LAMBDAS - INTERFACES FUNCIONALES

ÍNDICE

1. INTRODUCCIÓN

a. ¿Qué es una lambda?

Introducidas en **Java 8**, las expresiones **lambda** ofrecen una forma más concisa, funcional y eficiente de escribir código en comparación con la forma tradicional. Su principal ventaja es la eliminación de la necesidad de clases anónimas o clases normales que implementan interfaces funcionales, permitiendo una sintaxis más limpia y directa.

Una expresión **lambda** es una función anónima que puede usarse como argumento en métodos que esperan una interfaz funcional. Su sintaxis básica es:

```
( parámetros ) -> { cuerpo }
```

b. ¿Para que se usan?

A primera vista, las expresiones lambda pueden parecer innecesarias de poco uso o incluso liosas, sin embargo, su aplicación en el código permite reducir significativamente la cantidad de líneas de código necesarias para realizar diversas tareas, mejorando la legibilidad, el mantenimiento del código y una vez se entienden hacen mucho más eficiente la programación.

Aquí tenemos un ejemplo en el que se ve muy claro y además se entiende mejor que es una lambda

```
// Usando un foreach como hasta ahora
public void mostrarListaMultiplicada1(List<Integer> s) {
        for (Integer i : s) {
                System.out.println(i * 2);
        }
}
// Usando una lamda
public void mostrarListaMultiplicada(List<Integer> s) {
        s.forEach(i -> System.out.println(i * 2));
}
```

Cuando tu escribes un método indicas que parametros hay que pasarle y dentro escribe lo que hace, pues esto es lo mismo, a la izquierda de la flecha le pasas los parametros y a la derecha le dice lo que hace

Aquí otro ejemplo usando una interfaz funcional:

```
BiFunction<Integer, Integer, Integer> suma = (a, b) -> a + b;

System.out.println(suma.apply(3, 5));
// Devuelve 8
```

Las lambdas son especialmente útiles cuando se trabaja con objetos como **Streams** y **Collections**, ya que permiten escribir operaciones de filtrado y transformación, entre otras, de manera intuitiva y eficiente.

c. Programación funcional

Es una forma de programar en la que se usan funciones para resolver problemas, se inspiran en las matemáticas.
Tiene varias características como son:
Si le pasas siempre lo mismo, te devuelve siempre lo mismo, además, no modifica los atributos originales. crea nuevas variables y las devuelve.
Además y es importante, se le pueden pasar funciones como argumento o devolverlas, que es básicamente lo que se hace con las lambdas

2. CLASES ANÓNIMAS (Lo anterior a las lambdas)

Las clases anónimas en Java permiten instanciar una clase en una sola expresión. Normalmente, se usan para implementar interfaces o extender clases cuando se necesita una única instancia de esa clase en específico. Sin embargo, su uso también permite instanciar clases completas, lo que posibilita la modificación de los métodos de la misma o la implementación de interfaces en esa clase anónima en particular.

Ejemplo:

```
Secretaria secretaria = new Secretaria(10, listaAlumnos) {
        @Override
        public double averageByEveryoneV2() {
                return super.averageByEveryoneV2() / getListadoAlumnos().size();
        }
};
```

En el ejemplo anterior, instanciamos la clase **Secretaria** y la convertimos en una clase anónima añadiendo unas llaves tras la declaración de los parámetros. Dentro de esas llaves, modificamos el método **averageByEveryoneV2** que es el método abstracto que trae la clase.

Si bien las clases anónimas pueden parecer útiles para algunas ocasiones, su utilización puede traer diversos problemas a nuestro código como serian los siguientes:

- **Verbosidad**: requieren muchas líneas de código incluso para implementaciones simples.
- **Dificultad de lectura**: el código puede volverse más difícil de entender debido a la cantidad de sintaxis adicional.
- **Menor reutilización**: no pueden ser reutilizadas fácilmente en otros lugares del código.

En el ejemplo anterior se usó una clase normal de Java, por lo que no habría forma de simplificar su implementación con expresiones lambda. Sin embargo, en las interfaces funcionales, que son el objetivo principal de las lambdas, sí es posible simplificar el código.

Por ejemplo, en la siguiente imagen se muestra cómo una expresión multilínea usando una clase anónima:

```
Consumer<String> claseAnonima = new Consumer<String>() {

	@Override
	public void accept(String message) {
		System.out.println(message);
	}
};
```

Se convierte en una única línea gracias al uso de una expresión lambda:

```
Consumer<String> lambda = s -> {System.out.println(s);};
```

La diferencia principal entre clases anónimas y lambdas es que las primeras pueden implementar múltiples métodos si extienden una clase concreta o incluso crearlos, mientras que las expresiones lambda solo pueden usarse con interfaces funcionales.

Ejemplo con una clase anónima extendiendo una clase base:

```
public class Figura {
	public void dibujar() {
		System.out.println("Dibujando una figura.");
	}

	public void borrar() {
```

```
                System.out.println("Borrando una figura");
        }
}

public class Principal {
        public static void main(String[] args) {
                Figura figura = new Figura() {
                        @Override
                        public void dibujar() {
                                System.out.println("Estoy dibujando una figura desde el main.");
                        }
                        @Override
                        public void borrar() {
                                System.out.println("Estoy");
                                super.borrar();
                                System.out.println("Desde el main.");
                        }

                        public void crear() {
                                System.out.println("Estoy creando una figura en el main.");
                        }

                };
        }
}
```

No podríamos hacer esto con una expresión lambda porque **Figura** no es una interfaz funcional.

3. INTERFACES FUNCIONALES

Las interfaces funcionales son interfaces que contienen un único método abstracto, permitiendo así su implementación mediante expresiones lambda. Estas interfaces facilitan la programación funcional en Java y permiten escribir código más flexible y reutilizable.

Características de las Interfaces Funcionales

- **Un solo método abstracto**: Una interfaz funcional debe contener exactamente un método abstracto, aunque puede tener métodos default o static, sin que su presencia afecte a la implementación/uso de la interfaz como funcional..
- **Anotación @FunctionalInterface**: Aunque no es obligatoria, esta anotación ayuda a garantizar que la interfaz cumple con la restricción de tener un único método abstracto.
- **Uso con expresiones lambda**: Las interfaces funcionales permiten la implementación directa mediante lambdas, lo que reduce el código necesario y mejora la legibilidad.

Ejemplo de interfaz funcional:

```java
//@FunctionalInterface
public interface Operacion {

	int calcular(int a, int b);
}
```

```java
		Operación suma = (num1, num2) -> num1 + num2;
		System.out.println(suma.calcular(5, 3));
```

A continuación, veremos las interfaces funcionales aportadas directamente por Java y cómo las expresiones lambda simplifican su implementación.

a. Consumer

La interfaz **Consumer<T>** representa una operación que recibe un solo argumento y no devuelve ningún resultado, el tipo del argumento viene dado en el lugar de la T de la sintaxis mostrada anteriormente. Es comúnmente utilizada en estructuras como **forEach** para realizar acciones sobre cada elemento de una colección.

```java
Consumer<String> lambda = s -> {System.out.println(s);};
lambda.accept("Hola!!!");
```

Un uso real y útil de la interfaz funcional **Consumer** sería imprimir todos los elementos de una Lista, hasta el momento, esta tarea se realizaba mediante el uso de un **bucle** según se ve en el siguiente ejemplo:

```java
		List<String> nombres = new ArrayList<>();
		nombres.add("Juan");
		nombres.add("Carlos");
		nombres.add("Fernando");
		nombres.add("Lance");
		nombres.add("Noah");
		for (String nombre : nombres) {
			System.out.println("Hola " + nombre);
		}
```

Sin embargo, con el uso de las **clases anónimas + interfaces funcionales + lambdas**, podríamos pasar el codigo a lo siguiente, haciendo uso del método **forEach** el cual proviene de la **Interfaz Iterable**:

```java
		Consumer<String> consumerEj = new Consumer<String>() {
			@Override
			public void accept(String name) {
				System.out.println("Hola " + name);
			}};

		List<String> nombres = new ArrayList<>();
		nombres.add("Juan");
		nombres.add("Carlos");
		nombres.add("Fernando");
		nombres.add("Lance");
		nombres.add("Noah");
```

```
nombres.forEach(name -> consumerEj.accept(name));
```

Pese a que en el ejemplo anterior estamos haciendo uso de las lambdas para pasar el parámetro a la implementación del consumer, no estamos haciendo uso del potencial de la lambda al 100% puesto que aún estamos declarando la clase anónima lo que añade bastantes líneas al código así como también nos consume bastante tiempo, haciendo uso de únicamente las **Lambdas + interfaces funcionales**, podríamos obtener un código muchísimo más simple y más rápido de escribir que en los ejemplos anteriores:

```
List<String> nombres = new ArrayList<>();
nombres.add("Juan");
nombres.add("Carlos");
nombres.add("Fernando");
nombres.add("Lance");
nombres.add("Noah");
nombres.forEach(name -> System.out.println("Hola "+name));
```

Como se puede observar en el último ejemplo, la **lambda** se está convirtiendo a sí misma en un consumer y está consumiendo el dato facilitado por el método **forEach**, en este caso, el método **forEach** facilita cada uno de los datos almacenados en la Lista los cuales son posteriormente almacenados en la variable name y después son consumidos por el consumer para mostrar los datos mediante un **Syso**.

b.Function

La interfaz **Function<T, R>** representa una operación que recibe un solo argumento y devuelve un resultado, el tipo del argumento viene dado en el lugar de la T de la sintaxis mostrada anteriormente, al igual que el tipo del dato que retorna, viene especificado en la R de la sintaxis anterior. Es comúnmente utilizada en estructuras como **.map** de los streams para realizar conversiones sobre los elementos de una colección, aunque también es usado en otros ámbitos, según especificado anteriormente su principal uso es para convertir datos.
Un ejemplo sobre el uso de la interfaz Function sería el siguiente, en el cuál le facilitamos una palabra (String) y nos devuelve su longitud:

```
Function<String, Integer> extractLength = m -> m.length();
System.out.println(extractLength.apply("Patata"));
```

Por otro lado, un ejemplo con **.map** sería el siguiente en el que estamos convirtiendo los nombres de un array en su versión en mayúsculas, para más tarde imprimirlos haciendo uso de un consumer y de .forEach como vimos en puntos anteriores:

```
List<String> nombres = List.of("ana", "juan", "pedro");

nombres.stream()
        .map(name -> name.toUpperCase())
        .forEach(name -> System.out.println(name));
```

En este último ejemplo, usamos directamente la lambda con la interfaz funcional y podemos observar que tanto el tipo de dato que recibe como el que devuelve es del mismo tipo que el de la lista que estamos utilizando en el stream, estos datos los sabemos gracias a la explicación del método map en la API de Java.

c. Predicate

La interfaz **Predicate<T>** representa una operación que recibe un solo argumento (Tipo T) y devuelve un resultado de tipo Booleano. Es comúnmente utilizada en estructuras como **.filter** de los streams para realizar filtros, aunque también es usado en otros ámbitos, según especificado anteriormente su principal uso es para filtrar datos.

Como observamos en el siguiente ejemplo:

```
List<String> nombres = List.of("ana", "juan", "pedro");
nombres.stream()
        .filter(t -> t.length() > 2)
        .forEach(name -> System.out.println(name));
```

Podemos observar un stream con un **.filter** el cual está trabajando con un Predicate como lambda, con la Lambda indicamos que t será el nombre actual de la lista e implementando la interfaz funcional comprobamos si su longitud es mayor que 2, esto nos devolverá un true/false que es en lo que se basa el filtro para pasar el dato a la siguiente línea.

Pese a que se haya mostrado el uso de Predicate en un filter, también podemos hacer uso de esta interfaz funcional en otros ámbitos, como por ejemplo el siguiente if:

```
Predicate<Integer> checkAge = t -> t >= 18;
Map<String, Integer> edades = new HashMap<>();
edades.put("Ana", 18);
edades.put("juan", 16);
edades.put("pedro", 13);
for (String nombre : edades.keySet()) {
        if (checkAge.test(edades.get(nombre))) {
                System.out.println(nombre + " es mayor de edad");
        } else {
                System.out.println(nombre + " es menor de edad");
        }
}
```

d. Supplier

La interfaz **Supplier<R>** representa una operación que devuelve un argumento (Tipo R). Es comúnmente utilizada en los stream acompañado de **.generate** para generar listados de datos**,** aunque también se puede usar para generar datos en cualquier otro caso.

En el siguiente ejemplo podemos observar un uso de Supplier para generar números aleatorios:

```
Supplier<Double> generarNum = () -> {
        return Math.round(Math.random() * 100) / 10.0;
};
for (int i = 0; i < 5; i++) {
        System.out.println(generarNum.get());
}
```

e. Versiones alternativas

A continuación veremos unas cuantas de las múltiples versiones alternativas que tienen las interfaces funcionales vistas anteriormente junto con algunos de sus usos reales.

i. BiConsumer

La interfaz **BiConsumer<T,T>** representa una operación que recibe dos argumentos y los procesa, pero no devuelve nada, exactamente igual que su versión alternativa (Consumer), un uso real de esta interfaz sería la impresión de un map separando el par de clave valor:

```
Map<String, Integer> edades = new HashMap<>();
edades.put("Ana", 18);
edades.put("Juan", 16);
edades.put("Pedro", 13);
edades.forEach(
        (nombre, edad) -> System.out.println("La edad de " + nombre + " es: " + edad + " años")
);
```

En el ejemplo podemos observar que la sintaxis de la lambda ha cambiado un poco respecto a la sintaxis anterior, en este caso estamos añadiendo un () a los parámetros, esto es debido a que hay más de uno la sintaxis pasa de ser parámetro -> expresión a (parametro1, parametro2) -> expresión, al igual que en caso de ausencia de parámetros debe colocarse como () -> expresión, sin olvidar también que en todos los casos anteriores, si la expresión es de más de una sentencia, deberemos usar {} para contener la expresión.

ii. BiPredicate

Al igual que su versión alternativa **BiPredicate<T,T>** recibe, en este caso dos datos, que no tienen por que ser del mismo tipo y como resultado produce un booleano, siguiendo el ejemplo de su versión alternativa, **BiPredicate** es usado para filtrar, pero también puede usarse para comparar, entre muchos otros usos.

Ejemplo:

```
BiPredicate<String, Integer> checkLength = (texto, longitud) -> texto.length() >= longitud;
int largo = 5;
List<String> nombres = List.of("Paco", "Manuel", "Jose");
for (String nombre : nombres) {
        if (checkLength.test(nombre, largo)) {
                System.out.println(nombre + " es un nombre largo.");
        }
}
```

iii. BiFunction

La **interfaz BiFunction<T, U, R>** es una extensión de **Function<T, R>** que recibe dos argumentos **(T y U)** y devuelve un resultado **(R)**.

Un uso de **BiFunction** sería por ejemplo hacer uso de dos datos, un Precio y un Impuesto para devolver el precio final de un producto como se observa en el siguiente ejemplo:

```
List<Double> precios = List.of(4.21, 5.2, 102.3, 22.4);
int impuesto = 21;
BiFunction<Double, Integer, Double> calcularImpuesto = (precio, imp) -> {
        return precio + ((imp * precio) / 100);
};
precios.forEach(precio -> System.out.println(calcularImpuesto.apply(precio, impuesto)));
```

4. Observaciones Lambda

Scope/Alcance

Las expresiones lambda en Java pueden acceder a variables definidas fuera de su propio cuerpo, como variables locales o atributos de clase. Sin embargo, presentan una restricción importante: **no pueden modificar directamente las variables locales** del método en el que están definidas, a menos que estas sean efectivamente finales (es decir, que su valor no cambie después de ser asignado).

Solución: Uso de un Array como Contenedor
Para modificar valores dentro de una lambda sin generar un error de compilación, se pueden utilizar estructuras mutables, como un array o una instancia de una clase contenedora.

Por ejemplo, si queremos imprimir elementos de una lista con un índice incremental dentro de forEach, podemos usar un array para almacenar el contador, permitiendo su modificación dentro de la lambda:

```
int cont[] = {1};
List<String> nombres = List.of("ana", "juan", "pedro");
nombres.forEach(nombre -> {
        System.out.println(cont[0] + ". " + nombre);
        cont[0]++;
});
```

5. Métodos Referenciados

a. ¿Qué es un Método referenciado?

Un **método referenciado** es una forma abreviada de llamar a los diferentes métodos de **clases u objetos**, permitiendote usar métodos de **clases instanciadas(objetos) o sin instancia**

b. Referencia a un método estático

Este tipo de método referenciado se usa en ocasiones en las que necesitamos acceder a un **método estático**, este tipo de método puede ser usado sin necesidad de tener una **clase instanciada**, puesto que como se ve en la siguiente **sintaxis**, usamos la clase directamente y no un objeto instanciado como ella.

Sintaxis:

```
Clase::metodoEstatico
```

Ejemplo:

```
public class Ejemplo {
  public static int cuadrado(int x) {
    return x * x;
  }
  public static void main(String[] args) {
    Function<Integer, Integer> refMetodo = Ejemplo::cuadrado;
    System.out.println(refMetodo.apply(5));
  }
}
```

c. Referencia a un método de un objeto

Es una forma de referenciar un método que pertenece a un objeto que ha sido instanciado, básicamente le estás diciendo a un objeto “Toma, usa este método” y lo puede usar después.

¿Cómo se escribe?:

Objeto::MetodoDeInstacia

Ejemplo.

```
class Perro {
    void saludar(String nombre) {
        System.out.println("Hola, " + nombre + ", soy un perro!");
    }
}
public class Main {
    public static void main(String[] args) {
        Perro miPerro = new Perro();
        // Referenciamos al método saludar del objeto miPerro

        Consumer<String> accion = miPerro::saludar;

        // llamamos al metodo accion del consumer
        accion.accept("Juan");
            // Imprime "Hola, Juan, soy un perro!"
    }
}
```

d.Referencia a un método de un objeto arbitrario

En lugar de referirse a un método de un objeto concreto (como `miPerro::ladrar`), una referencia a un método de un objeto arbitrario (como podria ser para nosotros Alumno, o Trabajador) te permite apuntar a un método de una clase sin especificar un objeto particular. El objeto sobre el que se ejecutará el método se determina más tarde, cuando la referencia se use.
Ejemplo 1

```
class Perro {
        void ladrar(String mensaje) {
        System.out.println("¡Guau! " + mensaje);
         }
}
```

```
import java.util.function.BiConsumer;
        public class Main {
                public static void main(String[] args) {
       // Referencia a un método de un objeto arbitrario con un parámetro
         BiConsumer<Perro, String> hacerLadrar = Perro::ladrar;
         Perro miPerro = new Perro();
        hacerLadrar.accept(miPerro, "Soy un perro instanciado");
    }
}
```

Ejemplo 2

```
class Perro {
        public void ladrar() {
         System.out.println("¡Guau!");
    }
}


public class Main {
    public static void main(String[] args) {
      //referencias Perro a un método arbitrario.
        Consumer<Perro> perroLadrar = Perro::ladrar;
        // Creamos dos perros distintos
        Perro perro1 = new Perro();
        Perro perro2 = new Perro();
        // Aplicamos la referencia a diferentes objetos
        perroLadrar.accept(perro1);
        perroLadrar.accept(perro2);
    }
}
```

e. Referencia a un constructor

Este tipo de referencia se usa cuando queremos instanciar un objeto de una clase utilizando una **referencia a su constructor** en lugar de llamar directamente al **new**. Es útil cuando se trabaja con interfaces funcionales, ya que permite asignar un constructor a una expresión funcional sin necesidad de escribir una lambda.

Sintaxis:

```
Clase::new
```

Ejemplo:

```
class Persona {
  String nombre;
  public Persona(String nombre) {
     this.nombre = nombre;
  }
}
public class Ejemplo {
```

```
    public static void main(String[] args) {
        Function<String,Persona> crearPersona = Persona::new;
        Persona p = crearPersona.apply("JoseManue");
        System.out.println(p.nombre);
    }
}
```

Ejemplo 2:

```
class Persona {
    String nombre;
    int edad;
    String ciudad;
    String profesion;
    double salario;

    public Persona(String nombre, int edad, String ciudad, String profesion, double salario) {
        this.nombre = nombre;
        this.edad = edad;
        this.ciudad = ciudad;
        this.profesion = profesion;
        this.salario = salario;
    }
}

@FunctionalInterface
interface CreadorPersona {
    Persona crear(String nombre, int edad, String ciudad, String profesion, double salario);
}

public class Ejemplo {
    public static void main(String[] args) {
        CreadorPersona crearPersona = Persona::new;
        Persona p = crearPersona.crear("Jose Manuel", 30, "Madrid", "Ingeniero", 3500.50);
        System.out.println("Nombre: " + p.nombre);
        System.out.println("Edad: " + p.edad);
        System.out.println("Ciudad: " + p.ciudad);
        System.out.println("Profesión: " + p.profesion);
        System.out.println("Salario: " + p.salario);
    }
}
```