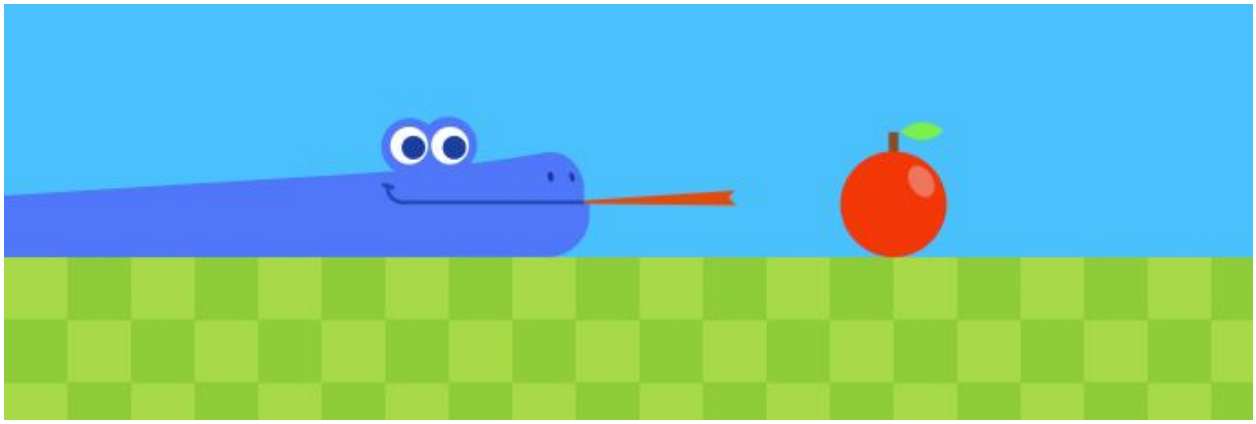
COMP 2140 Fall 2019 Section 1 and 2 Final Project - Snake!



Introduction

For the final project, we're going to build a famous video game that delighted gamers have been playing for many years. The game is called **Snake!** The player maneuvers a line with the arrow keys. The goal is to get the line to eat an apple, which appears at random points on the screen. The line itself grows in length. The player loses when the head of the line runs into the border around the screen, or itself.

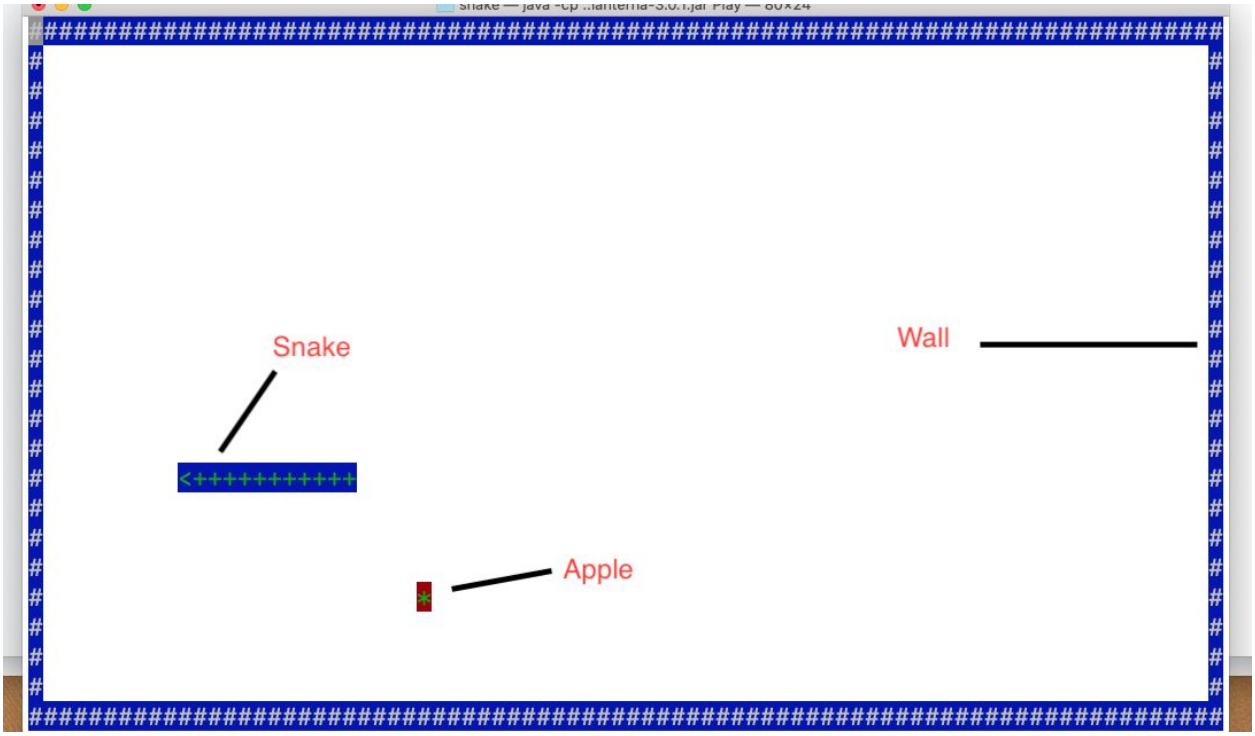
The final project exercises skills in:

- Reading code
- Working with conditionals, arrays, array lists, loops and functions
- Working with code in multiple files
- Working with objects and classes

It uses all that you've learned up to this point to create a real, working arcade game. Even if you don't have the game completely working by the due date, submit what you have, so long as it compiles and runs. You'll get credit for the parts that you managed to get working.

What It Should Look Like

The game should draw on the screen and display as below. The walls of the room are '#', the snake is '+' and an arrow for the head, and the apple is red and green, and is the asterisk '*'.



Doing Graphics

Our rendition of the game will use text graphics. That is, we will draw lines and characters using the terminal window of the screen. There is a powerful text graphics library for terminals called Lanterna. The library is already provided for you in the starter code you've been given. Because we use Lanterna, we have to compile and run our program with some extra options, as you can see in the highlighted bits below:

```
javac -cp .:lanterna-3.0.1.jar *.java      # compile all java files
java -cp .:lanterna-3.0.1.jar Play         # run main in the Play class
```

The highlighted section says to include the lanterna library (located in the `lanterna-3.0.1.jar` file) when we are compiling or running. `*.java` means all java files in the directory.

But we won't be using Lanterna directly. I've provided a stripped down interface for you to use in the file "Gui.java". The Gui[1] class boils the complex Lanterna functionality down into a set of simple functions for you to call.

[1] "Gui" is short for graphical user interface. In our particular case, we are drawing text graphics in the terminal window.

Starting the Gui

To start using the Gui, make a new Gui object, and call `start()` on it.

```
Gui gui = new Gui();
gui.start();
```

This clears the screen, and the Gui library takes control of the screen.

User Input

We can't use Scanner for user input for the game. The input has to be read immediately, as the game can't wait for the user to press the enter key each time. Instead, the Gui class gives you the ability to instantly read user key presses on the keyboard.

```
char c = gui.getKeypress();
```

The value of c is the key that was pressed. For example, if the user pressed 'q', then c would be equal to 'q'.

If the user presses the up arrow, down arrow, left arrow or right arrow, then c would be equal to one of the following values respectively:

- `Gui.UP_ARROW_KEY`
- `Gui.DOWN_ARROW_KEY`
- `Gui.LEFT_ARROW_KEY`
- `Gui.RIGHT_ARROW_KEY`

For example, this code snippet determines if the user pressed the up arrow key:

```
char c = gui.getKeypress();
if (c == Gui.UP_ARROW_KEY) {
    // the user pressed up arrow
}
```

If the user did not press any key when you called `gui.getKeypress()`, then `c` would be equal to `0`.

Screen Dimensions

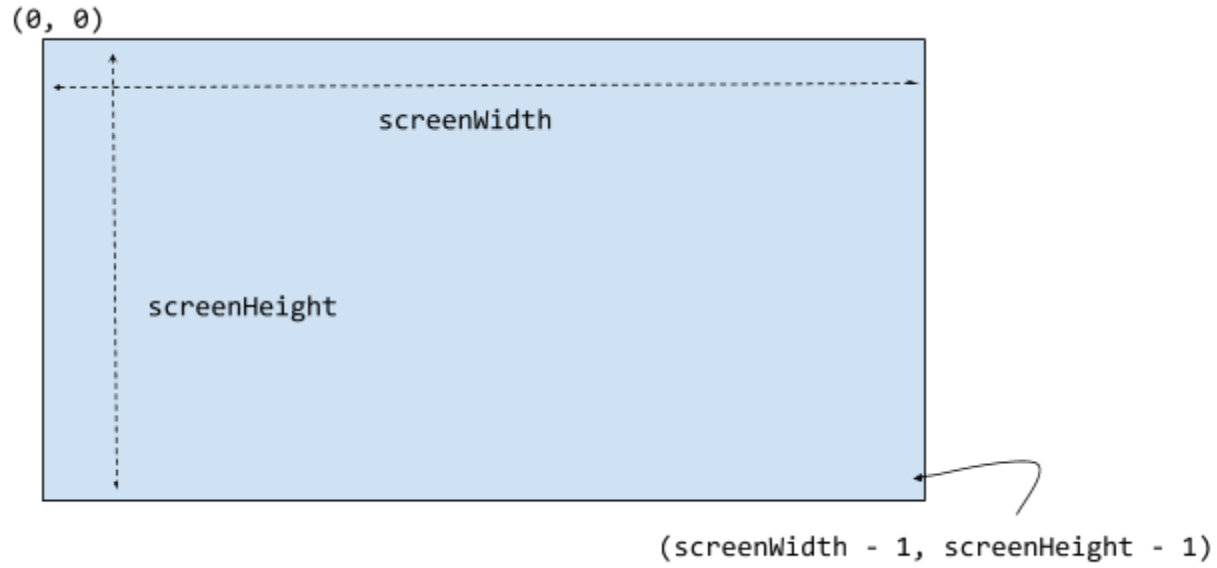
Because you can resize the terminal window, you need a way to find out how big it is. In order to get how big the screen is, you can call the following methods:

```
int screenHeight = gui.getScreenHeight();
int screenWidth = gui.getScreenWidth();
```

The values returned are the number of text rows for height and the number of text columns for width.

Drawing on the Screen

The screen coordinates are defined by `(x, y)` coordinates where `(0, 0)` is the upper left hand corner of the screen, and `(screenWidth - 1, screenHeight - 1)` is the lower right hand corner of the screen.



Drawing a Character

To draw a character, call the function:

```
gui.drawCharacter(x, y, c, foregroundColor, backgroundColor);
```

This draws the character `'c'` at the coordinates x and y with the given colors. The color arguments are strings, such as “RED”, or “YELLOW”. The valid values for colors are:

- `"WHITE"`
- `"BLACK"`
- `"RED"`
- `"ORANGE"`
- `"YELLOW"`
- `"GREEN"`
- `"BLUE"`
- `"INDIGO"`
- `"VIOLET"`

For example, to draw the character `'X'` at position `(15, 3)` and foreground color `"BLUE"` and background color `"YELLOW"` , we will call:

```
gui.drawCharacter(15, 3, 'X', "BLUE", "YELLOW");
```

Drawing a Line

To draw a line from the coordinates `(startx, starty)` to the coordinates `(endx, endy)`, with the character c and the given colors, call

```
gui.drawLine(startx, starty, endx, endy, c,
      foregroundColor, backgroundColor);
```

The coordinates and color arguments are the same as in drawCharacter.

Double Buffering

When you draw characters or lines on the screen, they don't appear straight away. The updates are done in an offscreen buffer. The buffer is shown only when the `Gui.refresh()` method is called.

This is done so that the complete picture is shown all at once. Otherwise you would see the system drawing parts of the picture, one at a time. Instead, we draw the characters and lines in an offscreen buffer. When the drawing of that frame is fully complete, we call `refresh()` and show it to the user all at once.

If you think about it, game animation is just like animating a cartoon. We show full consecutive frames of data with slight differences between them. If we do it fast enough, it appears as smooth motion on the screen.

Sleeping

Computers run really fast, so we have a method to slow things down so that humans have a chance to react. The method `Gui.sleep()` takes an int argument of the number of milliseconds to suspend execution. When you want to take a break from executing the program, calling `Gui.sleep()` does the trick. You'll see it in the main program loop.

Structure of the Main Program

The overall code flow of the game program is shown below.

```
Gui gui = new Gui();        // Make a new GUI object
gui.start();                // Clear screen and let the GUI take over
```

```
boolean continuePlaying = true;
while (continuePlaying) {   // Enter the main drawing loop
    char c = gui.getKeyPress();  // Get user input
    if (c == …) {                // React to user input here
                                 // Set “continuePlaying = false” to
    }                            // quit out of the loop

    gui.clear();            // Clear the screen for a fresh draw
    ...                     // Draw whatever you need to draw offscreen
    gui.refresh();          // Present the new drawing all at once

    gui.sleep(100);         // suspend execution to slow program down
                            // for 100 milliseconds = 1/10 of a second
}
                            // Game ends when you break out of loop
gui.stop();                 // Stop and return control to terminal
```

Starter Code

I’ve given you starter code for the project. Here’s what each file does. Study the code that I gave you to get an idea of where to start.

File	What it’s For
lanterna-3.0.1.jar	The library for text graphics. This is a large program that helps draw things on the screen. You don’t have to worry about this for the project. Use the Gui class provided for you instead.
Gui.java	This simplifies the Lanterna library and gives a set of methods to get key presses, screen dimensions, and draw characters and lines on the screen. You don’t have to worry about how Gui.java is written. Just use the methods described in the Gui section above.
Play.java	This is where the main program lives. It is the basic control loop for the entire game. **Study this one first and study it closely**. The basic control loop creates the Gui, creates the objects within the game, and monitors for keypresses. It also draws each object within the game. You move the snake within this loop as well.
Snake.java	This implements the Snake itself. Think of the Snake as an ArrayList of positions representing its entire length. The head is at index 0, with the tail at subsequent positions.

Apple.java	This implements the Apple, which is the object the snake should eat.
Room.java	This implements the Room, including the borders around the screen.
Explosion.java	This implements explosions that happen when the snake hits the walls of the room, or itself.
Position.java	This class represents an `(x, y)` position on the screen. It also has a helpful method to determine if the position collides with other positions.

What You Need To Do

Your task is to flesh out the starter code to get a complete Snake game.

- The tasks all build one one another, so do them in sequence.
- You may modify any of the code in any way you want.

Task 1 (0 points)

Your first task is to compile and run the starter code.

```
javac -cp .:lanterna-3.0.1.jar *.java
java -cp .:lanterna-3.0.1.jar Play
```

Notice that it doesn't do very much.

- The room is drawn with the top wall on the top edge of the screen.
- The other 3 walls are missing.
- There is no snake and no apple.

The game enters an infinite loop waiting for user input.

- **Press Control-C to interrupt the game**.
- You may press Control-C at any time to interrupt the game while you are testing and developing the game.

Task 2 (2 points)

Your next task is to detect when the user presses a `'q'`. When the user presses a `'q'`, the game should quit gracefully.

Hint:

- Start by reading the `main` program in Play.java.
- The program is heavily commented.

- The main loop continues as long as the variable `continuePlaying` is `true`.
- The main loop has a section that reads a user keypress.
- Add code to that section to break out of the main loop when the user presses ‘q’.

Remember to recompile and test your changes after each step!

Task 3 (2 points)

Your next task is to fully implement the Room. Modify the Room class so that the four walls of the room are drawn when `Room.draw()` is called.

Hint:

- You can see how the top wall of the room is drawn in the code for `Room.draw()`.
- Follow that example and draw the left, bottom and right walls of the room.

Task 4 (2 points)

Your next task is to implement a basic Snake. The Snake’s head (denoted by >) should start out at the center of the screen, and it should initially be three characters long, like so:

```
++>
```

The Snake doesn’t have to move in this step. It just has to show up on the screen when the `Snake.draw()` method is called.

Hint:

- The Snake object needs to know where the center of the screen is. Modify the constructor for Snake so that the center of the screen is passed to it when the Snake object is constructed.
- In the constructor:
    - The Snake is represented as an `ArrayList<Position>`.
    - The head’s position is at index 0. Create a position for the head, and add it to the list.
    - The tail’s position is represented by the rest of the list. Create two other positions that represent the tail, and add it to the list
- The `Snake.draw()` method should iterate over all positions.
    - Draw the head with a ‘>’, and tail section with ‘+’ using the Gui object.

Task 5 (3 points)

The Snake’s default direction to move is to the right, since it starts as ++>. Next, you should make the snake move right one step at a time. At each step, the snake moves one step to the right.

When you complete this step, the snake moves right one step at each iteration of the main loop, until it eventually runs off the edge of the screen on the right hand side. That's ok for now.

Hint:

- Add a `move()` method to the Snake class that takes no arguments.
- Call `snake.move()` in the main loop.
- The Snake is represented as an `ArrayList<Position>`, with the head at index 0 of the list.
- Starting from the end of the tail, **each element of the snake steps into the place of the element before it**.
- The head then moves right by **adding 1 to the x coordinate** of the head.

Task (7 points)

Make the Snake change directions when the user presses the up, down, left, or right arrow keys.

- When the Snake goes right, the head is >.
- When it goes down, the head is V.
- When it goes up, the head is ^.
- When it goes left, the head is <.

Hint:

- Add a member variable called `direction` to the Snake class. The variable tells the snake which direction the head should move.
- Add a `setDirection` method to the Snake class that takes a `String direction` as an argument. The argument can be either "RIGHT", "LEFT", "UP" or "DOWN".
    - When the user presses an arrow key, call `setDirection` from the main loop. The `changeDirection` should set the `direction` member variable to the new direction of movement.
- The head moves in the direction of motion. Edit the `Snake.move()` method so that:
    - If the `direction` is "RIGHT", add 1 to the x-coordinate of the head
    - If the `direction` is "LEFT", subtract 1 from the x-coordinate of the head
    - If the `direction` is "UP", subtract 1 from the y-coordinate of the head
    - If the `direction` is "DOWN", add 1 to the y-coordinate of the head
- Edit the `Snake.draw()` method to draw the head correctly, depending on the value of `direction`.

Task 7 (2 points)

The Snake shouldn't be able to double back on itself. For example, when it's going right, the only valid direction change is up or down. Left is not valid. Add to the code in `changeDirection()` to make sure that the snake can't double back on itself.

Task 8 (3 points)

Your next task is to implement the Apple. The Apple should be red in color, drawn with the '@' character, and show up in a **random** position on the screen. It shouldn't show up on top of the borders or anywhere the Snake is.

Hint:

- Add a `Position` variable to the Apple class to hold the position of the Apple.
- Write a function that generates random numbers in the range you want. See this link for how to generate random numbers in Java.
- In the constructor for Apple, call the random number function to generate the x and y coordinates of the Apple.

Task 9 (2 points)

The Apple shouldn't show up on top of the borders or anywhere the Snake is.

Hint:

- You'll need to pass the position of the Snake to the constructor for Apple, so that it knows where the Snake is.
- In the constructor for Apple, call the random number function to generate the x and y coordinates of the Apple. Keep generating random coordinates until the coordinate of the Apple **doesn't collide with the wall or the snake**.

Task 10 (4 points)

When the Snake's head hits the Apple, two things should happen:

1. The user gains 10 points
2. The Apple should disappear and reappear at a different spot

Hint:

- Initialize a variable that keeps the user's score, before the main loop of the program.
- Add to the score whenever the Snake hits an Apple.
- You'll have to write code in the main loop to determine when the Snake's head collides with the Apple's position.
- Replace the old apple object with a new instance of Apple when the Snake hits an Apple.

Task 11 (4 points)

When the Snake hits the Apple, it should also grow in length by one. Make this happen in the code.

Hint:

- Add a `grow()` method to Snake.
- In the `grow()` method, add to the end of the ArrayList<Position> that holds the snake's position.
- Call `grow()` when the Snake eats an Apple.

Task 12 (3 points)

The Snake shouldn't be able to hit the borders of the screen. If it does so, the player loses and the game is over. Implement border collision detection in the code.

Hint:

- This is implemented in the main loop

Task 13 (4 points)

The Snake shouldn't be able to hit its tail. If it does so, the player loses and the game is over. Implement self collision detection in the code.

Hint:

- Implement this in the main loop as well.

Congratulations -- you now have a fully working Snake game!

If you've got successfully to this point, and you have good style (there are 2 full points for writing with proper style), you have the **full 40 points for the final project.**

Bonus Tasks (2 points each)

From here on, you get bonus points for any of these extra tasks:

1. When the Snake collides with itself or the wall, draw an `Explosion` at the location where the collision occurred.
2. When the game ends, make it print out the score.
3. When the Snake eats an Apple, make the game run faster
4. Have the current running total of the score printed at the top of the screen
5. Have extra border walls appear at higher levels, e.g. after the Snake has eaten a multiple of 5 apples. The Snake is not supposed to collide with these extra walls either
6. Have multiple Apples at higher levels instead of just one

Submitting Your Code

When you have finished, submit all your code (use multi-select to submit multiple files in Mimir), including:

- All the .java files
- The lanterna-3.0.1.jar file as well.

If you've done any of the bonus tasks, send me an email (seemongt@google.com) telling me what you did.

Grading

Note that automatic grading is only done for style. Everything else will be manually graded to determine that you actually implemented it correctly. This means that all automated tests (except for the first style check) will always pass automatically, but it doesn't mean you have completed each task properly.

If you want style points, make sure Test Case 1 for coding quality and style passes cleanly.