Java 8 Features

Presented by

Sagar
Java Consultant

1. Lambda Expressions

- A Java lambda expression is a function which can be created without belonging to any class.
- A Lambda expression (or function) is just an anonymous function.
- They are written exactly in the place where it's needed.
- Java lambda expression is consisted of three components.

  1) Argument-list: It can be empty or non-empty as well.
  2) Arrow-token: It is used to link arguments-list and body of expression.
  3) Body: It contains expressions and statements for lambda expression.

- Ex1: lambda operator -> function body

  () -> System.out.println("Zero parameter lambda")

Lambda Expressions . . .

- Lambda Expression can be defined as an Anonymous Function that allows users to pass methods as arguments.
- Lambda Functions have no access modifiers(private, public or protected), no return type declaration and no name.
- Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only.

Traditional way of implementating Interface

```
public interface Greetings {

        public void sayHello(String name);

}
```

```
public class JavaTraditionalClass implements Greetings {

          @Override
          public void sayHello(String name) {
                   System.out.println("Hello World " + name);

          }

public static void main(String[] args) {
                  JavaTraditionalClass test = new JavaTraditionalClass();
                  greet(test, "Tom");

         }

private static void greet(Greetings greetingsInstance, String name) {
                  greetingsInstance.sayHello(name);
         }

}
```

Lambda way of implementing Interface

```
public interface Greetings {

        public void sayHello(String name);

}
```

```
public class JavaLambda {

public static void main(String[] args) {

                Greetings greetingsInstance=n -> System.out.println("Hello
                                                        World " + n);
                greet(greetingsInstance, "Tom");
        }

private static void greet(Greetings greetingsInstance, String name) {
                greetingsInstance.sayHello(name);
        }

}
```

Lambda Expressions …

- Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only.

  - Ex:
  - MathOperator sum = (int num1, int num2) -> num1 + num2; //define
  - ……
  - System.out.println("Sum = " +sum.operation(12, 23)); // invoke

- MathOperator is an Interface. Operation is a two argument abstract method.

2. Functional Interface

- @FunctionalInterface is a new interface added in Java 8.
- Functional interfaces have a single functionality to exhibit.

```
@FunctionalInterface
public interface Greetings {

        public void sayHello(String name);

}
```

- Annotation is optional. Even if not annotated with @FunctionalInterface, an interface can still be used as a functional interface.
- Interface should have only a single method.

Functional Interfaces…

- Functional interfaces are also called Single Abstract Method interfaces (SAM Interfaces).
- As name suggest, they permit exactly one abstract method inside them.
- Another important point to remember is that if an interface declares an abstract method overriding one of the public methods of java.lang.Object
- It also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere

Functional Interface… Example

```
@FunctionalInterface
public interface MyFirstFunctionalInterface
{
    public void firstWork();

    @Override
    public String toString();                    //Overridden from Object class

    @Override
    public boolean equals(Object obj);           //Overridden from Object class
}
```

3. Method Reference – (: :) operator

- Method reference is a shorthand notation of a lambda expression to call a method
- The :: operator is used in method reference to separate the class or object from the method name
- Ex: Lambda Expression:
  - str -> System.out.println(str)
  - Method Reference:
    - System.out::println(str)

Anonymous class vs lambda vs method reference

Anonymous class to print a list.

```
List<String> list = Arrays.asList("node", "java", "python", "ruby");
list.forEach(new Consumer<String>() {       // anonymous class
    @Override
    public void accept(String str) {
        System.out.println(str);
    }
});
```

Anonymous class vs lambda vs method reference …

Anonymous class -> Lambda expressions.

```
List<String> list = Arrays.asList("node", "java", "python", "ruby");
list.forEach(str -> System.out.println(str)); // lambda
```

Anonymous class vs lambda vs method reference …

Lambda expressions -> Method references.

```
List<String> list = Arrays.asList("node", "java", "python", "ruby");
list.forEach(System.out::println);          // method references
```

Example;

Types of method references

1. Reference to a static method - ClassName::staticMethodName

Ex:

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

public class MethodReferenceStaticMethod {

    public static void main(String[] args) {
        List<String> list = Arrays.asList("Java", "Python", "MySQL", "Angular");

        // anonymous class
        System.out.println("Using ananymous class");
        list.forEach(new Consumer<String>() {
            @Override
            public void accept(String str) {
                SimplePrinter.print(str);
            }
        });

        // lambda expression
        System.out.println("Using lamba expressions");
        list.forEach(str -> SimplePrinter.print(str));

        // method reference
        System.out.println("Using method references");
        list.forEach(SimplePrinter::print);
    }
}
```

Types of method references …

2A. Reference to an instance method of a particular object:

Instance methods can also be referenced.

Ex:

```
public void saySomething(){
    System.out.println("Hello, this is non-static method.");
  }
  public static void main(String[] args) {
    InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
    // Referring non-static method using reference
      Sayable sayable = methodReference :: saySomething;
```

Types of method references …

2B. Reference to an instance method of a particular object – Built-in class object:

Ex:

```
public class InstanceMethodReferenceThread {

    public void printnMsg(){
        System.out.println("Hello, this is instance method");
    }
    public static void main(String[] args) {
    Thread t2=new Thread(new InstanceMethodReferenceThread()::printnMsg);
        t2.start();
    }
}
```

Types of method references …

3. Reference to a Constructor:

A method can be referred by using new keyword.

Syntax :

```
ClassName::new
```

Example :

```
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message :: new;
        hello.getMessage("Hello World");
    }
}
```

4A. Static mehods

Static Methods in **Interface** are those methods, which are defined in the interface with the keyword static.

Unlike other methods in Interface, these static methods contain the complete definition of the function.

These methods cannot be overridden in Implementation Classes

Ex:

```
interface MyInterface {
        static void hello() {
                System.out.println("Hello, New Static Method");
        }
}
public class MyClass {
        public static void main(String args[]) {
                MyInterface.hello();
        }
}
```

4B. Default methods

```java
public interface MyInterface {

    // abstract method
    public void square(int side);

    // default method
    default void show() {
      System.out.println("Default Method Executed");
    }
}
```

```java
public class MyInterfaceImpl implements MyInterface{
  // implementation of square abstract method
  public void square(int side) {
     System.out.println("Area of square = " + side * side);
  }
}
```

```java
public class MainApp {
    public static void main(String args[])  {
        MyInterface myimpl = new MyInterfaceImpl();
       myimpl.square(4);

        // default method executed
        myimpl.show();
    }
}
```

5. Stream API

Streams : A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements.

Using stream, data is processed in a declarative way.

Ex: For a Collection of strings
- Filtering (Conditions)
- Converting into uppercase
- Sorting
- Traversing

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Streams are wrappers around a data source, allows to operate with that data source and making bulk processing convenient and fast.

Streams Features

Sequence of elements :

A stream provides a set of elements of specific type in a sequential manner.

A stream gets/computes elements on demand. It never stores the elements.

Source :

Stream takes Collections, Arrays, or I/O resources as input source.

Aggregate operations :

Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

Automatic iterations:

Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Lazy nature :

All the stream operations are lazy in nature which means they are not executed until they are needed. For example, if we want to display only the first 2 elements of a list using stream, the stream operation would stop at the end of second iteration after displaying the second element of list.

A. Stream's of(values) method

```
import java.util.stream.Stream;

public class MainValuesStream {

    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
        // iterates with single forEach() method no classic for loop
        stream.forEach(value -> System.out.println(value));
    }

}
```

B. Stream's of(arrayElements) method

```
import java.util.stream.Stream;

public class MainIntArrayStream {
   public static void main(String[] args) {

      Stream<Integer> stream = Stream.of(  new Integer[]  { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  );
      stream.forEach(element -> System.out.println(element));
   }
}
```

C. List's stream() method

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class MainListStream {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();

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

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

D. Stream's generate() mehtod

```
import java.util.Random;
import java.util.stream.Stream;

public class MainStreamDotGenerate {

   public static void main(String[] args) {

      System.out.println("Random numbers");
      Stream.generate(Math :: random)
        .limit(5)
        .forEach(System.out :: println);
      System.out.println("Random Integers");
      Stream.generate(new Random()::nextInt)
      .limit(5).forEach(System.out :: println);
      }
   }
```

E. Stream's iterate() method

```
import java.util.stream.Stream;

public class MainStreamDotIterate {

   public static void main(String[] args) {
      // Shows all multiples of 3 - no condition to stop - method to control the iteration
      // no for loop , no condition on no of iterations
       Stream.iterate(1, count->count+1)
          .filter(number->number%3==0)
          .limit(6)
          .forEach(System.out :: println);

   }

}
```

F. Stream’s anyMatch() method

The anyMatch() method returns true if “any of the elements” of the Stream matches the given predicate.

Predicate is a class that is associated with a condition

Ex: Predicate<Student> p1 = s -> s.stuName.startsWith("S");

where stuName is an attribute of a class Student.

This predicate stores true / false if a student name starts with S

boolean anymatch = list.stream().anyMatch(p1);

The Boolean value is assigned to anymatch variable.

Example

G. Stream’s allMatch() method

The allMatch() method returns true if “all of the elements” of the Stream matches the given predicate.

Predicate is a class that is associated with a condition

Ex: Predicate<Student> p1 = s -> s.name.startsWith("S");

where name is an attribute of a class Student.

This predicate stores true / false if a student name starts with S

boolean allmatch = list.stream().allMatch(p1);

The Boolean value is assigned to allmatch variable.

Example

H. Stream’s noneMatch() method

The stream noneMatch() method works just opposite to the anyMatch() method

It returns true if none of the stream elements match the given predicate.

It returns false if any of the stream elements matches the condition specified by the predicate.

Ex: Predicate<Student> p1 = s -> s.name.startsWith("S");

where name is an attribute of a class Student.

This predicate stores true / false if a student name starts with S

boolean nonematch = list.stream().noneMatch(p1);

The Boolean value is assigned to nonematch variable.

Example

I. filter() and collect()

The filter() method is to filter the collection.

The collect() method is to convert stream into a List.

Stream's collect() method is to collect the result of “stream processing” into a List, Set, and Map.

The result of a stream processing pipeline in a list is collected by Collectors.toList() method

In other words, it converts a given Stream into List, Set, and Map.

Ex:

```
List<String> listOfStringStartsWithJ
 = listOfString
      .stream()
      .filter( s -> s.startsWith("J"))
      .collect(Collectors.toList());
```

Example

filter(), collect() - Example

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

public class MainCollectionFilterAfter {

    public static void main(String[] args) {

        List<String> technologies = Arrays.asList("Spring", "Java", "Python");
        System.out.println("Using List");
        List<String> result = technologies.stream().filter(java -> !"Java".equals(java)).collect(Collectors.toList());

        result.forEach(System.out::println);
        System.out.println("Using Set");
        Set<String> technologiesSet = (Set<String>) technologies.stream().filter(java -> !"Java".equals(java))
                .collect(Collectors.toSet());

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

}
```

7A.forEach() for collections & streams - list

Java 8 newly introduced forEach() method to iterate over collections and streams.

This method takes a single parameter which is a functional interface.

A lambda expression is passed as an argument.

```
List<String> cities = new ArrayList<String>();
cities.add("New York");
cities.add("Paris");
cities.add("Bengaluru");
cities.add("Hyderabad");

// lambda expression in forEach Method - no boilerplate code
cities.forEach(str -> System.out.println("City name - " + str));
```

Example

7B.forEach() for collections & streams – set

```
public class MainForEachSet {

    public static void main(String[] args) {
        Set<String> citySet = new HashSet<String>();
        citySet.add("New York");
        citySet.add("Paris");
        citySet.add("Bengaluru");
        citySet.add("Hyderabad");

        // lambda expression in forEach Method - no boilerplate code
        citySet.forEach(str -> System.out.println("City name - " + str));


    }

}
```

7C.forEach() for collections & streams-map

```
public static void main(String[] args) {
    Map<Integer, String> productMap = new HashMap<Integer, String>();
     productMap.put(1001, "Refrigerator");
     productMap.put(1002, "Washing Machine");
     productMap.put(1003, "Woven");
     productMap.put(1004, "Television");

     // forEach to iterate and display each key and value pair of HashMap.

     productMap.forEach((key,value)->System.out.println(key+" - "+value));
     // forEach to iterate a Map and display the value of a particular   key

     productMap.forEach((key,value)->{
       if(key == 1002){
         System.out.println("Value associated with key 1002 is: "+value);
       }
     });
     // forEach to iterate a Map and display the key associated with a particular value

     productMap.forEach((key,value)->{
       if("Television".equals(value)){
         System.out.println("Key associated with Value Television is: "+key);
       }
     });

}
```

Example

8. StringJoiner class

StringJoiner is introduced in the java.util package.

Joining more than one string with the specified delimiter is done using this class.

Prefix and suffix to the final string while joining multiple strings can be added.

```
// "-" is delimiter, "[" is prefix and "]" is suffix
StringJoiner cities = new StringJoiner("-","[","]");
cities.add("New York");
cities.add("Paris");
cities.add("Bengaluru");
cities.add("Hyderabad");

System.out.println(cities);
```

Output:

```
[New York-Paris-Bengaluru-Hyderabad]
```

setEmptyValue and toString()

```
StringJoiner cities = new StringJoiner("-","[","]");

cities.setEmptyValue("This is a default string!");

System.out.println(cities);


cities.add("New York");
cities.add("Paris");
cities.add("Bengaluru");
cities.add("Hyderabad");
System.out.println("String Joiner One");
System.out.println(cities);
System.out.println(cities.toString());
```

Example

9. Collectors Class

- Collectors is a final class that extends Object class.
- As the name suggests, Collectors class is used to collect elements of a Stream into Collection.
- It acts as a bridge between Stream and Collection, and you can use it to convert a Stream into different types of collections like List, Set, Map.
- It even provides functionalities to join String, group by, partition by and several other reduction operators to return a meaningful result.
- It's often used along with collect() method of Stream class which accepts a Collectors.

Example

Collectors Class . . .

- Methods of Collectors class:
- Collectors.toSet()
- Collectors.toList()
- Collectors.toCollection()
- Collectors.toMap()
- Collectors.joining()
- Collectors.summaryStatistics()
- Collectors.groupingBy()
- Collectors.counting()
- Collectors.toConcurrentMap()
- Collectors.partition()

Collectors Class . . .

1. Collectors.toSet()

- This method is used to collect the result of a Stream into Set, or in other words, you can use this to convert a Stream to a Set.
- For example, if there is a stream of numbers which also contains duplicates.
- In order to convert them into Set, then we can use the following code:
- **Ex:**

```
Set<Integer> numbersWithoutDups =
        numbers.stream().collect(Collectors.toSet());
```

Collectors Class . . .

2. Collectors.toList() Example

- This method is very similar to the toSet() method of java.util.stream.Collectors class
- Instead of collecting elements into a Set it collects into a List.

Ex:

```
List<Integer> numbersWithDups =
                    numbers.stream().collect(Collectors.toList());
```

Collectors Class . . .

3. Collectors.toCollection()

- This method converts a Stream into any Collection class, e.g. ArrayList, HashSet, TreeSet, LinkedHashSet, Vector, PriorityQueue etc.

- **Ex: ArrayList<Integer> anArrayList = numbers.stream()**
  .collect(Collectors.toCollection(ArrayList::new));

Collectors Class . . .

4. Collectors.toMap()

- The Collectors class also provide a utility method to create Map from the elements of Stream

- **Ex:**

```
Map<Integer, String> intToString = numbersWithoutDups.stream()

.collect(Collectors.toMap(Function.identity(),String::valueOf));
```

Collectors Class . . .

5. Collectors.toConcurrentMap()

The Collectors class also provide a toConcurrentMap() function which can be used to convert a normal or parallel stream to a ConcurrentMap. Its usage is similar to the toMap() method.

Ex:

```
ConcurrentMap<Integer, String> concurrentIntToString
        = numbersWithoutDups.parallelStream()
        .collect(Collectors.toConcurrentMap(Function.identity(),
                String::valueOf));
```

Collectors Class …

6. Collectors.joining()

This method can be used to join all elements of Stream in a single Stream where parts are separated by a delimiter.

For example, if you want to create a long, comma-separated String from all the elements of Stream, then you can do so by using the Collectors.joining() method.

Ex:

```
String csv = numbers.stream()
                .map(String::valueOf)
                .collect(Collectors.joining(", "));
```

Collectors Class …

7. Collectors.summaryStatistics()

- This method can be used to find summary statistics from a Stream of numbers
- For example, if a Stream contains integers, then calculating the average,sum,min,max values of all the integers can be done by using IntSummaryStatistics object

Ex:

```
IntSummaryStatistics summary = numbers.stream()
        .collect(Collectors.summarizingInt(Integer::valueOf));
 double average = summary.getAverage();
 int maximum = summary.getMax();
int minimum = summary.getMin();
```

Collectors Class …

8. Collectors.groupBy()

- This method is like the group by clause of SQL, which can group data on some parameter.
- This method is like the group by clause of SQL, which can group data on some parameter. This way, you can convert a Stream to Map, where each entry is a group

Ex:

```
Stream<Locale> streamOfLocales =
                    Stream.of(Locale.getAvailableLocales());

Map<String, List<Locale>> countryToLocale = streamOfLocales
        .collect(Collectors.groupingBy(Locale::getCountry));
```

Collectors Class …

9. Collectors.partition()

The partitionBy() method can be used to partition the result of Stream in two parts, e.g. pass or fail. The Collectors.partitionBy() accept a Predicate and then partition all the elements into two categories, elements which pass the condition and elements which don't pass. The result is a Map of a List.

Ex:

```
Map<Boolean, List<Integer>> evenAndOddNumbers = numbers.stream()
        .collect(Collectors.partitioningBy(number -> number%2 ==0 ));
```

Collectors Class …

10. Collectors.counting()

- Use this method to count how many elements are going to collect or how many elements are present in the stream after processing.

Ex:

```
long count = numbers.stream()
        .filter( number -> number> 10)
        .collect(Collectors.counting());
```


return
THANK
YOU
alt
control