TESOEM

ING. EN SISTEMAS COMPUTACIONALES

NOMBRE: MANZO OLGUIN ALEX YAZMIN

GRUPO: 8S12

SEMESTRE: OCTAVO

PROGRMACION LOGICA Y FUNCIONAL

REFERENCIAS A UN MÉTODO

PROFESOR: LOPEZ CASTAÑEDA CESAR ALEJANDRO

Ha habido un denominador común en todos los posts dedicados a la programación funcional: se ha ido buscando cada vez más una mayor legibilidad en el código. Y esa es la premisa última de la referencia a método.

Como su propio nombre indica, es una forma de hacer referencia a un método. Es decir, llama a un método de una forma muy legible y sencilla. El operador que se utiliza es el '::'.

El requisito principal para poder realizar la referencia a método es que, dentro de la expresión lambda, haya el mismo número de variables a ambos lados de la expresión, y además, en el mismo orden. Esto es muy importante porque nos va a limitar en gran medida su uso.

Por ejemplo, si queremos ordenar una lista de números de manera inversa no podremos hacerlo mediante referencia a método directamente:

```
((a, b) -> Integer.compare(b, a))
```

Vemos que el orden está cambiado a, b -> b, a. Para que pueda aplicarse la referencia a método, la ordenación debe ser la siguiente:

```
((a, b) -> Integer.compare(a, b)) se puede convertir en Integer::compare
```

Y esto se traduce en una ordenación directa, de menor a mayor. Esa es la gran limitación de la referencia a método.

Hay una tipología dentro de las referencias a método. Las más importantes son las siguientes:

Referencia a un método estático

Cuando se quiere llamar a un método estático cuyo argumento es el parámetro de la expresión lambda. Por ejemplo, el caso anterior. Si queremos ordenar de menor a mayor una lista de números mediante lambda lo hacíamos así:

```
list.sort((a, b) -> Integer.compare(b, a));
```

Mediante referencia a método será así:

```
list.sort(Integer::compare);
```

Además de esta manera podemos también imprimir por consola cada elemento de la lista, ya que con lambda sería así:

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

Y con referencia a método se solucionaría así:

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

De manera que todo el código se resumiría en dos sencillas sentencias:

```
List<Integer> list = Arrays.asList(1, -1, 3, 5, 7, 9, 0, 2, 4, 6, 8);
list.sort(Integer::compare);
list.forEach(System.out::println);
```

Imprimiendo en consola cada uno de los números de la lista ordenados

Referencia a un método dinámico de un objeto cualquiera

Para este caso utilizaremos una lista de palabras, la cual ordenaremos alfabéticamente. El método de ordenación de cadenas es compareTo, que según la API es un método dinámico de la siguiente forma (compareToIgnoreCase no distingue mayúsculas y minúsculas):

```
s1.compareToIgnoreCase(s2)
```

Como ya sabemos, mediante lambda se haría de la siguiente manera:

```
(s1, s2) -> s1. compareToIgnoreCase(s2)
```

Pero en el terreno de la referencia a método todo se simplifica:

```
String:: compareToIgnoreCase
```

Quedándonos mucho más legible y sencillo:

```
List<String> list = Arrays.asList("Perro", "Caballo", "Gato", "Periquito", "Elefante",
                                  "Zorro", "Chanchito", "Burro");
list.sort(String:: compareToIgnoreCase);
list.forEach(System.out::println);
```

Referencia a un método dinámico de un objeto concreto

Aquí crearemos un método de ordenación en una clase interna Comparador Cadenas para tener la información de cómo vamos a ordenar, y dentro de esa clase definimos el método por Segunda Letra, que compara dos cadenas según su segunda letra:

```
class ComparadorCadenas {
        public int porSegundaLetra(String s1, String s2) {
                return Character.compare(s1.charAt(1), s2.charAt(1));
        }
}
```

Fuera de la clase interna y dentro del main, creamos una instancia de ComparadorCadenas que va a ser la que se encargue de la ordenación (de hecho, este va a ser el objeto concreto que usaremos en la referencia a método).

```
ComparadorCadenas comparando = new ComparadorCadenas();
```

Y a la hora de introducirla en el método sort, le indicamos mediante referencia a método que lo haga por SegundaLetra:

```
list.sort(comparando::porSegundaLetra);
```

De modo que el código al completo sería:

```
List<String> list = Arrays.asList("Perro", "Caballo", "Gato", "Periquito", "Elefante",
                "Zorro", "Chanchito", "Burro");

class ComparadorCadenas {
        public int porSegundaLetra(String s1, String s2) {
                return Character.compare(s1.charAt(1), s2.charAt(1));
        }
}

ComparadorCadenas comparando = new ComparadorCadenas();
list.sort(comparando::porSegundaLetra);
list.forEach(System.out::println);
```

Referencia a constructor

Podemos apuntar a la creación de objetos mediante la programación funcional. Simplemente llama al constructor de una clase. Con lambda se haría así:

```
() -> new Class()
```

Sin embargo, con referencia a método es:

```
Class::new
```

Un uso muy frecuente de este tipo de referencias a método es en streams, donde por ejemplo, podemos transformar de un flujo de cadenas en un flujo de objetos mediante un.

```
.map(Class::new)
```

Pero, ¿y si la clase Class tuviera más de un constructor? No pasa nada. El compilador sabe por inferencia que el constructor que debe utilizar es el que reciba por parámetro una cadena. De todas formas, los streams los veremos en próximos posts.

Y con esto ya habríamos visto los cuatro tipos principales de referencias a método.

La principal ventaja que ofrecen estas herramientas es un nivel de legibilidad altísimo. No obstante, y como ocurría con las expresiones lambda, es una forma de programar diferente a lo que solemos estar acostumbrados y requiere unas cuantas horas (quizás días) de práctica. Pero una vez que dedicamos tiempo a aprenderlo comprobamos que merece mucho la pena.