Expresiones Lambda

Escribir una expresión lambda se reduce a comprender tres pasos:

- Identificar el tipo de expresión lambda que desea escribir
- Encontrar el método adecuado para implementar
- Implementando este método.

En realidad, esto es todo lo que hay que hacer. Veamos estos tres pasos en detalle.

Identificación del tipo de una expresión lambda

En el lenguaje Java, todo tiene un tipo y este tipo se conoce en el momento de la compilación. Por lo tanto, siempre es posible encontrar el tipo de una expresión lambda. Puede ser el tipo de una variable, de un campo, de un parámetro de un método o el tipo devuelto por un método.

Existe una restricción en cuanto al tipo de expresión lambda: debe ser una interfaz funcional. Por lo tanto, una clase anónima que no implemente una interfaz funcional no puede escribirse como una expresión lambda.

La definición completa de lo que son las interfaces funcionales es un poco compleja. Todo lo que necesitas saber en este punto es que una interfaz funcional es una interfaz que tiene solo un método ***abstracto*.**

Debes tener en cuenta que, a partir de Java SE 8, se permiten métodos concretos en las interfaces. Pueden ser métodos de instancia, en cuyo caso se denominan *métodos* ***default***, y pueden ser métodos **estáticos.** Estos métodos no cuentan, ya que no son métodos ***abstractos***.

¿Necesito agregar la anotación ***@FunctionalInterface*** *en una interfaz para que sea funcional?*

No, no es necesario. Esta anotación está aquí para ayudarte a asegurar que tu interfaz sea realmente funcional. Si colocas esta anotación en un tipo que no es una interfaz funcional, el compilador generará un error.

Ejemplos de interfaces funcionales

Veamos algunos ejemplos extraídos de la API del JDK.

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

La interfaz ***Runnable*** es funcional, de hecho, porque solo tiene un método abstracto. Se agregó la anotación **@FunctionalInterface** como ayuda, pero no es necesaria.

```java
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        // the body of this method has been removed
    }
}
```

La interfaz **Consumer<T>** también es funcional: tiene un método abstracto y un método concreto predeterminado que no cuenta. Una vez más, **@FunctionalInterface** no es necesaria la anotación.

```java
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        // the body of this method has been removed
    }

    default Predicate<T> negate() {
        // the body of this method has been removed
    }

    default Predicate<T> or(Predicate<? super T> other) {
        // the body of this method has been removed
    }

    static <T> Predicate<T> isEqual(Object targetRef) {
        // the body of this method has been removed
    }

    static <T> Predicate<T> not(Predicate<? super T> target) {
        // the body of this method has been removed
    }
}
```

La interfaz **Predicate<T>** es un poco más compleja, pero sigue siendo una interfaz funcional:

- Tiene un método abstracto
- Tiene tres métodos predeterminados que no cuentan
- y tiene dos métodos estáticos que tampoco cuentan.

Encontrar el método correcto para implementar

En este punto has identificado el tipo de expresión lambda que necesitas escribir, y la buena noticia es que has hecho la parte más difícil: el resto es muy mecánico y más fácil de hacer.

Una expresión lambda es una implementación del único método abstracto en esta interfaz funcional. Por lo tanto, encontrar el método correcto para implementar es solo una cuestión de encontrar este método.

Puedes tomarte un minuto para buscarlo en los tres ejemplos del párrafo anterior.

Para la interfaz Runnable es:

```
public abstract void run();
```

Para la interfaz Predicate es:

```
boolean test(T t);
```

Y para la interfaz Consumer<T> es:

```
void accept(T t);
```

Implementando el método correcto con una expresión Lambda

Cómo escribir la primera expresión Lambda que implementa Predicate<String>

Ahora, la última parte: escribir la expresión lambda en sí. Lo que debes entender es que la expresión lambda que estás escribiendo es una implementación del método abstracto que encontraste. Con la sintaxis de la expresión lambda, puedes incorporar esta implementación en tu código.

Esta sintaxis se compone de tres elementos:

- un bloque de parámetros;
- un pequeño fragmento de arte ASCII que representa una flecha: ->. Tenga en cuenta que Java utiliza *flechas pequeñas* (->) y no *flechas gruesas* (=>);
- un bloque de código que es el cuerpo del método.

Veamos algunos ejemplos. Supongamos que necesita una instancia de Predicate que devuelva true para cadenas de caracteres que tengan exactamente 3 caracteres.

1. El tipo de su expresión lambda es Predicate
2. El método que necesitas implementar es `boolean test(String s)`

Luego escribes el bloque de parámetros, que es simple copiar/pegar de la firma del método: (String s).

Luego añade una pequeña flecha: ->.

Y el cuerpo del método. El resultado debería verse así:

```
Predicate<String> predicate =
    (String s) -> {
        return s.length() == 3;
    };
```

Simplificando la sintaxis

Esta sintaxis se puede simplificar gracias al compilador que puede adivinar muchas cosas para que no sea necesario escribirlas.

En primer lugar, el compilador sabe que estás implementando el método abstracto de la interfaz Predicate y que este método toma un String argumento. Por lo tanto, `(String s)`. se puede simplificar a `(s)`. En ese caso, donde solo hay un argumento, puedes ir un paso más allá eliminando los paréntesis. El bloque de argumentos se convierte entonces en `s`. Debes mantener los paréntesis si tienes más de un argumento o ningún argumento.

En segundo lugar, solo hay una línea de código en el cuerpo del método. En ese caso, no necesitas las llaves ni la palabra clave return.

Así que la sintaxis final es de hecho la siguiente:

```
Predicate<String> predicate = s -> s.length() == 3;
```

Y esto nos lleva a la primera buena práctica: mantener las lambdas cortas, de modo que sean solo una línea de código simple y legible.

Implementando una `Consumer<String>`

En algún momento, las personas pueden verse tentadas a tomar atajos. Escuchará a los desarrolladores decir "un consumidor toma un objeto y no devuelve nada" o "un predicado es verdadero cuando la cadena tiene exactamente tres caracteres". La mayoría de las veces, existe una confusión entre la expresión lambda, el método abstracto que implementa y la interfaz funcional que contiene este método.

Pero como una interfaz funcional, su método abstracto y una expresión lambda que la implementa están tan estrechamente vinculados entre sí, esta forma de hablar tiene todo el sentido. Así que está bien, siempre y cuando no genere ninguna ambigüedad.

Escribamos una lambda que consume a String y la imprime en `System.out`. La sintaxis puede ser la siguiente

```
Consumer<String> print = s -> System.out.println(s);
```

Aquí escribimos directamente la versión simplificada de la expresión lambda.

Implementación de un ejecutable

Implementar a `Runnable` resulta en escribir una implementación de `void run()`. Este bloque de argumentos está vacío, por lo que debe escribirse entre paréntesis. Recuerde: puede omitir los paréntesis solo si tiene un argumento; aquí tenemos cero.

Entonces, escribamos un ejecutable que nos diga que se está ejecutando:

```
Runnable runnable = () -> System.out.println("I am running");
```

Llamar a una expresión Lambda

Volvamos a nuestro ejemplo `Predicate` anterior y supongamos que este predicado se ha definido en un método. ¿Cómo se puede utilizar para comprobar si una cadena de caracteres dada tiene una longitud de 3?

Bueno, a pesar de la sintaxis que usaste para escribir una lambda, debes tener en cuenta que esta lambda es una instancia de la interfaz Predicate. Esta interfaz define un método llamado `test()` que toma un `String` y devuelve un `boolean`.

Escribámoslo en un método:

```
List<String> retainStringsOfLength3(List<String> strings) {

    Predicate<String> predicate = s -> s.length() == 3;
    List<String> stringsOfLength3 = new ArrayList<>();
    for (String s: strings) {
        if (predicate.test(s)) {
            stringsOfLength3.add(s);
        }
    }
    return stringsOfLength3;
}
```

Observa cómo definiste el predicado, tal como lo hiciste en el ejemplo anterior. Dado que la interfaz `Predicate` define este método `boolean test(String)`, es perfectamente legal llamar a los métodos definidos en `Predicate` mediante una variable de tipo `Predicate`. Esto puede parecer confuso al principio, ya que esta variable de predicado no parece definir métodos.

Por lo tanto, cada vez que escriba una lambda, podrá llamar a cualquier método definido en la interfaz que esté implementando esta lambda. Al llamar al método abstracto, se llamará al código de su lambda en sí, ya que esta lambda es una implementación de ese método. Al llamar a un método predeterminado, se llamará al código escrito en la interfaz. No hay forma de que una lambda pueda anular un método predeterminado.

Descubriendo el paquete java.util.function

Las interfaces funcionales en el paquete `java.util.function` en Java son interfaces que tienen un solo método abstracto, permitiendo que se utilicen como expresiones lambda o referencias de método. Estas interfaces facilitan el uso de programación funcional en Java y se emplean comúnmente en Streams y en otras APIs funcionales. A continuación, un resumen de las interfaces funcionales más utilizadas `en java.util.function`:

1. Function<T, R>

- **Descripción**: Representa una función que acepta un argumento de tipo T y produce un resultado de tipo R.
- **Método principal**: R apply(T t)
- **Uso típico**: Transformar datos, como convertir una cadena en su longitud (String -> Integer).

2. Consumer<T>

- **Descripción**: Representa una operación que acepta un solo argumento de tipo T y no devuelve ningún resultado.
- **Método principal**: `void accept(T t)`
- **Uso típico**: Ejecutar una acción sobre un elemento, como imprimir un valor o modificar un objeto.

3. Supplier<T>

- **Descripción**: Representa un proveedor de resultados; no toma ningún argumento y devuelve un valor de tipo T.
- **Método principal :** `T get()`
- **Uso típico**: Proveer o generar valores, como crear instancias o devolver valores por defecto.

4. Predicate<T>

- **Descripción**: Representa un predicado (una función booleana) que acepta un solo argumento de tipo T y devuelve un `boolean.`
- **Método principal**: `boolean test(T t)`
- **Uso típico**: Filtrar elementos o validar condiciones, como verificar si un número es par.

5. BiFunction<T, U, R>

- **Descripción**: Representa una función que toma dos argumentos de tipo `T` y `U` y produce un resultado de tipo `R.`
- **Método principal**: `R apply(T t, U u)`
- **Uso típico**: Combinar o transformar dos valores, como concatenar dos cadenas.

6. BiConsumer<T, U>

- **Descripción**: Representa una operación que acepta dos argumentos de tipo `T` y `U` y no devuelve ningún resultado.
- **Método principal**: `void accept(T t, U u)`

- **Uso típico**: Ejecutar acciones sobre dos elementos, como agregar una clave y un valor en un mapa.

7. UnaryOperator<T>

- **Descripción**: Especialización de `Function<T, T>` donde el argumento y el resultado son del mismo tipo.
- **Método principal**: `T apply(T t)`
- **Uso típico**: Realizar una operación sobre un solo valor que devuelve un valor del mismo tipo, como incrementar un número.

8. BinaryOperator<T>

- **Descripción**: Especialización de `BiFunction<T, T, T>` para operar sobre dos valores del mismo tipo y devolver un valor del mismo tipo.
- **Método principal :** `T apply(T t1, T t2)`
- **Uso típico**: Realizar una operación entre dos valores del mismo tipo, como sumar dos números o comparar valores.

Bibliografía: https://dev.java/learn/lambdas/