Programare avansata pe obiecte - laborator 9 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

Object oriented programming - lab 9

Streams:

- Java.util.stream - contains classes for processing sequences of elements. The central API class is the Stream<T>.

Stream Creation:

- Streams can be created from different element sources e.g. collection or array with the help of stream() and of() methods;
- A stream() default method is added to the Collection interface and allows creating a Stream<T> using any collection as an element source;

```
package com.paolabs.lab9.ex1;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class StreamCreationExample {

   public static void main(String[] args) {
       // stream creation with stream()
       String[] movies = new String[]{"Extraction", "Knives Out", "Sergio"};
       Stream<String> stream = Arrays.stream(movies);

       // stream creation with stream()
       Stream<String> anotherStream = Stream.of("Extraction", "Knives Out", "Sergio");

       // creating a stream using any collection as an element source
       List<String> actors = Arrays.asList("Robert De Niro", "Jack Nicholson", "Denzel Washington");
```

```java
        Stream<String> actorsStream = actors.stream();
    }
}
```

Stream Operations:

- **intermediate operations** (return *Stream<T>*) and **terminal operations** (return a result of definite type).
- Intermediate operations allow chaining.
- Operations on streams don't change the source.

```java
package com.paolabs.lab9.ex1;

import java.util.Arrays;
import java.util.List;

public class StreamOperationsExample {

    public static void main(String[] args) {
        List<String> actors = Arrays.asList("Robert De Niro", "Jack Nicholson", "Denzel Washington", "Robert De Niro");

        // stream operations
        long count = actors.stream().distinct().count();

        System.out.println("No of distinct elements: " + count);
    }
}
```

- the distinct() method represents an intermediate operation, which creates a new stream of unique elements of the previous stream. And the count() method is a terminal operation, which returns the stream's size.

Stream Iterating:

- Stream API helps to substitute for, for-each and while loops.
- It allows concentrating on operation's logic, but not on the iteration over the sequence of elements.

```
package com.paolabs.lab9.ex1;

import java.util.Arrays;
import java.util.List;

public class StreamIteratingExample {

   public static void main(String[] args) {
       List<String> actors = Arrays.asList("Robert De Niro", "Jack
Nicholson", "Denzel Washington", "Robert De Niro");

       // iterating
       boolean exists = actors.stream().anyMatch(actor ->
actor.equals("Jack Nicholson"));
       System.out.println("Jack Nicholson is in the list of actors? " +
(exists ? "Yes" :"No"));
   }
}
```

Filtering:

- The filter() method allows us to pick a stream of elements which satisfy a predicate.

```
package com.paolabs.lab9.ex1;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class StreamFilteringExample {

   public static void main(String[] args) {
       List<String> actors = Arrays.asList("Robert De Niro", "Jack
Nicholson", "Denzel Washington", "Samuel L. Jackson");

       // filtering
       Stream<String> result = actors.stream().filter(actor ->
actor.contains("Jack"));
   }
}
```

Mapping:

- map() method - convert elements of a Stream by applying a special function to them and to collect these new elements into a Stream;
- If you have a stream where every element contains its own sequence of elements and you want to create a stream of these inner elements, you should use the flatMap() method;

```
package com.paolabs.lab9.ex1;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class StreamMappingExample {

   public static void main(String[] args) {
       //map
       List<String> actors = Arrays.asList("Robert De Niro", "Jack
Nicholson", "Denzel Washington", "Samuel L. Jackson");

       Stream<String> result = actors.stream().map(actor ->
actor.toUpperCase());
       result.forEach(System.out::println);

       // flatMap
       List<String> actorsInMovie1 = Arrays.asList("Robert De Niro", "Jack
Nicholson");
       List<String> actorsInMovie2 = Arrays.asList("Denzel Washington",
"Samuel L. Jackson");

       List<List<String>> actorsInMovies = Arrays.asList(actorsInMovie1,
actorsInMovie2);

       Stream<String> allActors = actorsInMovies.stream().flatMap(strings
-> strings.stream());
       allActors.forEach(System.out::println);
   }
}
```

Matching:

- Stream API gives a handy set of instruments to validate elements of a sequence according to some predicate: anyMatch(), allMatch(), noneMatch().
- Those are terminal operations which return a boolean.

```java
package com.paolabs.lab9.ex1;

import java.util.Arrays;
import java.util.List;

public class StreamMatchingExample {

   public static void main(String[] args) {
       List<String> actors = Arrays.asList("Robert De Niro", "Jack Nicholson", "Denzel Washington", "Robert De Niro");

       boolean anyMatch = actors.stream().anyMatch(actor -> actor.contains("Robert"));
       boolean allMatch = actors.stream().allMatch(actor -> actor.contains("Robert"));
       boolean noneMatch = actors.stream().noneMatch(actor -> actor.contains("Robert"));

       System.out.println(anyMatch);
       System.out.println(allMatch);
       System.out.println(noneMatch);
   }
}
```

Reduction:

- Stream API allows reducing a sequence of elements to some value according to a specified function with the help of the reduce() method of the type Stream.
- This method takes two parameters: first -> **start value**, second -> **an accumulator function.**

```java
package com.paolabs.lab9.ex1;

import java.util.Arrays;
import java.util.List;
```

```
public class StreamReductionExample {

   public static void main(String[] args) {
       List<Integer> integers = Arrays.asList(1, 1, 1);

       Integer reduced = integers.stream().reduce(10, (a, b) -> a + b);

       System.out.println(reduced);
   }
}
```

Collecting:

- The reduction can also be provided by the collect() method of type Stream.
- This operation is very handy in case of converting a stream to a Collection or a Map and representing a stream in form of a single string.
- There is a utility class Collectors which provide a solution for almost all typical collecting operations.

```
package com.paolabs.lab9.ex1;

import org.w3c.dom.ls.LSOutput;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamCollectingExample {

   public static void main(String[] args) {
       List<String> actors = Arrays.asList("Robert De Niro", "Jack
Nicholson", "Denzel Washington", "Samuel L. Jackson");

       List<String> resultList
               = actors.stream()
               .map(element -> element.toUpperCase())
               .collect(Collectors.toList());

       resultList.forEach(System.out::println);
   }
}
```

Empty Stream:

- The empty() method should be used in case of a creation of an empty stream;
- Its often the case that the empty() method is used upon creation to avoid returning null for streams with no element;

```
package com.paolabs.lab9.ex2;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class EmptyStreamExample {

   public static void main(String[] args) {
       Stream<String> streamEmpty = Stream.empty();

       Stream<String> movies = streamOf(Arrays.asList("Robert De Niro",
"Jack Nicholson", "Denzel Washington", "Samuel L. Jackson"));
       Stream<String> emptyStream = streamOf(null);
   }

   public static Stream<String> streamOf(List<String> list) {
       return list == null || list.isEmpty() ? Stream.empty() :
list.stream();
   }
}
```

Stream.builder():

- When builder is used the desired type should be additionally specified in the right part of the statement, otherwise the build() method will create an instance of the Stream<Object>;

```
package com.paolabs.lab9.ex2;

import java.util.stream.Stream;

public class StreamBuilderExample {

   public static void main(String[] args) {
       Stream<String> actors = Stream.<String>builder()
```

```
                .add("Robert De Niro")
                .add("Jack Nicholson")
                .add("Denzel Washington")
                .build();
   }
}
```

Stream.generate():

- The generate() method accepts a Supplier<T> for element generation.
- As the resulting stream is infinite, we should specify the desired size or the generate() method will work until it reaches the memory limit;

```
package com.paolabs.lab9.ex2;

import java.util.stream.Stream;

public class StreamGenerateExample {

   public static void main(String[] args) {
       // creates a sequence of ten strings with the value - "element".
       Stream<String> streamGenerated =
               Stream.generate(() -> "element").limit(10);
   }
}
```

Stream.iterate():

- Another way of creating an infinite stream is by using the iterate() method;

```
package com.paolabs.lab9.ex2;
import java.util.stream.Stream;

public class StreamIterateExample {

   public static void main(String[] args) {
       Stream<Integer> streamIterated = Stream.iterate(40, n -> n +
2).limit(20);

       streamIterated.forEach(System.out::println);
   }
}

```

Stream of Primitives:

- Since version 8 Java offers a possibility to create streams out of three primitive types: int, long and double.
- As Stream<T> is a generic interface and there is no way to use primitives as a type parameter with generics, three new special interfaces were created: **IntStream, LongStream, DoubleStream.**

```
package com.paolabs.lab9.ex2;

import java.util.Random;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;

public class StreamOfPrimitivesExample {

   public static void main(String[] args) {
       IntStream intStream = IntStream.range(1, 3);
       intStream.forEach(i -> System.out.print(i + " "));

       System.out.println();

       LongStream longStream = LongStream.rangeClosed(1, 3);
       longStream.forEach(l -> System.out.print(l + " "));

       System.out.println();

       Random random = new Random();
       DoubleStream doubleStream = random.doubles(3);
       doubleStream.forEach(d -> System.out.print(d + " "));
   }
}
```

Stream of File:

- NIO class Files allows one to generate a Stream<String> of a text file through the lines() method. Every line of the text becomes an element of the stream:

```
package com.paolabs.lab9.ex2;

import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class StreamOfFileExample {

   public static void main(String[] args) throws IOException {
       Path path = Paths.get("src/resources/input.txt");

       Stream<String> streamOfStrings = Files.lines(path);
       streamOfStrings.forEach(System.out::println);


       Stream<String> streamWithCharset = Files.lines(path,
StandardCharsets.UTF_8);
       streamWithCharset.forEach(System.out::println);
   }
}
```