

EXPRESIONES LAMBDA

FRANCISCO MANUEL GAMERO RODRÍGUEZ Y PABLO MÉNDEZ ESPAÑA



INTRODUCCIÓN

En este documento tiene el objetivo de servir como tutorial y explicación de las expresiones lambda en java, se hablará sobre su estructura básica, sus usos y aplicaciones más comunes y algunos elementos del lenguaje java que se usan en conjunto con estas expresiones para sacar su máximo potencial.

ÍNDICE

1. Sintaxis de las expresiones lambda (Fco. Manuel)
2. Referencia de método (Fco.Manuel)
3. Interfaces funcionales (Pablo)
4. Predicados (Interfaz *predicate*) (Fco. Manuel)
5. Otras interfaces funcionales (*Supplier*, *Consumer*, *Function*, *BiFunction*) (Pablo)

1. Sintaxis de las expresiones lambda

Las funciones lambda son aquellas que se escriben en la misma línea de código donde se usa. Esta al igual que cualquier función puede recibir ningún o más parámetros y pueden devolver o no (pueden ser void). También puede recibir métodos de otras clases y objetos como cualquier función, y al declararse dentro de una clase, puede acceder a las variables de dicha clase, pero solo puede verlas y no modificarlas (solo puede hacer get).

Lambda se conforma de interfaces funcionales que se encuentran en `java.util.function`. Las más usadas son: `Supplier`, `Consumer`, `BiConsumer`, `Function`, `BiFunction` y `Predicate`.

La sintaxis de las expresiones lambda es la siguiente:

```
InterfazFuncional <LoQueRecibe,LoQueDevuelve>
NombreDeLambda(El que le quieras dar) = LoQueRecibe -> LoQueRealiza
```

Va todo en la misma línea, pero para que se vea bien lo hemos puesto en dos.

Lo que recibe y lo que devuelve la expresión, depende de la interfaz funcional que estemos usando, hay casos en las que no hay que poner nada, como en `Supplier`, en las que solo hay que poner lo que recibe como en `Predicate`, o hay que poner las dos cosas como en `Function`

2. Referencia de método

El operador `::` o “Referencia de Método” es un operador de java introducido en Java 8 y este tiene múltiples usos:

-En métodos:

```
public void mostrarListaClientes() {
    lista.forEach(System.out::println);
    /* Recorre toda la lista imprimiendo cada cliente */
}
```

-En constructores:

```
public static void main(String[] args) {
    //Llama al constructor de Empleado vacío
    Supplier<Empleado> empleadoSupplier = Empleado::new;
    //Le da a empleado ese empleado vacío creado en "empleadoSupplier"
    Empleado empleado = empleadoSupplier.get();

    //Pone un nombre y edad por defecto
    empleado.nombre = "John Doe";
    empleado.edad = 30;
}
```

-En expresiones lambda:

```
public void agregarCliente(Cliente c) {

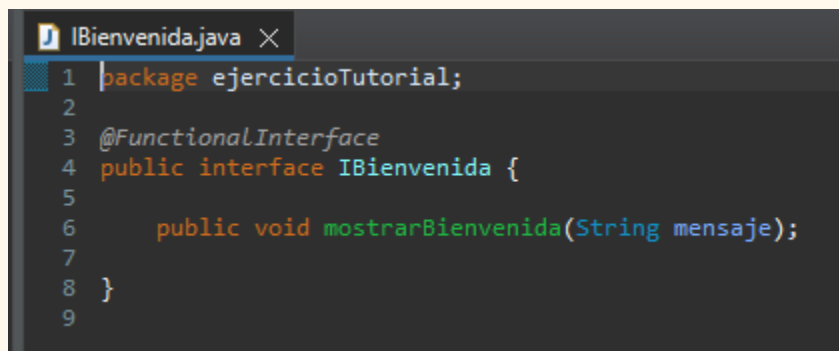
    Predicate<List<Cliente>> predicate = (l) -> l.add(c);
    Consumer<List<Cliente>> consumer = predicate::test;
    /* Un ejemplo tonto de como se puede reutilizar lambda */
    /* Poniendo que solo se pueda añadir si cumple el if */
    if (c.getCantidadBotellas() > 2) {
        consumer.accept(lista);
    }
}
```

3. Interfaces funcionales

Las expresiones lambda se componen de unas entidades básicas, las principales entidades de las que hablamos son interfaces con un único método que debe implementar el programador. Estas interfaces son conocidas como interfaces funcionales y pertenecen al paquete `java.util.function`, por lo que aparecen todas en el API de Java.

Las interfaces funcionales componen una lógica interna y cada una se relaciona con los objetos de diferente forma. En este caso hablaremos de las 5 interfaces funcionales más usadas, que son: *Supplier*, *Consumer*, *Function*, *BiFunction* y *Predicate*.

Además de las interfaces funcionales ya presentes en el lenguaje java, nosotros mismos podemos crear interfaces funcionales y aplicarlas a nuestro código, tal como sale en el ejemplo.



```
1 package ejercicioTutorial;
2
3 @FunctionalInterface
4 public interface IBienvenida {
5
6     public void mostrarBienvenida(String mensaje);
7
8 }
9
```

4. Predicados (Interfaz predicate)

La interfaz Predicate devuelve un booleano y esta tiene la siguiente sintaxis:

Predicate<TipoObjetoQueRecibe> nombreLambda = (TipoObjetoQueRecibe nombre) -> Lo que comprueba

Aquí podemos observar un ejemplo.

```
public void sumarBotella(String dni) {
    Predicate<Cliente> menor = (Cliente c) -> c.getEdad() >= 18;
    if (menor.test(listaClientes.buscarPorDni(dni))) {
        listaClientes.buscarPorDni(dni)
            .setCantidadBotellas(listaClientes.buscarPorDni(dni).getCantidadBotellas() + 1);
    }
}
```

5. Otras interfaces funcionales

La interfaz Supplier es una interfaz que no recibe ningún parámetro pero sí devuelve un objeto, haciendo uso de la siguiente sintaxis:

Supplier<TipoObjeto> nombreLambda = () -> Argumentos;

Cabe destacar que esta interfaz es parametrizada, es decir, debemos indicar el tipo de objeto.

Ejemplo de supplier:

```
case 2:
    Supplier<Double> kmR = () -> Math.random();
    for(Vehiculo v : lista) {
        kmRecorridos = kmR.get();
        v.setKmRecorridos(kmRecorridos);
    }
    break;
```

(Contextualizado en el proyecto tutorial)

La interfaz Consumer recibe un tipo de argumento y no devuelve nada, normalmente se suele usar para trabajar con listas, su sintaxis es:

Consumer<TipoObjeto> nombreLambda = (Parámetro) -> Argumentos;

También es parametrizada.

Este es un ejemplo:

```
case 3:
    Consumer<String> mostrarMatricula = (m) -> System.out.println(m);
    for(Vehiculo v : lista) {
        mostrarMatricula.accept(v.getMatricula());
    }
    break;
```

(Contextualizado en el proyecto tutorial)

La interfaz Function recibe un objeto y devuelve otro objeto tras pasar por unos argumentos, siguiendo la siguiente sintaxis:

Function <tipoObjeto1, tipoObjeto2> nombreLambda = (Parámetro) -> Argumentos;

Este es un ejemplo:

```
case 4:
    Function<Double, Double> aMetros = (km) -> km*1000;
    Function<Double, Double> aCm = (mt) -> mt*100;
    for(Vehiculo v : lista) {
        System.out.println(aMetros.apply(v.getKmRecorridos()));
    }
    for(Vehiculo v : lista) {
        System.out.println(aMetros.andThen(aCm).apply(v.getKmRecorridos()));
    }
    break;
```

Cabe destacar que la interfaz function puede anidarse a través del método andThen, tal como sale en el ejemplo.

La interfaz BiFunction puede recibir dos objetos y devolver otro objeto en resultado de argumentos aplicados a los otros objetos recibidos, a partir de la sintaxis:

```
BiFunction <tipoObjeto1, tipoObjeto2, tipoObjeto3> nombreLambda = (Parámetro1, Parámetro2) -> Argumentos;
```

Este es un ejemplo:

```
case 5:
    BiFunction<Integer, Double, Double> multiplicacion = (cl, kmr) -> cl*kmr;
    for(Vehiculo v : lista) {
        if(v instanceof Moto) {
            System.out.println(multiplicacion.apply(((Moto) v).getCilindrada(), v.getKmRecorridos()));
        }
    }
    break;
```

Los ejemplos anteriores muestran las instancias de las interfaces funcionales, para usarlas debemos llamar a los métodos de las interfaces.