COM2104: Advanced Programming

LECTURE 11: FUNCTIONAL PROGRAMMING

Objective

- Learn what is method reference
- Learn what streams are and how perform operations on Streams
- Recall how to use functional programming to simplify your code
- For lecture notes in today, only List/ArrayList will used.

Method Reference

Recall: Java Lambda Expressions

- **Lambda expressions in Java**, represent instances of functional interfaces (interfaces with a single abstract method).
- They provide a concise way to express instances of single-method interfaces using a block of code.

About method references

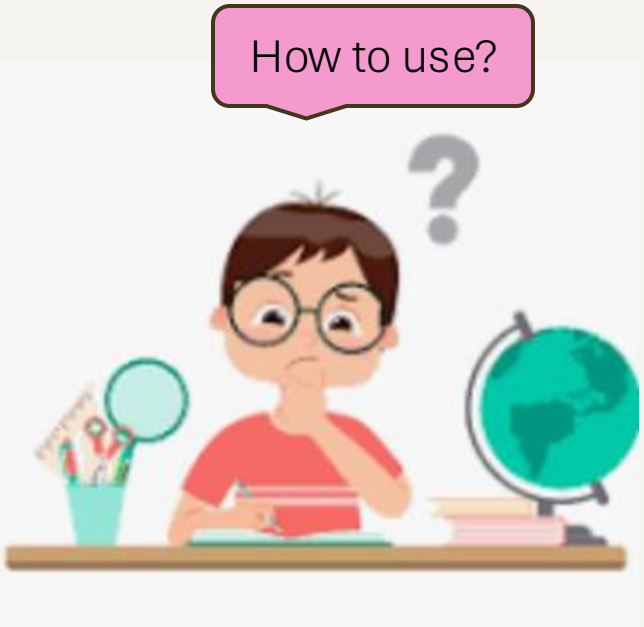
- **Method references are a special type of lambda expressions**. They're often used to create simple lambda expressions by referencing existing methods.
- There are four kinds of method references:
  - Static methods
  - Instance methods of particular objects
  - Instance methods of an arbitrary object of a particular type
  - Constructor

Syntax about using method reference

- A static method
  - ClassName::staticMethodName
- An instance method of a particular object (bound)
  - objectRef::methodName
- An instance method whose receiver is unspecified (unbound)
  - ClassName::instanceMethodName
- A constructor
  - ClassName::new

stream() and map() method

- stream() method in Java turns a collection into a flow of elements. They enable developers to perform functional operations on collections such as mapping and filtering.
- map() method: is an intermediate stream operation that transforms each element by some methods.



- https://belief-driven-design.com/functional-programm-with-java-map-filter-red

Combining with the method reference: static method

```
List<Double> o2 = Arrays.asList(1.9, 2.3,3.0);
```

We want to get the ceiling number for each value in the arraylist.

Using lambda

```
o2.stream().forEach(n->Math.ceil(n));
```

Using method reference

```
o2.stream().map(Math::ceil)
```

Map method will use Math.ceil for each value in the arraylist.

Reference to an Instance Method of a Particular Object

- In such case, we need to create one class with some methods.
- For example.
  BicycleComparator has overrode the compare method.
- And, BicycleComparator has one another method named addNumber which could add 10 to the frameSize of any Bicycle objects.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

class Bicycle {
    private String brand;
    private int frameSize;
    public Bicycle(String brand, int frameSize) {
        this.brand = brand;
        this.frameSize = frameSize;
    }
    // standard constructor, getters and setters
    public int getFrameSize() {
        return this.frameSize;
    }

    public String getBrand() {
        return this.brand;
    }
}
class BicycleComparator implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        Bicycle b1 = (Bicycle) o1;
        Bicycle b2 = (Bicycle) o2;
        if(b1.getFrameSize() < b2.getFrameSize()) return -1;
        if(b1.getFrameSize() > b2.getFrameSize()) return 1;
        return 0;
    }
        public int addNumber(Bicycle e) {
        return e.getFrameSize() + 10;
    }
}
```

Reference to an Instance Method of a Particular Object

We could use method reference for addNumber method first.

```
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ArrayList<Bicycle> obj = new ArrayList<>();
        obj.add(new Bicycle("a", 50));
        obj.add(new Bicycle("b", 20));
        obj.add(new Bicycle("c", 30));
        BicycleComparator bcompare = new BicycleComparator();
        obj.stream().map(bcompare::addNumber).forEach(System.out::println);
    }
}
```

Method reference

output

```
60
30
40
```

- We can create an ArrayList containing 3 bicycle objects.
- Then create an instance of class BicycleComparator.
- Convert the ArrayList to a stream and use the map method to add 10 to the frameSize of each object in the array list and print out the result.

Reference to an Instance Method of a Particular Object

We could use method reference for compare method as well.

```
public class Test {
    public static void main(String[] args) {
        ArrayList<Bicycle> obj = new ArrayList<>();
        obj.add(new Bicycle("a", 50));
        obj.add(new Bicycle("b", 20));
        obj.add(new Bicycle("c", 30));
        BicycleComparator bcompare = new BicycleComparator();
        List<Bicycle> a2 = obj.stream().sorted(bcompare::compare).collect(Collectors.toList());
        for(Bicycle a:a2) {
            System.out.println(a.getBrand() + " " + a.getFrameSize());
        }
    }
}
```

Method reference

- For the List a2, we firstly sort the objects in obj. After sorting, all sorted objects were put in one stream.
- Then using collect(Collectors.toList()) to convert the stream to an ArrayList.

Instance methods of an arbitrary object of a particular type

In such case, we don't use the objects of one class created by our own. But using the classes in some java packages instead.

```java
import java.util.ArrayList;
import java.util.Collections;
public class TestParticularObject {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ArrayList<Integer> obj2 = new ArrayList<>();
        Collections.addAll(obj2, 1,5,3,2,9,10);
        obj2.stream().sorted(Integer::compareTo).forEach(System.out::println);
    }
}
```

Method reference

output

```
1
2
3
5
9
10
```

One tip for using sorted() method in one stream

- If our stream contains some objects OF one class created by our own. After using sorted method, we need to convert the stream containing class objects to the List type for better printing them.
- If our stream contains some primitive data (int, char, byte, short, long, float, double, and boolean) or String data. After using sorted method, we could use forEach(System.out::println) to print them.

One tip for using sorted() method in one stream

- If we use forEach(System.out::println) to output a stream of sorted objects of our self-defined class, you'll see the representation of these objects in the memory instead of their respective property values.

For example:

```
obj.stream().sorted(bcompare::compare).forEach(System.out::println)
```

Output:

```
LabEleven.Bicycle@880ec60
LabEleven.Bicycle@3f3afe78
LabEleven.Bicycle@7f63425a
```

Reference to a Constructor

```java
class Bicycle {
    private String brand;
    private int frameSize;
    public Bicycle(String brand) {
        this.brand = brand;
        this.frameSize = 0;
    }
    // standard constructor, getters and setters
    public int getFrameSize() {
        return this.frameSize;
    }

    public String getBrand() {
        return this.brand;
    }

    public String toSting() {

        return "brand is " + this.brand + "framsize is " + this.frameSize;
    }
}
```

If we change the constructor of class Bicycle

Reference to a Constructor

```
public class Test {
    public static void main(String[] args) {
        ArrayList<String> obj2 = new ArrayList<>();
        Collections.addAll(obj2, "Giant", "Scott", "Trek", "GT");
        Bicycle[] a2 = obj2.stream().map(Bicycle::new).toArray(Bicycle[]::new);
        for(Bicycle a : a2) {
            System.out.println(a.getBrand() + ". " + a.getFrameSize());
        }
    }
}
```

Using map() method to create one object about class Bicycle.

toArray() could allow us to put all objects of Bicycle we created to one Array.

Methods to generate one stream

Methods

- There are around four methods to generate one stream:
  - IntStream
  - DoubleStream
  - Arrays.stream(any arrays)
  - Stream<Class *type*>

IntStream

- The Intstream is a sequence of raw integer values. We could use IntStream.of(some integers) to create one integer stream.
- We can use the following methods to operate on Intstream:
  - rangeclosed(a,b): provides a stream of integers from a to b
  - range(a,b): provides a stream of integers from a to b-1
  - summaryStatistics(): provides the summary statistics for some integers
  - sum(): calculates the sum of the values
  - sorted(): sorts the values
  - min(): gets the minimum
  - max(): gets the maximum
  - average(): gets the average

IntStream example

output

```
1
2
3
4
5
%%%%%%%
6
7
8
9
%%%%%%%
2
3
4
5
6
%%%%%%%
Sum is 5050
max is OptionalInt[100]
average is OptionalDouble[50.5]
Statistics is IntSummaryStatistics{count=100, sum=5050, min=1, average=50.500000, max=100}
```

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.*;

public class IntStreamDemo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        IntStream.rangeClosed(1,5).forEach(System.out::println);
        System.out.println("%%%%%%%%");
        IntStream.range(6,10).forEach(System.out::println);
        System.out.println("%%%%%%%%");
        IntStream.rangeClosed(2, 6).sorted().forEach(System.out::println);
        System.out.println("%%%%%%%%");
        System.out.println("Sum is " + IntStream.rangeClosed(1, 100).sum());
        System.out.println("max is " + IntStream.rangeClosed(1, 100).max());
        System.out.println("average is " + IntStream.rangeClosed(1, 100).average());
        System.out.println("Statistics is " +
        IntStream.rangeClosed(1, 100).summaryStatistics());
    }
}
```

DoubleStream

- The DoubleStream is a sequence of raw double values. Using DoubleStream.of(*some double values*) to generate one double stream.
- We can use the following methods to operate on DoubleStream:
  - min(): gets the minimum
  - max(): gets the maximum
  - average(): gets the average
  - summaryStatistics(): provides the summary statistics for some integers

https://www.tutorialspoint.com/doublestream-of-method-in-java

DoubleStream Example

```java
import java.util.stream.*;
public class Demo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Double values in the stream are " + DoubleStream.of(20.5, 35.9, 55.8, 68.7, 80.5));
        System.out.println("Min is " + DoubleStream.of(20.5, 35.9, 55.8, 68.7, 80.5).min());
        System.out.println("Max is " + DoubleStream.of(20.5, 35.9, 55.8, 68.7, 80.5).max());
        System.out.println("Average is " + DoubleStream.of(20.5, 35.9, 55.8, 68.7, 80.5).average());
        System.out.println("Statistics are " + DoubleStream.of(20.5, 35.9, 55.8, 68.7, 80.5).summaryStatistics());
    }
}
```

output

```
Double values in the stream are java.util.stream.DoublePipeline$Head@42eca56e
Min is OptionalDouble[20.5]
Max is OptionalDouble[80.5]
Average is OptionalDouble[52.279999999999994]
Statistics are DoubleSummaryStatistics{count=5, sum=261.400000, min=20.500000, average=52.280000, max=80.500000}
```

Arrays.stream()

- The stream() method of the Arrays class in Java is a utility method that allows you to obtain a sequential stream of elements from an array.
- The stream() method is overloaded for different types of arrays, such as arrays of primitive types (int[], double[], etc.) and arrays of reference types (Object[]).
- We could use stream() method to any ArrayList/List directly.

Arrays.stream() example

Convert a string array to a stream

```
import java.util.Arrays;
import java.util.stream.Stream;
class StudyTonight{
    public static void main(String args[])
    {
        String[] array = { "java", "cpp", "c", "python" };
        Stream<String> myStream = Arrays.stream(array);
        myStream.forEach(str -> System.out.print(str + " "));
    }
}
```

Convert an int array to a stream

```
import java.util.Arrays;
import java.util.stream.IntStream;
class StudyTonight{
    public static void main(String args[])
    {
        int[] array = {12, 41, 18, 4, 5, 31};
        IntStream  myStream = Arrays.stream(array);
        myStream.forEach(str -> System.out.print(str + " "));
    }
}
```

Stream<Class type>

- We could use Stream<Class type> to create a stream containing the object of the class type. For example:
- Stream<Integer>, Stream<Double>, Stream<String>...
- Stream<Objects>
- We need to use Stream.of(*data*) to convert one group of any data to one stream.

Stream<Class type> Example

```java
import java.util.stream.Stream;

public class FilterExample {
    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);

        // Use filter() to include only even numbers
        Stream<Integer> evenStream = stream.filter(n -> n % 2 == 0);

        // Print the filtered elements
        evenStream.forEach(System.out::println);
    }
}
```

output

2
4
6

filter() method will find the elements satisfied the conditions.

Stream<Class type> Example

```java
import java.util.stream.Stream;

public class ComplexFilterExample {
    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Filter numbers greater than 3 and even
        Stream<Integer> filteredStream = stream.filter(n -> n > 3 && n % 2 == 0);

        // Print the filtered elements
        filteredStream.forEach(System.out::println);
    }
}
```

output

```
4
6
8
10
```

Stream<Class type> Example

Static inner class

- We store some objects about class User to one stream.
- We could use the User::isActive to find the users with an active status of true

```java
import java.util.stream.Stream;

public class FilterInactiveUsersExample {
    static class User {
        String name;
        boolean active;

        User(String name, boolean active) {
            this.name = name;
            this.active = active;
        }

        boolean isActive() {
            return active;
        }

        @Override
        public String toString() {
            return name;
        }
    }

    public static void main(String[] args) {
        Stream<User> users = Stream.of(
            new User("Alice", true),
            new User("Bob", false),
            new User("Charlie", true),
            new User("David", false)
        );

        // Filter only active users
        Stream<User> activeUsers = users.filter(User::isActive);

        // Print the active users
        activeUsers.forEach(System.out::println);
    }
}
```

Output

```
Alice
Charlie
```

Stream<Class type> Example

Considering this example, we used filter method to find the employees with the age larger than 30 and the department of Engineering.

```java
import java.util.stream.Stream;
class Employee {
    String name;
    int age;
    String department;

    Employee(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
    }
    int getAge() {
        return age;
    }
    String getDepartment() {
        return department;
    }
   @Override
    public String toString() {
        return name + " (" + age + ", " + department + ")";
    }
}
public class FilterEmployeesExample {
    public static void main(String[] args) {
        Stream<Employee> employees = Stream.of(
            new Employee("Rajesh", 35, "Engineering"),
            new Employee("Anita", 28, "HR"),
            new Employee("Vikram", 40, "Engineering"),
            new Employee("Meera", 25, "Sales")
        );
        // Filter employees older than 30 and in the Engineering department
        Stream<Employee> filteredEmployees =
        employees.filter(e -> e.getAge() > 30 && e.getDepartment().equals("Engineering")
        // Print the filtered employees
        filteredEmployees.forEach(System.out::println);
        }
```

Output

```
Rajesh (35, Engineering)
Vikram (40, Engineering)
```

Using multiple methods together

```
Stream<String> rows2 = Files.lines(Paths.get("data.txt"));
rows2
    .map(x -> x.split(","))
    .filter(x -> x.length == 3)
    .filter(x -> Integer.parseInt(x[1]) > 15)
    .forEach(x -> System.out.println(x[0] + "  " + x[1] + "  " + x[2]));
rows2.close();
```

We could use map to split each line content

First filter to get the lines with three values.

Close the connection finally.

The second filter converts the values in the second column to integer types and finds values greater than 15.

```
A,12,3.7
B,17,2.8
C,14,1.9
D,23,2.7
E
F,18,3.4
```

data.txt

More Methods to operate one stream

More methods

- You have learned some methods for operating one stream, like map(), filter(), forEach(), sorted() and so on.
- There are also other more methods, the following methods are popularly used:

Methods	Description
distinct():	Remove duplicated elements in the ArrayList
findFirst()	Find the first element in the ArrayList.
orElse()	returns the value if present, otherwise returns *other*
limit(int number)	Truncates a stream to a specific number of elements.

Example 1

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Stream<Integer> si = Stream.of(5,2,3,2,4,1,5,6).distinct();
    si.forEach(System.out::println);
}
```

Remove duplicates

Output

```
5
2
3
4
1
6
```

Example 2

- In the following example, we will generate a random number from 0 to 1.
- Use filter to determine if the number is greater than 0.5.
- then use findFirst to get the number.
- If it is not found, return -1.

```
double firstBg = Stream.generate(Math::random)
                .filter(r -> r>0.5)
                .findFirst()
                .orElse(-1.0);
```

Output `0.9252003804025993`

Example 3

Using limit() to generate 5 random numbers.

```
Stream<Double> numr = Stream.generate(Math::random).limit(5);
```

reduce() method for one stream of integers

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
int result = numbers
  .stream()
  .reduce(0, (subtotal, element) -> subtotal + element);
```

Initial sum value subtotal = 0.

Here is the lambda expression being the accumulator since it takes the partial sum of *Integer* values and the next element in the stream.

To use method reference:

```
int result = numbers.stream().reduce(0, Integer::sum);
```

reduce() method for one stream of Strings

```
List<String> letters = Arrays.asList("a", "b", "c", "d", "e");
String result = letters
   .stream()
   .reduce("", (partialString, element) -> partialString + element);
```

Here is the lambda expression to concatenate all letter together.

Initial sum value
subtotal = 0.

To use method reference:

```
String result = letters.stream().reduce("", String::concat);
```

End