PRA2003 Programming Skills
Week 3 Tasks

File I/O and Exceptions

Cameron Browne & Andreea Grigoriu
13 November 2019

This week, you will explore reading from and writing to files, and making and catching your own custom exceptions.

Task 3.1. Understanding Exceptions

Study the `Exceptions.java` file and make sure you understand how you can build user-defined exceptions. Make sure that you understand how the exception is constructed.

Switch the order of the first two catch blocks in main() (lines 42-45 and 47–50) and try to compile and run the code. What happens and why?

Task 3.2. Runtime Exceptions

Write a short program that declares a reference to an object (e.g. a List reference or an array reference) and initialises it to null. Then, try to run something based on this reference (e.g. a call to List's size() method or checking the array length) using this reference. Run the program to see what happens.

Comment out the code above and create an array of 3 integers. Try to assign a value for the element at index 3. Run the program to see what happens.

Uncomment the code above and put them both into a `try` block. Add two catch cases, one for a `NullPointerException` and one for an

`ArrayIndexOutOfBoundsException`. In each block, print the name of the type of the exception caught.

Run the program to see what happens. Then switch the order of the catch cases and run the program again. What happens and why?

Task 3.3. Heat Filter

Using the code `FileReader.java` provided with the slides as a base, write a program that filters out all the lines not containing the word "heat" from a file pepper.txt. Save the file into your project's folder (not the src folder where you save your code, but one level up).

Your program must read each line from the source file (pepper.txt), and if a line contains the word "heat", then it should write it to the output file (pepper-out.txt). This way the filtered lines appear in the same order as in the original file.

You may need to modify the `FileReader` constructor to take a second parameter describing the output file name. Then create an object of this type and run its parse method to perform the filtering.

Hint: Think of how can you check if a line (which is represented by a String) contains a specific word. Which String method can you use? Check "String API" to see the available `String` methods if needed, and discuss with each other and your instructor.

Assignment 3 **Due 23:59 on Monday 19 November**

In this assignment, you will add a constructor to World that will build the world from a file and add simple gravity to the game.

First, create a new class (in its own file) called `Tests` . Here you will test your static test methods. In this case, we will use it for testing whether new game worlds are created properly (or if there are errors in the files provided in the input). Your main method invokes them, for example by using a statement like this: `Tests.testLoadFile();`

Create a custom exception, like the one in previous tasks, called `BadFileFormatException` (use a separate file). The constructor has three parameters: a string, and two integers. The string contains an error message explaining the problem. The integers represent the row and (possibly) column of the problem. Implement a `toString()` method that returns a string containing the exception's name, the error message, and the row and (if necessary) column it occurred on.

Create *another* constructor for the World class which takes as a parameter a single string: the name of the input file which represents the initial board. The constructor parses the file to create the world as specified from the specific format below. If the format of the file is incorrect the constructor throws a `BadFileFormatException` with the information as specified above. Write a test method that creates a new world by using this new constructor for the example file. Print out the board using the world's toString() method to be certain that it is being read correctly.

Then, create three other board files with illegal formats to make sure that the exception is being thrown in each case (you can have invalid characters, wrong number of rows/columns, not enough emeralds, etc.).

Finally, add simple gravity to the game, by implementing the following method (in your World class) and *invoking it once at the end of applyMove* (after the player and alien have moved) returning the appropriate value for a loss if a massive object fell on the player:

```
// Returns true if an object with mass falls on
the player,
// false otherwise

private boolean applyGravity(int exceptRow, int
exceptCol) {
  boolean fellOnPlayer = false;
  // flag to indicate if an object with mass fell
on the player

  // apply gravity from bottom-up! So, if there
are two rocks,
  // (one on top of the other), they both fall...
  // start at rows-2 because gravity does not
apply to
  // objects on bottom row
```

```
  for (int r = rows-2; r >= 0; r--) {
    for (int c = 0; c < cols; c++) {
  /**
  * ... stuff ... (remember to modify
fellOnPlayer if necessary)
  */
  }
 }

return fellOnPlayer;
}
```

This method moves all the objects *with mass* one single row downward (but not off the board!), as long as what is below them *is vulnerable* (i.e. open space, diamonds, players, and aliens). If a valid row and column are passed as arguments, then that one location should not be affected by gravity: this exception is needed for when the player enters a square with dirt or an emerald that has a rock on top of it; that rock should not immediately fall on the player (boom!) it only does so the turn after if the player doesn't move! The alien is not as strong as the player, so this exception does not apply to it. In other words, if the alien enters a square with dirt or an emerald that has a rock on top of it, then that rock will indeed fall on the alien immediately after it eats the dirt or emerald.

File Format

The first line contains a number of rows, **R**. The second line contains a number of columns, **C**. The third line contains a number of emeralds remaining. There should be exactly **R** lines following these first three, which should contain exactly **C** characters each. There should be exactly one player, and total worth of all the emeralds and diamonds should be greater than or equal to the emeralds remaining. (There can be multiple aliens, but your program need not handle moving more than one of them for now.) See below for an example of a correctly-formatted file.

Notes: You may assume that the first three lines will always be properly formatted. Use Integer.parseInt(...) to convert from string to integer, and String.charAt(...) to get individual characters in each line.

Contents of a correctly formatted example file example.txt :

```
10
```

```
10
16
e##.p#d###
e##e#####d
.###re####
####.##..#
.###d#####
e#####r###
rd#####ee#
##########
##e##a####
#r######.#
```

In this example world: After the player's first move, the two emeralds at the top-left should each fall down one row, and the rock at (row 2, col 4) should also fall down one row. After the second move, the falling rock at (row 3, col 4) should squash the diamond. The rock should not fall anymore until the player or the alien removes the dirt below it.

Honour code, coding style and deliverable

Please refer to Infosheet → Paragraph 7 → Honour policy.

You are welcome to expand your program to do extra things but they are not mandatory. Do not deviate from the specific requirements of the assignment!

If you use some code found online (or anywhere else, e.g. StackOverflow), clearly state it (with comment about where you found it) in your .java files. That means mention the author and the www source you found it. Be sure that you read the terms of the licence for the code that you are reusing. Most code that is found openly in the internet has specific licences that you should not violate.

Through courses portal, find Assignments→Assignment 3, submit a zipped file (.zip) that contains your source files (.java) for the assignment.

Please refer to the coding style guidelines uploaded to courses portal (under Course materials) on how to write readable programs.

Hard deadline for submission:

23:59 on Monday 19 Nov. 2018