Lambda Expression Stream API

Lambda expressions

- Lambda expressions are block of code without a name
- Lambda expression Implementation code can be provided when it is called
- It is a representation of anonymous class
- It can be referred as method without a name

Use of Lambda Expressions in Java

- Advantage to use it is less coding
- The code to be used exactly once can be wrapped as lambda expression
- Encourages functional Programming

How to write Lambda Expressions in Java

- Syntax:-
- (one_parameter) -> { lambda expression };
- (first_parameter , second_parameter) -> { lambda expression };
- ( ) -> {lambda expression };

Functional Interface

- Interface having single abstract method are called functional interface
- These methods can have n number of static default methods
- Example:- Runnable, Predicate etc.

Functional Interface implementation

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

public class Example implements Predicate
{
@Override
public boolean test(Object o)
{
return o == null;
}
}
```

Functional interface with Lambda Expressions

```
public class Example {
public static void main(String a1[])
{
Predicate<String> p1 = x -> {
if(x == "Hello")
return true;
else
return false;
};
System.out.println(p1.test("Hello"));
}
}
```

Rules

- The variable type should be of Interface type, where Interface has one method
- The lambda expression should have same number of parameters and same return type as that of the method in that Interface

Method Reference

- Method reference is used to refer to a method of functional interface.
- It is a compact and easy form of Lambda expression and also minimizes lines of code even more than Lambda expression.

Syntax to Write Method References

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass:: staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject:: instanceMethodName	s::toString	s -> s.toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType:: methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

Constructor Reference

```
Interface StringMessage{
getString(String st);
}

Class StringExample{
String Example(String s1){
System.out.println(" Great " +s1)}
}

public class Example{
public static void main(String args[]){
String Message m1 = StringExample :: new;
m1.getString("Learning platform");
}
}
```

Java 8 Stream API

- Stream API is a newly added feature to the Collections API in Java Eight.
- A stream represents a sequence of elements and supports different operations (Filter, Sort, Map, and Collect) from a collection.
- We can combine these operations to form a pipeline to query the data, as shown in the below diagram:

- There are many ways to create a stream instance of different sources.
- Once created, the instance **will not modify its source**, therefore allowing the creation of multiple instances from a single source.

How Java Streams Work

- The simplest way to think of Java streams is to imagine a list of objects disconnected from each other, entering a pipeline one at a time.
- You can control how many of these objects are entering the pipeline, what you do to them inside it, and how you catch them as they exit the pipeline.
- We can obtain a stream from a collection using the .stream() method.

```
List<String>languages = new ArrayList <String>();
languages.add("English");
languages.add("German");
languages.add("French");
```

Using For loop:

```
for(String language:languages)
{
System.out.println(language);
}
```

Usage of Stream API Methods:

```
languages.stream().forEach(System.out::println)
```

Stream Creation

```
Stream<String> streamEmpty = Stream.empty();

Collection<String> collection = Arrays.asList("a", "b", "c"); Stream<String>
streamOfCollection = collection.stream();

Stream<String> streamOfArray = Stream.of("a", "b", "c");

String[] arr = new String[]{"a", "b", "c"}; Stream<String> streamOfArrayFull =
Arrays.stream(arr); Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

Stream of File

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =
Files.lines(path, Charset.forName("UTF-8"));
```

Referencing a Stream


```
Stream<String> stream =
        Stream.of("a", "b", "c").filter(element -> element.contains("b"));
Optional<String> anyElement = stream.findAny();
```

```
List<String> elements = Stream.of("a", "b", "c").filter(element -> element.contains("b"))
                .collect(Collectors.toList());
Optional<String> anyElement = elements.stream().findAny();
Optional<String> firstElement = elements.stream().findFirst();
```