Programación funcional

Programación funcional

- **La programación funcional** es un paradigma de programación declarativo donde el programador especifica lo que quiere hacer, en lugar de lidiar con el estado de los objetos.
- Es decir, las funciones estarían en un primer lugar y nos centraremos en expresiones que pueden ser asignadas a cualquier variable.
- Se basa en un lenguaje matemático formal.
- Es más expresivo, se usa menos código para hacer las mismas operaciones y es más elegante.

Programación funcional

- **La programación funcional en java es un nuevo estilo de programación** que se enfoca en que vas a resolver y no en cómo resolverlo.
- Para trabajar con programación funcional en java podemos usar **lambdas** y **streams**

Expresiones lambda

- **Una expresión lambda** es una función anónima que se utilizan para programar de una forma más concisa y directa.

- Funciones anónimas, que se utilizan para programar de una manera más concisa y directa.
- Se pueden usar donde se acepte una **interfaz funcional**.

Interfaz funcional

- **Una interfaz funcional es una interfaz que tiene un solo método abstracto y se utiliza para representar contratos de función, lo que permite el uso de lambdas y referencias de métodos de manera eficiente en programación funcional.**
- **Se puede anotar con @FunctionalInterface**
- Pueden tener un montón de métodos estáticos, por defecto o abstractos, pero de la clase **object**.
- Si solamente tienen un método abstracto entonces es una interfaz funcional de forma que se puede utilizar como una expresión **Lambda**.
- En lugar de implementar una clase o una clase anónima, podemos utilizar una expresión **Lambda**.

Interfaz funcional

- **Forma “tradicional”**

```
public static int multiplicar (int a , int b){
return a*b;
}
```

- **Podemos convertirlo a interfaz functional**

```
@FunctionalInterface
public interface Multiplicador{
int multiplicar(int a, int b);
}
```

Interfaz funcional

- Teniendo

```
@FunctionalInterface
public interface Multiplicador{
int multiplicar(int a, int b);
}
```

- Podemos usarlo como:

```
Multiplicador multiplicador = (x,y) ->x*y;
int d = multiplicador.multiplicar (2, 3);
```

Expresiones lambda

- **Sintáxis**:
- ()-> expresión
- (parámetros) ->{expresión} // unparámetro->{expresión}
- (parámetros) ->{sentencias;}

- EL operador arrow separa la declaración de los parámetros del cuerpo de la función.
- Si sólo hay un parámetro no son necesarios los paréntesis
- Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

Expresiones lambda

- **Sintáxis**:
- ()- > expresión
- (parámetros)- >{expresión}
- (parámetros)- >{sentencias;}

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

Expresiones lambda

- **Ejemplos**:
- ()->System.out.println(“Hola”)
- x -> x + 10
- x->x.lenght
- (x,y)->x+y

Expresiones lambda

- **Ejemplos**:
- () -> new ArrayList<>()
- (int a, int b)-> a+b
- (int longitud, int altura) -> { return altura * longitud; }
- (x,y)->{
  System.out.println(x + "+" + y)
  return x+y;
  }

Expresiones lambda

```java
@FunctionalInterface
public interface Saludar {
    String decirHola(String s);
}
```

```java
public class Principal{

    public static void main(String[] args) {

    Saludar s = (nombre) -> "Hola " + nombre;
    System.out.println(s.decirHola("Pepito"));
    System.out.println(s.decirHola("a todos"));
    System.out.println(s.decirHola(""));
}
```

Expresiones lambda

```
@FunctionalInterface
public interface Multiplicador {
int multiplicar (int a, int b);
}
```

```
public class Principal{

    public static void main(String[] args) {

    Multiplicador multiplicador = (x,y) ->x*y;
    int d = multiplicador.multiplicar(2, 3);

  }
}
```

Ejemplos

- () -> new ArrayList<>()
- (int a, int b)-> a+b
- (a) -> {

```
System.out.println(a);
return true;
}
```

Clasificación

- **Supplier**: función que no tiene parámetros y devuelve un resultado
- **Consumidores**: función que acepta un solo valor y no devuelve ninguno
- **Biconsumidores**: tienen dos parámetros y no devuelven valor
- **Proveedores**: función que no tienen parámetros, pero devuelven un resultado
- **Funciones**: expresiones que aceptan un argumento y devuelven un valor como resultado. Los tipos no tienen por qué ser iguales
- **Operadores Unarios**: si argumento y valor son del mismo tipo
- **Operadores Binarios**: los dos argumentos y el valor son del mismo tipo
- **Predicados**: aceptan un parámetro y devuelven un valor lógico

Colecciones y forEach

- Las colecciones incorporan el método **forEach**, el cual nos va a permitir usar expresiones Lambda.
- Este método espera una instancia de un objeto tipo **Consumer<T>** que se puede sustituir al ser una interfaz funcional por una expresión Lambda.

```
List<String> list=new ArrayList<String>();
 list.add("Pepito");
 list.add("Eva");
 list.forEach(
        (nombre)->System.out.println(nombre)   //expresión lambda
       );
```

Incluso `list.forEach(System.out::println);`

API Stream

- Los "streams " son secuencias de elementos que permiten procesar **colecciones de datos** de manera eficiente y legible.
- Permite realizar fácilmente operaciones de filtrado, transformación, ordenación, agrupación y presentación de información.
- Stream es **una librería** que facilita mucho el trabajo con Collections, como List, Sets, y consiste en llamar al método **stream()** para después llamar a las funciones de filtrado, transformación, ordenación, agrupación y presentación de información.
- Permiten realizar estas operaciones de manera más concisa y funcional que los enfoques tradicionales de bucles.

Streams

Para poder trabajar con Streams necesitamos trabajar con clase que implementen la interfaz java.util.Collection.

```
// Creamos una lista de números
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);


// Usamos filter para seleccionar solo los números pares


List<Integer> numerosPares = numeros.stream() // de lista a stream
        .filter(numero -> numero % 2 == 0)        //operación intermedia
        .collect(Collectors.toList());            //op. terminal (de stream a lista)
    // Imprimimos los resultados
System.out.println("Números pares: " + numerosPares);
```

Ejemplos

- List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

- **Recorrer e imprimir todos los elementos de la lista**

```
lista
    .stream()
    .forEach((n)-> System.out.println(n));
```

Ejemplos

- **Realizar operaciones de filtrado**
- Por ejemplo, imprimir solo los mayores o iguales que 5:

```
lista
    .stream()
    .filter((x)-> x>=5)
    .forEach((n)-> System.out.println(n));
```

Ejemplos

- **imprimir solo los mayores o iguales a 15 ordenados inversamente:**

```
lista
    .stream()
    .filter((x)-> x>=15)
    .sorted((n1,n2)-> -(n1.compareTo(n2)));
    .forEach((n)-> System.out.println(n));
```

Ejemplos

- **Sumar todos los elementos mayores o iguales a 5:**

```
int resultado = lista
                    .stream()
                    .mapToInt(v-> v.intValue())    //Pasarlo de array a entero
                    .filter((x)-> x>=5)
                    .sum();
System.out.println(resultado);
```

Operaciones intermedias

- **filter (Predicate <T> predicate)**
- Filtra los elementos del stream según un criterio definido por el predicado y crea un nuevo "stream" con los elementos que cumplen ese criterio.

```
.filter(numero -> numero % 2 == 0)
```

- **map(Function <T , R> mapper)**
- Transforma cada elemento del stream según la función especificada y crea un nuevo stream con los resultados de la transformación

```
.map(numero -> numero * 2)
```

Operaciones intermedias

- **distinct ()**
- Elimina elementos duplicados del stream dejando sólo valores únicos

```
.stream().distinct()
```

- **limit(long maxSize)**
- Limita el stream a un número máximo de elementos especificados por maxSize

```
stream().limit(5)
```

- **skip(long n)**
- Omite los primeros n elementos y devuelve un nuevo stream con los elementos restantes

```
stream().skip(3)
```

Operaciones intermedias

- **forEach (Consumer <T> action)**
- **collect()**
- **reduce()**
- **min()**
- **max()**
- **count()**
- **average()**