PREGRADO

UNIDAD 1 | OVERVIEW

JAVA FUNCTIONAL FEATURES

SI705 | Arquitectura de Aplicaciones Web

Al finalizar la unidad de aprendizaje, el estudiante describe el proceso de software realizado aplicando el paradigma orientado a objetos combinado con aspectos de la programación funcional, utilizando el lenguaje Java y frameworks de actualidad, para desarrollar aplicaciones web básicas en un ambiente de desarrollo colaborativo.

AGENDA

INTRO

LAMBDA EXPRESSIONS

STREAMS

Intro

En 2014 Java SE introdujo cambios significativos que impactaron a la forma como se programaba con este lenguaje.

Entre estos cambios resaltan dos aspectos relacionados: las características de programación funcional y el Stream API.

AGENDA

INTRO

LAMBDA EXPRESSIONS

STREAMS

Lambda expressions

- Una expresión Lambda es una función anónima
- Es un método abstracto, ya que está definido en una interfaz pero no implementado
- El programador puede implementarlas dónde el crea conveniente sin heredar ninguna interfaz.
- También conocidas como lambda functions, son funciones sin nombre y que no están vinculadas a un identificador (nameless functions).

Partes de una expresión lambda

Su sintáxis básica se detalla a continuación:

```
( parámetros ) -> { cuerpo-lambda }
```

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

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.

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 clausula 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 clausula return en el caso de que la función deba devolver un valor .

Ejemplo

A continuación se usa una expresión lamba en el ArrayList usando el método forEach(), para imprimir cada elemento de la lista:

```
package demoEjemplo01;

import java.util.ArrayList;

public class Main {
public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );//expresión Lambda
  }
}
```

Ejemplos

```
(int a, int b) -> a * b                  // takes two integers and returns
                                         // their multiplication

(a, b) -> a - b                          // takes two numbers and returns
                                         // their difference

() -> 99                                 // takes no values and returns 99

(String a) -> System.out.println(a)      // takes a string, prints its value
                                         // to the console, and returns nothing

a -> 2 * a                               // takes a number and returns
                                         // the result of doubling it

c -> { //some complex statements }       // takes a collection and do some processing
```

Functional interface

Las interfaces con un solo método, conocidas como Single Abstract Method interfaces (SAM interfaces) se referencian a partir de Java 8 como functional interfaces.

Functions

El caso más simple y general de lambda es una functional interface que recibe un valor y retorna otro.

Esta función de un solo argumento es representada por la interfaz *Function* que tiene como parámetros los tipos de su argumento y valor de retorno.

```
public interface Function<T, R> { … }
```

Functions

Ejemplo:

Map.computeIfAbsent retorna un valor de un map por key, pero calcula un valor si la clave no está ya presente en el map.

Para calcular un valor, usa la implementación de Function que recibe.

```
Map<String, Integer> nameMap = new HashMap<>();
Integer value = nameMap.computeIfAbsent("John", s -> s.length());

// An object on which the method is invoked is the implicit first argument
// of a method. It allows casting an instance method "length" reference
// to a Function interface.
Integer value = nameMap.computeIfAbsent("John", String::length);
```

Functions

La interfaz Function tiene un método por defecto *compose* para combinar varias funciones en una y ejecutarlas de forma secuencial.

```
Function<Integer, Integer> multiply = (value) -> value * 2;
Function<Integer, Integer> add      = (value) -> value + 3;

Function<Integer, Integer> addThenMultiply = multiply.compose(add);

Integer result1 = addThenMultiply.apply(3);
System.out.println(result1);
```

Primitive Function Specializations

Una primitiva no puede ser argumento de un tipo genérico.

Java ofrece versiones de la interfaz Function para la mayoría de tipos primitivos como double, int, long y sus combinaciones.

- IntFunction, LongFunction, DoubleFunction
- ToIntFunction, ToLongFunction, ToDoubleFunction
- DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, Int ToLongFunction, LongToIntFunction, LongToDoubleFunction

@FunctionalInterface

Se refuerza el principio de single responsibility proporcionando a estas interfaces una anotación: @FunctionalInterface.

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

Un intento de agregar un método nuevo a un Functional Interface sería considerado un error en la compilación.

@FunctionalInterface and lambda expressions

Los parámetros en Java tienen un tipo.

Cuando una lambda expression se utiliza en un parámetro, debe ser convertida a un tipo para ser aceptada como parámetro.

Dicho tipo es siempre un tipo **Functional Interface.**

@FunctionalInterface and lambda expressions

```
// Runnable is a functional interface with single method run()

new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}).start();

// In this version, compiler tries to convert the expression
// into Runnable code. If compiler succeeds then everything runs fine.

// Lambda expression is converted to type Runnable.

new Thread(
            () ->   {
                        System.out.println("Thread is running");
                    }
        ).start();
```

@FunctionalInterface and lambda expressions

```
@FunctionalInterface
public interface ShortToByteFunction {

    byte applyAsByte(short s);

}
```

```
// method that transforms an array of short to an array of byte
// using a rule defined by a ShortToByteFunction

public byte[] transformArray(short[] array, ShortToByteFunction function) {
    byte[] transformedArray = new byte[array.length];
    for (int i = 0; i < array.length; i++) {
        transformedArray[i] = function.applyAsByte(array[i]);
    }
    return transformedArray;
}
```

```
// Transforming an array of shorts to array of bytes multiplied by 2.

short[] array = {(short) 1, (short) 2, (short) 3};
byte[] transformedArray = transformArray(array, s -> (byte) (s * 2));

byte[] expectedArray = {(byte) 2, (byte) 4, (byte) 6};
assertArrayEquals(expectedArray, transformedArray);
```

Two-Arity Function Specializations

Java proporciona functional interfaces que contienen "Bi" keyword en su nombre, las cuales permiten definir lambdas con dos argumentos.

BiFunction: Dos argumentos y un tipo de retorno generified.

ToDoubleBiFunction, *ToIntBiFunction*, *ToLongBiFunction*: Dos argumentos y tipo de retorno un valor primitivo.

Two-Arity Function Specializations

Ejemplo típico es el método Map.replaceAll, que permite reemplazar todos los elementos en un map con algún valor calculado.

```
// Use a BiFunction implementation that receives a key and an old value
// to calculate a new value for the salary and return it.

Map<String, Integer> salaries = new HashMap<>();
salaries.put("John", 40000);
salaries.put("Freddy", 30000);
salaries.put("Samuel", 50000);

salaries.replaceAll((name, oldValue) ->
  name.equals("Freddy") ? oldValue : oldValue + 10000);
```

Ejemplo de Lambda expressions

Iterar sobre una lista y realizar operaciones.

```
List<String> lista = new ArrayList();

lista.add("1");
lista.add("2");

lista.forEach(p -> {
                    System.out.println(p);
                    //Do more operations
                  }
              );
```

Ejemplo de Lambda expressions

Crear un Runnable y pasarlo a un Thread.

```
new Thread(
    () -> System.out.println("This Thread is running!");
).start();
```

Ejemplo de Lambda expressions

Sorting Employees by name

```
class Employee {
  String name;

  Employee(String name) {
    this.name = name;
  }

  public static int nameCompare(Employee one, Employee another) {
    return one.name.compareTo(another.name);
  }

  public String toString() {
    return name;
  }
}

// ...
```

Ejemplo de Lambda expressions

Sorting Employees by name

```
// ...
public class LambdaIntroduction {

  public static void main (String[] ar){
          Employee[] employees  = {
                  new Employee("David"),
                  new Employee("Naveen"),
                  new Employee("Alex"),
                  new Employee("Richard")};

          System.out.println("Before Sorting Names: "+Arrays.toString(employees));
          Arrays.sort(employees, Employee::nameCompare);
          System.out.println("After Sorting Names "+Arrays.toString(employees));
      }
}

Output:

Before Sorting Names: [David, Naveen, Alex, Richard]
After Sorting Names [Alex, David, Naveen, Richard]
```

AGENDA

INTRO

LAMBDA EXPRESSIONS

STREAMS

Stream API

- java.util.stream fue introducido en Java 8.
- Contiene clases para procesar secuencias de elementos.
- La clase principal del API es Stream<T>.
- Un Stream en Java se puede definir como una secuencia de elementos de una fuente .
- La fuente de elementos aquí se refiere a una colección o matriz que proporciona datos a la secuencia.

Stream creation

Puede crearse a partir de diversas fuentes de elementos, como collection o array, con los métodos stream() y of().

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> stream = Arrays.stream(arr);
stream = Stream.of("a", "b", "c");


// A stream() default method is added to the Collection interface
// and allows creating a Stream<T> using any collection as an element source

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

Multi-threading con Streams

Stream API proporciona el método parallelStream() que ejecuta operaciones sobre elementos del stream en parallel mode.

```
// Method doWork() runs in parallel for every element of the stream

list.parallelStream().forEach(element -> doWork(element));
```

Ejemplo:

Cualquier flujo en Java se puede transformar fácilmente de secuencial a paralelo.
Podemos lograr esto agregando el método *paralelo* a un flujo secuencial o creando un flujo usando el método de flujo *paralelo* de una colección :

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
    listOfNumbers.parallelStream().forEach(number ->
            System.out.println(number + " " +
          Thread.currentThread().getName()) );
```

Los flujos paralelos nos permiten ejecutar código en paralelo en núcleos separados. El resultado final es la combinación de cada resultado individual.
Sin embargo, el orden de ejecución está fuera de nuestro control. Puede cambiar cada vez que ejecutamos el programa:

Stream Operations

Realizan operaciones sobre el stream, sin cambiar la fuente.

Se dividen en:

Intermediate operations: retornan Stream<T>, soportan encadenar llamadas:map(), filter(), distinct(), sorted(), limit(), skip().

Terminal operations: retornan resultado de tipo definido: forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny().

```
// The distinct() method represents an intermediate operation, which creates
// a new stream of unique elements of the previous stream.
// The count() method is a terminal operation, which returns stream's size.

long count = list.stream().distinct().count();
```

Iterar

Stream API ayuda a sustituir los loops como for, for-each y while.

Permite enfocarse en la lógica de la operación en vez de la iteración sobre la secuencia de elementos.

```
for (String string : list) {
    if (string.contains("a")) {
        return true;
    }
}


// The for loop can be replaced by one line of Java code

boolean isExist = list.stream().anyMatch(element -> element.contains("a"));
```

Filtrar

El método filter() toma elementos que cumplen con un predicado.

```
ArrayList<String> list = new ArrayList<>();
list.add("One");
list.add("OneAndOnly");
list.add("Derek");
list.add("Change");
list.add("factory");
list.add("justBefore");
list.add("Italy");
list.add("Italy");
list.add("Thursday");
list.add("");
list.add("");

// Creates a Stream<String> of the List<String>, finds all elements of this
// stream which contain char “d” and creates a new stream containing only
// the filtered elements.

Stream<String> stream = list.stream().filter(element -> element.contains("d"));
```

Mapping

map() convierte elementos de un Strem aplicandoles una función y recolecta estos nuevos elementos en un Stream.

```
List<String> uris = new ArrayList<>();
uris.add("C:\\My.txt");
Stream<Path> stream = uris.stream().map(uri -> Paths.get(uri));
```

Mapping

flatMap() actúa sobre un stream donde cada elemento contiene su propia secuencia de elementos, creando un stream con todos estos elementos internos.

```
// In this example, we have a list of elements of type Detail.
// The Detail class contains a field "parts", which is a List<String>.
// With the help of the flatMap() method every element from field "parts"
// will be extracted and added to the new resulting stream.
// After that, the initial Stream<Detail> will be lost

List<Detail> details = new ArrayList<>();
details.add(new Detail());
Stream<String> stream
  = details.stream().flatMap(detail -> detail.getParts().stream());
```

Matching

Stream API permite validar elementos de una secuencia según un predicado: anyMatch(), allMatch(), noneMatch(). Todos son operaciones terminales que retornan boolean.

```
boolean isValid = list.stream().anyMatch(element -> element.contains("h")); // true
boolean isValidOne = list.stream().allMatch(element -> element.contains("h")); // false
boolean isValidTwo = list.stream().noneMatch(element -> element.contains("h")); // false
```

Reduction

Stream API permite reducir una secuencia de elementos a un valor según una función especificada. Para ello ofrece reduce() que toma dos parámetros: valor inicial y función de acumulación.

```
// Given a List<Integer>, you want to have a sum of all these elements
// and some initial Integer (23 on this case).
// After the following code the result will be 26 (23 + 1 + 1 + 1).

List<Integer> integers = Arrays.asList(1, 1, 1);
Integer reduced = integers.stream().reduce(23, (a, b) -> a + b);
```

Collecting

El tipo Stream también ofrece el método collect() para reducir. Es muy útil para convertir un stream en un Collection, un Map o una cadena de texto.

Java ofrece la clase Collectors que brinda solución a las operaciones típicas de conversión.

```
// Reduce a Stream<String> to the List<String>.

List<String> resultList = list.stream().map(element ->
  element.toUpperCase()).collect(Collectors.toList());
```

REFERENCIAS

Para profundizar

Package java.util.stream
https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Data streaming and functional programming in Java
https://opensource.com/article/20/1/javastream

https://www.baeldung.com/java-8-streams
https://www.adictosaltrabajo.com/2016/06/23/uso-basico-de-java-8-stream-y-lambdas/
https://www.baeldung.com/java-map-computeifabsent

PREGRADO

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova