Harokopio University
Department of Informatics and Telematics

Anonymous Inner Class, Lambda Expressions & Streams

Object-oriented programming II

Dr. John Violos

Urls

Types of Anonymous Inner Class in Java:
https://www.tutorialspoint.com/how-many-types-of-anonymous-inner-classes-are-defined-in-java

Lambda Expressions:
https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm

Difference between Anonymous Inner Class and Lambda Expression:
https://www.geeksforgeeks.org/difference-between-anonymous-inner-class-and-lambda-expression/

Streams:
https://www.tutorialspoint.com/java8/java8_streams.htm

Anonymous Inner Class

An inner class **without a name** and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually subclass a class.

Anonymous inner classes are useful in writing implementation classes for listener interfaces in graphics programming.

Anonymous Inner Class that **extends a class**

```
class Car {
  public void engineType() {
    System.out.println("Turbo Engine");
  }}
public class AnonymousClassDemo {
  public static void main(String args[]) {
    Car c1 = new Car();
    c1.engineType();
    Car c2 = new Car() {
      @Override
      public void engineType() {
        System.out.println("V2 Engine");
      }};
    c2.engineType();}}
```

Anonymous Inner Class that **implements an Interface**

```
interface Software {
  public void develop();
}
public class AnonymousClassDemo1 {
  public static void main(String args[]) {
    Software s = new Software() {
      @Override
      public void develop() {
        System.out.println("Software Developed in Java");
      }};
    s.develop();
    System.out.println(s.getClass().getName());
  }}
```

Anonymous Inner Class **as an argument**

```
abstract class Engine {
  public abstract void engineType();
}
class Vehicle {
  public void transport(Engine e) {
    e.engineType();
  }}
public class AnonymousInnerClassDemo2 {
  public static void main(String args[]) {
    Vehicle v = new Vehicle();
    v.transport(new Engine() {
      @Override
      public void engineType() {
        System.out.println("Turbo Engine");
      }});}}
```

Lambda expression

Lambda expression facilitates functional programming, and simplifies the development a lot. A lambda expression is characterized by the following syntax:

parameter -> expression body

parameter -> expression body

- **Optional type declaration** – No need to declare the **type** of a parameter. The compiler can inference the same from the value of the parameter.
- **Optional parenthesis around parameter** – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
- **Optional curly braces** – No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword** – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

parameter -> expression body

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

Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

Using lambda expression, you can refer to any final variable or effectively final variable (which is assigned only once). Lambda expression throws a compilation error, if a variable is assigned a value the second time.

parameter -> expression body

```
final static String salutation = "Hello! ";

  public static void main(String args[]) {

    GreetingService greetService1 = message ->

    System.out.println(salutation + message);

    greetService1.sayMessage("Mahesh");

  }

  interface GreetingService {

    void sayMessage(String message);

  }
```

Anonymous Inner Class vs Lambda Expression

ANONYMOUS INNER CLASS	LAMBDA EXPRESSION
It is a **class without name**.	It is a **method without name**.(anonymous function)
It can **extend abstract** and **concrete class**.	It **can't** extend abstract and concrete class.
It can **implement an interface** that contains **any number of abstract methods.**	It can **implement an interface** which contains a **single abstract methods**.
Inside this we can declare instance variables.	It does not allow declaration of instance variables, whether the variables declared simply act as local
Anonymous inner class **can be instantiated**.	Lambda Expression **can't be instantiated**.
Inside Anonymous inner class, "**this**" always refers to **current anonymous inner class object** but not to outer	Inside Lambda Expression, "**this**" always refers to **current outer class object** that is, enclosing class
It is the best choice if we want **to handle multiple methods**.	It is the best choice if we want to **handle interface**.
At the time of compilation, a separate .class file will be generated.	At the time of compilation, no separate .class file will be generated. It simply convert it into private method outer
Memory allocation is on demand, whenever we are creating an onject.	It resides in a permanent memory of JVM.

Imperative Programming

Much like the imperative mood in speech, it tells the program to perform some command.

Statements: commands that express some action to be carried out. These statements usually change the state of the program.

```
Integer x = 0;

x++;
```

Functional Programming

It also has commands, but the commands are treated more like mathematical functions (declarative programming). Their task is to **take some input, perform some action, and return a result**

Do not modify the state of the program. The input variable(s) remain unchanged, and the returned result is always a new variable

```
function max(a, b) {return a > b ? a : b;}

var x = 10; var y = 5;

var maximum = max(x, y);
```

JDK 1.8: Combination OOP & FP

Use both OOP, FP, or a mix of the two:

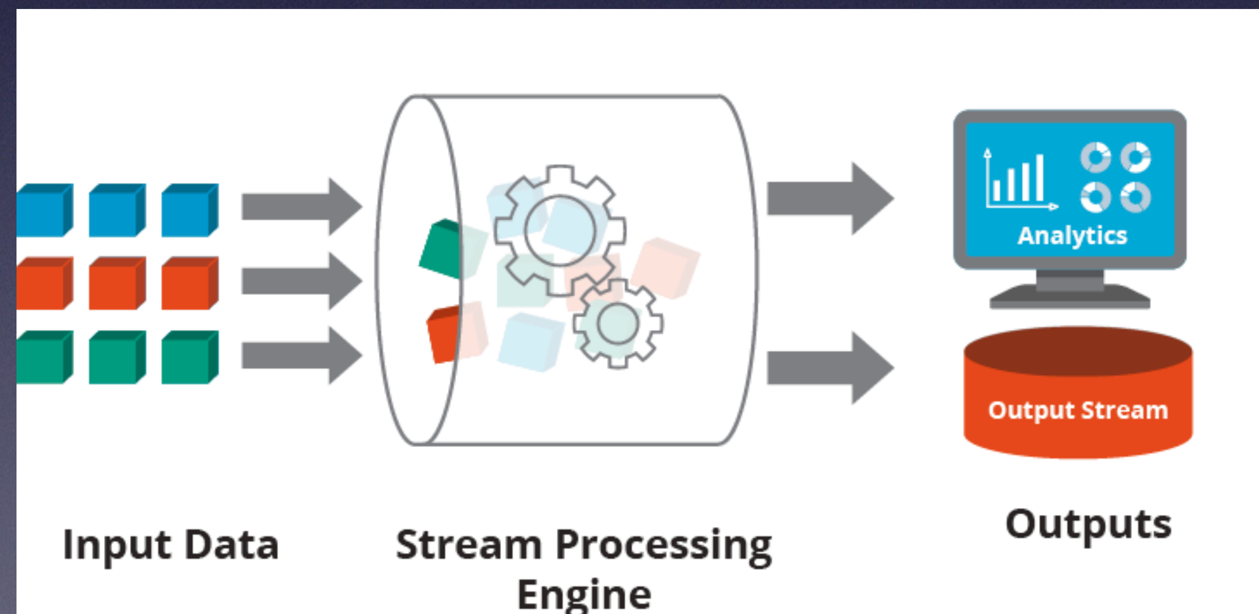
- Object-Oriented Programming: Java 1.7
- Functional Programming that comes with JDK 1.8 (new features)

Stream

A stream is a sequence of elements supporting sequential and parallel aggregate operations.

Aggregate operations:

- filter
- map
- limit
- reduce
- find



Stream represents a sequence of objects from a source, which supports aggregate operations.

- **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, I/O resources as input source (**Files**), or API.

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

Pub/Sub

Publisher B

Message 2

Topic B

Subscription B

Message 2

Subscriber B1

Subscriber B2

Point-to-Point

Publisher C

Message 3

Topic C

Subscription YC

Subscription ZC

Message 3

Message 3

Subscriber Y

Subscriber Z

Publish-Subscribe

Cloud Pub/Sub

Stream represents a sequence of objects from a source, which supports aggregate operations.

- **Pipelining** – Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. *collect*() method is a terminal operation which is normally present at the end of the pipelining operation to mark the *end of the stream*.

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

stream()

Collection interface has two methods to generate a Stream

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
```

map()

To **map each element** to its **corresponding result**. The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

List<Integer> squaresList = numbers.stream()
.map( i -> i*i).distinct().collect(Collectors.toList());
```

filter()

To **eliminate** elements based on criteria.
The following code segment prints a count of empty strings using filter.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

int count = strings.stream().**filter**(string -> string.*isEmpty*()).count();

forEach
forEach
filter

limit()

To **reduce the size of the stream**.
The following code segment shows how to print 10 random numbers using limit.

```
Random random = new Random();

random.ints().limit(10).forEach(System.out::println);
```

sorted()

To sort the stream.
The following code segment shows how to print 10 random numbers in a sorted order.

```
Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::
println);
```

forEach

forEach

forEach()

Stream has provided a new method ‘forEach’ to iterate each element of the stream.
The following code segment shows how to print 10 random numbers using forEach.

```
Random random = new Random();

random.ints().limit(10).forEach(System.out::println);
```

forEach
forEach
filter

parallelStream()

To **divide the provided task into many and run them in different threads**, utilizing multiple cores of the computer vs sequential streams that work just like for-loop using a single core

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg",
"abcd","", "jkl");

long count = strings.parallelStream().filter(string ->
string.isEmpty()).count();
```

forEach
forEach
filter

collect()

To **combine the result of processing on the elements of a stream**.
Collectors can be used to return a **list** or a **string**.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !
string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string -> !
string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);
```

forEach
forEach
filter

stats

To calculate all statistics when stream processing is being done.

```
List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = numbers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());
```

```
Map < String, List < String >> phoneNumbers = new HashMap < String, List <
String >> ();

phoneNumbers.put("John Lawson", Arrays.asList("3232312323", "8933555472"));

phoneNumbers.put("Mary Jane", Arrays.asList("12323344", “492648333"));

phoneNumbers.put("Mary Lou", Arrays.asList("77323344", "938448333"));

Map < String, List < String >> filteredNumbers = phoneNumbers.entrySet().stream()

    .filter(x - > x.getKey().contains("Mary"))

    .collect(Collectors.toMap(p - > p.getKey(), p - > p.getValue()));

filteredNumbers.forEach((key, value) - > {

    System.out.println("Name: " + key + ": ");

    value.forEach(System.out::println);});:
```

```
Map < String, List < String >> phoneNumbers = new HashMap < String, List <
String >> ();

phoneNumbers.put("John Lawson", Arrays.asList("3232312323", "8933555472"));

phoneNumbers.put("Mary Jane", Arrays.asList("12323344", “492648333"));

phoneNumbers.put("Mary Lou", Arrays.asList("77323344", "938448333"));

Map < String, List < String >> filteredNumbers = phoneNumbers.entrySet().stream()

    .filter(x - > x.getKey().contains("Mary"))

    .collect(Collectors.toMap(p - > p.getKey(), p - > p.getValue()));

filteredNumbers.forEach((key, value) - > {

    System.out.println("Name: " + key + ": ");

    value.forEach(System.out::println);});:
```