Java SE & Spring

Module 1: Java SE

11.Lambda Expressions, Functional Interfaces & Stream API

What is Lambda Expression?

- Lambda expressions are nameless functions given as constant values, and written exactly in the place where it's needed, typically as a parameter to some other function.
- Lambda expressions provide many benefits to functional programming over object-oriented programming (OOP). Most OOP languages evolve around classes and objects and these languages treat only the classes as their first-class citizens. The other important entity, i.e. functions, takes the back seat.
- The lambda expression was introduced first time in Java 8. Its main objective to increase the expressive power of the language. But, before getting into lambdas, we first need to understand functional interfaces.

Further reading: https://howtodoinjava.com/java8/lambda-expressions/

Functional Interface

If a Java interface contains one and only one abstract method then it is termed as functional interface. This only one method specifies the intended purpose of the interface.

For example, the Runnable interface from package java.lang; is a functional interface because it constitutes only one method i.e. run().

Define Functional Interface Example:

```java
@FunctionalInterface
public interface CalculateSquare<T extends Number> {
    public T getSquare(T number);
    default void test(){
        System.out.println("This is a default method in a functional interface.\n  " +
                "In functional interfaces, only one abstract method is allowed.");
    }
    //public void testMethod();
}
```

Functional Interface

In the example in previous slide, the interface **CalculateSquare** has only one abstract method getSquare(). Hence, it is a functional interface. Here, we have used the annotation @FunctionalInterface. The annotation forces the Java compiler to indicate that the interface is a functional interface. Hence, does not allow to have more than one abstract method.

Lambda Expressions

Lambda expression is, essentially, an anonymous or unnamed method. The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.

We can define a lambda expression like below.

Syntax:

```
(parameter list) -> {lambda body}
```

The new operator (->) used is known as an arrow operator or a lambda operator.

Suppose we have a method like below:

```
public int calculateCube(int number){
   return number * number * number;
}
```

We can write equivalent of this method with lambda expression like this:

```
(number) -> {number * number * number};
```

Lambda Expressions

- If there is only one parameter on parameter list we can omit braces for the sake of simplicity.

```
(number) -> lambda body
// we can omit braces
number -> lambda body
```

- If there is only one statement in the lambda body, then we can omit curly braces. Otherwise, if there are multiple statements, the body must enclosed with curly brackets. Every statements must end with a semicolon.

```
number -> {
      number += 3;
      System.out.println(number);}
```

- If the lambda expression has no parameter, then open and close brackets with empty parameter list.

```
() -> 3.141592 // equivalent to return 3.141592;
```

Stream API

- A Stream in Java 8 can be defined as a sequence of elements from a source. Streams supports aggregate operations on the elements. The source of elements here refers to a Collection or Array that provides data to the Stream.
- Stream keeps the ordering of the elements the same as the ordering in the source. The aggregate operations are operations that allow us to express common manipulations on stream elements quickly and clearly.
- Java 8 Streams are designed in such a way that most of the stream operations return a stream. This helps us to create a chain of stream operations. This is called as pipe-lining.

Further reading: https://howtodoinjava.com/java8/java-streams-by-examples/

Stream API

- In Java, java.util.Stream represents a stream on which one or more operations can be performed. Stream operations are either intermediate or terminal.
- The terminal operations return a result of a certain type and intermediate operations return the stream itself so we can chain multiple methods in a row to perform the operation in multiple steps.
- Streams are created on a source, e.g. a java.util.Collection like List or Set. The Map is not supported directly, we can create stream of map keys, values or entries.
- Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them.

Creating Streams

Example-1

```java
public class StreamBuilders
{
      public static void main(String[] args)
      {
           Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
           stream.forEach(p -> System.out.println(p));
      }
}
```

Example-2

```java
public class StreamBuilders
{
      public static void main(String[] args)
      {
           Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );
           stream.forEach(p -> System.out.println(p));
      }
}
```

Creating Streams

We can also create a streams from Collections. To do that, we can use .stream() method of the related collection object.

```java
List<Integer> list =  new ArrayList<Integer>();

for(int i = 1; i< 10; i++){
   list.add(i);
}

Stream<Integer> stream = list.stream();
stream.forEach(p -> System.out.println(p));
```

Stream Intermediate Operations

The methods below are commonly used intermediate operations in streams.

- **Stream filter()**: We can use filter() method to test stream elements for a condition and generate filtered list. The filter() method accepts a **Predicate** to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation on the result.
- **Stream map()**: The map() intermediate operation converts each element in the stream into another object via the given function.
- **Stream sorted()**: The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator. note that the sorted() method only creates a sorted view of the stream without manipulating the ordering of the source Collection.

Stream Terminal Operations

The methods below are commonly used terminal operations in streams.

- **Stream foreach**: The forEach() method helps in iterating over all elements of a stream and perform some operation on each of them. The operation to be performed is passed as the lambda expression.
- **Stream collect()**: The collect() method is used to receive elements from a steam and store them in a collection.
- **Stream match()**: Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.
- **Stream count()**: The count() is a terminal operation returning the number of elements in the stream as a long value.

Questions ?

Next:Exception Handling