Expresiones LAMBDA

JAVA Bootcamp

DigitalHouse >
Coding School

Temario

- Programación declarativa vs. Programación imperativa.
- Expresiones **lambda**
- Repaso de **interfaces funcionales**
- **Sintaxis:** Cómo **escribir** expresiones lambda
- Cómo **invocar los métodos** de una interfaz funcional a través de una expresión lambda
- **Interfaces funcionales** del paquete java.util.function
- Procesar datos utilizando **Lambdas, Collections y Streams**
- Introducción a **operaciones** de **Stream API**

Programación Imperativa vs. Declarativa

Imperativa

Un programa consiste en una **secuencia claramente definida de instrucciones para un ordenador.**

Centrada en el «cómo».

Son **instrucciones paso a paso** que describen de forma explícita qué pasos deben llevarse a cabo y en qué secuencia para alcanzar finalmente la solución deseada.

Declarativa

Un programa describe directamente el resultado final deseado.

Centrada en el «qué»

Sintaxis más legible, elegante.

Código optimizado y menor cantidad de líneas.

A través de las expresiones Lambda, Java busca implementar el paradigma funcional, transformando funciones imperativas en funciones declarativas

Expresiones Lambda

DigitalHouse > Coding School

Definición

Las expresiones Lambda son expresiones que implementan interfaces funcionales.

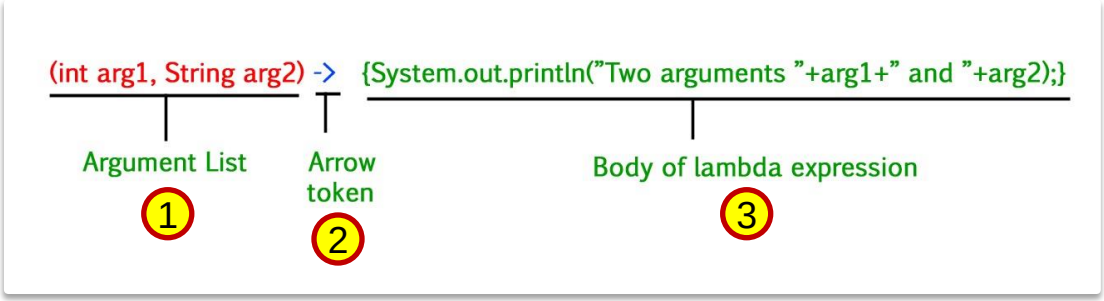
¿Qué es una interfaz funcional?

- Es una interfaz que tiene un solo método abstracto. Será la expresión lambda la que proveerá de la implementación para dicho método.
- No es obligatorio, pero puede anotarse con **@FunctionalInterface**, en este caso si la interfaz que escribimos no es funcional, producirá un error de compilación.

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

Expresiones Lambda

Sintaxis de la expresión lambda



Implementando una interfaz funcional a través de una función lambda en 3 pasos

```
public interface Supplier <T> {
    T get();
}
```

```
Supplier<String> supplier =
    () -> "Hello!";
```

1. Tomamos los **parámetros** del método get() en este caso, como no tiene parámetros, los paréntesis irán vacíos por lo que no tendrá **argumentos**.
2. Agregamos el **operador lambda** -> que separa la declaración de parámetros de la declaración del cuerpo de la función.
3. Proveemos una **implementación**, es decir en este caso un string “Hello!”, esto sería el **cuerpo** de la expresión.

Ejemplos de sintaxis

*Si hay **una sola línea de código** dentro de la lambda no es necesario utilizar los corchetes {} ni explicitar el return.*

Sin argumentos:

```
() -> System.out.println("Hello");
```

Un argumento:

```
m -> System.out.println(m);
```

Dos argumentos:

```
(x, y) -> x + y;
```

*En este caso no es necesario explicitar el tipo ya que se **infiere de la interfaz que se está implementado**.*

Dos argumentos explicitando el tipo:

```
(Integer x, Integer y) -> x + y;
```

Dos argumentos y múltiples expresiones:

```
(a, b) ->  {
    int suma = a + b;
    return sum;
}
```

*Si hay **más de una línea de código** debe usarse la cláusula **return** en el caso de que el método no sea void y añadir los **corchetes {}**.*

Interfaces funcionales del paquete java.util.function

Este paquete está formado por más de 40 interfaces funcionales. Los principales tipos son:

Supplier: No tienen parámetros, pero devuelven un resultado.

Consumer: Aceptan un solo valor y no devuelven ninguno.

Predicate: Aceptan un parámetro y devuelven un valor lógico (booleano). Es muy utilizada en la stream API para filtrar.

Function: Puede retornar cualquier tipo de objeto. T es el objeto que toma la función. R es objeto que produce. Se utiliza en el map de la stream API.

```
public interface Supplier <T> {
    T get();
}
```

```
public interface Consumer <T> {
    void accept(T t);
}
```

```
public interface Predicate <T> {
    boolean test(T t);
}
```

```
public interface Function <T, R> {
    R apply(T t);
}
```

DigitalHouse > Coding School

Ejemplo: Implementando interfaces funcionales

```java
public class PrimerasLambdas {

    public static void main(String[] args) {

        // Supplier
        Supplier<String> supplier = () -> {

            System.out.println("Estoy dentro de supplier");
            return "Hello!";
        };

        String string = supplier.get();
        System.out.println("string = " + string);

        // Consumer
        Consumer<String> consumer =
                (String s) -> {
                    System.out.println("Estoy dentro de consumer");
                    System.out.println(s);
                };
        consumer.accept("Hello");

    }

}
```

```java
public interface Supplier <T> {
    T get();
}
```

```java
public interface Consumer <T> {
    void accept(T t);
}
```

DigitalHouse > Coding School

Ejemplo Integrador

Dada una lista de String, remover de ella todas las palabras que no empiezan con “t” y luego imprimir por consola las que quedaron.

```
public class Lambdas {

    public static void main(String[] args) {

        List<String> palabras =
                new ArrayList<>(List.of("uno", "dos", "tres", "cuatro"));

        palabras.removeIf(p -> !p.startsWith("t"));
        palabras.forEach(p -> System.out.println(p));
    }
}
```

Expresiones Lambda

DigitalHouse > Coding School

Ejercitación:

Escribir una función lambda que:

1) Devuelva el número 5.

2) Duplique el valor de x y lo retorne

3) Tome dos valores y retorne su diferencia.

4) Tome dos enteros y retorne su suma.

5) Tome un String y lo imprima en consola.

Respuestas:

```
() -> 5
```

```
x -> 2 * x
```

```
(x, y) -> x - y
```

```
(int x, int y) -> x + y
```

```
(String s) -> System.out.print(s)
```

Procesar Datos utilizando Lambdas, Collections y Streams:

Podemos definir los **Streams** como una **secuencia de funciones que se ejecutan una detrás de otra**, en forma anidada.

Ejemplo: Obteniendo un Stream de una Collection

```
List<String> items =  new ArrayList<>();
items.add("uno");
items.add("dos");
items.add("tres");

Stream<String> stream = items.stream();
```

Ejemplo: Streams un antes y un después en el código

El código se simplifica utilizando Java 8 con Streams y Lambdas para iterar sobre una Collection, en este caso una lista.

```
String frase = "La imaginación es más importante que el conocimiento";
String[] fraseCortada = frase.split("\\s+");


List<String> palabras =  Arrays.asList(fraseCortada);
int count = 0;
for (String palabra : palabras) {
    if(palabra.startsWith("i")) {
        count++;
    }
}
```

es lo mismo que…

```
palabras.stream().filter(p -> p.startsWith("i")).count();
```

Introducción a Operaciones con Streams

OPERACIONES INTERMEDIAS: El stream devolverá otro stream permitiendo la continuidad de pasos o funciones sobre ella misma. Esto es llamado 'pipelining'.

1) map: Retorna un stream que cosiste del resultado de aplicar una determinada función a los elementos del stream.

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

Resultado: [4, 9, 16, 25]

2) filter: Selecciona elementos basándose en el predicado que se pasa como argumento.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
```

Resultado: [Stream]

3) sorted: Ordena los elementos del stream.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().sorted().collect(Collectors.toList());
```

Resultado: [Collection, Reflection, Stream]

Expresiones Lambda

DigitalHouse > Coding School

OPERACIONES TERMINALES: Terminan el pipeline de **operaciones** de un **Stream** y activan su ejecución.

1) collect: Retorna el resultado de operaciones intermedias realizadas sobre el stream.

```
List number = Arrays.asList(2,3,4,5,3);
Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

Resultado: [16, 4, 9, 25]

2) forEach: Itera sobre cada elemento del stream.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

Resultado: 4 9 16 25

3) reduce: Reduce los elementos del stream a un solo valor. Toma como parámetro un BinaryOperator.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```

Resultado: 6

Expresiones Lambda

DigitalHouse > Coding School

Ejercitación:

Procesar una lista de productos y realizar las siguientes operaciones:

- Recorrer la lista
- Filtrar durante el recorrido los elementos del tipo «limpieza»
- Ordenar el resultado anterior
- Imprimir el resultado

Clase del modelo:

```
public class Producto {
    private long id;
    private String nombre;
    private String tipo;
    private Double precio;
}
```

Respuesta:

```
List<Producto> productos =  productService.findAll();

productos.stream()
        .filter(p -> "limpieza".equals(p.getTipo()))
        .sorted(comparing(Producto::getPrecio))
        .forEach(System.out::println);
```