

Due

Saturday by 11:59pm

Points

2

Available

Nov 6 at 10am - Nov 13 at 11:59pm

Lab 6 - Maps & Code-jutsu



Home

(<https://northeastern.instructure.com/courses/123246/pages/home>)



Modules

(<https://northeastern.instructure.com/courses/123246/modules>)



Practice with Maps and Art of Design 1(Green Belt Code-justu)

Applying Concepts You've Learned

1.1 Introduction

The purpose of this lab is to give you practice with the Map collections (covered before Exam 1) and some Code-jutsu techniques we are covering in lecture this week (e.g. factory methods and double dispatch)

1.2 What to do

There are two parts to the lab. Both parts are relatively simple and straightforward, but there are a number of moving pieces (especially for double dispatch) so do not wait until the last minute to

work on the lab.

NO UML IS REQUIRED for either problem.

NO Unit Test submission is required for either problem. Of course we suggest you write tests to validate your code, but you will not be graded on producing tests for this lab.

Problem 1

Implement a utility class named **Analytics** in a package named **frequency** that provides a public class method (static) with the following signature:

```
public static Map<String, Double> wordFrequency(String message)
```

This method takes a String message as its input parameter and returns a frequency count of the words in the message. The return value is a Map that contains the relative frequency of the times that a given word appears in the message. Relative frequency is computed as the number of occurrences of the word divided by the total number of words in the message.

For example, given the string message:

```
"Really? Like, really? I do need another cookie to cook?"
```

The frequency Map should contain the following information as key-value pairs:

```
{NEED=0.1, REALLY=0.2, LIKE=0.1, COOKIE=0.1, COOK=0.1, I=0.1, DO=0.1, TO=0.1, ANOTHER=0.1}
```

Note the Map keys are in uppercase.

Tokenize the message using blanks as delimiters between words. In the example above, there are 10 words. Also: treat the message as case-insensitive and remove punctuation.

If the message string is null or an empty string, return null rather than a Map instance.

For this lab, you'll likely find the String.replaceAll() method useful. And, if you're not a Regex (regular expressions) expert (I certainly am not), the Java documentation on using the Pattern class will make your life much, much, much easier. Find the documentation here:

<https://docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern.html> 

(<https://docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern.html>)

Problem 2

You'll practice some double-dispatch "code-jutsu" for this part of the lab. Create all of the required assets for this problem in a package named **doubledispatch**.

Given the following interfaces for planets and space explorers

```
public interface IPlanet {
    void accept(ISpaceExplorer explorer);
}
```

```
public interface ISpaceExplorer {
    void visit(Mercury mercury);

    void visit(Mars mars);

    void visit(Venus venus);

    default void visit(IPlanet aPlanet) {
        SimulationBuilder.addToLog("Visiting an unknown planet");
    }
}
```

And this code for a concrete class that implements **ISpaceExplorer**:

```
public class LifeExplorer implements ISpaceExplorer {
    @Override
    public void visit(Mercury mercury) {
        SimulationBuilder.addToLog("Landing on Mercury...exploring for life");
    }


    @Override
    public void visit(Mars mars) {
        SimulationBuilder.addToLog("Landing on Mars...exploring for life");
    }

    @Override
    public void visit(Venus venus) {
        SimulationBuilder.addToLog("Landing on Venus...exploring life");
    }
}
```

Create the rest of a simple simulation that allows us to send different explorers to various planets.

- Currently, the 3 known planets are: **Mars**, **Mercury** and **Venus**. However, the code you develop must be able to accommodate new planets as long as they adhere to the **IPlanet** protocol/interface.
- Currently, there are 2 types of space explorers: **LifeExplorer** and **TerrainExplorer**. However, the code you develop must be able to accommodate new explorer craft as long as they adhere to the **ISpaceExplorer** protocol/interface.
- The key behavior for planets to achieve when they **accept()** an explorer is to have the current explorer **visit()** the planet.

You must provide a concrete class that is a simple "simulation engine". Name this class **SimulationBuilder** and provide the following methods:

```
public static IPlanet createPlanet(String name)  https://docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern.html
```

The `createPlanet` class method is a factory method that returns an instance of the concrete planet requested by (case-insensitive) name. For example, `createPlanet("mars")` returns a new instance of the **Mars** planet. If the planet requested is unknown to your factory method, return null.

```
public static ISpaceExplorer createExplorer(String name)
```

Likewise, `createExplorer("LifeExplorer")` creates an instance of one of the concrete space explorers. If the explorer requested is unknown to your factory, return null.

```
public static void addToLog(String message)
```

```
public static List<String> getSimulationLog()
```

These two methods work in tandem. The `getSimulationLog()` method returns a List of String that is the "captain's log" of all planetary exploration. In practical terms, each log entry is a element in the **List<String>**. See the example code given to you above for the **LifeExplorer**. You should call `addToLog()` when an explorer visits a planet and allow us to inspect your log via `getSimulationLog()` when the simulation is complete.

Note: You are free to store the log in any type of internal data structure you wish, but you **must** return the log to external clients as a **List<String>** with each log entry as a separate list element, as specified by the signature above.

Notes

Be sure to use access modifiers, private, *default* (no keyword), protected, and public appropriately.

Include JavaDoc for your classes. For this lab, you can leave your classes relatively "sparse"; no `.equals()` or `.toString()` (or even constructors, really) are required. Focus on applying the concepts and using code-jutsu to make things work.

NO UML or unit tests required for submission.

Lab 4 - rubric

Full Credit - Pass	Partial Credit - Pass	No Credit
2 pts	1 pt	0 pt

Both solutions pass all automated tests.	Solutions pass 50% of the automated tests	Solutions pass < 50% of the automated tests. Or assignment not submitted
--	---	---