Introduction to Programming I

Lab 13

Alexey Shikulin, Munir Makhmutov, Sami Sellami and Furqan Haider

Agenda

- Introduction of the topic
- Warm-up exercises
- Solving Problems
- Q&A

Learning outcome:

- Enhance your knowledge on:
    - Lambdas, concurrency
- Practical / applicable knowledge

λambdas

Java Lambda Expressions

- Lambda Expressions were added in Java 8.
- A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Java Lambda Expressions. Syntax

The simplest lambda expression contains a single parameter and an expression:

parameter -> expression

To use more than one parameter, wrap them in parentheses:

(parameter1, parameter2) -> expression

To have function body more than one line, wrap them in parentheses:

(parameter1, parameter2) -> { code block }

Java Lambda Expressions. Usage

Use a lambda expression in the ArrayList's *forEach()* method to print every item in the list:

```java
import java.util.ArrayList;

public class Main {

   public static void main(String[] args) {
       ArrayList<Integer> numbers = new ArrayList<Integer>();
       numbers.add(5);
       numbers.add(9);
       numbers.add(8);
       numbers.add(1);
       numbers.forEach( (n) -> { System.out.println(n); } );
   }
}
```

Java functional interface

A *functional interface* in Java is an interface that contains only a single abstract (unimplemented) method. A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method.

Here is a Java functional interface example:

```
public interface MyFunctionalInterface {

        public void execute();

}
```

Functional Interfaces Implemented by a Lambda Expression

A Java functional interface can be implemented by a **Java Lambda Expression**.

Here is an example that implements the functional interface ***MyFunctionalInterface*** defined earlier:

```
MyFunctionalInterface lambda = () -> {

        System.out.println("Executing...");

}
```

A Java lambda expression implements a single method from a Java interface. The interface can only contain a single unimplemented method. In other words, the interface must be a Java functional interface.

Matching lambdas to interface:

Matching a Java lambda expression against a functional interface is divided into these steps:

- Does the interface have only one abstract (unimplemented) method?
- Does the parameters of the lambda expression match the parameters of the single method?
- Does the return type of the lambda expression match the return type of the single method?

If the answer is yes to these three questions, then the given lambda expression is matched successfully against the interface.

Java Lambda Expressions. Example

Use a lambda expression in the ArrayList's *forEach()* method to print every item in the list:

```java
interface StringFunction {
   String run(String str);
}
public class Main {
   public static void main(String[] args) {
       StringFunction exclaim = (s) -> s + "!";
       StringFunction ask = (s) -> s + "?";
       printFormatted("Hello", exclaim);
       printFormatted("Hello", ask);
   }
   public static void printFormatted(String str, StringFunction format) {
       String result = format.run(str);
       System.out.println(result);
   }
}
```

Built-in Functional Interfaces in Java

- Java contains a set of functional interfaces designed for commonly occuring use cases, so you don't have to create your own functional interfaces for every little use case.

- java.util.function package contains many functional interfaces:
  - Predicate<T>
  - UnaryOperator<T>
  - BinaryOperator<T>
  - Function<T,R>
  - Consumer<T>
  - Supplier<T>
  - ...

Predicate<T>

The Java Predicate interface, represents a simple function that takes a single value as parameter, and returns true or false

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

```
import java.util.function.Predicate;

public class LambdaApp {
   public static void main(String[] args) {
       Predicate<Integer> isPositive = x -> x > 0;
        System.out.println(isPositive.test(5)); // true
        System.out.println(isPositive.test(-7)); // false
   }
}
```

UnaryOperator<T>

The Java UnaryOperator interface is a functional interface that represents an operation which takes a single parameter and returns a parameter of the same type.

```
public interface UnaryOperator<T>
{
    T apply(T t);
}
```

```
import java.util.function.UnaryOperator;

public class LambdaApp {
   public static void main(String[] args) {
        UnaryOperator<Integer> square = x -> x*x;
         System.out.println(square.apply(5)); // 25
   }
}
```

BinaryOperator<T>

```
public interface BinaryOperator<T>
{
    T apply(T t1, T t2);
}
```

The Java BinaryOperator interface is a functional interface that represents an operation which takes two parameters and returns a single value. Both parameters and the return type must be of the same type.

```
import java.util.function.BinaryOperator;

public class LambdaApp {
   public static void main(String[] args) {
        BinaryOperator<Integer> multiply = (x, y) -> x*y;
         System.out.println(multiply.apply(3, 5));   // 15
         System.out.println(multiply.apply(10, -2)); // -20
   }
}
```

Function<T,R>

The Java Function interface interface represents a function (method) that takes a single parameter and returns a single value.

```
public interface Function<T,R> {
    R apply(T t);
}
```

```
import java.util.function.Function;

public class LambdaApp {
   public static void main(String[] args) {
        Function<Integer, String> convert =
              x -> String.valueOf(x) + " dollars";
        System.out.println(convert.apply(5)); // 5 dollars
   }
}
```

Consumer<T>

The Java Consumer interface is a functional interface that represents a function that consumes a value without returning any value.

```
public interface Consumer<T> {
    void accept(T t);
}
```

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

public class LambdaApp {
   public static void main(String[] args) {
        Consumer<Integer> printer =
                x -> System.out.printf("%d dollars \n", x);
        printer.accept(600); // 600 dollars
   }
}
```

The Java Supplier interface is a functional interface that represents a function that supplies a value of some sorts.

Supplier<T>

```java
public interface Supplier<T> {
    T get();
}
```

```java
class User {
    private String name;
    String getName(){
        return name;
    }

    User(String n){
        this.name = n;
    }
}
```

```java
import java.util.Scanner;
import java.util.function.Supplier;

public class LambdaApp {
   public static void main(String[] args) {
     Supplier<User> userFactory = () -> {
            Scanner in = new Scanner(System.in);
            System.out.println("Enter the name: ");
            String name = in.nextLine();
            return new User(name);
    };
    User user1 = userFactory.get();
    User user2 = userFactory.get();
    System.out.println("user1 name: " + user1.getName());
    System.out.println("user2 name: " + user2.getName());

    }
}
```

Stream API

- One of the major new features in Java 8 is the introduction of the stream functionality – java.util.stream – which contains classes for processing sequences of elements
- Simply put, streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.
- **A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**
- Stream API is based on lambda expressions and uses functional interfaces
- There are 2 types of operations in Stream API:
  - Terminal (forEach(), findFirst(), findAny(), toArray(), collect(), ...)
  - Intermediate (filter(), map(), flatMap(), sorted(), distinct(), peek(), skip(), limit(), ...)
- Each chain of Stream API calls ends with a single terminal operation preceded by any number of intermediate operations
- Collections have a method stream() to allow working with Stream API

Example

Adding non-unique values to the list and displaying only first value out of unique sorted list

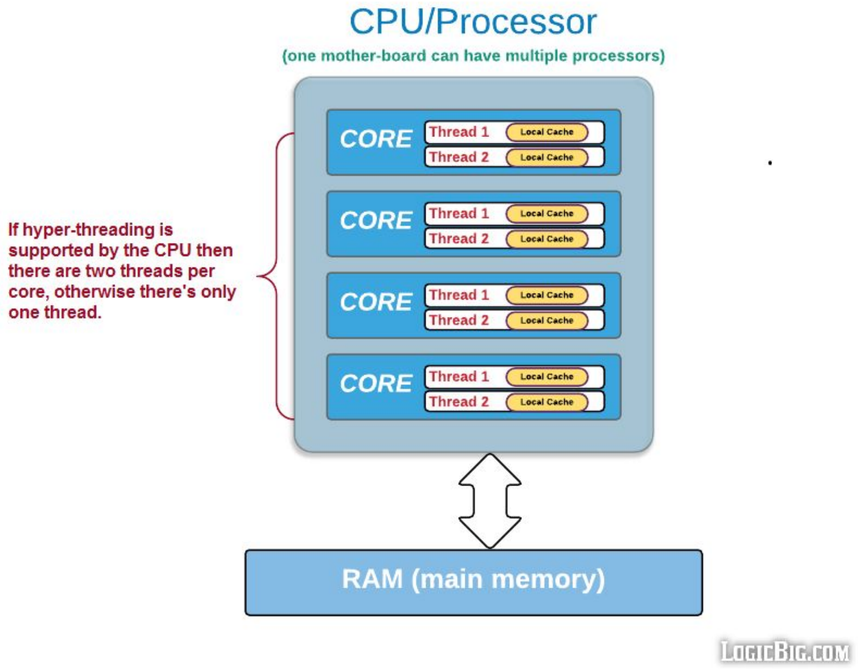
```java
import java.util.ArrayList;

public class StreamApp {
   public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
         numbers.add(5);
         numbers.add(9);
         numbers.add(8);
         numbers.add(8);
         numbers.add(1);
         numbers.stream().forEach((n)-> System.out.print(n));
         System.out.println();
         numbers.stream()
                  .filter(n -> n > 5)
                  .distinct()
                  .sorted()
                  .limit(1)
                  .forEach((n)-> System.out.print(n));
        System.out.println("\n" + numbers);
   }
}
// Output: 59881
//          8
//          [5, 9, 8, 8, 1]
```

multithreaded programming

Concurrency

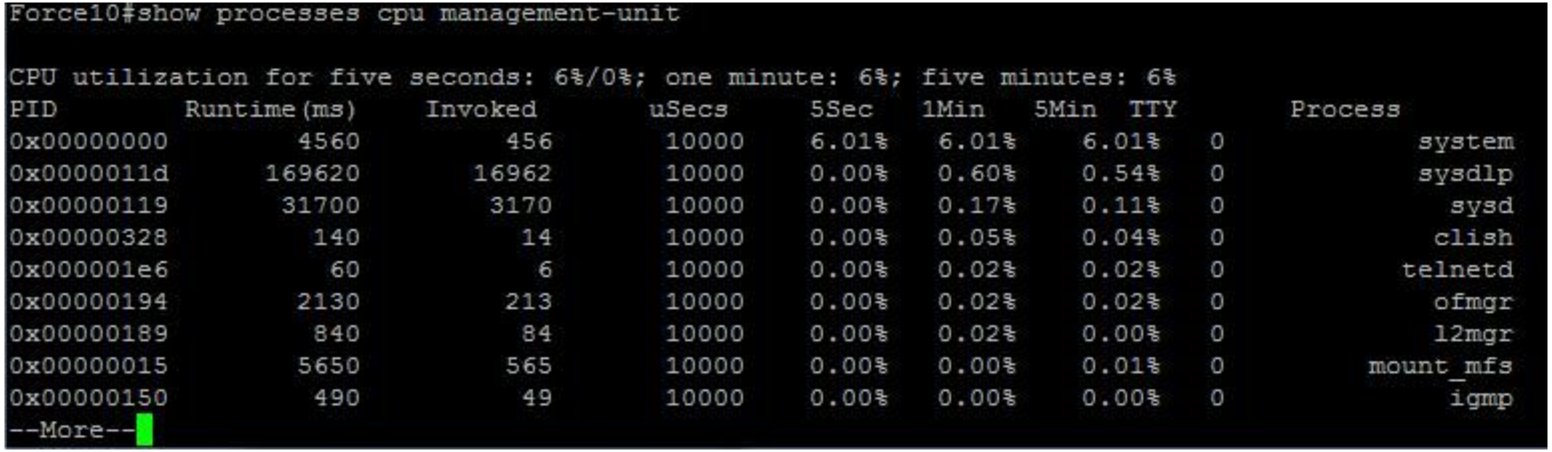
Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program.

A modern computer has several CPUs or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.


CPU/Processor
(one mother-board can have multiple processors)
If hyper-threading is supported by the CPU then there are two threads per core, otherwise there's only one thread.
CORE
Thread 1
Local Cache
Thread 2
Local Cache
RAM (main memory)
LogicBig.com

Process

A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.



```
Force10#show processes cpu management-unit

CPU utilization for five seconds: 6%/0%; one minute: 6%; five minutes: 6%
PID         Runtime(ms)     Invoked        uSecs     5Sec    1Min    5Min  TTY        Process
0x00000000         4560         456        10000    6.01%    6.01%    6.01%    0          system
0x0000011d       169620       16962        10000    0.00%    0.60%    0.54%    0          sysdlp
0x00000119        31700        3170        10000    0.00%    0.17%    0.11%    0            sysd
0x00000328          140          14        10000    0.00%    0.05%    0.04%    0           clish
0x000001e6           60           6        10000    0.00%    0.02%    0.02%    0         telnetd
0x00000194         2130         213        10000    0.00%    0.02%    0.02%    0           ofmgr
0x00000189          840          84        10000    0.00%    0.02%    0.00%    0           l2mgr
0x00000015         5650         565        10000    0.00%    0.00%    0.01%    0       mount_mfs
0x00000150          490          49        10000    0.00%    0.00%    0.00%    0            igmp
--More--
```

Threads

A thread is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data, it stores this data in its own memory cache.

A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Java Threads. Defining and Starting a Thread

An application that creates an instance of *Thread* must provide the code that will run in that thread. There are two ways to do this:

1. Provide a *Runnable* object. The *Runnable* interface defines a single method, *run()*, meant to contain the code executed in the thread. The *Runnable* object is passed to the *Thread* constructor, as in the *HelloRunnable* example:

```java
public class HelloRunnable implements Runnable {
   public void run() {
        System.out.println("Hello from a thread!");
   }
   public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
   }
}
```

Java Threads. Defining and Starting a Thread

An application that creates an instance of *Thread* must provide the code that will run in that thread. There are two ways to do this:

2. Subclass *Thread*. The *Thread* class itself implements *Runnable*, though its run method does nothing. An application can subclass *Thread*, providing its own implementation of *run()*, as in the *HelloThread* example:

```java
public class HelloThread extends Thread {
   public void run() {
        System.out.println("Hello from a thread!");
   }
   public static void main(String args[]) {
        (new HelloThread()).start();
   }
}
```

Warm-Up Exercises (1)

Describe the attributes of the following lambda expression:

```
(o) -> o.toString();
```

Warm-Up Exercises (2)

Does the modification of a Set with Stream API and without assigning the value to this set do any updates in the original set?

Warm-Up Exercises (3)

How many non-abstract (default) methods are allowed in functional interfaces?

Warm-Up Exercises (4)

What is the difference between *start()* and *run()* method of Thread class?

Warm Up Exercises (5)

What happens when an *Exception* occurs in Java thread?

Exercise 1: Lambda Expressions

Create a list of integer numbers and fill it with random positive and negative values. By using lambda expressions display all of them which are divisible by 3 and remove “-” sign if present

Exercise 2: stream() with Lambdas

Create a list of strings and fill it with random strings of different length containing English letters and numbers. Duplicate all values and add them to the same list. By using lambda expressions display the sorted list containing non-empty unique strings without numbers

Hint: use regular expression

Exercise 3: Concurrency. Race Condition

Reference a *Counter* object is from multiple threads (i.e. two threads, *thread1* and *thread2*). We can decompose the *counter*++ statement into 3 steps:

1. Retrieve the current value of counter
2. Increment the retrieved value by 1
3. Store the incremented value back in counter

```
class Counter {
   private int counter = 0;
   public void increment() {
       counter++;
   }
   public int getValue() {
       return counter;
   }
}
```

Exercise 3: Concurrency. Race Condition (cont.)

Reference a *Counter* object is from multiple threads (i.e. two threads, *thread1* and *thread2*). We can decompose the *counter*++ statement into 3 steps:

1. Retrieve the current value of counter
2. Increment the retrieved value by 1
3. Store the incremented value back in counter

```
class Counter {
   private int counter = 0;
   public void increment() {
       counter++;
   }
   public int getValue() {
       return counter;
   }
}
```

Now fix the race condition, so that counter object is referenced correctly. That is the threads read / write the counter sequentially.

Exercise 4: Concurrency. Deadlock

A Deadlock problem describes a situation in which two or more threads are blocked forever, waiting for one another. The deadlock occurs when several subprocesses need the same object but they get it in a different order.

Consider the code in the snippet:

https://gist.github.com/Rhtyme/ec91195d98aee16f24e78f61bd1970b8

Your task is to solve the deadlock problem, so that threads are not blocked by each other

Exercise 5: Simple 2D Game (optional)

Very primitive game *VehicleTrackerGraphDemo* whose source code you can obtain from the link:

http://xyzcode.blogspot.com/2016/12/a-simple-multi-threading-java-game-with.html

Try to understand how a new thread is used there. Try to implement new features, such as faster movement etc.

References

- https://www.w3schools.com/java/java_lambda.asp
- https://www.vogella.com/tutorials/JavaConcurrency/article.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html
- https://metanit.com/java/tutorial/9.3.php
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
- https://annimon.com/article/2778