Expresiones Lambda

Antonio Espín Herranz

Introducción

- Java 1.8 permite escribir código como este:

- Toma dos parámetros enteros y devuelve su suma:

  (int x, int y) → x+y

  (x, y) → x+y

- Toma un parámetro y lo muestra por pantalla:

  msg → System.out.println(msg)

Expresiones Lambda

- ¿Cuál es el tipo de las siguientes expresiones en Java?

(**String** str) → str.length()

  – **Se especifica el tipo …**

person -> (if (person.age > 40) return “Mayor”;

else return “joven”;

  – **No especifica el tipo**

Expresiones Lambda

- La primera puede ser:

```
@FunctionalInterface
interface StringToIntMapper {
    int map(String str);
}
```

Expresiones Lambda

- En el caso de la 2ª, no hay tipado explícito.

- Hay dos tipos de expresiones en Java:
  - **Standalone expressions**
  - **Poly expressions**

Standalone expressions

- En una **expresión Standalone**, Java puede **conocer su tipo sin necesidad de contexto** …

- Ejemplos:
  - new String(“Hello”)
  - “Hello”
  - new ArrayList<String>()

Poly expressions

- En estas expresiones Java No puede conocer su tipo como en las anteriores:

  new ArrayList<>(); **// Puede generar varios tipos …**

- Dependiendo del contexto la expresión puede tener distintos tipos:
  - List<Long> idList = new ArrayList<>();
  - List<String> nameList = new ArrayList<>();

Poly expressions

- Las expresiones **Lambda** en Java son "**poly expressions**".
- El tipo será inferido siempre en un contexto determinado (aunque no sea necesario, como en el ejemplo (String s) → s.length()
- De este modo la expresión **"(x,y)-> x+y"** es una poly expression cuyo **tipo dependerá del contexto**:
  - @FunctionalInterface interface intIntToIntMapper{ int map(int, int); }
  - @FunctionalInterface interface doubleDoubleToDoubleMapper{ double map(double, double); }
  - @FunctionalInterface interface stringStringToStringMapper{ String map(String,String); }

Exp.Lambda

- Las expresiones Lambda son una “versión reducida” de las clases anónimas:

- Se pueden definir como “azúcar sintáctico” para **clases anónimas** que:
  - **No tengan estado**
  - **Contengan un solo método**.

Exp.Lambda

```
// Clase anónima (sin estado y con un solo método):

class StringToIntMapper mapper = new StringToIntMapper(){
    @Override
    public int map(String str){
        return str.length()
    }
}

// Expresión lambda
StringToIntMapper mapper = (String str) → str.length();
```

parámetro → resultado

Los tipos en las e.Lambda

- // Los tipos de los parámetros son declarados:
  (int x, int y) → {return x+y; }

- // Los tipos de los parámetros se omiten:
  (x, y) → {return x+y; }

- // Error de compilación
  (int x, y) → {return x+y; }

- ***O tipamos todos los parámetros o no se tipa nada, no se pueden mezclar!!***

Ejemplo

- // Declara el tipo del parámetro:

  (String s) → {System.out.println(s);}

- // Omite el tipo del parámetro:

  (s) → {System.out.println(s);}

- // Omite el parámetro de tipo y los paréntesis:

  s → {System.out.println(s);}

Ejemplos

- // También se puede con parámetros final:

  (final int x, final int y)→ {return x+y;}

  (int x, final int y) → {return x+y;}

- // Forma no válida:

  (final x, final y)→ {return x+y;}

El cuerpo de la e.Lambda

- El cuerpo de la expresión lambda puede ser:
  - Un **bloque**:

    (int x, int y) → {return x+y;}

  - Una **expresión**, que se evalúa y es el valor de la expresión:

    (int x, int y) → x+y

El cuerpo de la e.Lambda

- También puede haber más expresiones elaboradas:

```
(oldState, newState)→{
    System.out.println(“old State: “ + oldState);
    System.out.println(“new State: “ + newState);
}
```

Tipos

- Las expresiones lambda son instancias de Interfaces Funcionales:
  - Una **interface funcional** es una interface que contiene **un solo método abstracto** (los métodos default, static y los públicos heredados de Object no cuentan)

- En la librería de Java hay múltiples casos de interfaces de este tipo:

```
@FunctionalInterface
interface Comparator<T>
{
    int compare(T o1, To2);
}
```

Tipos

- Una expresión lambda se puede utilizar en cualquier sitio que se espera una instancia de una clase "**Single Abstract Method**" (ni siquiera de una @FunctionalInterface, que es un tipo particular de SAM).

Runnable r1=() → System.out.println("Un Runnable");

Tipos

- *Los interface SAM que llevan la anotación **@FunctionalInterface** son las que Java espera o recomienda que se implementen por medios de expresiones lambda.*

Comparator<Humano> r = (Humano h1, Humano h2) → h1.getName().compareTo(h2.getName);

Target typing

- Para que la expresión lambda sea asignable a una interfaz T:
  - T debería ser una interfaz funcional
  - La expresión lambda tiene que tener el mismo número de parámetros que el método abstracto de T, y los tipos deben ser compatibles.
  - El tipo del valor de retorno de la lambda debe ser compatible con el método abstracto de T.
  - Si el cuerpo de la lambda lanza (throw) alguna excepción, estas deben ser compatibles con las que lanza el método abstracto de T.

Tipos

Si tenemos estas dos interfaces funcionales:

```
@FunctionalInterface
public interface Adder {
double add(double n1, double n2);
}

@FunctionalInterface
public interface Joiner {
String join(String s1, String s2);
}
```

```
(x, y) -> x + y;
```
NO

```
Adder adder = (x, y) -> x + y;
```
OK

```
Adder adder = (x, y) -> x - y;
```
OK

```
Adder adder = (double x, double y) -> x + y;
```
OK

```
Joiner joiner = (x, y) -> x + y;
```
OK

Tipos

```
public class Ejemplo {

        public void test(Adder adder)
        {
                double x = 190.90;
                double y = 8.50;
                double sum = adder.add(x, y);
                System.out.println(x + " + " + y + " =
                " + sum);
        }

        public void test(Joiner joiner)
        {
                String s1 = "Hello";
                String s2 = "World";
                String s3 = joiner.join(s1,s2);
                System.out.println("\"" + s1 + "\" + \""
                + s2 + "\" = \"" + s3 + "\"");
        }
}
```

Creamos un objeto de Ejemplo:
`Ejemplo util = new Ejemplo();`

¿Y ahora, esto compila?

`util.test((double x, double y) -> x + y);` ✓
`util.test((double x, double y) -> x - y);`

`util.test((Adder)(x, y) -> x + y);` ✓

`Adder adder = (x, y) -> x + y;` ✓
`util.test(adder);`

`util.test((x, y) -> x + y);` ✗

Interfaces Funcionales

Nombre de la Interfaz	Método	Descripción
Function<T,R>	R apply(T t)	Representa una función que coge un parámetro de tipo T y devuelve un resultado de tipo R
BiFunction<T,U,R>	R apply(T t, U u)	Representa una función que coge dos parámetros de tipos T y U, y devuelve un resultado de tipo R
Predicate<T>	boolean test(T t)	Dado un parámetro de tipo T, comprueba algo y devuelve un booleano
BiPredicate<T,U>	boolean test(T t, U u)	Predicado de dos parámetros
Consumer<T>	void accept(T t)	Representa una operación que toma un parámetro, opera con él para hacer algo y no devuelve un resultado
BiConsumer<T,U>	void accept(T t, U u)	Análogo al anterior con dos parámetros
Supplier<T>	T get()	Devuelve un elemento de tipo T
UnaryOperator<T>	T apply(T t)	Hereda de Function<T,T>. Representa una función que coge un parámetro y devuelve un resultado del mismo tipo
BinaryOperator<T>	T apply(T t1, T t2)	Hereda de BiFunction<T,T,T>. Representa una función que coge dos parámetros del mismo tipo y devuelve un resultado del mismo tipo también

Lista completa:
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Ejemplos

```
// Function square = x -> x*x; //NO SE PUEDE

// Las siguientes tres son equivalentes:

Function<Integer,Integer> square = x -> x*x;

UnaryOperator<Integer> square2 = x -> x*x;

IntFunction square3 = x -> x*x;

System.out.println(square.apply(4));
```

Funciones como parámetros

- Ordenación de listas:
  - Definimos una lambda que instancie la interface **Comparator** sobre tipos genéricos, Java puede determinar el tipo según los parámetros:

    Comparator<Comparable> comp = ((a,b)→a.compareTo(b));

    List<Integer> numeros = Arrays.*asList(1,5,4,0,-1,5,3,22);*
    numeros.sort(*comp);*
    System.***out.println(numeros);***

  - ***También valdría:***

    numeros.sort(*comp.reversed());*
    Valdría para cualquier clase que implemente Comparable

Funciones como parámetros

```
Empleado Eloy = new Empleado("Eloy", 25000, "Famoso corredor", 50);

Empleado Felix = new Empleado("Félix", 5000, "Currante serio y madrugador", 31);

Empleado Julio = new Empleado("Julio", 75000, "Rector", 52);

Empleado Laure = new Empleado("Laure", 100000, "Amante de los ordenadores", 52);
```

Podemos ordenar de forma muy fácil por cualquiera de los campos:

```
listaEmpleados.sort(Comparator.comparing(Empleado::getNombre));
listaEmpleados.sort(Comparator.comparing(Empleado::getSueldo));
listaEmpleados.sort(Comparator.comparing(Empleado::getEdad));
```

Java Lambda: Variables

- Al igual que ocurre en las clases anónimas, una expresión lambda puede acceder a variables locales “de fuera”:
  - Si son finales, y
  - Si no son finales pero se han inicializado una sola vez.

```
public Printer test() {
        final String msg = "Hello";
        Printer printer = msg1 -> System.out.println(msg + " " + msg1);
        return printer;
}
```

OK

```
 public Printer test() {
         String msg = "Hello";
         Printer printer = msg1 -> System.out.println(msg + " " + msg1);
         msg = “How are you?”;
         return printer;
 }
```

Error de compilación

Lambda Serializable

```
public class Main {
  public static void main(String[] argv) {
   Serializable ser = (Serializable & Calculator) (x,y)-> x + y;
  }
}

@FunctionalInterface
interface Calculator{
  long calculate(long x, long y);
}
```

Conclusiones

- No se pueden utilizar expresiones lambda solas, se necesita una interfaz funcional.
- Las lambda facilitan crear instancias de interfaces funcionales, sobre todo en comparación con clases anónimas.
- Sintaxis concisa.
- Se pueden utilizar para pasar funciones como parámetros.