Make A Movie

Submission Date: 11th of May 11:59 PM

1 Introduction

1.1 Submission

Submit a **folder** that is **only** containing your Java source files (*.java) to the course's Homework folder.

Full path: **F:\COURSES\UGRADS\COMP130\Homework\**

Note: MAVA 130 students, please submit to the folder under the COMP 130 directory.

Please use the following naming convention for the submitted folders:

YourPSLetter_CourseCode_Surname_Name_HWNumber_Semester

Example folder names:

- **PSA_COMP130_Surname_Name_HW4_S18**
- **PSB_MAVA130_Surname_Name_HW4_S18**

Additional notes:

- Using the naming convention properly is important, failing to do so may be **penalized**.
- **Do not** use Turkish characters when naming files or folders.
- Submissions with unidentifiable names will be **disregarded** completely. (ex. ”homework1”, ”project” etc.)
- Please write your name into the Java source file where it is asked for.

1.2 Academic Honesty

Koç University's *Statement on Academic Honesty* holds for all the homeworks given in this course. Failing to comply with the statement will be penalized accordingly. If you are unsure whether your action violates the code of conduct, please consult with your instructor.

1.3 Aim of the Project

In this project you are expected to create a Command Line Interface (CLI) that will allow the user to create animations. This project also expects you to display the created animation to the user. Your program must be able to save and load animation files.

1.4 Given Code

This part is **optional** but advised as it will allow you to understand the given partitions of the code better. **Do not** change anything in the code if it is indicated to you with a comment. The code given to you has something called **JavaDoc** comments above all the methods. These comments allow you to view various information about the method when you mouse over the name of the method. Below are the methods given to you in the code with their explanation.

1.4.1 Given Methods

-
  ```
  void methodWithNoArguments()
  ```

  This is an example method with no arguments.

-
  ```
  int methodWithIntegerValue()
  ```

  This is an example method that returns an **int**.

-
  ```
  void methodWithArguments(int arg1, int arg2)
  ```

  This method constructs the roads and the crossing and adds them to the screen. See below for how the roads are created individually.

1.4.2 Given Constants

Constants are given at the bottom of the project. All constants provide **JavaDoc** comments above them. Please read these to understand what constant is used for what. **Do not** use another variable or a static value for something if there is a constant variable defined for that purpose.

1.5 Further Questions

For further questions **about the project** you may contact **Kaan Yıldırım** at [kyildirim14@ku.edu.tr]. Note that it may take up to 24 hours before you receive a response so please ask your questions **before** it is too late. No questions will be answered when there is **less than two days** left for the submission.

2 Project Tasks

The project tasks are divided into subsections to make it easier to understand and implement. You are strictly advised to follow the given order of tasks, however should you choose not to follow the tasks in order, it is possible to implement the project in any order.

2.1 Creating a Command Line Interface

A Command Line Interface (CLI) is a means of interacting with a program where the user issues commands in the form of successive lines of texts. In this task you are expected to create a simple CLI that needs to be capable of recognizing the following commands:

- create: Creates a new project.
- open: Opens an existing project.
- settitle: Sets the title for the project.
- setgrammar: Sets the grammar for the project.
- addscene: Adds a scene to the current project.
- removescene: Removes a scene from the current project.
- listscenes: Lists all the scenes in the project.
- play: Plays the scenes in order.
- save: Saves the project to a file.
- close: Closes a project.
- quit: Quits the program.
- help: Display additional information.

2.1.1 Building a Basic CLI

In its most basic form a CLI reads the user input as a string and parses it to call different commands according to the input text. You may want to start by implementing a simple code that asks for user input continuously. Once you have accomplished this, try detecting the input text of ”quit”. Modify your code so that it asks for a new input from the user as long as the input is not ”quit”.

Advice: You may want to echo (print back to the screen) the input you receive from the user to clearly see that you can read the input.

2.1.2 Detecting All Commands

Now that you have a basic CLI that is capable of recognizing a specific input, you can proceed by detecting all the commands shown in Section 2.1. You may modify the names of the commands, however you are required to state all the changes when the "help" command is called. Note that as you implement the tasks below, you are expected to link them to these commands.
Advice: You can implement temporary methods for all the commands you have detected and for now print a text that states the command is executed properly.

2.1.3 Distinguishing Commands From Output

If you have implemented the suggested way of checking your CLI, you may have noticed that the input and the output lines are mixing up. To prevent this confusion you are expected to print a constant string before taking an input from the user. This is the string defined as **CLI_INPUT_CONST** in the given code.

2.2 Creating a New Project

In this task you will create a new project from scratch. To be able to achieve this, you need to first understand the animation file format. Below, you can find an example valid project file.

```
Title New Movie
Images ball tree
Grammar time color image from to
Scenes
For 5 seconds red ball from left to right
For 1500 milliseconds green tree from bottom to top
```

The first line always starts with the keyword **Title** and is followed by the title of the animation. Note that the animation title is not necessarily a single word. The second line always starts with the keyword **Images** and is followed by the name of all image files that are going to be used in the animation. The third line always starts with the keyword **Grammar** and is followed by the grammar specification of the animation. The fourth line is the keyword **Scenes** which is followed by scene descriptions, one per line. The scene descriptions are required to follow the grammar.

The grammar allows the user to change the way they want to describe the scene. As a real world example, you can see that Turkish and English have different grammars. Below is a list of all grammar descriptors defined for this project.

- color: A single word describing a color. Example: red
- image: A single word that is the name of a image file. Example: plan
- from: A two word descriptor that starts with the keyword **from** and is followed by a location. Example: from left

- to: A two word descriptor that starts with the keyword **to** and is followed by a location. Example: to right
- time: A three word descriptor that starts with the keyword **for** and is followed by an integer value which is followed by either the word **second(s)** or **millisecond(s)**.

Note: The locations described in the **from** and **to** descriptors can take the following values: left, right, top, bottom, center.

2.2.1 Creating a New Project, Actually.

In this task you are required to start implementing the commands that your CLI can detect. Start with the **create** command, which is expected to ask the user for a new project name. (Note that this is different from the project title.) Once you get this name for the project, modify your CLI so that it displays this name before the **CLI_INPUT_CONST** seperated with a '@' character.

Example for project name Example: **Example@MakeAMovie ->**

Now that you have created a new project (not yet, really) you may also implement the **settitle** command. This command will ask the user for a title, later on you will be expected to save these to a file so please make sure that you do not lose any information the user inputs.

Once you achieved getting the title from the user, you may proceed with the **setgrammar** command. This command will take the ordering of the grammar descriptors defined in Section 2.2 from the user. You will later on use this to parse the animation scenes.

Hint: You may want to save the grammar to either a string or an array of strings.

As you have acquired the grammar specification, you may now allow the user to start inputting new scenes. Implement the **addscene** command so that the user can add a scene by describing it according to the grammar that they defined. This command will read a **single** line of user input.

After an excessive usage of **addscene** command a user may lose track of the scenes added, and may want to see what is currently in the project. At this point you are expected to implement the **listscenes** command that will list all the current scenes in order, starting their enumeration from 1. (Meaning the first scene is the scene number 1)

Once seeing all the scenes present in the project, a user may want to remove a scene from the animation. Notice how you enumerated the scenes in the **listscenes** command, you are now required to implement **removescene** command. This command will ask the user to input a scene number, and will remove it from the project. After a removal, the enumeration of the scenes may change if the scene removed was not the last scene. (In an animation with four scenes, the removal of second scene would make the third the second and the fourth the third.)

2.2.2 Task 2 of Task Group 2

Second task. Or *italic*.

2.2.3 Task 3 of Task Group 2

Third task. Or underlined.

2.3 Task Group 3

Third Task Group. Another way of better expressing your task is the inclusion of colors.

2.3.1 Task 1 of Task Group 3

First task. This text is colored red.

2.3.2 Task 2 of Task Group 3

Second task. The word **blue** is both colored and **bold** in this example.

2.3.3 Task 3 of Task Group 3

Third task. Please use coloring **sparingly** as it is may become hard to read.

2.4 End of Project

Your project ends here. You may continue to tinker with the code to implement any desired features and discuss them with your section leader. Below in the **Section 3** are further tasks for you to implement if you are willing to continue practicing the topics. However, **do not** include any additional features that you implement after this point in to your submission.

Final Warning: Do not include anything beyond this point to your submission. Points may be deducted from your grade as Section 3 alters the normal behavior of the simulation.

3 Further Tasks

Tasks described in this section are **not** included to your project, but are provided for studying the topics further. **Do not** submit your project with any of these tasks completed. You will only be graded for the tasks in **Section 2**. Also note that tasks below are meant to be implemented on their own but may function together as well.

3.1 Further Task 1

Any further tasks that may be used by the students to study further and improve their skills is described here.

3.1.1 Task 1 of Further Task 1

First task.

3.1.2 Task 2 of Further Task 1

Second task.

3.1.3 Task 3 of Further Task 1

Second task.

3.1.4 Further Task 2

Alternatively further tasks may be short enough to not include task. In this case just explain the task in this section.

3.1.5 Further Task 3

Third further task.