Java Course
Lecture 11 - Lambdas and Stream API

IT Learning &
Outsourcing Center

www.pragmatic.bg

Summary

- What is Lambda
- What is functional interface
- Streaming API

Lambdas

- Lambdas bring anonymous function types in Java (JSR 335):

```
(parameters) -> {body}
```

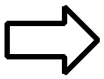
- Example:

```
(x,y) -> x + y
```

- Lambdas can be used in place of functional interfaces (interfaces with just one method such as **Runnable**)

```
new Thread(new Runnable() {

@Override
public void run() {
        System.out.println("It
runs !");
   }
}
).start();
```



```
new Thread(() ->
{ System.out.println("It runs !"); }
).start();
```

Functional Interfaces

- Every interface with only one method is @FunctionalInterface
- Examples of such functional interfaces:

java.lang.Runnable -> run()
java.util.concurrent.Callable -> call()
java.security.PrivilegedAction -> run()
java.util.Comparator -> compare(T o1, T o2)
java.awt.event.ActionListener ->
actionPerformed (ActionEvent e)

Extension(default) methods

- Default aka extension methods provide a mechanism for extending an existing interface without breaking backward compatibility

```
public interface Iterable<T> {

 default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
 }
}
```

Static methods in interfaces

- You can declare static methods on interfaces:

```
public interface ContentFormatter {
    public void format();
    static String convertRegex(String
        regex) {
        ...
    }
}
```

Note about static methods on interfaces

- None of the implementing classes will have this method available to them
- The idea : less utility classes

Some new functional interfaces

- Additional functional interfaces are provided by the **java.util.function** package for use by lambdas such as:
  - **Predicate<T>** - one method with param of type T and boolean return type
  - **Consumer<T>** - one method with param T and no return type
  - **Function<T, R>** - one method with param T and return type R
  - **Supplier<T>** - one method with no params and return type T

Simple functional interface

```
@FunctionalInterface
interface Test {
    public void doSomething();
}
```

However this is NOT FunctionalInterface:

```
@FunctionalInterface
interface Test2 {
  public default void doSomething()
{//Something}
}
```

Other valid functional interfaces


```
@FunctionalInterface
interface Test3 {
        public void doSomething();
        public default void doSomethingDefault() {}
}

@FunctionalInterface
interface Test4 {
        public void doSomething();
        public static void doSomethingDefault() {}
}
```

So .. Consuming functional interface

- Lets suppose you have a method like this

```
public static void something(Test t) {
        System.out.println(t.doSomething());
}
```

- You can invoke this method like this:

```
something(new Test<Integer>() {
                        @Override
                        public Integer doSomething() {
                                return 5;
                        }
                });
//or
something(() ->  5);
```

Stream API

- Databases and other programming languages allow us to specify aggregate operations explicitly
- The streams API provides this mechanism in the Java platform
- The notion of streams is derived from functional programming languages

Stream API (2)

- The stream API makes use of lambdas and extension methods
- Streams can be applied on collections, arrays, IO streams and generator functions
- Streams can be finite or infinite
- Streams can apply intermediate functions on the data that produce another stream (e.g. map, reduce)

Stream API usage

java.util.stream.Stream<T> **collection.stream();**

java.util.stream.Stream<T> **collection.parallelStream();**

Stream API useful methods

- filter(Predicate), map(Function), reduce(BinaryOperator), collect(Collector)
- min(Comparator), max(Comparator), count()
- forEach(Consumer)
- findAny(), findFirst()
- average(), sum()

- Prior to Java 8, printing the even numbers:

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4,
    5, 6);
for (Integer anInt : ints) {
    if (anInt % 2 == 0) {
        System.out.println(anInt);
    }
}
```

- After java 8 you can write this:

```
List<Integer> ints = Arrays.asList(1, 2, 3,
4, 5, 6);
ints.stream()
    .filter(i -> i % 2 == 0)
    .forEach(i -> System.out.println(i));
```

Optional

The main point behind Optional is to wrap an Object and to provide convenience API to handle nullability in a fluent manner.

```
Optional<String> stringOrNot = Optional.of("123");

//This String reference will never be null s
String alwaysAString = stringOrNot.orElse("");
```

But why do we need Optional?

- Let's see an example
- Task is : Double the first even number greater than 3
- Having as example

```
List<Integer> values = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
```

Old Pre java 8 way

```
int result = 0;
for(int e : values){
    if(e > 3 && e % 2 == 0) {
                result = e * 2;
                break;
            }
}
System.out.println(result);
```

The New way... Lambda way


```
System.out.println(
      values.stream()
        .filter(value -> value > 3)
        .filter(value -> value % 2 == 0)
        .map(value -> value * 2)
        .findFirst());
```

Lets read it, it's not that that hard?
Also lets check it in eclipse.

So in summary Optional is heavily used in streaming API …!

```
// This Integer reference will be wrapped again
Optional<Integer> integerOrNot =
    stringOrNot.map(Integer::parseInt);

// This int reference will never be null
int alwaysAnInt = stringOrNot
    .map(s -> Integer.parseInt(s))
    .orElse(0);
```

More at : http://java.dzone.com/articles/optional-will-remain-option

Stream internals

```
int sum = widgets.stream()
        .filter(w -> w.getColor() == RED)
        .mapToInt(w -> w.getWeight())
        .sum();
```

- Stream operations are composed into a pipeline
- Streams are lazy: computation is performed when the terminal operation is invoked

Creating a Stream

- You can take a Stream from a collection or you can also create a stream

```
Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4});
```

- However keep in mind you cannot write :

```
Stream<Integer> stream1 = Stream.of(new int[]{1,2,3,4});
//Compile time error, Type mismatch: cannot convert from Stream<int[]> to Stream<Integer>
```

Creating Streams(2)

- If you need to create a Stream of int or longs you should use IntStream or LongStream and so on

- LongStream is = Arrays.stream(new long[]{1,2,3,4});

Converting Stream to Collection

- There are several ways through which we can get a Collection or Array from a Stream.

```
Stream<Integer> intStream = Stream.of(1,2,3,4);
List<Integer> intList =
    intStream.collect(Collectors.toList());
System.out.println(intList); //prints [1, 2, 3, 4]

intStream = Stream.of(1,2,3,4); //stream is
    closed, so we need to create it again
Map<Integer,Integer> intMap =
    intStream.collect(Collectors.toMap(i -> i, i ->
i+10));
System.out.println(intMap); //prints {1=11, 2=12,
3=13, 4=14}
```

Converting Stream to Array

```
Stream<Integer> intStream =
  Stream.of(1,2,3,4);

Integer[] intArray =
  intStream.toArray(Integer[]::new);

System.out.println(Arrays.toString(i
ntArray)); //prints [1, 2, 3, 4]
```

Note: ignore the “::” bit for a while will speak about it in few slides

Stream Intermediate Operations

- **Stream filter() example** (use filter() method to test stream elements for a condition and generate filtered list.)

```
List<Integer> myList = new ArrayList<>();

for(int i=0; i<100; i++) myList.add(i);

Stream<Integer> sequentialStream = myList.stream();
Stream<Integer> highNums = sequentialStream.filter(p -> p
> 90); //filter numbers greater than 90

System.out.print("High Nums greater than 90=");

highNums.forEach(p -> System.out.print(p+" "));
//prints "High Nums greater than 90=91 92 93 94 95 96 97
98 99 "
```

Stream Intermediate Operations

- **Stream map() example**: We can use map() to apply functions to a stream.

```
Stream<String> names = Stream.of("aBc", "d", "ef");

List<String> upperCaseNames = names.map(s ->
s.toUpperCase()).collect(Collectors.toList());

System.out.println(upperCaseNames);
//prints [ABC, D, EF]
```

Stream Intermediate Operations

- **Stream sorted() example**: We can use sorted() to sort the stream elements by passing Comparator

```
Stream<String> names2 = Stream.of("aBc", "d", "ef",
"123456");
List<String> reverseSorted =
names2.sorted(Comparator.reverseOrder()).collect(Coll
ectors.toList());
System.out.println(reverseSorted);
// [ef, d, aBc, 123456]
Stream<String> names3 = Stream.of("aBc", "d", "ef",
"123456");
List<String> naturalSorted =
names3.sorted().collect(Collectors.toList());
System.out.println(naturalSorted);
//[123456, aBc, d, ef]
```

Stream Terminal Operations

- **Stream reduce() example**: We can use reduce() to perform a reduction on the elements of the stream

```
Stream<Integer> numbers = Stream.of(1,2,3,4,5);

Optional<Integer> intOptional =
    numbers.reduce((i,j) -> {return i*j;});
if(intOptional.isPresent())
    System.out.println("Multiplication =
      "+intOptional.get()); //120
```

Stream Terminal Operations

- **Stream count() example**: We can use this terminal operation to count the number of items

```
Stream<Integer> numbers1 = Stream.of(1,2,3,4,5);

System.out.println("Number of elements in
stream="+numbers1.count()); //5
```

Stream Terminal Operations

- **Stream forEach() example**: This can be used for iterating over the stream.

```
Stream<Integer> numbers2 = Stream.of(1,2,3,4,5);
numbers2.forEach(i -> System.out.print(i
+",")); //1,2,3,4,5,
```

Stream Terminal Operations

- Stream match() examples:

```
Stream<Integer> numbers3 = Stream.of(1,2,3,4,5);
System.out.println("Stream contains 4?
"+numbers3.anyMatch(i -> i==4));
//Stream contains 4? true

Stream<Integer> numbers4 = Stream.of(1,2,3,4,5);
System.out.println("Stream contains all elements less
than 10? "+numbers4.allMatch(i -> i<10));
//Stream contains all elements less than 10? true

Stream<Integer> numbers5 = Stream.of(1,2,3,4,5);
System.out.println("Stream doesn't contain 10?
"+numbers5.noneMatch(i -> i==10));
//Stream doesn't contain 10? true
```

Stream Terminal Operations

- **Stream findFirst() example**: This is a short circuiting terminal operation

```
Stream<String> names4 =
    Stream.of("Pankaj","Amit","David", "Lisa");

Optional<String> firstNameWithD = names4.filter(i
-> i.startsWith("D")).findFirst();

if(firstNameWithD.isPresent()){
     System.out.println("First Name starting with
     D="+firstNameWithD.get()); //David
}
```

Method references

- Intended to be used in lambda expressions, preventing unnecessary boilerplate
- Example:

```
books.stream().map(b -> b.getTitle())   ➔

books.stream().map(Book::getTitle)
```

- Lambda parameter list and return type must match the signature of the method

Method references – static methods


```
public class Printers {
      public static void print(String s) {...}
 }

Arrays.asList("a", "b", "c").forEach(Printers::print)
```

Method references – instance methods


```
public class Document {
    public String getPageContent(int pageNumber) {
        return this.pages.get(pageNumber).getContent();
    }
}

public static void printPages(Document doc, int[]
pageNumbers) {
  Arrays.stream(pageNumbers)
        .map(doc::getPageContent)
        .forEach(Printers::print);
 }
```

Method references – constructors


```
public static Stream<Page> createPagesFrom(Stream<String> contents)
{
        return contents.map(Page::new).
 }
```

Q and A ?