I.3 Programación funcional

Acceso a Datos

Alejandro Roig Aguilar
alejandro.roig@iesalvarofalomir.org

IES Álvaro Falomir
Curso 2023-2024

¿Qué es la programación funcional?

Es un paradigma de programación declarativo, no imperativo. En estos paradigmas se dice cómo es el problema a resolver, en lugar de los pasos a seguir para resolverlo.

Entre los lenguajes de programación funcionales destacamos:

- Puros: Miranda, Haskell
- Híbridos (adaptados a otros paradigmas): Clojure, Scala

La mayoría de lenguajes actuales no se pueden considerar funcionales, pero sí han adaptado su sintaxis y funcionalidad para ofrecer parte de este paradigma.

Características

- Transparencia referencial. La salida de una función depende solo de sus argumentos. Si la llamamos varias veces con los mismos argumentos, debe producir siempre el mismo resultado.
- Inmutabilidad de los datos. Los datos no modifican su valor para evitar posibles efectos colaterales.
- Composición de funciones. Las funciones se tratan como datos, de modo que la salida de una función se puede tomar como entrada de la siguiente.
- Funciones de primer orden. Funciones que permite tener otras funciones como parámetros, a modo de callback.

Imperativo vs Declarativo

Ejemplo en Java: obtener una sublista con los mayores de edad entre una lista de personas.

```
List<Persona> adultos = new ArrayList<>();
for (int i = 0; i < personas.size(); i++) {
        if (personas.get(i).getEdad() >= 18)
                adultos.add(personas.get(i));
}
```

IMPERATIVO

```
List<Persona> adultos =
        personas.stream()
                .filter(p -> p.getEdad() >= 18)
                .toList();
```

DECLARATIVO

- \+ compacto
- – propenso a errores

composición de funciones

Funciones lambda

Son funciones que no necesitan una clase, por lo que también se denominan funciones anónimas.

Nos permiten simplificar la implementación de elementos más costosos en cuanto a líneas de código.

Por su sintaxis, en algunos lenguajes se les suele denominar funciones flecha, ya que una flecha separa la cabecera de la función de su cuerpo.

```
.filter(p -> p.getEdad() >= 18)
```

expresión lambda

Ejemplo: comparaciones

Para comparar elementos en Java tenemos distintas opciones:

1) Usando el método Collections.sort() e implementando el método compare() de la interfaz funcional Comparator.

   OJO!!! La interfaz Comparator es similar pero distinta a Comparable, vista en la clase anterior. ¿Qué diferencias hay?

   El valor devuelto decide la posición del primer objeto en relación con el segundo objeto.

   int compare(T o1, T o2)

   - Si devuelve un entero negativo, o1 es menor que o2.
   - Si devuelve un cero, o1 es igual a o2.
   - Si devuelve un entero positivo, o1 es mayor que o2.

Ejemplo: comparaciones

Ordenar una lista de personas de mayor a menor edad usando Comparator:

```
class Persona {
    private String nombre;
    private int edad;
}
```

```
class ComparadorPersona implements Comparator<Persona> {
    @Override
    public int compare (Persona p1, Persona p2) {
        return p2.getEdad() – p1.getEdad();
    }
}
```

```
ArrayList<Persona> personas = new ArrayList<>();
personas.add(new Persona("Pepe", 18));
personas.add(new Persona("Ana", 20));
personas.add(new Persona("Juan", 22));
personas.sort(new ComparadorPersona());
for (int i = 0; i < personas.size(); i++)
    System.out.println(personas.get(i).mostrar());
```

Ejemplo: comparaciones

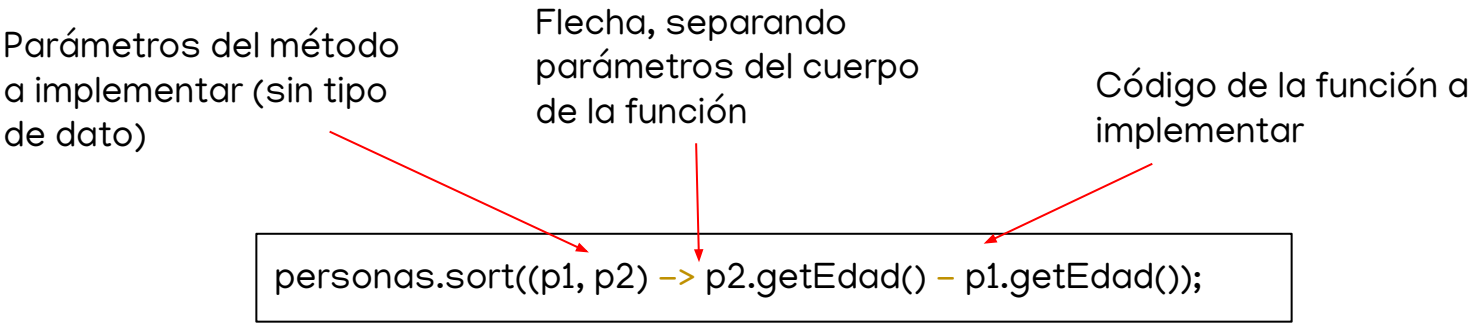
2) Implementación con funciones lambda:

```
ArrayList<Persona> personas = new ArrayList<>();
personas.add(new Persona("Pepe", 18));
personas.add(new Persona("Ana", 20));
personas.add(new Persona("Juan", 22));

personas.sort((p1, p2) -> p2.getEdad() - p1.getEdad());

for (int i = 0; i < personas.size(); i++)
   System.out.println(personas.get(i).mostrar());
```

Estructura de una función lambda



- Los paréntesis del lado izquierdo pueden omitirse si solo hay un parámetro.
- Si el código a la derecha de la flecha necesita hacer más que un simple “return”, se pone entre llaves.

Gestión de colecciones en Java: Streams

Desde Java 8, es posible procesar grandes cantidades de datos aprovechando la paralelización que permita el sistema.

Los streams son envoltorios de colecciones de datos que nos permiten operar con estas colecciones y hacer que el procesamiento masivo de datos sea rápido y fácil de leer.

No modifican la colección original, sino que crean copias.

Dos tipos de operaciones:

- Intermedias: devuelven otro stream resultado de procesar el anterior, para ir enlazando operaciones.
- Finales: cierran el stream devolviendo algún resultado.

Operaciones intermedias con streams: filtrado

El método filter es una operación intermedia que permite filtrar los datos de una colección que cumplan el criterio indicado como parámetro.

```
ArrayList<Persona> personas = new ArrayList<>();
// Aquí añadiríamos personas a la lista de personas

Stream<Persona> adultos =
        personas.stream()
                .filter(p -> p.getEdad() >= 18);
```

filter recibe como parámetro una interfaz Predicate, cuyo método test recibe como parámetro un objeto y devuelve si ese objeto cumple o no una determinada condición

Aquellas personas "p" de la colección cuya edad sea mayor o igual que 18

Operaciones intermedias con streams: mapeo

El método map es una operación intermedia que permite transformar la colección original para quedarnos con cierta parte de la información o crear otros datos.

```
ArrayList<Persona> personas = new ArrayList<>();
// Aquí añadiríamos personas a la lista de personas

Stream<Integer> edades =
        personas.stream()
                .map(Persona::getEdad);
                // .map(p -> p.getEdad());
```

Las edades de aquellas personas "p" de la colección

map recibe como parámetro una interfaz Function, cuyo método apply recibe como parámetro un objeto y devuelve otro objeto diferente, normalmente derivado del parámetro

Se puede utilizar tanto una función lambda como una referencia a un método.

Operaciones intermedias con streams: combinar

Se pueden combinar operaciones intermedias (composición de funciones) para producir resultados más complejos.

Por ejemplo, las edades de las personas adultas:

```
ArrayList<Persona> personas = new ArrayList<>();
// Aquí añadiríamos personas a la lista de personas

Stream<Integer> edadesAdultos =
        personas.stream()
                .filter(p -> p.getEdad() >= 18)
                .map(Persona::getEdad);
```

Operaciones intermedias con streams: ordenar

El método sorted es una operación intermedia que permite ordenar los elementos de una colección según cierto criterio.

```
Stream<Persona> personasOrdenadas =
        personas.stream()
                .filter(p -> p.getEdad() >= 18)
                .sorted((p1, p2) -> p2.getEdad() - p1.getEdad());
```

sorted recibe como parámetro una interfaz Comparator, que ya conocemos

Para cada pareja de personas p1 y p2, ordénalas en función de la resta de la edad de p1 menos la edad de p2

Operaciones finales con streams: colección

El método collect es una operación final que permite obtener algún tipo de colección a partir de los datos procesados por las operaciones intermedias.

```
ArrayList<Persona> personas = new ArrayList<>();
// Aquí añadiríamos personas a la lista de personas

Stream<Integer> edadesAdultos =
        personas.stream()
                .filter(p -> p.getEdad() >= 18)
                .map(Persona::getEdad)
                .toList();
```

Operaciones finales con streams: cadena

El método collect también permite obtener una cadena de texto que una los elementos resultantes a través de un separador común.

En la función Collectors.joining se puede indicar también un prefijo y un sufijo para el texto.

```
ArrayList<Persona> personas = new ArrayList<>();
// Aquí añadiríamos personas a la lista de personas

Stream<Integer> edadesAdultos =
        personas.stream()
                .filter(p -> p.getEdad() >= 18)
                .map(Persona::getEdad)
                .collect(Collectors.joining(",", "Adultos: ", ""));
```

Operaciones finales con streams: forEach

El método forEach permite recorrer cada elemento del stream resultante, y hacer lo que se necesite con él.

```
ArrayList<Persona> personas = new ArrayList<>();
// Aquí añadiríamos personas a la lista de personas

Stream<Integer> edadesAdultos =
        personas.stream()
                .filter(p -> p.getEdad() >= 18)
                .map(Persona::getEdad)
                .forEach(System.out::println);
                // .forEach(p -> System.out.println(p));
```

Operaciones finales con streams: media

El método average permite, junto a la operación intermedia mapToInt, obtener una media de un stream que haya producido una colección numérica.

```
ArrayList<Persona> personas = new ArrayList<>();
// Aquí añadiríamos personas a la lista de personas

double mediaEdadAdultos =
        personas.stream()
                .filter(p -> p.getEdad() >= 18)
                .mapToInt(Persona::getEdad)
                .average().getAsDouble();
```