que la función deba devolver un valor.

Algunos **ejemplos de expresiones lambda**:

- z -> z + 2

- () -> System.out.println(“ Mensaje 1”)

- (int longitud, int altura) -> {return altura * longitud;}

- (String x) -> {
  String retorno = x;
  retorno = retorno.concat(“***”);
  return retorno;
  }

Como hemos visto, las expresiones lambda son *funciones anónimas* y pueden ser utilizadas allá donde el tipo aceptado sea una ***interfaz funcional***.

¿Qué es una función anónima y para qué sirve?

El uso de una clase anónima nos permite **crear un objeto que implementa una interfaz** en particular y poder usarlo libremente sin tener que definir explícitamente una clase. Un ***ejemplo*** es el uso en la ordenación de colecciones:

```
ArrayList milista= new ArrayList();

milista.add(new Persona("Miguel"));

milista.add(new Persona("Alicia"));

Collections.sort(milista, new Comparator() {

      public int compare(Persona p1,Persona p2)

      { return p1.getNombre().compareTo(p2.getNombre()); }

 });
```

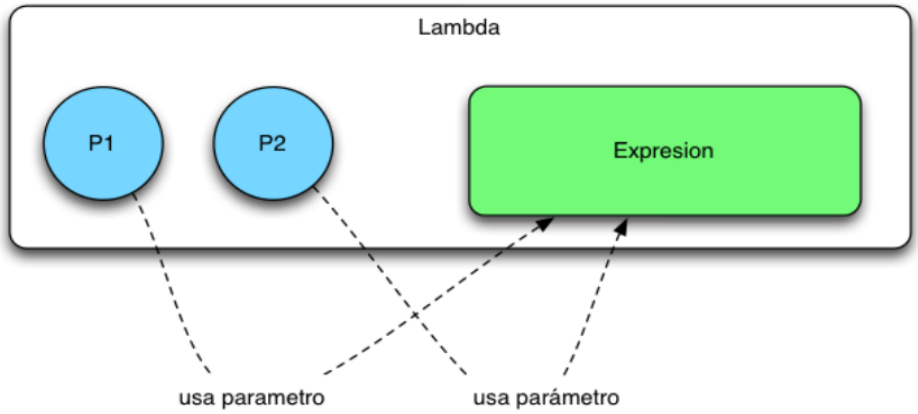
Si lo pensamos, lo más importante del código es:

```
(Persona p1, Persona p2) {

   return p1.getNombre().compareTo(p2.getNombre());

}
```

Que representa de forma “anónima” al método compare.

Dados dos objetos Persona, devolvemos el resultado de aplicarles el método compareTo, es decir, lo que viene siendo una función.

Como vimos al principio de este apartado, una expresión lambda se compone de dos elementos. En primer lugar, de un conjunto de parámetros y en segundo lugar de una expresión que opera con los parámetros indicados.



Aplicando esto, la función anónima del ejemplo quedaría:

```
Collections.sort(milista, (Persona p1, Persona p2)->
                                    p1.getNombre().compareTo(p2.getNombre()));
```

Acortando el código de manera significativa.

->Ver antes las características de la versión 8 y 9 de Java para Interfaces en el aula virtual<-

2.2 ¿Qué es una interfaz funcional?

Una **interfaz funcional** es una interfaz con **uno y solo un** método abstracto. La declaración es exactamente igual que las interfaces normales con dos características adicionales:

- Tienen un **único método abstracto**. Pueden tener otros tipos de métodos.
- De manera opcional puede estar anotada como **@FunctionalInterface**. (Java 8)

El motivo de que la interfaz tenga un único método abstracto es que será una expresión lambda la que proveerá de la implementación para dicho método.

Ejemplos de interfaz funcional:

```
@FunctionalInterface
public interface Runnable {
	public abstract void run();
}

public interface MiInterfaz {
	default void saluda() {
		System.out.println("Un saludo!");
	}
	public abstract int calcula(int dato1, int dato2);
}


@FunctionalInterface
public interface Comparator {
	// Se eluden los métodos default y estáticos
	int compare(T o1, T o2);

    // El método equals(Object obj) es implícitamente implementado por la clase
    //objeto.
	boolean equals(Object obj);
}
```

2.3 Uso

Las lambdas se pueden usar donde el tipo de parámetros aceptados sea una interfaz funcional.

Este es el caso de muchas de las funcionalidades que ofrece **java.util.stream**, nuevo API que aparece con Java 8, que permite la programación funcional sobre un flujo de valores sin estructura. **Ejemplo:**

```
@FunctionalInterface
interface Calcular {
	Integer op(Integer a, Integer b);
	…. // puede tener otros métodos de los permitidos
}
```

Y ahora se puede asignar la expresión lambda:

```
Calcular sum = (Integer x, Integer y) -> {return x+y;};
Calcular mult = (Integer x, Integer y) -> x * y;
Calcular rest = (x,y) -> x-y;
```

->Ver ejemplo Listener con Lambda y más<-

En **Java**, las **funciones de orden superior** son un concepto importante que se relaciona con la programación funcional:

1. **Definición**:
   - Las funciones de orden superior son aquellas que pueden **tomar otras funciones como argumentos** o **devolver funciones como resultado**.
   - En términos más sencillos, una función de orden superior es aquella que interactúa con funciones de manera más flexible y dinámica.
2. **Usos comunes**:
   - **Pasar como argumentos a otras funciones**: Puedes enviar una función como argumento a otra función. Esto es útil para personalizar el comportamiento de una función según las necesidades específicas.
   - **Construir el resultado de una función de orden superior**: A veces, necesitas crear una función que retorne otra función. Esto es especialmente útil en casos como la creación de funciones de filtrado o mapeo.
   - **Trabajar con interfaces funcionales**: Las funciones de orden superior son fundamentales al trabajar con interfaces funcionales, como java.util.function.Function o java.util.function.Predicate.
   - **Operar en colecciones y flujos (Streams)**: Las funciones de orden superior son esenciales al usar la API Stream en Java. Puedes aplicar transformaciones, filtrar elementos y realizar otras operaciones de manera concisa.
3. **Ejemplo práctico**:
   - Supongamos que tienes una lista de números y deseas filtrar los números pares e impares por separado.
   - En lugar de escribir dos funciones separadas para esto, puedes crear una función de orden superior que acepte un predicado (una función que evalúa una condición) como argumento.
   - Ejemplo en Java:

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

public class FuncionesOrdenSuperior {

  public static void main(String[] args) {
    List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    // Filtrar números impares
    List<Integer> impares = filtrar(numeros, n -> n % 2 != 0);
    System.out.println("Números impares: " + impares);

    // Filtrar números pares
    List<Integer> pares = filtrar(numeros, n -> n % 2 == 0);
    System.out.println("Números pares: " + pares);
  }
```

```
    // Función de orden superior para filtrar
    public static List<Integer> filtrar(List<Integer> lista, Predicate<Integer> condicion) {
        return lista.stream().filter(condicion).toList();
    }
}
```

En este ejemplo, la **función filtrar** toma una lista de números y un predicado como argumentos. El predicado determina la condición de filtrado (por ejemplo, números impares o pares). La función filtrar utiliza la API Stream para aplicar la condición y devuelve una lista filtrada.

En resumen, las funciones de orden superior en Java nos permiten escribir código más modular, reutilizable y expresivo al trabajar con funciones como ciudadanos de primera clase.

2.4 Tipos

Paquete java.util.function de java

Son “functional interfaces” que sirven como “variables” para expresiones lambda.

T es el tipo del parámetro, R tipo del resultado

Las expresiones lambda pueden clasificarse de la siguiente manera:

- Consumidores.
  - Consumers (T -> void)
- Proveedores.
  - Supplier (nil -> R)
- Funciones.
  - Function (T -> R)
- Predicados.
  - Predicate (T -> boolean)

Functional Interface	Parameter Types	Return Type	Description
Supplier<T>	None	T	Supplies a value of type T
Consumer<T>	T	void	Consumes a value of type T
BiConsumer<T, U>	T, U	void	Consumes values of types T and U
Predicate<T>	T	boolean	A Boolean-valued function
ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>	T	int long double	An int-, long-, or double-valued function
IntFunction<R> LongFunction<R> DoubleFunction<R>	int long double	R	A function with argument of type int, long, or double
Function<T, R>	T	R	A function with argument of type T
BiFunction<T, U, R>	T, U	R	A function with arguments of types T and U
UnaryOperator<T>	T	T	A unary operator on the type T
BinaryOperator<T>	T, T	T	A binary operator on the type T

3 Streams

Stream hace referencia a un flujo de elementos sobre los cuales se aplicará una serie de operaciones usando Programación Funcional.

Para utilizar los streams en Java 8, seguiremos estos pasos:

1. **Obtenemos un Stream**: Se puede crear un stream a partir de una colección o cualquier otra fuente de datos (un array, un archivo e incluso valores individuales).

2. Usamos alguna/s de las siguientes **operaciones en el Stream**:
    a. **Filtrar**: Puedes filtrar elementos según ciertos criterios.
    b. **Mapear**: Transforma los elementos del stream.
    c. **Ordenar**: Organiza los elementos según tus preferencias.
    d. **Reducir**: Realiza una operación final en el stream.
    e. **Terminar el Stream**: Recolecta los elementos resultantes o realiza una operación final.

Un **stream** en Java es una secuencia de elementos que podemos procesar de manera eficiente y funcional:

- Un **stream** es una abstracción que te permite trabajar con colecciones de datos de manera más concisa y expresiva.
- Puedes crear un **stream** a partir de una colección (como una lista o un conjunto), un array, un archivo o incluso valores individuales.
- Los **streams** te permiten realizar operaciones como filtrado, mapeo, ordenamiento y reducción de elementos.

Son especialmente útiles para procesar grandes cantidades de datos de manera eficiente.