CSE216 – Programming Abstractions
Recitation 7

Objectives:

- Understanding lambda expressions in Java
- Understanding Java 8 functional interface
- Lambda expression examples and exercises

Download Recitation7.zip.

Lambda expressions in Java

In programming, a Lambda expression (or function) is just an anonymous function, i.e., a function with no name and without being bounded to an identifier. Most OOP languages evolve around objects and instances and treat only them their first-class citizens. Another important entity i.e. functions take back seat. This is especially true in java, where functions can't exist outside an object. But in functional programming, you can define functions, give them reference variables and pass them as method arguments and much more. It's very useful feature and has been lacking in java from beginning. Now with java 8, we can also use these lambda expressions.[1]

Syntax of lambda expression:

```
either
(parameters) -> expression          //1
or
(parameters) -> { statements; }     //2
or
() -> expression                    //3
```

Some examples:

```
(int a, int b) -> a * b // takes two integers and returns their multiplication

(a, b)-> a - b // takes two numbers and returns their difference

() -> 99 // takes no values and returns 99

(String a) -> System.out.println(a) // takes a string, prints its value to
the console, and returns nothing

a -> 2 * a // takes a number and returns the result of doubling it

c -> { //some complex statements } // takes a collection and do some
processing
```

[1] For details, see Lambda Expressions, https://howtodoinjava.com/java8/lambda-expressions/

Features of Lambda expressions:

- A lambda expression can have zero, one or more parameters.
- The type of the parameters can be explicitly declared or it can be inferred from the context.
- Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
- When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a.
- The body of the lambda expressions can contain zero, one or more statements.
- If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

Introduction to Java 8 Streams – Filter, Map and Reduce Operations[2]

See StreamDemo.java

A stream is just a sequence of items. In Java 8, every class which implements the java.util.Collection interface has a stream method which allows you to convert its instances into Stream objects. Therefore, it's trivially easy to convert any list into a stream.

```
// Create an ArrayList

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

myList.add(1);

myList.add(5);

myList.add(8);

// Convert it into a Stream

Stream<Integer> myStream = myList.stream();
```

The features of Java stream are –

- 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 lazily 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.

The map Method:

It takes a lambda expression as its only argument, and uses it to change every individual element in the stream. Its return value is a new Stream object containing the changed elements.

[2] Reference material: https://www.sitepoint.com/java-8-streams-filter-map-reduce/

```
String[] myArray = new String[]{"bob", "alice", "paul", "ellie"};

Stream<String> myStream = Arrays.stream(myArray);

Stream<String> myNewStream =

             myStream.map(s -> s.toUpperCase());
```

The Stream object returned contains the changed strings. To convert it into an array, you use its toArray method:

```
String[] myNewArray =

              myNewStream.toArray(String[]::new);
```

The filter Method:

Sometimes, you might want to work with only a subset of the elements. To do so, you can use the filter method. Just like the map method, the filter method expects a lambda expression as its argument. However, the lambda expression passed to it must always return a Boolean value, which determines whether or not the processed element should belong to the resulting Stream object.

For example, if you have an array of strings, and you want to create a subset of it which contains only those strings whose length is more than four characters, you would have to write the following code:

```
Arrays.stream(myArray)

      .filter(s -> s.length() > 4)

      .toArray(String[]::new);  //notice chaining of Stream operations
```

The sorted Method:

The sorted method is used to sort the stream.

```
List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().sorted().collect(Collectors.toList());
```

The forEach Method:

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

```
List number = Arrays.asList(2,3,4,5);

number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

Reduction Operations (Terminal operations)

A reduction operation is one which allows you to compute a result using all the elements present in a stream. Reduction operations are also called terminal operations because they are always present at the end of a chain of Stream methods. We've already been using a reduction method in our previous example: the toArray method. It's a terminal operation because it converts a Stream object into an array.

```
int myArray[] = { 1, 5, 8 };

int sum = Arrays.stream(myArray).sum();
```

The reduce Method:

If you want to perform more complex reduction operations, however, you must use the reduce method. The reduce method is used to reduce the elements of a stream to a single value. Unlike the map and filter methods, the reduce method expects two arguments: an identity element, and a lambda expression. The lambda expression you pass to the reduce method must be capable of handling two inputs: a partial result of the reduction operation, and the current element of the stream.

The following is a sample code snippet which uses the reduce method to concatenate all the elements in an array of String objects:

```
String[] myArray = { "this", "is", "a", "sentence" };

String result = Arrays.stream(myArray)

                .reduce("", (a,b) -> a + b);
```

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);
```

The collect Method:

Stream Collectors class provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc. Java Collectors class provides various methods to deal with elements. 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());
```

See CollectorsExample.java.

Double colon (::) operator in Java[3]

See DColonDemo.java

The double colon (::) operator, also known as method reference operator in Java, is used to call a method by referring to it with the help of its class directly. They behave exactly as the lambda expressions. The only difference it has from lambda expressions is that this uses direct reference to the method by name instead of providing a delegate to the method.

Syntax: `<Class name>::<method name>`

Example: To print all elements of the stream:

```
//Print the stream using Lambda expression:

stream.forEach( s-> System.out.println(s));
```

[3] See for details: https://www.geeksforgeeks.org/double-colon-operator-in-java/

```
//Print the stream using double colon operator
stream.forEach(System.out::println);
```

Java 8 functional interface

Single Abstract Method (SAM) interfaces means interfaces with only one single method. From java 8, they are also referred as functional interfaces. Java 8, enforces the rule of single responsibility by marking these interfaces with a new annotation i.e. @FunctionalInterface. For example, new definition of Runnable interface is like this:

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

In java, lambda expressions are converted to a functional interface type. E.g. If we need to write a thread which will print “howtodoinjava” in console then simplest code will be:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("howtodoinjava");
    }
}).start();
```

If we use the lambda expression for this task then code will be:

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

We have also see that Runnable is an functional interface with single method run(). So, when you pass lambda expression to constructor of Thread class, compiler tries to convert the expression into equivalent Runnable code as shown in first code sample. If compiler succeed then everything runs fine, if compiler is not able to convert the expression into equivalent implementation code, it will complain. Here, in above example, lambda expression is converted to type Runnable.

More Lambda expression examples

See Recitation7.java, Example1.java , Example2.java, Example3.java

Homework

Task 1:

- Create a class Person with enum Sex{MALE, FEMALE}.
- Add attributes name, age, emailAddress and gender to Person class where gender is of type Sex.
- Create get methods corresponding to above attributes.
- The Person class constructor should take as input name, age, gender and emailAddress.
- The Person class has main() method which looks like the following:

```
public static void main(String args[]){

    List<Person> roster = new ArrayList<Person>();

    Person mary = new Person("Dajung", 20, Sex.FEMALE,
"dajung@stonybrook.edu");

    Person john = new Person("Jong", 22, Sex.MALE,
"jong@stonybrook.edu");

    Person cogi = new Person("David", 16, Sex.MALE,
"david@stonybrook.edu");

    Person steve = new Person("Daniel", 21, Sex.MALE,
"daniel@stonybrook.edu");

    Person bush = new Person("Bilal", 19, Sex.MALE,
"bilal@stonybrook.edu");

    Person seoyoung = new Person("Kelly", 18, Sex.FEMALE,
"kelly@stonybrook.edu");

    roster.add(mary);

    roster.add(john);

    roster.add(cogi);

    roster.add(steve);

    roster.add(bush);

    roster.add(seoyoung);

    Person.lambdaPrintPersonsOlderThan(roster, 18);

    Person.lambdaPrintEmailPersonsOlderThan(roster, 18);

    Person.lambdaPrintEmailMalesBetweenAge (roster, 16, 20);

}
```

- Create implementations of following methods which use lambda expressions:

    - `public static void lambdaPrintEmailMalesBetweenAge (List<Person> roster, int low, int high) {...}`
    - `public static void lambdaPrintPersonsOlderThan(List<Person> roster, int age) {...}`
    - `public static void lambdaPrintEmailPersonsOlderThan (List<Person> roster, int age) {...}`
- **Submit Person.java on blackboard.**

Task 2:

- Given a list of numbers, create a library StatFunctions.java which uses lambda expressions to calculate following statistical values: mean, median, mode, standard deviation and variance.
- For meaning of these terms, see: https://towardsdatascience.com/understanding-descriptive-statistics-c9c2b0641291
- Submit **StatFunctions.java** on blackboard.