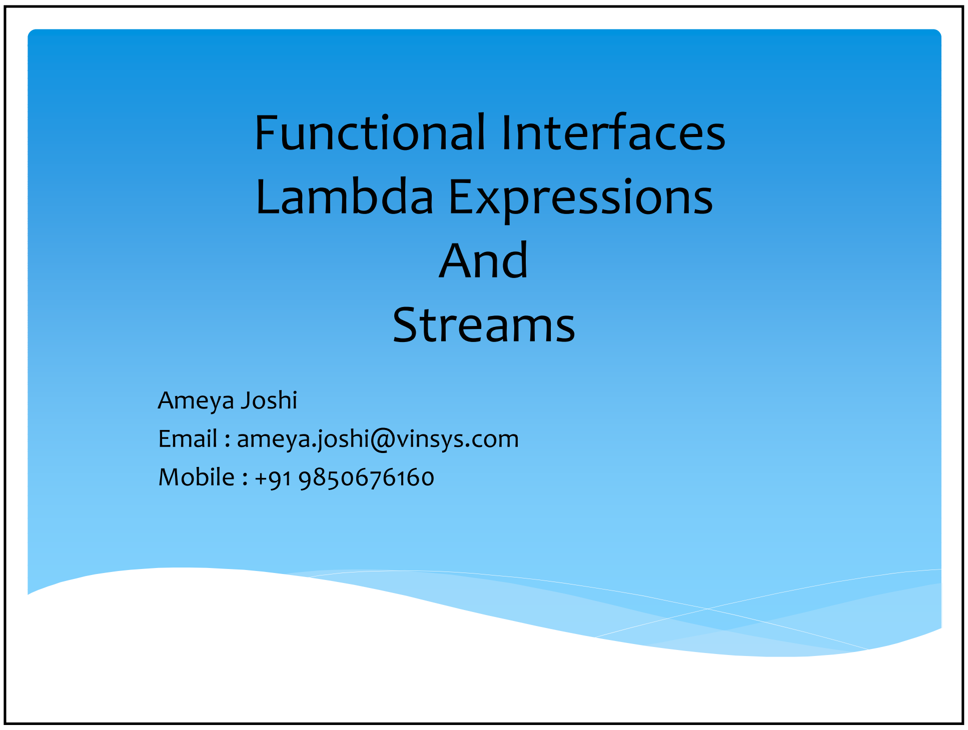
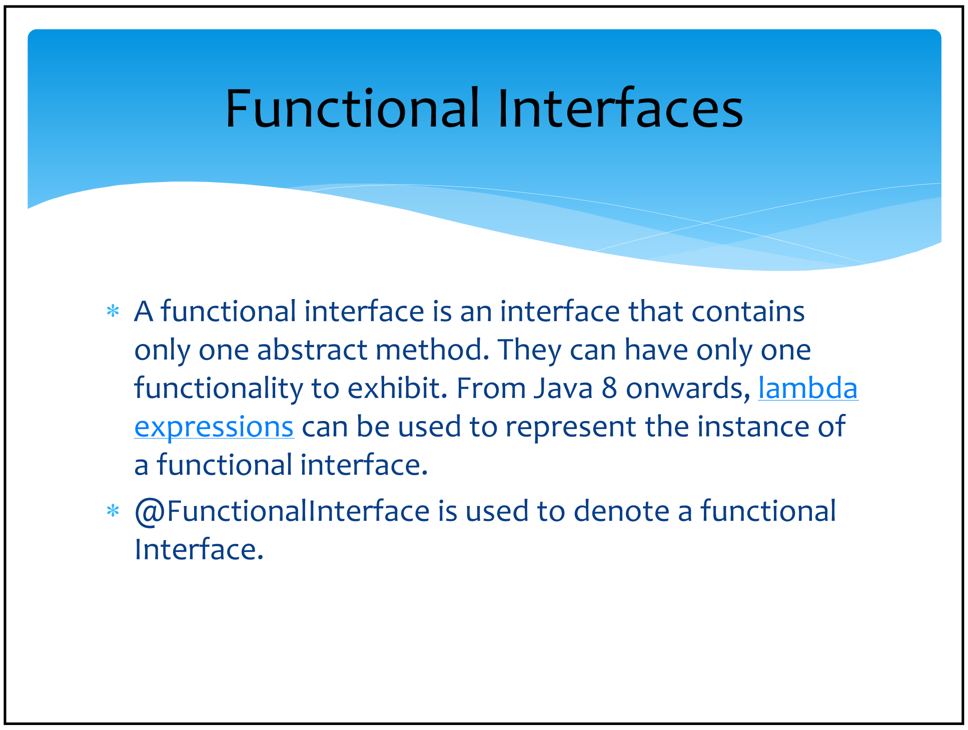


Functional Interfaces Lambda Expressions And Streams

Ameya Joshi
Email : ameya.joshi@vinsys.com
Mobile : +91 9850676160



Functional Interfaces

- A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.
- @FunctionalInterface is used to denote a functional Interface.

Lambda Expressions

- Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming, and simplifies the development a lot

Syntax

- parameter -> expression body

- The body of a **lambda expression** can contain zero, one or more statements. When there is a single statement curly brackets are not mandatory, and the **return type** is the same as that of the body **expression**

Example

```
public class LambdaDemo {

interface MathOperation {
int operation(int a, int b);
}
interface GreetingService {
void sayMessage(String message);
}

private int operate(int a, int b, MathOperation mathOperation) {
return mathOperation.operation(a, b);
}
```

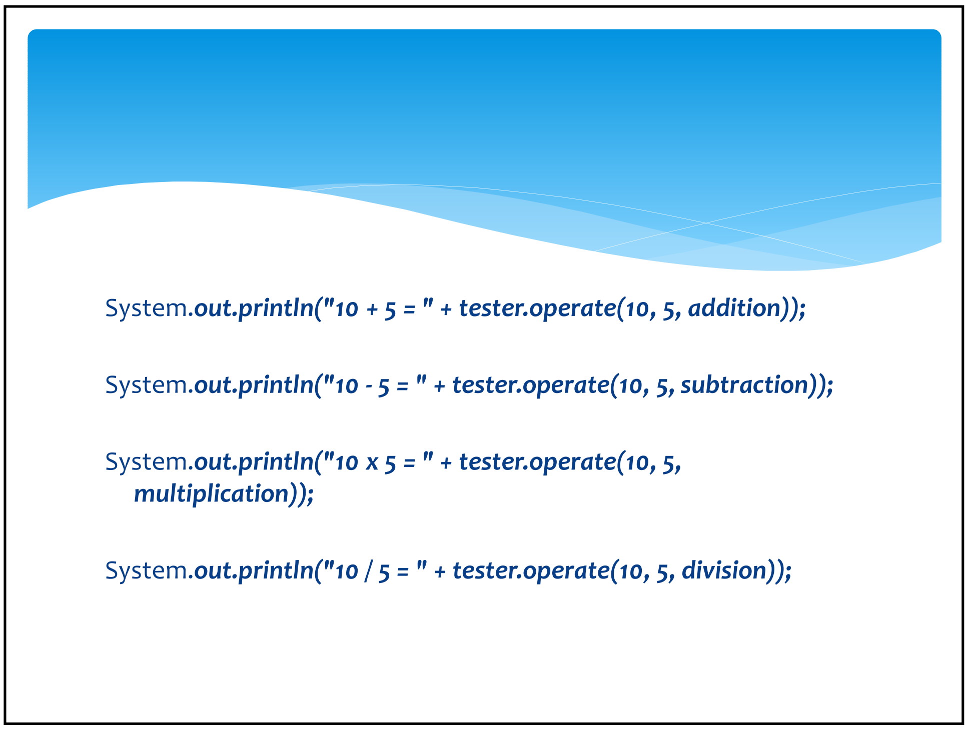
```
public static void main(String args[]) {
LambdaDemo tester = new LambdaDemo();

// with type declaration
MathOperation addition = (int a, int b) -> a + b;

// with out type declaration
MathOperation subtraction = (a, b) -> a - b;

// with return statement along with curly braces
MathOperation multiplication = (int a, int b) -> {
return a * b;
};

// without return statement and without curly braces
MathOperation division = (int a, int b) -> a / b;
```

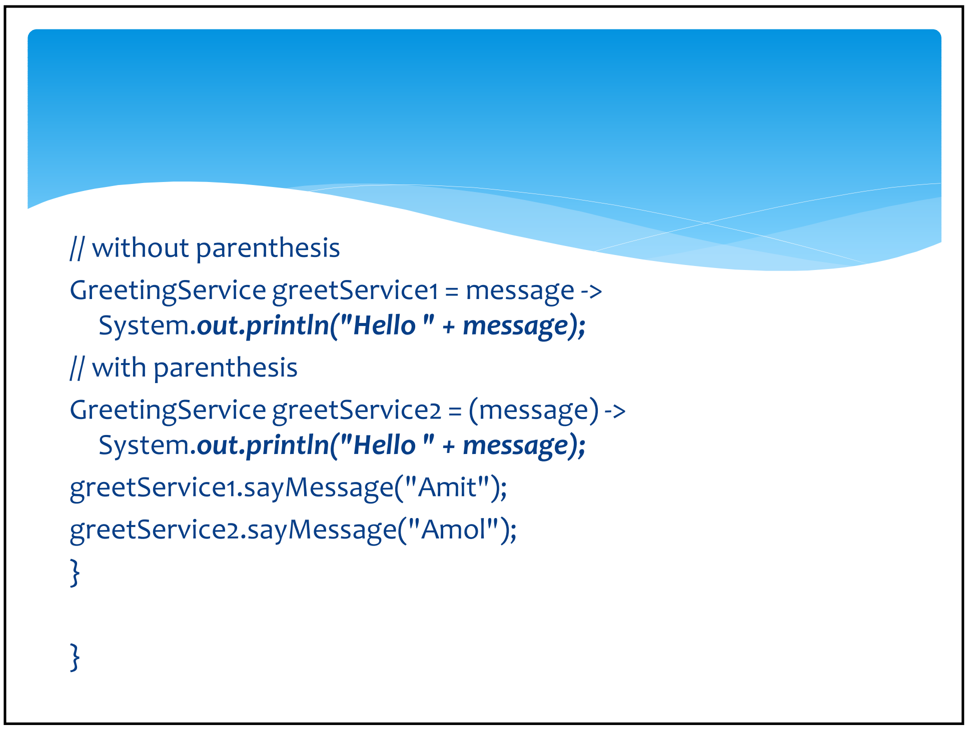


```
System.out.println("10 + 5 = " + tester.operate(10, 5, addition));

System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));

System.out.println("10 x 5 = " + tester.operate(10, 5,
   multiplication));

System.out.println("10 / 5 = " + tester.operate(10, 5, division));
```



```
// without parenthesis
GreetingService greetService1 = message ->
   System.out.println("Hello " + message);
// with parenthesis
GreetingService greetService2 = (message) ->
   System.out.println("Hello " + message);
greetService1.sayMessage("Amit");
greetService2.sayMessage("Amol");
}

}
```

Stream

- The Stream interface is defined in *java.util.stream* package. Starting from Java 8, the java collections will start having methods that return Stream. This is possible because of another cool feature of Java 8, which is default methods. Streams can be defined as *a sequence of elements from a source that supports aggregate operations.*

- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams

Intermediate Operations:
1.map: The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

2.filter: The filter method is used to select elements as per the Predicate passed as argument.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s-> s.startsWith("S")) .collect
(Collectors.toList());
```

3.sorted: The sorted method is used to sort the stream.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().sorted().collect(Collectors.toList());
```

Different Operations On Streams

Terminal Operations:
1.collect: The collect method is used to return the result of the intermediate operations performed on the stream.

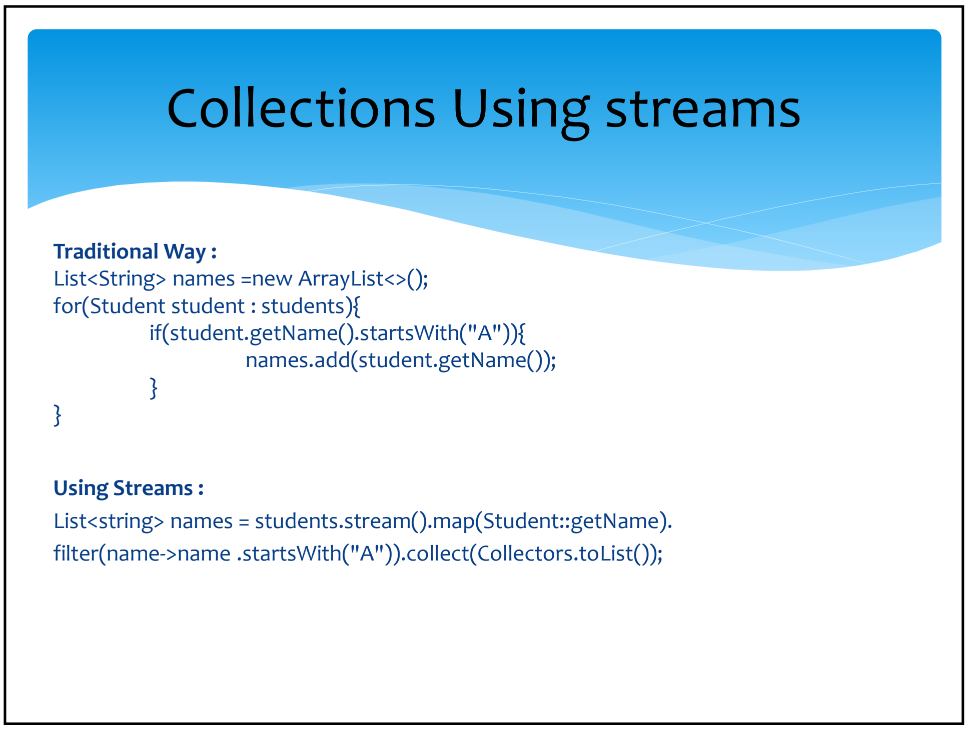
```
List number = Arrays.asList(2,3,4,5,3);
Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

2.forEach: The forEach method is used to iterate through every element of the stream.
3.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

3.reduce: The reduce method is used to reduce the elements of a stream to a single value.
The reduce method takes a BinaryOperator as a parameter.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```



Collections Using streams

Traditional Way :

```
List<String> names =new ArrayList<>();
for(Student student : students){
        if(student.getName().startsWith("A")){
                names.add(student.getName());
        }
}
```

Using Streams :

```
List<string> names = students.stream().map(Student::getName).
filter(name->name .startsWith("A")).collect(Collectors.toList());
```