CS5004, Spring 2024

Lab 9: Functional Programming in Java.

Tamara Bonaci and Abi Evans

tbonacin@northeastern.edu, ab.evans@northeastern.edu

1. Summary

In today's lab, we will continue our conversation about functional programming in Java. More specifically, we will focus on streams and lambdas in Java, and we will explore some design patterns in functional programming paradigm.

Note 1: Labs are intended to help you get started, and give you some practice while the course staff is present and able to provide assistance. You are not required to finish all the questions during the lab, but you are expected to push your lab work to a designated repo on the Khoury GitHub.

Problem 1: Functional programming in Java

Please use functional programming elements (Streams and lambdas) to identify trending topics on some social media platform.

The first step to identifying trending topics is to develop a **counter of topics**, and this is your task in this problem. Please design a class `TrendingTopics`, and within that class implement a method `countTopics()`.

Method `countTopics()` takes a List<String> as input, and returns the number of occurrences of every `String` in the input list as a `Map`, where every distinct `String` represents the `Map` key, and the number of the `String's` occurrences the `Map` value. For example, given the list of `Strings`:

Seattle, wildfires, DEFCON26, NEU, NEU, Seattle, Seattle, NEU, DEFCON26, wildfires

the count should contain the following information:

- **Seattle** → 3
- **wildfires** → 2
- **DEFCON26** → 2
- **NEU** → 3

Problem 2: Functional programming in Java

Please consider classes `Vehicle.java` and `OlderVehiclesFilter.java`, provided below.

```
public class Vehicle {

     private String make;
     private String model;
     private Integer year;
     private Color vehicleColor;

/*

   • * Constructor for an object Vehicle, based on the provided
     input arguments.
   • * @param make - String, make of a vehicle
   • * @param model - String, make of a vehicle
   • * @param year - Integer, year of the vehicle (expected to
     be in the range 1950-2020)
   • * @param vehicleColor - Color, color of a vehicle */

public Vehicle(String make, String model, Integer year, Color
vehicleColor){

     this.make = make;
     this.model = model;
     this.year = year;
     this.vehicleColor = vehicleColor;

}

/*
 * Getter for the property 'make'
 * @returns value for property 'make' (String)
 */

public String getMake() { return make; }


/*
 * Getter for the property 'model'
 * @returns value for property 'model' (String)
```

```
 */

public String getModel() { return model; }

/*
* Getter for the property 'year'
* @returns value for property 'year' (Integer in the range 1950-
2020) */

public Integer getYear() { return year; }

/*
 * Getter for the property 'vehicleColor'
 * @returns value for property 'vehicleColor' (Color)
 */

public Color getVehicleColor() { return vehicleColor; }

@Override
public boolean equals(Object o) {

if (this == o) return true;
if (!(o instanceof Vehicle)) return false;
Vehicle vehicle = (Vehicle) o;
return Objects.equals(getMake(), vehicle.getMake());

}

@Override
public int hashCode() {

return Objects.hash(getMake()); }

@Override
public String toString() {

return "Vehicle{" +
"make='" + make + '\'' +
", model='" + model + '\'' +
", year=" + year +
", vehicleColor=" + vehicleColor + '}'; } }



import java.util.ArrayList;
```

```java
import java.util.List;


public class OlderVehiclesFilter {

	private List<Vehicle> vehicles = new ArrayList<>();

	public OlderVehiclesFilter(List<Vehicle> vehicles) {
	this.vehicles = vehicles;

	}

	public OlderVehiclesFilter(Vehicle vehicle1, Vehicle
	vehicle2, Vehicle vehicle3){

		this.vehicles.add(vehicle1);
		this.vehicles.add(vehicle2);
		this.vehicles.add(vehicle2);

	}

	public List<String> filterVWvehilces(){

	//YOUR CODE HERE
	return null;

	}

}
```

In `OlderVehiclesFilter.java`, you will notice method `filterOlderVehicles()`. The goal of this method is to print out make, model and year of all the vehicles manufactured after 1999.

Please provide code, and Javadoc (no tests needed) for `method filterOlderVehicles()`.

In doing so, please use the elements of functional Java. Please feel free to develop any helper methods that you might need within class `OlderVehiclesFilter.java`, but please do not modify class `Vehicle.java`.

Additional Reading: Functional programming in Java and Factory Design Pattern

You likely still remember design pattern, and in particular the factory design pattern. The factory design pattern is used to assist in the creation of objects.

To refresh your memory about the factory pattern, let's take a look at the following code example, originally presented in the book "Learning Java Functional Programming", by R. M. Reese.

```
public interface VacuumCleaner() {

     public void vacuum();
     public void clean();

}
```

The given interface is implemented by two classes:

```
public class DirtVacuumCleaner implements VacuumCleaner() {

     public DirtVacuumCleaner() {
          System.out.println("Creating DirtVacuumCleaner");
     }

     @Override
     public void vacuum() {
          System.out.println("Vacuuming dirt!");
     }
     @Override
     public void clean () {
          System.out.println("Cleaning Dirt Vacuum Cleaner!");
     }

}

public class WaterVacuumCleaner implements VacuumCleaner() {

     public WaterVacuumCleaner() {
          System.out.println("Creating WaterVacuumCleaner");
     }

     @Override
     public void vacuum() {
          System.out.println("Vacuuming water!");
     }
```

```
    @Override
    public void clean () {
        System.out.println("Cleaning Water Vacuum Cleaner!");
    }

}
```

A `VacuumCleanerFactory` class is shown below. It provides a static method `getInstance()` that takes a string indicating the type of vacuum cleaner to create:

```
public class VacuumCleanerFactory() {

    public static VacuumCleanter getInstance (String type) {

        VacuumCleaner vacuumCleanter = null;

        if("Dirt".equals(type)){
            vacuumCleaner = new DirtVacuumCleaner();
        } else if ("Water".equals(type)){
            vacuumCleaner = new WaterVacuumCleanter();
        } else {
            //Handling all bad cases
        }
        return vacuumCleaner;
    }
}
```

We could also use lambda expressions to eliminate the need for an explicit factory class. The next example illustrates how we can use two `Supplier` functional interfaces, and lambda expressions to support the creation of `DirtVacuumCleanter` and `WaterVacuumCleaner` instances.

The variable, `dvcSupplier` represents an object that supports the `Supplier` interface's get method, which returns an instance of a `DirtVacuumCleaner`. The `wvcSupplier` variable works in a similar way for an instance of `WaterVacuumCleaner`.

```
Supplier<DirtVacuumCleaner> dvcSupplier =
DirtVacuumCleaner::new;

DirtVacuumCleaner dvc = dvcSupplier.get();

dvc.vacuum();

dvc.clean();
```

```
Supplier<WaterVacuumCleaner> wvcSupplier = WaterVacuumCleaner
::new;

WaterVacuumCleanter wvc = wvcSupplier.get();

wvc.vacuum();

wvc.clean();
```