Argentina programa 4.0

UNIVERSIDAD TECNOLOGICA NACIONAL

Lambda y aplicación en colecciones

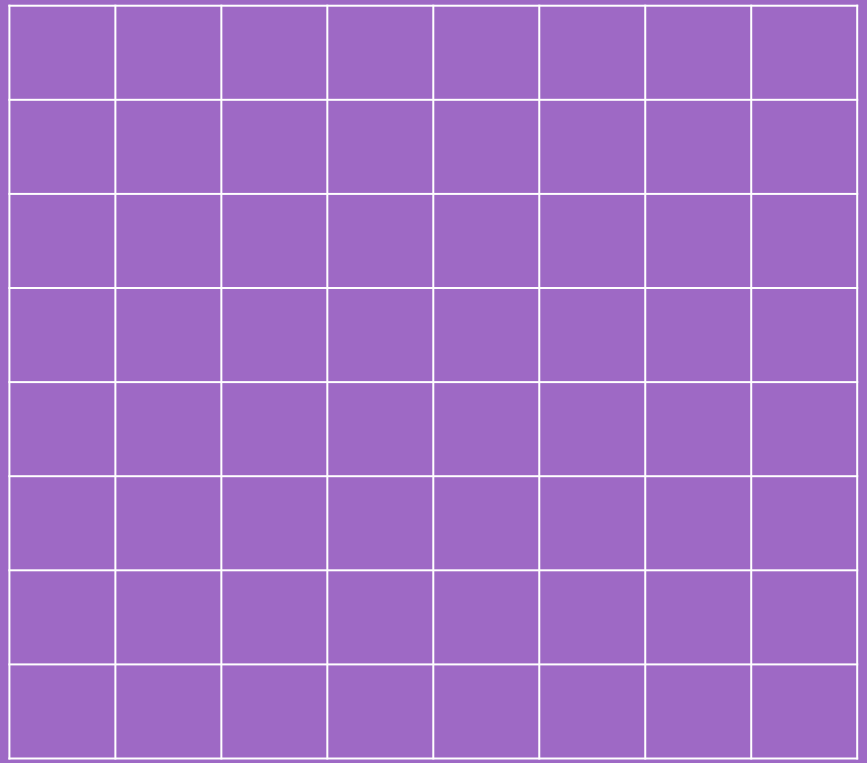
Etapa 2 - Desarrollador Java Intermedio

Agenda

Argentina programa 4.0

- Aplicación de Lambdas en colecciones
  - Introducción
    - Streams
  - forEach
  - anyMatch
  - allMatch
  - map
    - variantes
- Aplanación de Colecciones
  - min
  - max
  - sum
  - joining

Aplicación de Lambdas en colecciones



Introducción

Argentina programa 4.0

Uno de los mayores usos de las expresiones Lambda los encontraremos en la manipulación de colecciones.

Gracias a estas expresiones, el manejo de colecciones es mucho más sencillo, legible y simplificado en código.

En esta clase, veremos algunos de los métodos más útiles, que reciben interfaces funcionales como parámetro, para la manipulación de colecciones:

- forEach
- anyMatch
- allMatch
- map

Introducción - Streams

Argentina programa 4.0

Antes de empezar a ver los métodos mencionados, es importante entender que estos no son métodos propios de las colecciones sino que son definidos por la interfaz Stream.

La interfaz Collection nos proporciona un método stream() que devuelve un Stream secuencial con la colección correspondiente como fuente.

Un Stream envuelve una fuente de datos, lo que nos permite operar con esa fuente de datos y hacer que el procesamiento masivo sea conveniente y rápido.

Es importante aclarar que no almacena datos y, en ese sentido, no es una estructura de datos. Tampoco modifica la fuente de datos a la cuál se está envolviendo.

forEach

El método forEach recibe como único parámetro “action” de tipo Consumer, que es una interfaz funcional. La firma del método abstracto de dicha interfaz define que es de tipo void y recibirá un único parámetro cuyo tipo es el correspondiente a la colección en cuestión.

```
void forEach(Consumer<? super T> action) ;

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

La funcionalidad de forEach será ejecutar la acción, enviada por parámetro, en cada elemento de la colección.

Supongamos que tenemos una colección de enteros y queremos imprimirlos por pantalla:

```
List<Integer> numeros =  new ArrayList<>();

numeros.add(1);
numeros.add(2);
numeros.add(3);

numeros.stream().forEach((numero) -> System.out.println(numero));
```

anyMatch

Argentina programa 4.0

Recibe una interfaz funcional Predicate como parámetro y devuelve un booleano.
Devuelve true si algún elemento de la colección coincide con el predicado proporcionado.

```
boolean anyMatch(Predicate<? super T> predicate)
```

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Siguiendo con el ejemplo del listado de números:

```
List<Integer> numeros =  new ArrayList<>();

numeros.add(1);
numeros.add(2);
numeros.add(3);

boolean coincide = numeros.stream().anyMatch((numero) -> numero > 3);
```

“coincide” será false, ya que ningún número es mayor que 3. En cambio si cambiamos el predicado de la siguiente forma:

```
boolean coincide = numeros.stream().anyMatch((numero) -> numero > 2);
```

“coincide” será true.

allMatch

Argentina programa 4.0

Similar a anyMatch, solo que devolverá true si todos los elementos de la colección coinciden con el predicado. Las firmas son las mismas.

```
boolean anyMatch(Predicate<? super T> predicate)
```

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

```
List<Integer> numeros =  new ArrayList<>();

numeros.add(1);
numeros.add(2);
numeros.add(3);

boolean coincide = numeros.stream().allMatch((numero) -> numero > 1);
```

“coincide” será false, ya que no todos los números son mayores que 1. En cambio si cambiamos el predicado de la siguiente forma:

```
boolean coincide = numeros.stream().allMatch((numero) -> numero > 0);
```

“coincide” será true.

map

Argentina programa 4.0

Permite transformar estructuras de datos.

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Donde ‘T’ es el tipo de dato de la colección original y ‘R’ es el tipo de dato de la colección resultante. La función “mapper”, que recibe por parámetro, define cómo se deben transformar los elementos de la colección original para producir la colección resultante.

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Notar que map devuelve un Stream con el tipo de dato resultante. Para convertir el Stream en una lista se puede utilizar el método collect de Stream pasándole por parámetro el Collector que corresponda para la necesidad.

```
collect(Collectors.toList())
```

map - Algunos ejemplos

Argentina programa 4.0

Definamos una clase Persona que tiene como atributo String nombre. Imaginemos que tenemos un listado de objetos Persona y queremos convertirlo a un listado de Strings en el que cada String sea el nombre de la Persona. Lo podemos resolver:

```
List<Persona> personas =  new ArrayList<>();

personas.add(new Persona("Jose"));
personas.add(new Persona("Maria"));
personas.add(new Persona("Carlos"));

List<String> personasString = personas.stream().map((persona) ->
persona.getNombre()).collect(Collectors.toList());
```

También podríamos a partir de un listado de nombres generar un listado de objetos Persona:

```
List<Persona> otroListadoDepersonas = personasString.stream().map(persona -> new
Persona(persona)).collect(Collectors.toList());
```

map - Algunos ejemplos

Argentina programa 4.0

Otro caso de uso que es válido, es generar una nueva colección del mismo tipo pero con alguna modificación aplicada para todos los elementos. Por ejemplo, dada una colección de números, generar una nueva colección que tenga cada número de la colección original duplicado.

```
List<Integer> numeros =  new ArrayList<>();

numeros.add(1);
numeros.add(2);
numeros.add(3);

List<Integer> numerosDuplicados =
numeros.stream().map(numero -> numero * 2).collect(Collectors.toList());
```

La nueva colección “numerosDuplicados” tendrá como números el 2, 4 y 6.

map - Otras variantes

Argentina programa 4.0

El método map tiene otras variantes similares:

- mapToInt
- mapToDouble
- mapToLong

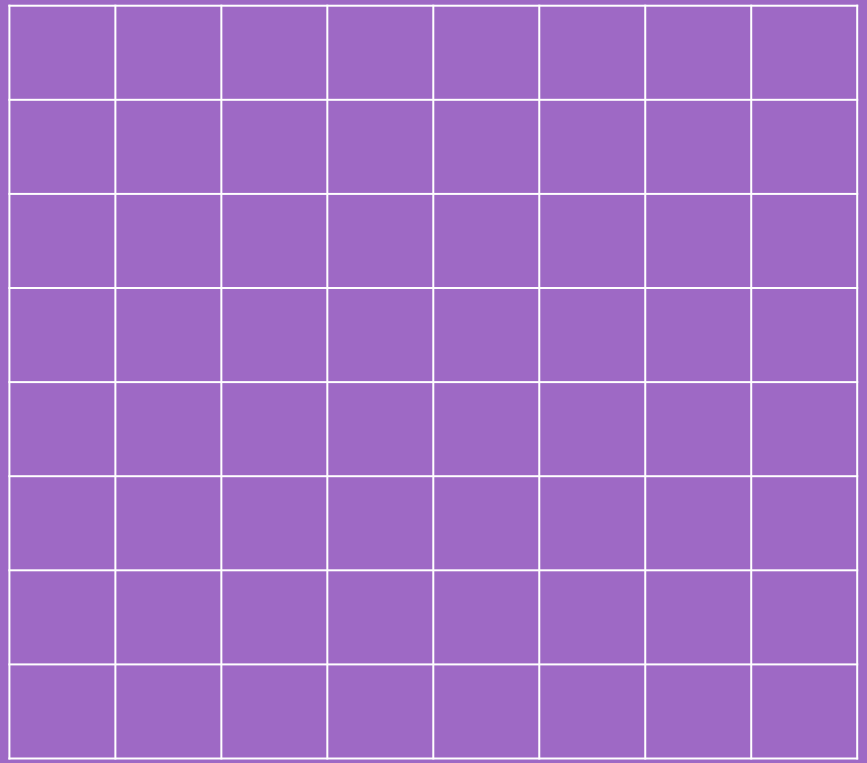
Todos son similares al map explicado, únicamente que la nueva colección será de Integer, Double o Long respectivamente. (Originalmente devolverán Streams de tipos primitivos. Para poder obtener una colección de las mencionadas se hace uso del método boxed() que convierte el Stream de tipo primitivo a el Stream del objeto correspondiente)

```
List<String> numerosString =  new ArrayList<>();

numerosString.add("1");
numerosString.add("2");
numerosString.add("3");

List<String> numeros = numerosString.stream().mapToInt(numero ->
Integer.valueOf(numero)).boxed().collect(Collectors.toList());
```

Aplanación de colecciones



Introducción

Argentina programa 4.0

La aplanación de colecciones es una operación que se utiliza en la programación funcional para procesar y reducir colecciones de objetos en valores simples. Algunas operaciones de aplanación comunes en Java que estaremos viendo son:

- Collections.min()
- Collections.max()
- IntStream.sum()
- Collectors.joining()

Collections.min()

Argentina programa 4.0

```
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)
```

Devuelve el elemento mínimo de la colección dada, según el orden natural de sus elementos. Todos los elementos de la colección deben implementar la interfaz Comparable. Además, todos los elementos de la colección deben ser comparables entre sí.

```
List<Integer> numeros =  new ArrayList<>();

numeros.add(4);
numeros.add(1);
numeros.add(8);

Integer resultado = Collections.min(numeros);
```

La variable resultado tendrá como valor 1.

Collections.max()

Argentina programa 4.0

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

A la inversa de Collections.min(), devolverá el elemento máximo.

```
List<Integer> numeros =  new ArrayList<>();

numeros.add(4);
numeros.add(1);
numeros.add(8);

Integer resultado = Collections.max(numeros);
```

La variable resultado tendrá como valor 8.

IntStream.sum()

Argentina programa 4.0

```
int sum();
```

Devuelve la suma de los elementos en el Stream.

```
List<Integer> numeros =  new ArrayList<>();

numeros.add(4);
numeros.add(1);
numeros.add(8);

int suma = numeros.stream().mapToInt(Integer::intValue).sum();
```

La variable suma tendrá como valor 13.

Collectors.joining()

```
public static Collector<CharSequence, ?, String> joining()
```

Como vimos anteriormente, es posible transformar un Stream en una lista. También es posible transformar un Stream de tipo String en un único String con todos los elementos del Stream. Esto se puede resolver pasándole al método collect de Stream, pasándole como parámetro el Collector que nos devuelve joining().

```
List<String> textos =  new ArrayList<>();

textos.add("Primero");
textos.add("Segundo");
textos.add("Tercero");

String textoResultante = textos.stream().collect(Collectors.joining());
```

La variable textoResultante tendrá como valor un String "PrimeroSegundoTercero". El método joining se encuentra sobrecargado de tal forma que también admite como parámetro un separador que se agregará entre cada elemento del Stream.

```
public static Collector<CharSequence, ?, String> joining(CharSequence delimiter)
```

Por lo que para obtener un String "Primero, Segundo, Tercero" podríamos cambiar el llamado a joining como:

```
String textoResultante = textos.stream().collect(Collectors.joining(  ", "));
```

Referencias

Argentina programa 4.0

- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
- https://stackify.com/streams-guide-java-8/
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

Gracias!