**COMPUTER SCIENCE 1027B – Foundations of Computer Science II**

**Assignment 1**

**Due Date: February 10, 11:55 pm**

# 1. Purpose

To gain experience with

- Java and Java libraries.
- 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 on a rectangular board. 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 and the goal is 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.

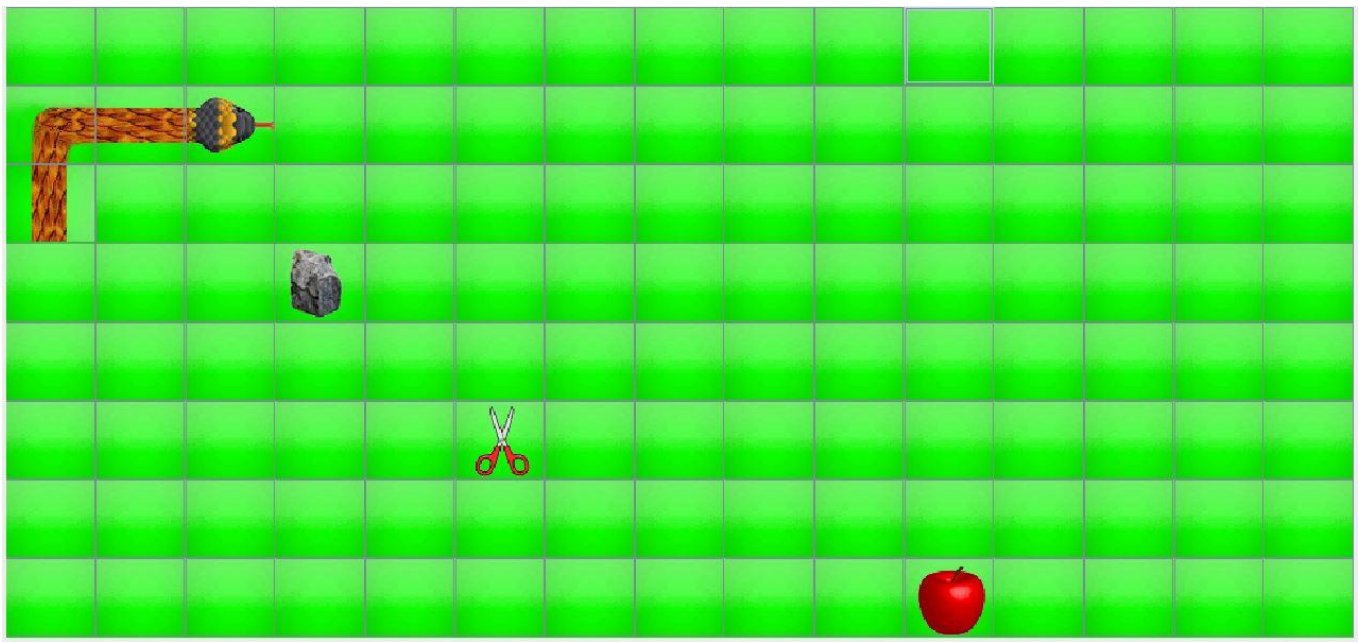


**Figure 1.** A gameboard of 8 rows and 15 columns containing one apple, one rock, one pair of scissors, and a snake of length 4.

This assignment is an example of a collaborative project with modular design. Some modules of the program were designed by the CS1027 team and other modules need to be designed by you. The modules that you must design are specified below. It is very important that you follow exactly the specifications of the modules 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 3 Java classes: *Position*, *Snake*, and *Board*.

### 3.1 Class *Position*

Each square in the grid of the board game can be specified by two numbers: the row where the square is and the column where it is. An object of class *Position* represents the position of a square of the grid. This class must have two private integer variables: *positionRow* and *positionColumn*. In this class you must implement the following public methods:

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

### 3.2 Class *BoardGame*

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

- *int board_length:* the number of columns of the grid on the game board.
- *int board_width*: the number of rows of the grid.
- *Snake theSnake:* and object of the class Snake representing the playing snake.
- *String[][] matrix:* a 2-dimensional matrix that will store the content of each one of the squares of the grid. Each entry of *matrix* 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:

- *BoardGame(String boardFile):* this is the constructor for the file. The parameter is the name of a file containing the dimensions of the game board, the initial position of the snake, and the objects placed on the game board. You must open the file named by *boardFile*, read it and sore 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 the *boardFile* is as follows. The first 6 lines contain each one number. For the rest of the file, the next 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 first 2 numbers 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 *boardFile* 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 length of the board, which you must store in *board_length*.
- The fourth number is the width of the board, which you must store in *board*_**width**.
- The fifth number is the row and the sixth number is the column where the snake is initially positioned on the board. Initially the snake has length 1. A new object of the class *Snake* must be created, and its address stored in *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_width][board_length].* 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 triplet *number1*, *number2*, *string1* read your code must store *string1* in *matrix*[*number1*][*number2*].

An example *boardFile* is shown below, where the grid squares are of size 100 by 100 pixels, the board 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. Note that 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 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 matrix instance variable will be a 2-dimensional array of size [8][15].

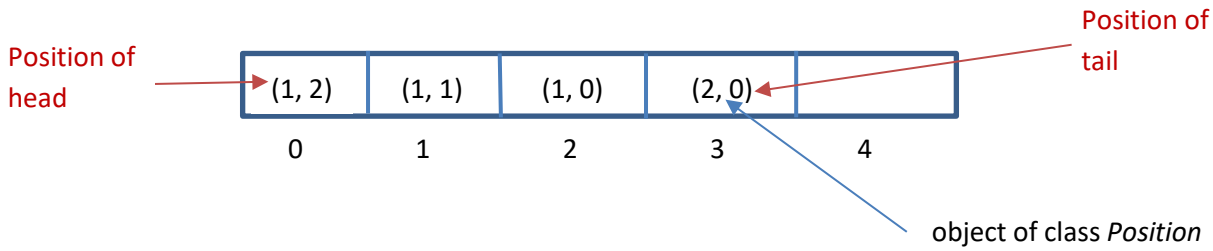
Another note (not so important). If you want to write your own board files, make it sure the length of the board is at least 15 and the width is at least 6, so it displays correctly on the screen.

- *String getObject(int row, int col):* returns the string stored in *matrix*[*row*][*col*].
- *void setObject(int row, int col, String newObject):* stores *newObject* in *matrix*[*row*][*col*].
- *Snake getSnake():* returns *theSnake*.
- *void setSnake(Snake newSnake):* stores the value of *newSnake* in instance variable *theSnake*.
- *int getLength():* returns *board_length*.
- *int getWidth():* returns *board_width*.
- *String getType(int row, int col):* returns *matrix*[*row*][*col*].

### 3.3 Class Snake

The 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 has a length of 4.
- *Position[] 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 the snake in the figure, array *snakeBody* will store the following information:

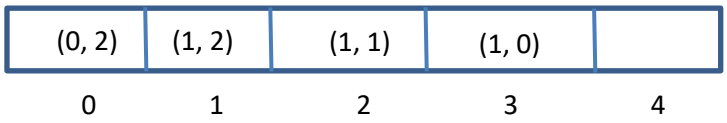



In this class you need to implement the following public methods.

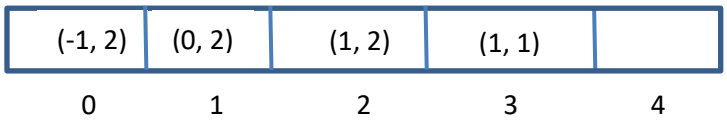
- *Snake(int row, int col):* this is the constructor for the class; the parameters are the coordinates of the head of the snake. Initially the snake has length 1, so in this method the value of the instance variable *snakeLength* will be set to 1. Instance variable *snakeBody* is initialized to an array of length 5 of *Position* objects. An object of class *Position* will be created storing the values of *row* and *col* and this *Position* object will then be stored in the first entry of array *snakeBody*.
- *int getLength():* returns the value of instance variable *snakeLength*.
- *Position getPosition(int index):* returns the *Position* object stored in *snakeBody*[*index*]. It returns *null* if *index* < 0 or *index* >= *snakeLength*.
- *void shrink():* decreases the value of *snakeLength* by 1.
- *boolean snakePosition(Position pos):* returns true if *pos* is in array *snakeBody*, and it returns false otherwise. Notice that you must use method *equals* from class *Position* to compare two objects of the class *Position*.
- *Position newHeadPosition(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 *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 above figure array *snakeBody* is as specified above. If *direction* = “up” then array *snakeBody* must be this

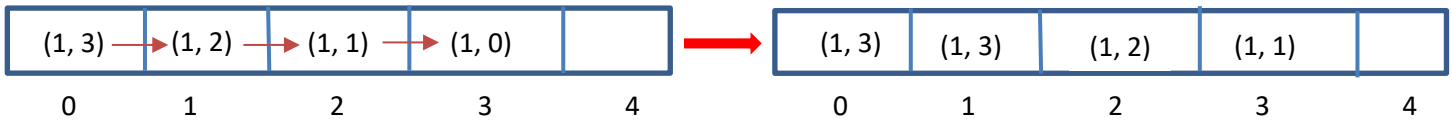



If *direction* is “up” again then array snakeBody must be this



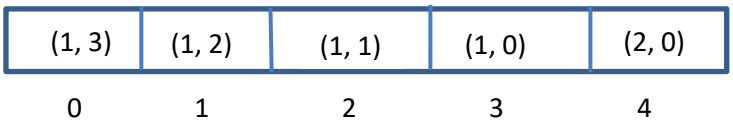

Notice that to determine the new array snakeBody what you must do is this:

  - 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 figure 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 *newHeadPostion*(); in the above example we would store (1,4) in the first entry of the array.

- *void grow(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 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, and *direction* = “right” then the new content of array *snakeBody* is




Notice that since the length of the snake grows 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 before storing the new positions of the snake in it. To do this you must use the below private method *increaseArraySize*().

You must also implement the following private method.

- *void increaseArraySize*(): this method doubles the size of array *snakeBody* preserving the information that was stored in it.

In all three above classes, *Position*, *Snake*, and *BoardGame* 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. How to Run the Program

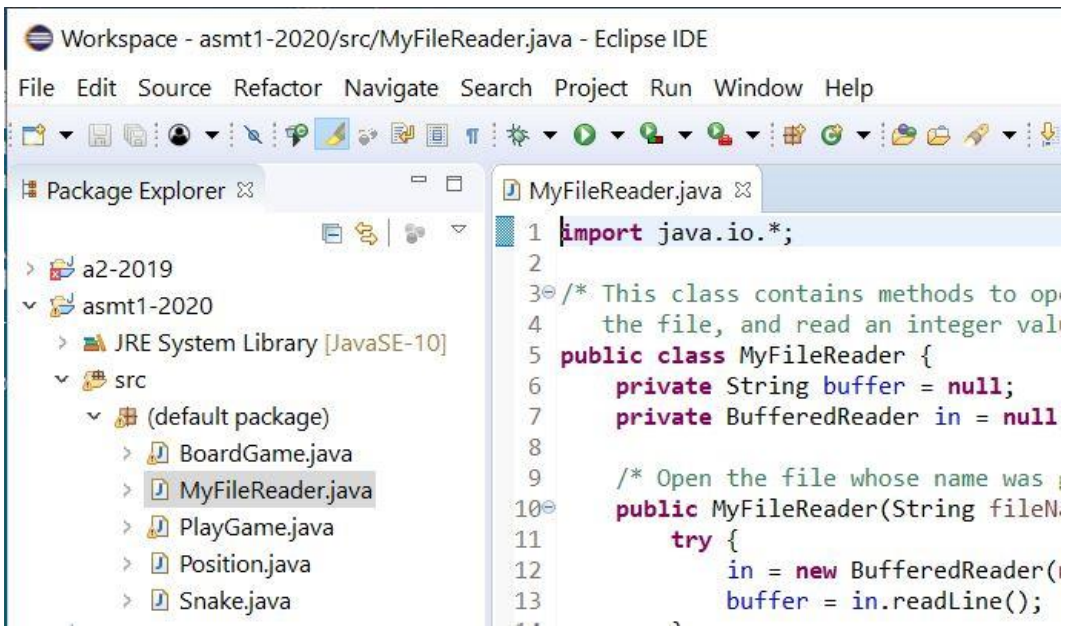
Download from the course's webpage the following java classes: PlayGame.java and MyFileReader.java, and all the image files needed to display the game board. If you are running the program from the terminal place all the files in the same directory and then compile the program by running `javac PlayGame.java` and then run it with `java PlayGame`. If you run the program from Eclipse, read the instructions in the next section.

The program will ask for the name of the file containing the objects initially placed on the game board and the n it will display the board. 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:

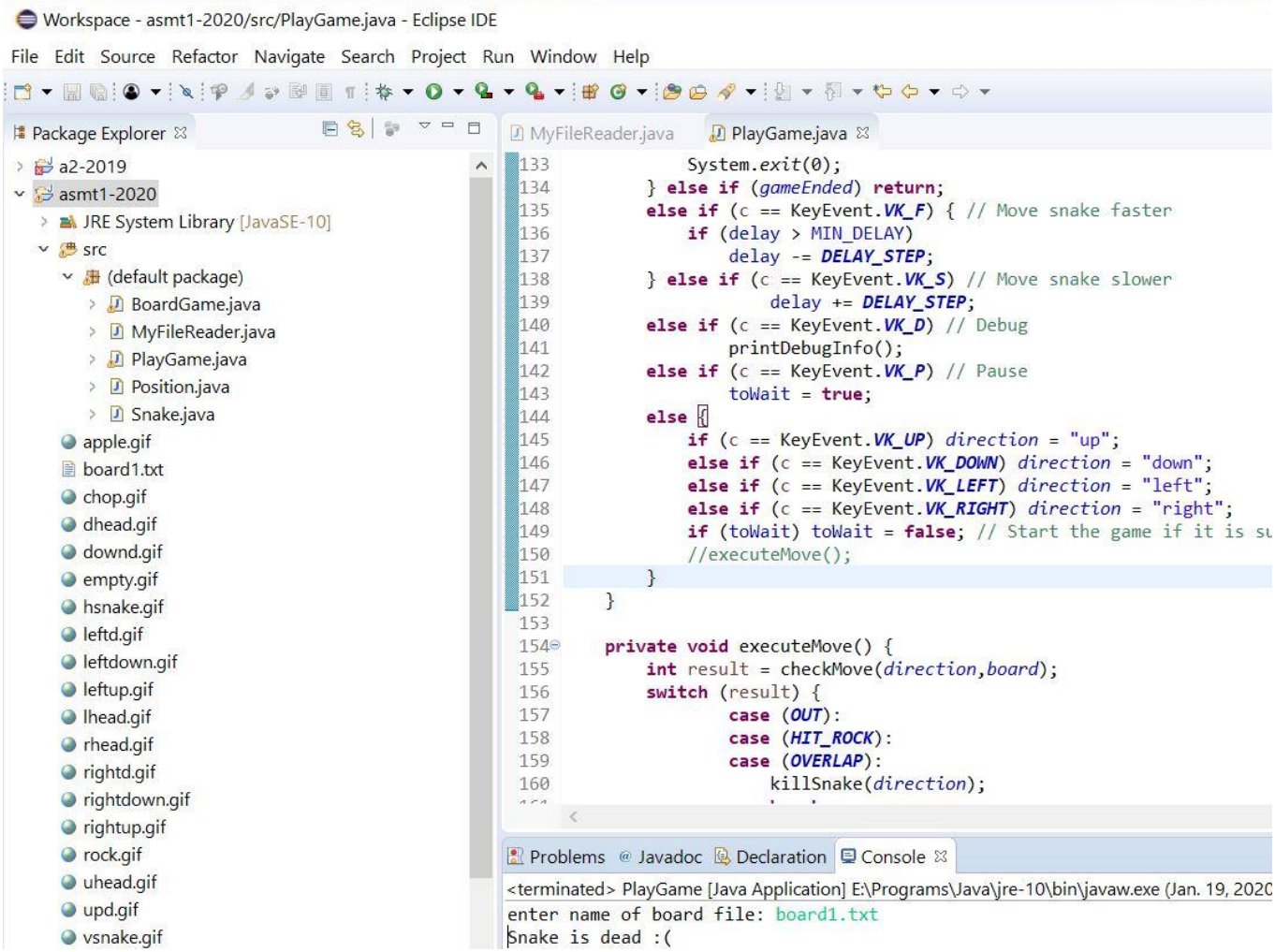
- `f`: increases the speed of the snake
- `s`: reduces the speed of the snake
- `p`: pauses the game
- `x`: terminates the game
- `d`: prints some debugging information that you might find useful when testing and debugging your program. You must first pause the game to print this information.

### 4.1 Running the Program from Eclipse

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



Place the image files and the sample board files in the root directory for your project (not inside the "(default package)" folder). If Eclipse does not find the image files you will have to move them around until you find the directory where Eclipse will find them. Sometimes different versions of Eclipse work in different ways.




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

## 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 Javadoc comments where appropriate. Add Javadoc 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 Javadoc 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 *xxy*, or names, like *flower*, 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 like *symbol* and *table* can be combined, for example, using "*camelCasing*" (i.e. the words are concatenated, but the second word starts with a capital letter, for example, *symbolTable*) 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, tabs, and white spaces in a consistent manner to improve the readability of your code. The body of a for loop statement, for example, should have a larger indentation than the statement itself:

    ```
    for (int i = 0; i < TABLE_SIZE; ++i)
        table[i] = 0;
    ```

  - Positioning of brackets, '{' and '}' to delimit blocks of code should be consistent. For example, if you put an opening bracket at the end of the header of a method:

    ```
    private int method() {
        int position;
    ```

    then you should not put the bracket in a separate line for another method:

    ```
    private String anotherMethod()
    {
        return personName;
    ```

## 7. Submitting your Work

You **MUST SUBMIT ALL YOUR JAVA** files through OWL. **DO NOT** put the code inline in the text-box provided by the submission page of OWL. **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!

## 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: 6 marks.
- Testing. Program produces the correct output for all individual method tests and the program plays the game as required: 10 marks.
- Programming Style: 4 marks
    - Meaningful names for variables and constants.
    - Code is well designed (simple to follow, no redundant code, no repeated code, no overly complicated code, …)
    - Readability:
        - Good indentation.
        - Appropriate code comments.