FUNDAMENTAL PROGRAMMING TECHNIQUES

PROCESSING SENSOR DATA OF DAILY ACTIVITIES

ASSIGNMENT 5



Teacher: Ioan Salomie

Teacher Assistant: Cristina Pop

Student: Suster Elena -Roxana

Group: e\_30421

1. OBJECTIVE:

Consider designing, implementing and testing an application for analysing the behaviour of a person recorded by a set of sensors installed in its house. The historical log of the person’s activity is stored as tuples (start time, end time, activity label), where start time and end time represent the date and time when each activity has started and ended while the activity label represents the type of activity performed by the person: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare Time/ TV, Grooming. The data is spread over several days as many entries in the log Activities.txt, which is the input file.

The input file is taken from the assignment site at the following link:

http://coned.utcluj.ro/~salomie/PT\_Lic/4\_Lab/Assignment\_5 .

The new program should use functional programming in Java with lambda expressions and stream processing to perform the tasks listed in the table presented in the assignment description. The results of each task must be written in a separate .txt file (each .txt file must be named according to the given template).

1. PROBLEM ANALYSIS, SCENARIOS, USE CASES

Input and Output:

The input file is taken from the assignment site at the following link:

http://coned.utcluj.ro/~salomie/PT\_Lic/4\_Lab/Assignment\_5 .

The new program should use functional programming in Java with lambda expressions and stream processing to perform the tasks listed in the table presented in the assignment description. The results of each task must be written in a separate .txt file (each .txt file must be named according to the given template).

Use cases

In software and systems engineering, a use case is a list of actions or event steps, typically defining the interactions between a role (known in the Unified Modeling Language as an actor) and a system, to achieve a goal. The actor can be a human, an external system, or time. In systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals. Another way to look at it is a use case describes a way in which a real-world actor interacts with the system. In a system use case you include high-level implementation decisions. System use cases can be written in both an informal manner and a formal manner.

Following example will illustrate on how to plan use cases:

**Use Case:** What is the main objective of this use case. For eg. Adding a software component, adding certain functionality etc.

**Primary Actor:** Who will have the access to this use case. In the above examples, administrators will have the access.

**Scope:** Scope of the use case

**Level:** At what level the implementation of the use case be.

**Flow:** What will be the flow of the functionality that needs to be there. More precisely, the work flow of the use case.

1. DESIGN

\*Packages:

Java packages help in organizing multiple modules and group together related classes and interfaces.

In object-oriented programming development, model-view-controller (MVC) is the name of a methodology or design pattern for successfully and efficiently relating the user interface to underlying data models. The MVC pattern is widely used in program development with programming languages such as Java, Smalltalk, C, and C++.

The MVC pattern has been heralded by many developers as a useful pattern for the reuse of object code and a pattern that allows them to significantly reduce the time it takes to develop applications with user interfaces.

\*Lambda Expressions:

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

The previous section, Anonymous Classes, shows you how to implement a base class without giving it a name. Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-method classes more compactly.

\*Method References

You use [lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

### Reference to a Static Method

The method reference Person:: compare By Age is a reference to a static method.

### Reference to an Instance Method of a Particular Object

The following is an example of a reference to an instance method of a particular object:

class Comparison Provider {

public int compare By Name(Person a, Person b) {

return a.get Name().compare To(b.get Name());

}

public int compare By Age(Person a, Person b) {

return a.get Birthday().compare To(b.get Birthday());

}

}

Comparison Provider my Comparison Provider = new Comparison Provider();

Arrays.sort (roster As Array, **my Comparison Provider::compare By Name**);

The method reference my Comparison Provider::compare By Name invokes the method compare By Name that is part of the object my Comparison Provider. The JRE infers the method type arguments, which in this case are (Person, Person).

### Reference to an Instance Method of an Arbitrary Object of a Particular Type

The following is an example of a reference to an instance method of an arbitrary object of a particular type:

String[] string Array = { "Barbara", "James", "Mary", "John",

"Patricia", "Robert", "Michael", "Linda" };

Arrays.sort (string Array, String::compare To Ignore Case);

The equivalent lambda expression for the method reference String::compare To Ignore Case would have the formal parameter list (String a, String b), where a and b are arbitrary names used to better describe this example. The method reference would invoke the method a.compare To Ignore Case(b).

### Reference to a Constructor

You can reference a constructor in the same way as a static method by using the name new. The following method copies elements from one collection to another:

public static <T, SOURCE extends Collection<T>, DEST extends Collection<T>>

DEST transfer Elements(

SOURCE source Collection,

Supplier<DEST> collection Factory) {

DEST result = collection Factory.get();

for (T t : source Collection) {

result. add(t);

}

return result;

}

The functional interface Supplier contains one method get that takes no arguments and returns an object. Consequently, you can invoke the method transfer Elements with a lambda expression as follows:

Set<Person> roster Set Lambda =

Transfer Elements(roster, () -> { return new HashSet<>(); });

You can use a constructor reference in place of the lambda expression as follows:

Set<Person> roster Set = transfer Elements(roster, HashSet::new);

The Java compiler infers that you want to create a HashSet collection that contains elements of type Person. Alternatively, you can specify this as follows:

Set<Person> roster Set = transfer Elements(roster, HashSet<Person>::new);

\*STREAM OPERATIONS

Stream operations have two fundamental characteristics that make them very different from collection operations:

* **Pipelining:** Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as *laziness* and *short-circuiting*, which we explore later.
* **Internal iteration:** In contrast to collections, which are iterated explicitly (*external iteration*), stream operations do the iteration behind the scenes for you.
* Tream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example filter, map and sorted are intermediate operations whereas for Each is a terminal operation. For a full list of all available stream operations see the Stream Javadoc. Such a chain of stream operations as seen in the example above is also known as *operation pipeline*.
* Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both *non-interfering* and *stateless*. What does that mean?
* A function in non interfering when it does not modify the underlying data source of the stream, e.g. in the above example no lambda expression does modify myList by adding or removing elements from the collection.
* A function is stateless when the execution of the operation is deterministic, e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

### Different kind of streams

### Streams can be created from various data sources, especially collections. Lists and Sets support new methods stream() and parallel Stream() to either create a sequential or a parallel stream. Parallel streams are capable of operating on multiple threads and will be covered in a later section of this tutorial.

1. IMPLEMENTATION

TASK 1 : Define a class Monitored Data with 3 fields: start time, end time and activity as string. Read the data from the file Activity.txt using streams and split each line in 3 parts: start time, end time and activity label, and create a list of objects of type Monitored Data.

TASK 2 : Count the distinct days that appear in the monitoring day.

TASK 3 : Count how many times each activity has appeared over the entire monitoring period.

* Return a structure of type Map < String, Integer > representing the mapping of each distinct activity to the number of occurrences in the log; therefore the key of the Map will represent a String object corresponding to the activity name, and the value will represent an Integer object corresponding to the number of times the activity has appeared over the monitoring period.

TASK 4 : Count for how many times each activity has appeared for each day over the monitoring period.

• Return a structure of type Map < Integer, Map < String, Integer > > that contains the activity count for each day of the log; therefore the key of the Map will represent an Integer object corresponding to the number of the monitored day, and the value will represent a Map<String, Integer> (in this map the key which is a String object corresponds to the name of the activity, and the value which is an Integer object corresponds to the number of times that activity has appeared within the day

TASK 5 : For each activity compute the entire duration over the monitoring period.

•Return a structure of type Map<String, Local Time> in which the key of the Map will represent a String object corresponding to the activity name, and the value will represent a Local Time object corresponding to the entire duration of the activity over the monitoring period.

TASK 6 :

Filter the activities that have more than90% of the monitoring records with duration less than 5 minutes, collect the results in a List<String> containing only the distinct activity names and return the list.

1. BIBLIOGRAPHY :

https://archive.ics.uci.edu/ml/datasets/Activities+of+Daily+Living+(ADLs)+Recognition+Using+Binary+Sensors[3] https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html[4] https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html[5] https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html[6] <https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

Programming Techniques – Lectures of prof. Ioan SALOMIE