CS 1302 – HW 6

*Text Files*

Contents

[1 Problem 1 – Martian Manager App, Ver 2 1](#_Toc192340773)

[1.1 Overview 1](#_Toc192340774)

[1.2 Text File Format 3](#_Toc192340775)

[1.3 Requirement: *writeMartiansFile* Method 3](#_Toc192340776)

[1.4 Requirement: *readMartiansFile* Method 3](#_Toc192340777)

[1.5 Requirement: *Video* 4](#_Toc192340778)

[2 Problem 2 5](#_Toc192340779)

[2.1 Overview 5](#_Toc192340780)

[2.2 Details 5](#_Toc192340781)

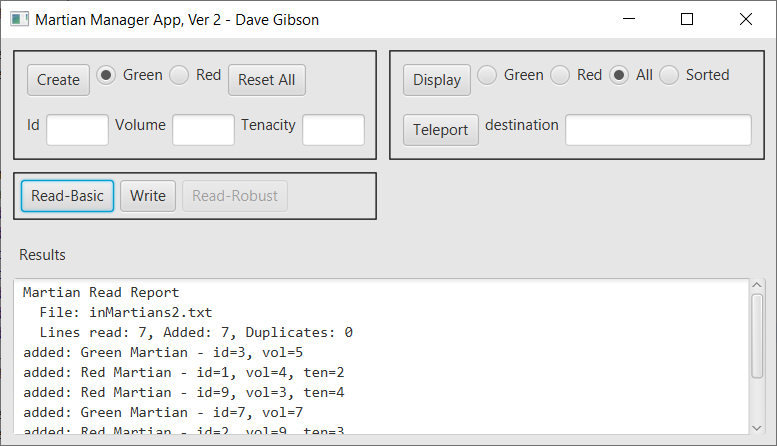
[2.3 Suggested Steps 6](#_Toc192340782)

[3 Submission Requirements 7](#_Toc192340783)

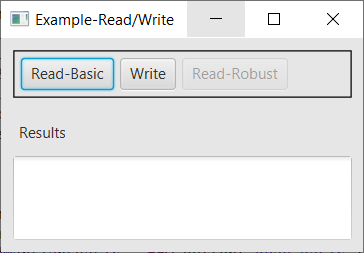
# Problem 1 – Martian Manager App, Ver 2

## Overview

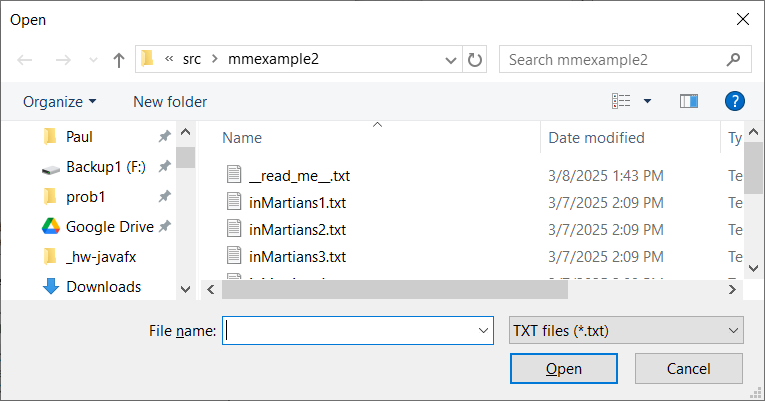
(60 Points) You will add a component (highlighted in red in the figure below) to the Martian Manager App to allow the user to read a text file of marians, populating the app. The user can also write the martians in memory to disk.



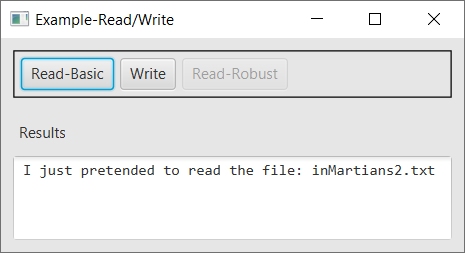
You have been provided code that produces:



When “Read-Basic” is pressed, the standard File Open Dialog is displayed where the user can choose a file:



When a file is choosen, the code displays:



When “Write” is pressed, similarly, a File Output Dialog is displayed and a sample message is displayed. Thus, you will do the following:

1. Create a copy of your Martian Manager App, ver 1. This will be your ver 2.
2. Integrate the code provided into your code
3. Write two methods: *readMartiansFile* and *writeMartiansFile* (stubs provided).

## Text File Format

The format for the martians text file is shown below (left column). Thus, a *GreenMartian* is represented with a ‘G’ followed by the *Id* and optionally, the *Volume.* A *RedMartian* is represented by a ‘R’ followed by the *Id,* optionally a *Volume,* and a*Tenacity.* An example file is shown below (right column). So, we can see that each *Martian* is on a separate line. For example, the first line in the example is a *GreenMartian* with id=3 and volume=5. The second line is a *RedMartian* with id=1, volume=4, and tenacity=2.

|  |  |  |
| --- | --- | --- |
| **Format, either of these is valid** | **Key** | **Example** |
| G I [V]  R I [V] T | |  |  |  | | --- | --- | --- | | **Code** | **Type** | **Meaning** | | G | Character | Green Martian | | R | Character | Red Martian | | I | Integer | Id | | V | Integer | Volume | | T | Integer | Tenacity | | G 3 5  R 1 4 2  G 5  R 9 4  G 7 7  R 2 9 3  G 8 2  G 4 5 |

## Requirement: *writeMartiansFile* Method

The sample code provides this method stub:

**protected** **void** writeMartiansFile(File file) **throws** FileNotFoundException {

Write this method. This method should loop through the martians in the *MartianManager* and write the martians tothe *file* object that is passed as an argument, using the format [described above](#_Text_File_Format_1); however, the volume should always be present. In other words, these are the valid formats for writing:

G I V

R I V T

The output after a write should show a message like this, which shows the file name and the number of martians writen:

File: out1.txt, Num Martians written:7

Hint: What *MartianManager* methods will you need to loop over the *Martians*? Reference the Javadoc for HW 5 (where the *MartianManager* and related classes were originally written).

## Requirement: *readMartiansFile* Method

1. The sample code provides this method stub:

**protected** **void** readMartiansFile(File file) **throws** FileNotFoundException {

This method accepts a file object, reads the martians in the file, add the martians to the martian manager (there may already be martians in there, just add to it), and display the added martians along with some statistics.

1. Note the following:
2. When you read martian data, you use it to create either a *RedMartian* or a *GreenMartian*. Note that there are 4 cases (which correspond with the 4 constructors). Then, you will attempt to add the martian to the *MartianManager.* The *MartianManager*’s *addMartian* method will only add a martian, if there doesn’t already exist a martian with the same *id.* The *addMartian* method returns *true* if the martian was added and *false* otherwise. Your code will keep track of how many martians were successfully added and how many were not added because they already exist.
3. As lines are being read, martians created, and attempting to add them to the *MartianManager*, your code should keep track of:
4. *numLinesRead* – the total number of lines read (valid or ill-formed)
5. *numSuccessfullyAdded* – the total number of martians successfully added to the *MartianManager*
6. *numDuplicates* – the total number of martians that could not be added to the *MartianManager* because they are duplicates, *i.e. already exist.*

Note: *numLinesRead* = *numSuccessfullyAdded* + *numDuplicates*

1. The output should be similar to what is shown in an example below. In this example, the duplicates

|  |  |
| --- | --- |
| **Sample File** | **Output** |
| G 3 5  R 1 4 2  R 9 3 4  G 7 7  R 9 9 3  G 8 2  G 7 5 | Martian Read Report  File: inMartiansBasicDups.txt  Lines read: 7, Added: 5, Duplicates: 2  added: Green Martian - id=3, vol=5  added: Red Martian - id=1, vol=4, ten=2  added: Red Martian - id=9, vol=3, ten=4  added: Green Martian - id=7, vol=7  \*\*\*not added, id=9 already exists  added: Green Martian - id=8, vol=2  \*\*\*not added, id=7 already exists |

1. You have been provided 4 test input files: *inMartians1.txt, inMartians2.txt, inMartiansBasicDups.txt* (has a few duplicates), and *inMartiansBasicNoDups.txt.*

## Requirement: *Video*

Create a video in Blazeview demoing:

1. Launch the app and immediately read in: *inMartians2.txt.*
2. Use “Display” to display “All” the martians.
3. Add two martians manually.
4. Use “Display” to display “All” the martians (there should be 2 more).
5. Write the martians to a file, *out1.txt*.
6. Open the output text file just created and display the contents.
7. Press, “Reset All”
8. Read in the output file (*out1.txt*) you just created.

# Problem 2

## Overview

(40 points) For this problem you will read a file of numbers adding them as you go, but skipping some of the number as indicated by a code in the file.

In the provided *hw06\_prob2.zip,* you will find a *prob2* folders, a *NumberAdder* class, and some sample test files. In *NumberAdder,* you will write the *getSum* method, which returns the sum of the integers read from a file. However, some numbers are skipped. When you encounter a line whose first character is ‘s’, then an integer will follow it. The integer tells you how many of the next consecutive numbers should be skipped. For Example 1 below, the yellow numbers should be added to produce 60, skipping the values as indicated. As shown in Example 3, a skip of zero doesn’t skip anything, effectively ignoring the skip. Example 4 shows a skip beyond the length of the file which should just skip to the end of the file, ignoring the invalid skips. Finaly, Example 5 shows overlapping (or embedded) skips. The embedded skip, s5, is effectively ignored, meaning it is treated as any other skipped value.

|  |  |  |  |
| --- | --- | --- | --- |
| Example 1 – Sum=60 | Example 2 – Sum=90 | Example 3 – Sum=10 | Example 5 – Sum=10 |
| E:\Data-Classes\CS 1302 - Programming 2\homework\Fall, 2015\hw7\a.jpg | s5  2  43  3  6  9  12  4  56  s2  10  4  8  s2  1  5  s2  4  2  10 | 2  3  s0  4  1 | 2  3  s3  4  s5  2  4  1 |
| Example 4 – Sum=5 |
| 2  3  s7  4  1 |

## Details

**Drag the *prob2* package into project where you have Martian Manager App, ver 2.**

In the *prob2* package you will find a *NumberAdder* class as shown below. Write the *getSum* method (stub provided). This is also the test class as *main* calls *getSum* 10 times, passing it a different text file each time.

**public** **class** NumberAdder {

**private** **static** **final** String ***IN\_FILE\_PATH*** = "src/prob2/";

// I have supplied test files: t1.txt, t2.txt, ..., t10.txt

// You will need to verify the expected output by hand to ensure

// that your code gives the correct value.

**static** String[] *inFileNames* = {"t1.txt", "t2.txt", "t3.txt", "t4.txt", "t5.txt",

"t6.txt", "t7.txt", "t8.txt", "t9.txt", "t10.txt"};

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

// Comment out most of these as you test/debug.

File file;

file = **new** File(***IN\_FILE\_PATH*** + *inFileNames*[0]);

System.***out***.println("Sum=" + *getSum*(file));

file = **new** File(***IN\_FILE\_PATH*** + *inFileNames*[1]);

System.***out***.println("Sum=" + *getSum*(file));

file = **new** File(***IN\_FILE\_PATH*** + *inFileNames*[2]);

...

}

/\*\*

\* YOU WRITE THIS METHOD

\*/

**public** **static** **int** getSum(File file) {

**return** Integer.***MAX\_VALUE***;

}

...

}

## Suggested Steps

If you need help getting started, I’d use baby steps:

1. Create a simpler text file to start with:
   1. Write a text file named: *test.txt* with these values: 1 2 a 3 4 5
   2. Comment out all code in *main.*
   3. Add these lines to *main*:

File file;

file = **new** File(***IN\_FILE\_PATH*** + *test.txt*);

System.***out***.println("Sum=" + *getSum*(file));

1. Write *getSum* so that it simply adds up all the numbers in the file. Thus, you need to read each token as a string, and then ask if it is an integer. If it is, add it. If it is not, then skip it. See Chapter 7, Section 8 to see how to determine if a string is an inteter.
2. Run, and verify that the output is 15. If not, debug and fix your code.
3. Modify *test.txt* so that the values are: 1 2 a b 3 c 4 5.
4. Run and verify that the output is 15. If not, debug and fix your code.
5. Add a skip to *test.txt*: 1 2 s2 3 4 5. Thus, the expected result is 8 once you modify *getSum.*
6. Modify *getSum* to detect and obey the skip:
   1. When you detect that a token is not an integer, then you can assume it is a skip, *i.e.* the first character is an ‘s’. So, write code to strip off the strip length. See Chapter 1, Appendix 1.4 to see how to use the substring method.
   2. Put an inner *for* loop (*i.e.* inside the while(scanner.hasNext()) loop) that goes from 0 to less than the strip length. Each time it should simply read the next token and do nothing with it.
   3. Run and verify that the output is 8. If not, debug and fix your code.
7. Comment out the two lines in *main* and then uncomment the first test case:

File file;

file = **new** File(***IN\_FILE\_PATH*** + *inFileNames*[0]);

System.***out***.println("Sum=" + *getSum*(file));

1. Run and verify the output. If incorrect, then debug and fix your code.
2. Repeat steps 8 and 9 with the second test case, then the third, *etc.*

# Submission Requirements

Checklist:

|  |  |  |
| --- | --- | --- |
|  | **Complete?** | **Requirement** |
| 1. |  | Your *application* and *prob2* folders are zipped into a file name: *hw6\_yourLastName.zip.*   * See Lab 2, Stage 9 for exact instructions. * Do not zip your workspace folder * Do not zip your *src* folder. * Do not zip just the java files * Do zip just your *application* and *prob2* folders |
| 2. |  | Video complete |
| 3. |  | Submit video and zip file in the *hw 6* dropbox on Blazeview by the deadline. |