

# CS 1027 Computer Science Fundamentals II

## Assignment 1

**Due Date: October 8, 11:55 pm**

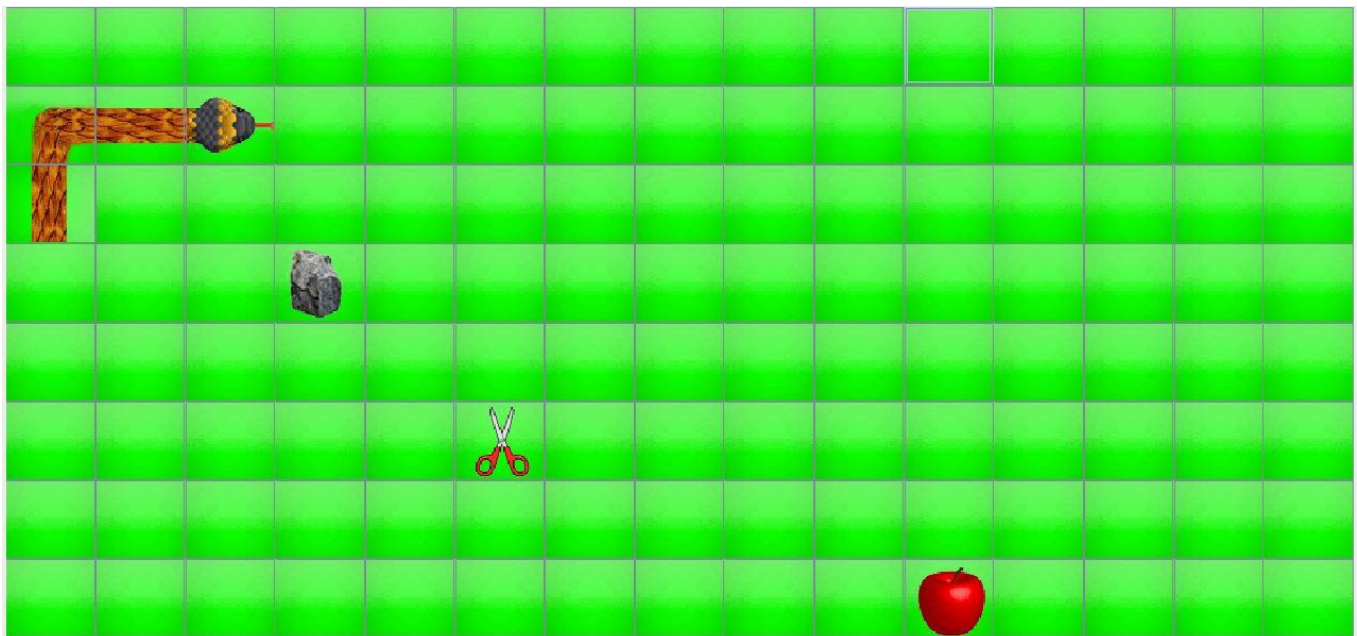
### 1. Learning Outcomes

To gain experience with

- Creating and using classes and objects.
- Using arrays and two-dimensional arrays.
- Reading input from a file.
- Algorithm design and modular design.

### 2. Introduction

The snake game is played on a rectangular grid. There are 4 classes of objects placed on the squares of the grid: rocks, apples, scissors, and a snake. The snake moves around the board trying to eat as many apples as it can, but avoiding the rocks, exiting the board, or overlapping with itself. Whenever the snake eats an apple, it grows and whenever it touches a pair of scissors it shrinks. The user controls the movement of the snake using the keyboard and the goal is for the snake to eat as many apples as possible. The game ends when the head of the snake touches a rock, it tries to move outside the board, or it touches any part of its body. The following figure shows a screenshot of the game.



**Figure 1.** A gameboard of 8 rows and 15 columns containing one apple, one rock, one pair of scissors, and a snake of length 4.

This assignment is an example of a collaborative project with modular design. Some classes of the project were designed by the CS1027 team and other classes need to be designed by you. The classes that you must design are

specified below. It is **very important** that you follow exactly the specifications of the classes that you are to design, as otherwise your code will not work correctly with the code that is provided to you.

### 3. Classes to Implement

For this assignment you need to design and implement 3 Java classes: *GridPosition*, *Snake*, and *GridBoard*.

#### 3.1 Class *GridPosition*

Each square in the grid can be specified by two numbers: its row and its column. An object of class *GridPosition* represents the position of a square of the grid. This class must have two private integer variables: *posRow* and *posCol*. In this class you must implement the following public methods:

- *GridPosition(int row, int col)*: this is the constructor for the class. The value of the first parameter must be stored in instance variable *posRow* and the second in *posCol*.
- *int getRow()*: returns the value of *posRow*.
- *int getCol()*: returns the value of *posCol*.
- *void setRow(int newRow)*: stores the value of *newRow* in *posRow*.
- *void setCol(int newCol)*: stores the value of *newCol* in *posCol*.
- *boolean equals(GridPosition otherPosition)*: returns true if this *Position* object and *otherPosition* have the same values stored in *posRow* and *posCol*; otherwise it returns false.

#### 3.2 Class *GridBoard*

This class represents the board game where the snake moves around eating apples. This class will have 4 private instance variables:

- *int board\_cols*: the number of columns of the grid.
- *int board\_rows*: the number of rows of the grid.
- *Snake theSnake*: and object of the class *Snake* (described below) representing the playing snake.
- *String[][] gridMatrix*: a 2-dimensional array of Strings that will store the content of each one of the squares of the grid. So, *gridMatrix[r][c]* is the value stored in row *r* and column *c* of the 2-dimensional array. Each entry of *gridMatrix* can contain the following possible values:
  - “empty”: if the corresponding square of the grid is empty.
  - “apple”: if the corresponding square of grid contains an apple.
  - “scissors”: if the corresponding square of the grid contains a pair of scissors.
  - “rock”: if the corresponding square of the grid contains a snake-killing rock.

In this class you need to implement the following public methods:

- *GridBoard(String gridFile)*: this is the constructor for the class. The parameter is the name of a file containing the dimensions of the game grid, the initial position of the snake, and the objects placed on the grid. You must open the file named by *gridFile*, read it and store in the instance variables of this class the appropriate values. To help you with this task, you are provided with a java class called *MyFileReader* which contains methods to open a text file, read a String or an integer, and check whether the whole file has been read. You are also given a Java class called *TestMyFileReader* that shows how some of these

methods are used to read and print the content of a file. Study these classes carefully, so you know how to use *MyFileReader* for this assignment.

The format of *gridFile* is as follows. The first 6 lines contain each one number. For the rest of the file, the following 3 lines contain a number, a number, and a string; then the next 3 lines contain a number, a number, and a string, and so on until the end of the file.

- The numbers in the first 2 lines of the file are not going to be used by the code that you will write. So, your constructor will just read them and ignore them. The first number is the width of each grid square and the second number is the length of each grid square. The code given to you will use these two numbers to determine how the game board will be displayed in the screen. When running the program, if you see that the board is too small, then simply increase these two values in the *gridFile* and re-run the program; if the board is too large then decrease these numbers and re-run the program.
- The third number is the number columns of the grid, which you must store in *board\_cols*.
- The fourth number is the number of rows of the grid, which you must store in *board\_rows*.
- The fifth number is the row and the sixth number is the column where the snake is initially positioned on the grid. Initially the snake has length 1. A new object of the class *Snake* must be created and stored in instance variable *theSnake*:

*theSnake* = new *Snake*(value of fifth number, value of sixth number);

Once your code has read the first 6 lines of the file, it must create a 2-dimensional array of type and dimensions: *String[board\_rows][board\_cols]*. All entries of the array are initialized to contain the string "empty" (in lowercase; **it is very important that all strings that you store in matrix are lowercase**). Then, the rest of the file is read and for each group of 3 lines your code will read two numbers (*number1* and *number2*) and a string *s* and then you must store *s* in *gridMatrix[number1][number2]*.

An example *gridFile* is shown below, where the grid squares are of size 100 by 100 pixels, the grid has 15 columns and 8 rows, the snake is initially positioned in row 5 and column 8, a rock is placed in row 3 and column 3, an apple is in row 7 and column 10, and a pair of scissors is placed in row 5 and column 5.

```
100
100
15
8
5
8
3
3
rock
7
10
apple
5
5
scissors
```

**Important Note.** Rows and columns are indexed starting at 0. So, the figure in page 1 shows the correct positioning for the above rock, apple, and scissors. Note also that in that figure the snake has already


eaten some apples and moved to a different location on the board than the one initially specified in the file. For the above example, the *gridMatrix* instance variable will be a 2-dimensional array of size [8][15].

Another note (not so important). If you want to write your own board files, make it sure the number of columns of the board is at least 15 and the number of rows is at least 6, so it displays correctly on the screen.

The other public methods that you must implement in this class are the following:

- *String getObject(int row, int col)*: returns the string stored in *gridMatrix* [row][col].
- *void setObject(int row, int col, String newObject)*: stores *newObject* in *gridMatrix* [row][col].
- *Snake getSnake()*: returns *theSnake*.
- *void setSnake(Snake newSnake)*: stores the value of *newSnake* in instance variable *theSnake*.
- *int getNumCols()*: returns *board\_cols*.
- *int getNumRows()*: returns *board\_rows*.
- *String getType(int row, int col)*: returns *gridMatrix* [row][col].
- *void invertGrid()*: modifies *gridMatrix* so the first column is switched with the last column, the second column is switched with the second to last column and so on, so at the end *gridMatrix* is “inverted” (a mirror image of itself). For example, if *gridMatrix* contains the information in the following matrix on the left, then after invoking this method *gridMatrix* will contain the information shown on the matrix on the right.

apple				scissors		
		rock				
	rock					



		scissors				apple
				rock		
					rock	

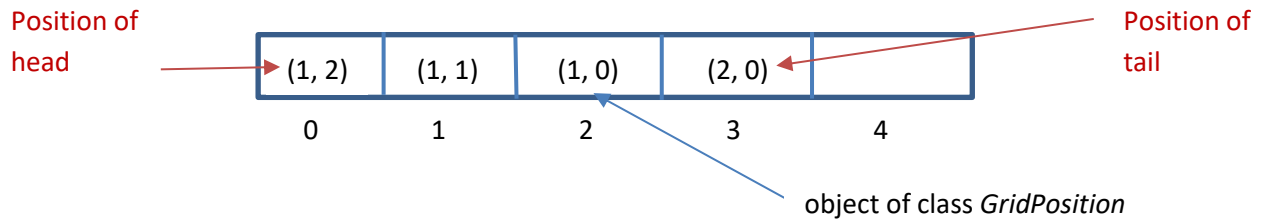
In this method **you CANNOT use another 2-dimensional array**. You can use an array if you want, but it cannot be a 2-dimensional array.

After inverting *gridMatrix* this method must invoke method *theSnake.invertSnake()* to invert the position of the snake; this method is described below.

### 3.3 Class Snake

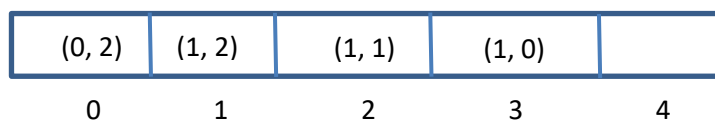
This class stores the information about the snake as it moves around the board. This class will have two private instance variables:

- *int snakeLength*: this is the number of grid squares occupied by the snake. For example, the snake shown in the above figure in page 1 has a length of 4.
- *GridPosition[] snakeBody*: the grid squares occupied by the snake will be stored in this array. The grid square with the head of the snake will be stored in index 0 of the array; the grid square where the tail of the snake is will be stored in index *snakeLength*-1 of the array. For example, for the snake in the figure in page 1, the array *snakeBody* will store the following information:



In this class you need to implement the following public methods.

- *Snake(int row, int col)*: this is the constructor for the class; the parameters are the coordinates of the head of the snake. Initially the snake has length 1, so in this method the value of the instance variable *snakeLength* will be set to 1. Instance variable *snakeBody* is initialized to an array of length 5 of *GridPosition* objects. An object of class *GridPosition* will be created storing the values of *row* and *col* and this *GridPosition* object will then be stored in the first entry of array *snakeBody*.
- *int getLength()*: returns the value of instance variable *snakeLength*.
- *GridPosition[] getSnakeBody()*: returns instance variable *snakeBody*.
- *GridPosition getGridPosition(int index)*: returns the *GridPosition* object stored in *snakeBody[index]*. It returns *null* if *index < 0* or *index >= snakeLength*.
- *void shrink()*: decreases the value of *snakeLength* by 1.
- *boolean snakeGridPosition(GridPosition pos)*: returns true if *pos* is in array *snakeBody*, and it returns false otherwise. Notice that you must use method *equals* from class *GridPosition* to compare two objects of the class *GridPosition*.
- *GridPosition newHeadGridPosition(String direction)*: returns the new position of the head of the snake when the snake moves in the direction specified by the parameter. The values that parameter *direction* can take are "right", "left", "up" and "down". If, for example, the head of the snake is at (2,3) and *direction* is "right" then the new position would be (2,4); if *direction* is "down" then the new position would be (3,3). If the head is at (0,0) and *direction* is "up" the new position would be (-1,0).
- *void moveSnake(String direction)*: moves the snake in the specified direction; this means that array *snakeBody* must be updated so it contains the positions of the grid squares that the snake will occupy after it moves in the direction specified by the parameter. For example, for the snake in the figure in page 1 array *snakeBody* is as specified at the top of this page. If *direction* = "up" then array *snakeBody* must change to this

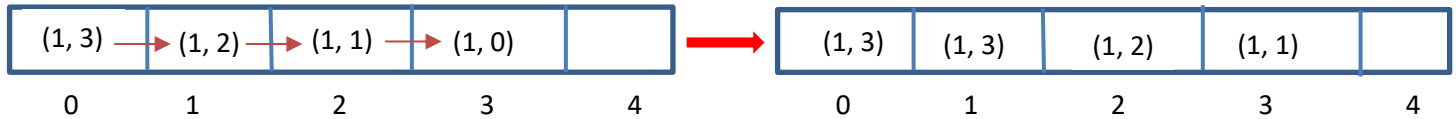


If *direction* is "up" again then array *snakeBody* must change to this



Notice that to determine the new array *snakeBody* what you must do is this:

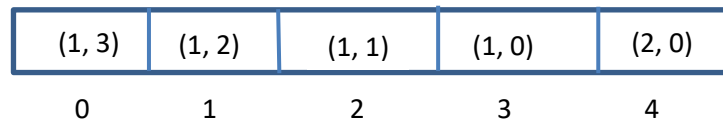
- shift one position to the right all values in the array stored in indices 0 to  $snakeLength - 2$ . For example, if the snake is as in the array on the left and *direction* = “right”, then shifting produces this array:



- and then store in index 0 of the array the new position of the snake’s head (which you can compute using method *newHeadGridPostion()*); in the above example we would store (1,4) in the first entry of the array.
- *void invertSnake(int gridCols)*: receives as parameter the number of columns in the grid and it changes the value stored in every entry of array *snakeBody* as follows:
  - if *snakeBody[i]* stores a *GridPosition* object *g*, then change the value of instance variable *g.posCol* to  $gridCols - 1 - g.posCol$ .

The above operation moves the snake to a mirror position of its initial location.

- *void growSnake(String direction)*: increases the length of the snake by 1 and moves the snake’s head in the direction specified. This method is very similar to method *moveSnake*, but instead of shifting the values stored in *snakeBody* from index 0 to  $snakeLength - 2$ , we need to shift all values from index 0 to index  $snakeLength - 1$ . For example, if the snake is as shown in the figure in page 1, and *direction* = “right” then the new content of array *snakeBody* will be



Notice that since the length of the snake increases you need to make sure that array *snakeBody* is large enough to store the new information. If instance variable *snakeLength* has the same value as *snakeBody.length()*, the size of the array, then you must

- double the size of the array if  $snakeLength < 10$ ,
- otherwise you must increase the size of the array by 8.

To increase the size of the array you must implement the following private method called *increaseArraySize()*:

- *void increaseArraySize()*: increases the size of array *snakeBody* as specified above, preserving the information that was stored in it.

In all three above classes, *GridPosition*, *Snake*, and *GridBoard* you can implement more private methods, if you want to, but you cannot implement more public methods. You can also add more private instance variables, but only if they are required. **The use of unnecessary instance variables will be penalized.**

## 4. Classes Provided

You can download from the course’s webpage the following java classes: *PlayGame.java*, *MyFileReader.java*, *TestMyFileReader.java*, and *TestAsmt1.java*. You can also download several image files needed to display the game board and some grid input files.



Class `PlayGame` has the *main* method. Class `TestAsmt1` contains several tests that you can use to check that your code works correctly. Please note that the tests in this class are not exhaustive, so there might be errors in your code that are not detected by this testing program. Hence, you are required to perform additional tests on your code to ensure that it works correctly. When the TA's mark your assignment, they will perform additional tests on your code.

## 5. How to Run the Program

If you are running the program from the terminal place all the files in the same directory and compile the program by running `javac PlayGame.java` and then run it with `java PlayGame` (in this case the program will ask for the name of the input grid file) or `java PlayGame nameOfGridFile`; sample grid files `board1.txt`-`board3.txt` have been provided . If you run the program from Eclipse, read the instructions in the next section.

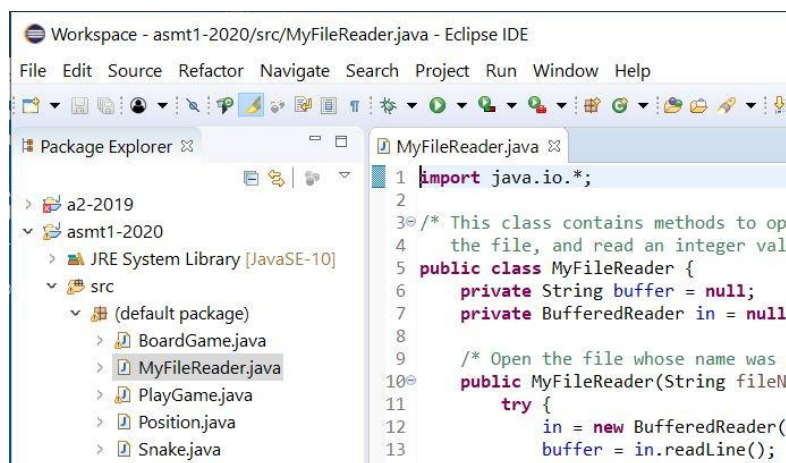
The program will display the grid with the objects specified in the input grid file. To start the game press any of the arrow keys. The arrow keys can then be used to change the direction in which the snake moves.

Additionally, you can type the following keys:

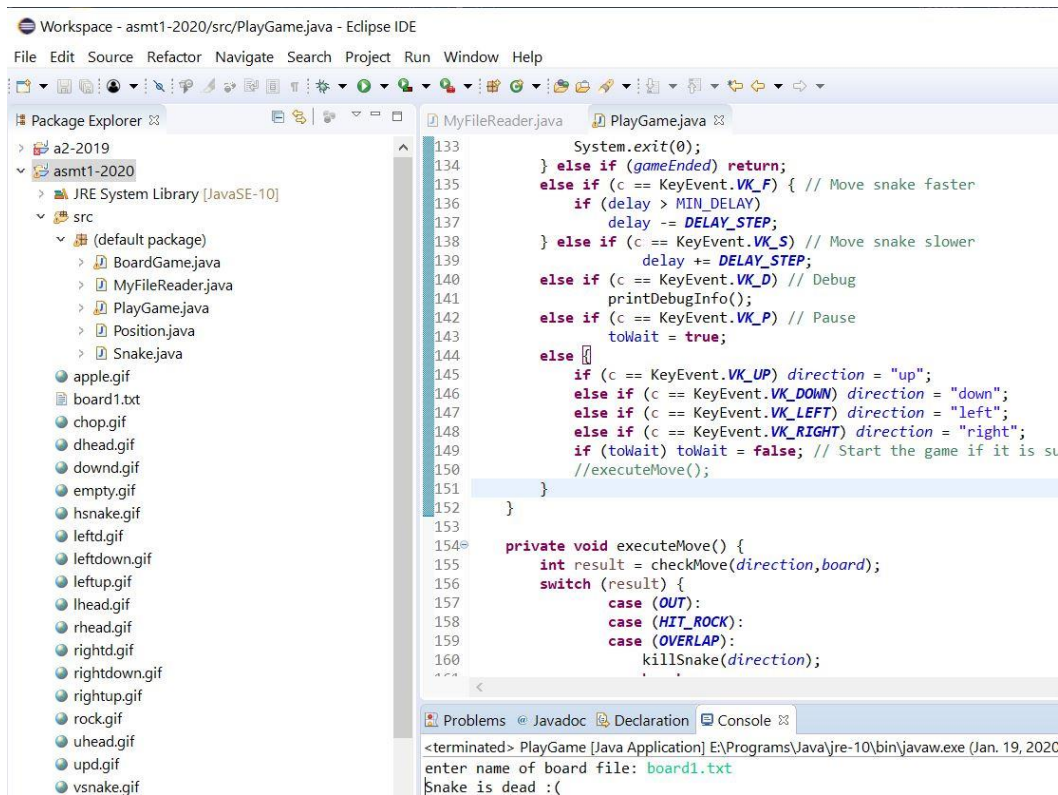
- `f`: increases the speed of the snake
- `s`: reduces the speed of the snake
- `p`: pauses the game
- `i`: inverts the grid. The grid will also invert at random times while you are playing the game to make it a bit more challenging
- `x`: terminates the game
- `d`: prints some debugging information that you might find useful when testing and debugging your program. You must first pause the game to print this information.

### 5.1 Running the Program from Eclipse

Place the java files in your Eclipse workspace under the Java project you have created for this assignment.

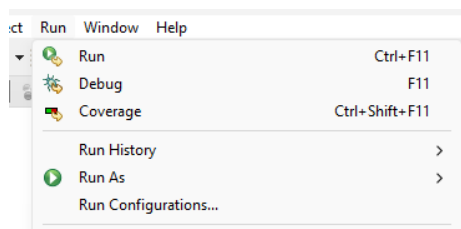


Place the image files and the sample input grid files in the root directory for your project (not inside the "(default package)" folder).

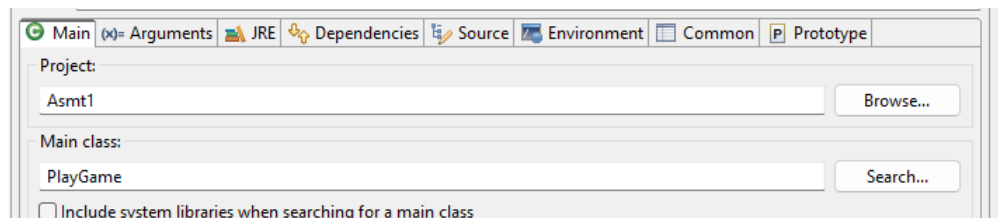


Double-click PlayGame in the Package Explorer and select “run”.

If Eclipse cannot find the image and grid files, then you can place all of the image and grid files in the same folder as the java files and then click on Run -> Run Configurations

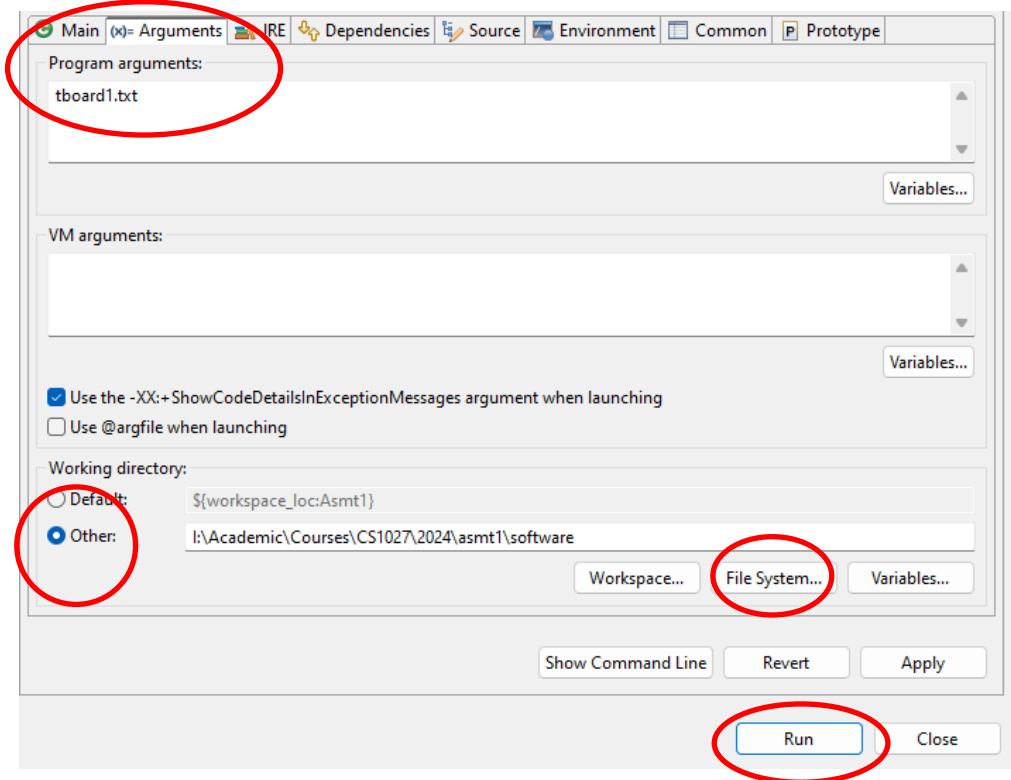


Select the correct Project and Main Class:



Then click on Arguments and in “Program arguments” you can enter the name of the grid input file. Under “Working directory” select “Other:” click on “File System” and select the folder where the image and txt files are stored. Finally click on the “Run” button.





## 6. Non-Functional Specifications

- **Assignments are to be done individually and must be your own work. Software will be used to detect cheating.**
- You must properly document your code by adding comments where appropriate. Add comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add comments to methods to explain what they do and to instance variables to explain their meaning and/or purpose. Also add comments to explain the meaning of potentially confusing parts of your code.

When deciding where to add comments, you need to use your own judgment. If the meaning of a method, instance variable, or fragment of code is obvious, you do not need to add a comment. If you think that someone else reading a fragment of your code might struggle to understand how the code works, then write a comment. However, try to avoid meaningless comments like these:

```
i = 1;           // initialize the value of i to 1
i = i + 1;       // increase the value of i
if (i == j) // compare i and j
```

- Use Java coding conventions and good programming techniques:
  - Use meaningful variable and method names. A name should help understand what a variable is used for or what a method does. Avoid the use of variable names without any meaning, like `xyy`, or names that do not relate to the intended purpose of the variable or method.
  - Use consistent conventions for naming variables, methods, and classes. For example, you might decide that names of classes should start with a capital letter, while names of variables and methods should start with a lower-case letter. Names that consist of two or more words can be combined, for example, using “*camelCasing*” (i.e. the words are concatenated, but the second word starts with a capital letter) or they can be combined using underscores as in *symbol\_table*. However, you need to be consistent.

- Use consistent notation for naming constants. For example, you can use capital letters to denote constants (`final` instance variables) and constant names composed of several words can be joined by underscores: `TABLE_SIZE`.
- Use constants where appropriate.
- Readability. Use indentation and white spaces in a consistent manner to improve the readability of your code.

## 7. Submitting your Work

You **MUST SUBMIT ALL THE JAVA FILES THAT YOU WROTE** through Gradescope.

**DO NOT** put a `package` line at the top of your java files.

**DO NOT** submit a compressed file (.zip, .tar, .gzip, ...); **SUBMIT ONLY** .java files.

**Do not submit** your .class files. If you do this and do not submit your .java files, your assignment cannot be marked.

Submit your assignment on time. Late submissions will receive a penalty as specified in the course outline.

## 8. Marking

What You Will Be Marked On:

- Functional specifications:
  - Does the program behave according to specifications?
  - Does it run with the sample input files provided and produce the correct output?
  - Are your classes implemented properly?
  - Are you using appropriate data structures?
- Non-functional specifications: as described above.
- **Assignment has a total of 20 marks.**

### 8.1 Marking Rubric

- Program Design and Implementation. All methods in student's java classes are correctly designed and implemented as required: 5 marks.
- Testing. Program produces the correct output for all tests: 12 marks.
- Programming Style: 3 marks
  - Meaningful names for variables and constants: 1 mark
  - Code is well designed (simple to follow, no redundant code, no repeated code, no overly complicated code, ...) and readable (good indentation): 1 mark
  - Code comments: 1 mark