UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08014 INFORMATICS 1 - OBJECT-ORIENTED PROGRAMMING

Tuesday 8th May 2018

14:30 to 16:30

INSTRUCTIONS TO CANDIDATES

1. **Note that all questions are compulsory.**
2. **Remember that a file that does not compile, does not pass the simple JUnit tests provided, or uses Java packages will get no marks.**
3. **This is an Open Book exam. You may bring in your own material on paper. No electronic devices are permitted.**
4. **CALCULATORS MAY NOT BE USED.**

Convener: I. Simpson
External Examiner: I. Gent

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. In this task you will implement a buffer and functionality to add and remove elements or check the contents of the buffer. In computer science a buffer is used to temporarily store data while it is being moved from one place to the other. The buffer you will implement is able to store data of the type `int`. All its data is kept in an internal integer array which has a maximum capacity specified in the constructor.

   Your task is to implement the subclass `SectionBuffer` of `Buffer`, representing a special kind of `Buffer` which is split in the middle into a front and a back section for storing data. Data can be added or removed to the end of either section. To efficiently modify data and avoid overspill, the buffer keeps track of the last data entry of each section.

   To illustrate its functionality, consider the following examples:

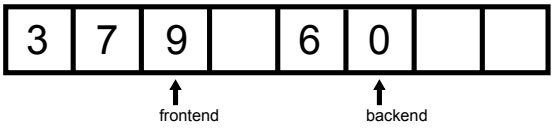

Figure 1: A section buffer of size eight filled with three integer entries in the front and two in the back. *Frontend* and *Backend* markers keep track of the last element of each section.

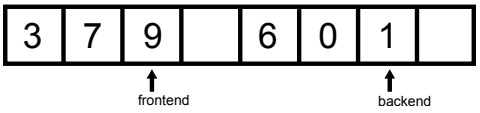

Figure 2: Adding an element to a section, moves its end marker forward. For example, when adding the integer 1 to the back section.

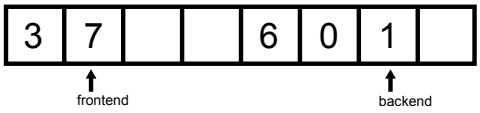

Figure 3: Removing an element from a section, moves its end marker backwards. For example when removing the last element of the front section.

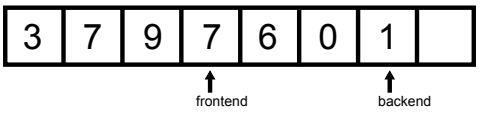

Figure 4: If a section end marker reaches the end of its assigned section, the corresponding section is considered to be full. Here, the front section is full.

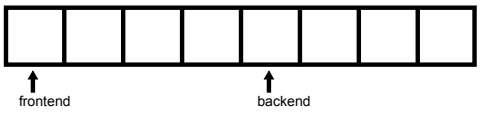

Figure 5: When the section buffer is empty, both markers point to the first free space of their section.

QUESTION CONTINUES ON NEXT PAGE

QUESTION CONTINUED FROM PREVIOUS PAGE

From the exam template directory, please use all of the following files to answer this question (if you are using Eclipse, make sure you import ALL of them):

- Buffer.java
  Work with the `Buffer` class provided in the exam's template folder, NOT the `Buffer` class from the Java API!
- SectionBufferBasicTest.java

Please execute the following steps for your implementation of `SectionBuffer`:

(a) Define the class `SectionBuffer`, extending `Buffer`. It should have four instance variables:

- a private variable `frontEnd` of type `int` representing the end marker of the front section
- a private variable `backEnd` of type `int` representing the end marker of the back section
- a private variable `frontElementCount` of type `int` representing how many elements are currently in the front section
- a private variable `backElementCount` of type `int` representing how many elements are currently in the back section

[*5 marks*]

(b) Write two public constructors for `SectionBuffer`. Your first constructor must take an `int` which is the buffer's total capacity. Invoke the `Buffer` constructor as appropriate. It will take care of the array initialisation and checks if the given capacity is in a valid range. Make sure that `SectionBuffer`'s marker instance members are initialised to zero for the front and half the initial capacity for the back. **If the capacity is uneven, the back section will be one element larger than the front section.** Also, initialise corresponding element counts to zero.

Your second constructor is a zero argument constructor that invokes your first constructor with a default capacity of ten.

NOTE: The buffer's capacity refers to the length of the underlying array and is constant throughout the execution of the program. The element counts refer to the amount of valid data currently stored in a section and can vary over the course of the program's execution.

[*10 marks*]

(c) Write a public instance method `isFull`, taking a `boolean` argument indicating which section to check and returning a `boolean` value indicating whether the specified section is full. If the given argument is `true`, the front is checked, and if it is `false`, the back is checked. **HINT consider using elementCount here.**

[*5 marks*]

QUESTION CONTINUES ON NEXT PAGE

(d) Write a public instance method `isEmpty`, taking a `boolean` argument indicating which section to check and returning a `boolean` value indicating whether the specified section is empty. If the given argument is `true`, the front is checked, and if it is `false`, the back is checked. **HINT consider using elementCount here.** [*5 marks*]

(e) Override `Buffer`'s instance method `clear`, which empties the entire buffer. Use the same access modifier for your overridden method as is used in `Buffer`. In your implementation, use `Buffer`'s `clear` method to clear out all entries from the array. Then, make sure all markers and element counters are updated accordingly. [*5 marks*]

(f) Write a public instance method `addToBuffer`, taking, in that order, an `int` argument and a `boolean` argument and returning nothing. This method will add the given integer to the end of the buffer section specified with the `boolean` argument. **All input elements can expected to be positive integers or zero.** If the given `boolean` argument is `true`, the integer is added to the front, if it is `false`, the integer is added to the back.

If the specified section is full, the integer argument should be ignored and the following **exact** message should be printed (including dot and newline): `Buffer is full.`

Make sure you move the end markers as described above. Also, keep track of the element counts. [*10 marks*]

(g) Write a public instance method `getFromBuffer`, taking a `boolean` argument and returning an `int`. This method will remove the last integer element from the specified buffer section and return it. **An empty space in the underlying buffer array must be set to -1.** If the given `boolean` argument is `true`, the integer is added to the front, if it is `false`, the integer is added to the back.

If the specified section is empty, -1 should be returned and the following **exact** message should be printed (including dot and newline): `Buffer is empty.`

Make sure you move the end markers as described above. Also, keep track of the `elementCount`. [*10 marks*]

The file you must submit for this question is `SectionBuffer.java`. Before you submit, check that it compiles, passes the basic JUnit tests provided and uses no packages, otherwise it will get 0 marks.

2. In this question you will implement parts of a Minesweeper game.

Game In Minesweeper a set of explosive mines are placed within a square game grid. Mines have the size of a single field in the grid. The player is trying to isolate and disarm some or all of the mines.

Game Grid In your program the game grid is a square two-dimensional array of characters called `grid` where the first index is for y coordinates and the second index for x coordinates. Each character in a game grid is a field in the grid and can either be a '0' (zero) representing an empty field or a 'X' representing a mine.

You must never change the contents of a game grid array.

	0	1	2	3	4
0	0	0	X	0	X
1	0	0	0	0	0
2	0	X	0	X	X
3	X	0	0	0	0
4	0	X	X	0	0

Figure 6: This example grid has a side length of five and 8 mines.

Utils Class A utils class `MineSweeperUtils` is provided and a method `parseGrid` for parsing a game grid from a given filename as well as two overloaded methods `printGrid` for printing a game grid. You can use those methods to retrieve and output data.

Game Data Files Two game data files called `game01.txt` and `game02.txt` with example game grids are provided for you. You can alter them or create your own with a simple text editor to test your program.

MineSweeper Skeleton For this question you are provided with a `MineSweeper` skeleton implementation in file `MineSweeper.java`. The skeleton has method stumps for your implementation where you can fill in your solutions.

MineSweeper main method The skeleton also comes with a main function which is already filled with an example execution of the game. You are not marked on the contents of the main function, so feel free to alter it for testing your code. However, you must make sure that your final solution has *no compiler errors!* You can execute the main method the same way as you are used to from the labs. The required command line argument is the name of a game txt file, e.g. `game01.txt`.

Position Class The provided position class is used to encapsulate x and y coordinates into a single object. You will have to use it for your solutions, however, you must not change it.

QUESTION CONTINUES ON NEXT PAGE

QUESTION CONTINUED FROM PREVIOUS PAGE

JDK Library Classes In this question you will use the collection class `Hashtable` which is familiar from lectures and labs. You may need to look at the JDK documentation for it. Note that the types of your methods will involve `Map`, thus hiding, from clients, which implementation of the `Map` interface is used; your code is expected to create, concretely, `Hashtable`s.

Argument Assumptions You may assume that none of the object arguments to your methods are `null`.

From the exam template directory, please use all of the following files to answer this question (if you are using Eclipse, make sure you import ALL of them):

- MineSweeper.java
- MineSweeperUtils.java
- Position.java
- MineSweeperBasicTest.java
- game01.txt
- game02.txt

For your solution, implement the following three game aspects:

(a) Implement the method `findMines` as given in the skeleton file. This method locates all mines on the game grid. It takes a game grid and returns a `Map` mapping `String`s to `Position`s. Your implementation should find all locations of the mines in the grid, save x and y coordinates in a `Position` object and map a `String` representation of those coordinates to the `Position` object. A `String` representation of a `Position` must look **exactly** like the following for example coordinates x=2 and y=3: `(2,3)`. If no mines are on the grid, an empty `Map` should be returned.

In the case of the sample grid specified in Figure 6, the following mapping should be returned (not necessarily in that order):

```
"(2,0)" -> Position object with coordinates x=2 and y=0
"(4,0)" -> Position object with coordinates x=4 and y=0
"(1,2)" -> Position object with coordinates x=1 and y=2
"(3,2)" -> Position object with coordinates x=3 and y=2
"(4,2)" -> Position object with coordinates x=4 and y=2
"(0,3)" -> Position object with coordinates x=0 and y=3
"(1,4)" -> Position object with coordinates x=1 and y=4
"(2,4)" -> Position object with coordinates x=2 and y=4
```

QUESTION CONTINUES ON NEXT PAGE

Hint: The `Position` class conveniently implements the `toString` method. Remember to use the concrete type Hashtable as return type here. [*15 marks*]

(b) Implement the method `isIsolated` as given in the skeleton file. This method checks if a field in the game grid is isolated from other mines. It takes a game grid and a `Position`, `minePos`, and returns a `boolean` indicating whether the coordinates specified by `minePos` are isolated from other mines or not. Isolation means that for specific coordinates no mine is directly *above, below, left or right* of it. In the case of the sample grid specified above, consider the following examples:

- The mines at coordinates (1,2) and (0,3) are isolated and your method should return `true` in each case.
- The mine at coordinates (3,2) is not isolated because a mine is directly to the right of it, hence your method should return `false`.

For your calculation it does not matter whether `minePos`'s coordinates actually point at a mine in the grid or at an empty field. Only its adjacent fields matter. [*15 marks*]

(c) Implement the method `disarmMines` as given in the skeleton file. This method disarms isolated mines. It gets a game grid and a `Map` of armed mines, called `armedMines`, as produced by method `findMines` from the first part of this question. It returns the number of mines the method was able to disarm. Your implementation should find out which mines are isolated in `armedMines` and remove them from the `Map`. Only isolated mines are allowed to be removed. You may find it convenient to use methods implemented in previous parts of this question.

In the case of the sample mapping produced in part one of this question, the value 4 should be returned and the `armedMines` map should be changed to the following mapping (not necessarily in that order):

```
"(3,2)" -> Position object with coordinates x=3 and y=2
"(4,2)" -> Position object with coordinates x=4 and y=2
"(1,4)" -> Position object with coordinates x=1 and y=4
"(2,4)" -> Position object with coordinates x=2 and y=4
```

[*20 marks*]

QUESTION CONTINUES ON NEXT PAGE

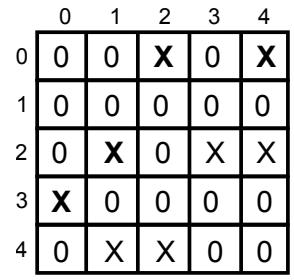

Figure 7: Game grid from Figure 6 with disarmed mines displayed in bold font.

The file you must submit for this question is `MineSweeper.java`. Before you submit, check that it compiles, passes the basic JUnit tests provided and uses no packages, otherwise it will get 0.