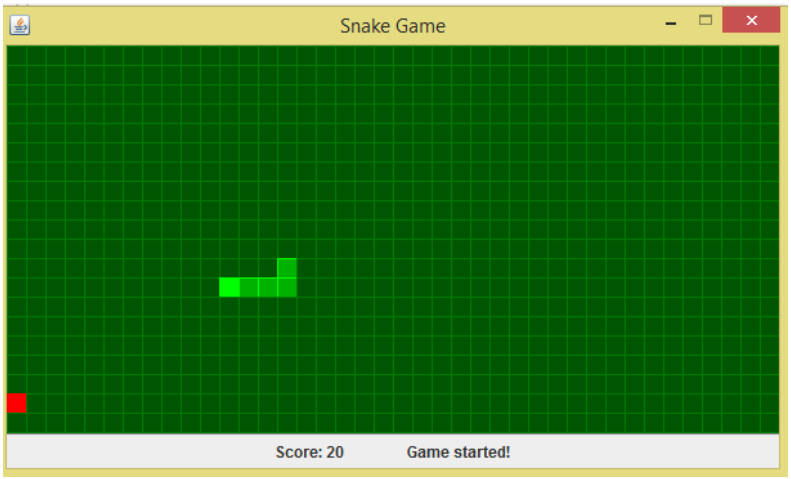


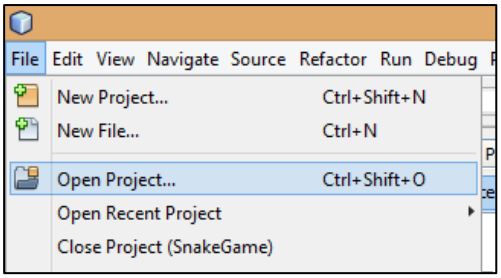
Introduction

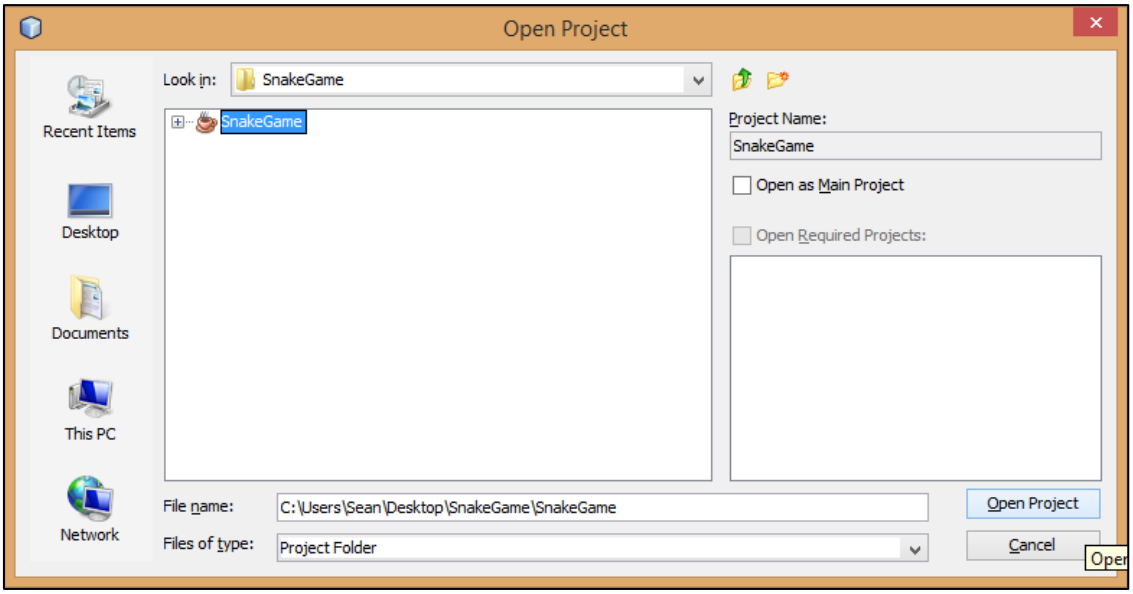
Now that we've covered some basic Java, let's get started writing the Snake game!

Please read the instructions in this worksheet carefully as they will contain advice on how best to go about creating the Snake game! You have some of the code provided already to create the UI, but it's up to you to implement the rest…

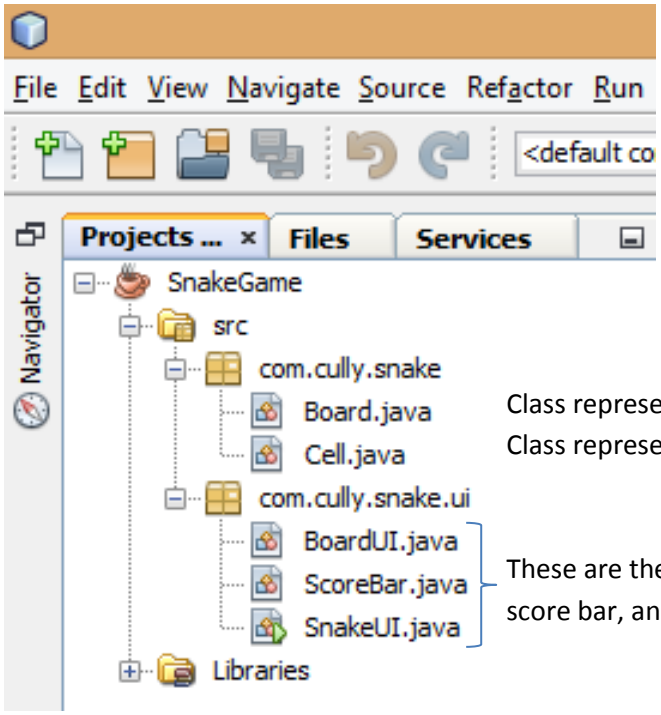
- First, please grab a copy of **SnakeGame.zip** off the Engine, and extract the contents. It doesn't really matter where you extract it to, just as long as you remember where it is for the future. This archive contains the Netbeans project with some skeleton code that creates the UI and starts the game.

- Now open **Netbeans 7.0** from the start menu, and go to *File > Open Project*, then navigate to the location of the SnakeGame files you extracted.





You should now be able to see the following files if you expand the Projects tree:



Class representing the game board

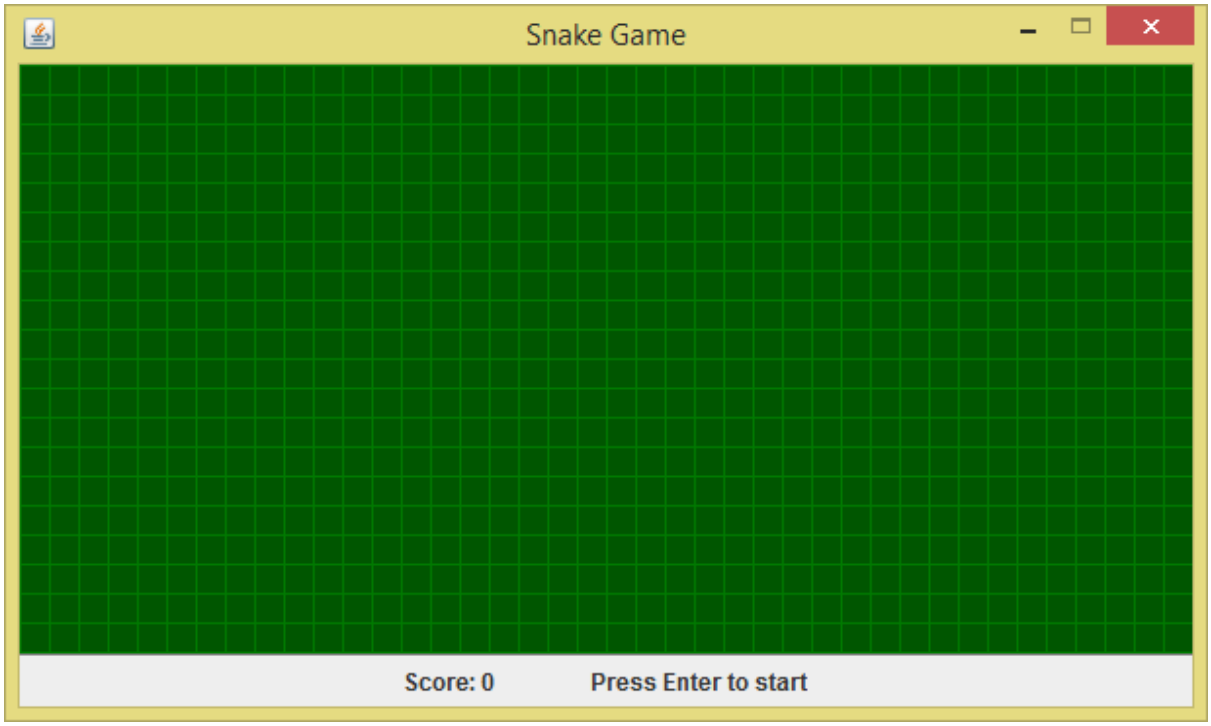
Class representing a square "cell" on the game board

These are the UI classes, representing the board, the score bar, and SnakeUI represents the actual window.

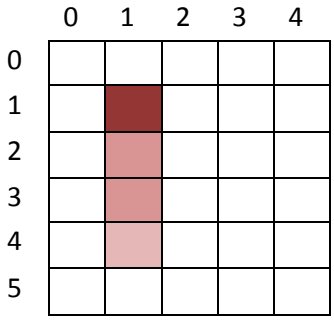
Feel free to look through these files now, to get a picture of how the code is laid out, and how the different classes interact. There will be a lot of unfamiliar syntax, which I aim to explain in this worksheet as we go along. The code is commented to give a description of the methods and what they do.

If you open one of the classes in the editor, you will be able to run the program (by pressing the green arrow button in Netbeans) and should see the following:

This is just the basic UI with a grey background, but soon you'll cover how to create the game grid:



This grid allows us to store some **state** (basically information such as what is in that cell) about the game, and we can access that information in a grid by accessing it with a row number and column number:



Here you can see an example of a Snake on a board, with a list of the co-ordinates of different types of Snake piece below, the left value is the row, and the right is the column:

(1, 1) – Snake Head

(2, 1) – Snake Body

(3, 1) – Snake Body

(4, 1) – Snake Tail

This is the purpose of the 2-Dimensional array of Cell objects that you can see in the `Board` class (the array is called grid), so we will be able to access the cells using row and cell co-ordinates (which will be useful when we want to loop over all the cells to draw them!).

Tasks

- ## Change the title of the window

In the `SnakeUI` class, we use the `setTitle()` method to set the title of the window. We do this by passing a String as an argument to the `setTitle()` method. Currently the title is "Your Title Here", but you should change this to something appropriate.

In the SnakeUI class:

- On line 35, change the call to the setTitle method
- Run the program afterwards so you can see the results.

- ## Set up the grid array

In the `Board` class, we want to store a "grid" made up of cells to keep track of what each cell contains. We want to set this grid up when the Board object is created, so we want to add code to the constructor (the method with the same name as the class).

The grid variable in the Board class is a 2D array, like the second example above. You can use the rows and cols variables to declare the sizes (i.e `new Cell[rows][cols]`) .

In the Board class:

- In the Board constructor (the method called Board):
  - Declare a new 2D array of type Cell, you can use the following code:
    - `this.grid = new Cell[rows][cols]`

Note: the grid variable has been declared as a class variable, or **field**. A variable is a field if it is declared outside of any methods in the class, and the variable can be accessed or changed in any method in that class. This means you can just declare it as 'grid = …', as it has already been declared on line 17:

```
14      // This variable will hold the actual grid behind the game board!
15      // Notice it is made up of 'Cell' objects, a custom class that we
16      // will be extending!
17      private Cell[][] grid;
```

Populate the array

Now that we've told Java how many items will be in the array, it's time to create the Cell objects that make up the array. We want to create a new **Cell object for each space in the array**, and if you look in the Cell class you'll see the constructor (shown below) takes three arguments: The size of a side (e.g the height of each cell in pixels), and the row and column index of the Cell (in the array).

```
public Cell(int sideSize, int row, int col) {
```

The easiest way to do this is to loop over all the row and column values. We can do this by **nesting** one for loop inside another. What we want to do inside the for loop is to create a new Cell object at each index. We want to do this to allow each Cell to be responsible for drawing itself.

In the Board class:

- After your code form the previous task:
- Write a nested for loop as follows:
  -
```
for (int row = 0; row < rows; row++) {
    for (int col = 0; col < cols; col++) {
        // Create a Cell object and place it
        // in the array at position [row][col]
    }
}
```
- Inside the for loop:
  - Create a new Cell object and place it in the array at the position [row][col]
  - The code for this is:
    - grid[row][col] = new Cell(cellSize, row, col);

- **Draw the Cells on screen**

4.1 Create a method to get the Cell objects

This first bit deals with the Board class. As the grid variable is private, only methods in the Board class can access it. We'll need to be able to access the Cell objects from another class in the next part, so we should create a new method in the Board class called something along the lines of getCell. It should return a Cell object, which can be accessed from the grid array, using the row and col variables.

In the Board class:

- Create a new method called getCell
  - It should return a Cell object, and take two integer arguments: one called row and one called col

```
/**
 * Return the Cell object at the given coordinates
 */
public Cell getCell(int row, int col) {

}
```

  - Inside the method, return the Cell object in the grid array at position [row][col]

4.2 Set up the Cell class

There is the skeleton code for the `Cell` class included in the project. This has a constructor, and one other method, drawCell, which will be explained in a bit. Firstly we need to create **fields** for each of the values that are passed into the constructor. Remember a field is a private variable, almost always declared at the top of the class, that can be accessed by other methods in the class:

```
/**
 * Class representing a square c
 * @author Sean
 */
public class Cell {

    // Declare your fields here
    private final int sideSize;
    private final int row;
    private final int col;
```

You can then assign a value to your field in the constructor:

```
this.sideSize = sideSize; // sideSize is one of the arguments in the constructor
```

Using this.sideSize means that Java knows we want to use the field called sideSize, rather than the method argument called sideSize.

In the Cell class:

- Create a new field for each of the arguments in the constructor method
  - E.g private int sideSize;
- In the constructor, assign the value of the argument to the field of the same name
  - E.g this.sideSize = sideSize;

The other method in the Cell class is drawCell(). This method takes a Graphics object as an argument, which allows us to draw the cell onto the screen! One of the benefits of using an Object Oriented approach such as this is that each Cell object only has to worry about drawing itself.

There are instructions in comments in the Cell class for how to implement the method. Read through the comments to see how you can draw the rectangle on the screen. Because the Cell object knows which row and column it resides in, we can use this to calculate where on the screen to draw it:

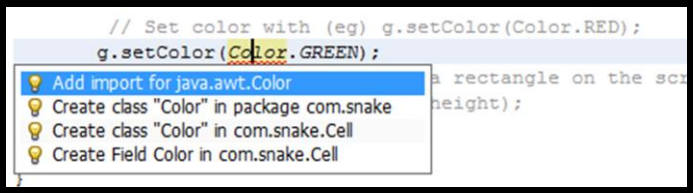
```
int x = (col * sideSize);

int y = (row * sideSize);
```

Make sure you get the row and col the right way around here. The x value is how far along the screen we are, so we start drawing after space for 'col' number of cells.

In the Cell class:

- In the drawCell method (read the comments for the necessary code)
  - Set the colour to Color.GREEN with g.setColor
    - This tells Java what colour to draw on the screen
    - The Color class will have a red underline because it needs to be **imported.**
    - You can get netbeans to do this for you, by moving the cursor to where you typed Color, and press Alt + Enter, which should give you an option as follows:



    - You can then press enter to select 'Add import for java.awt.Color'
  - Use g.fillRect to draw a **square**, below is a list of the parameters:
    - x – the x position of the Cell (check the instructions above)
    - y – the y position of the Cell
    - width – the width, use sideSize
    - height, the width, use sideSize

When you import a class, as you just did above, that tells Java where to find the class you are importing. In this case the class Color is in one of the standard java packages: java.awt.

4.3 Let's draw a Cell!

We've written all the code for this part so far without really being able to test if it works or not, so it's time to change that by drawing a cell onto the screen.

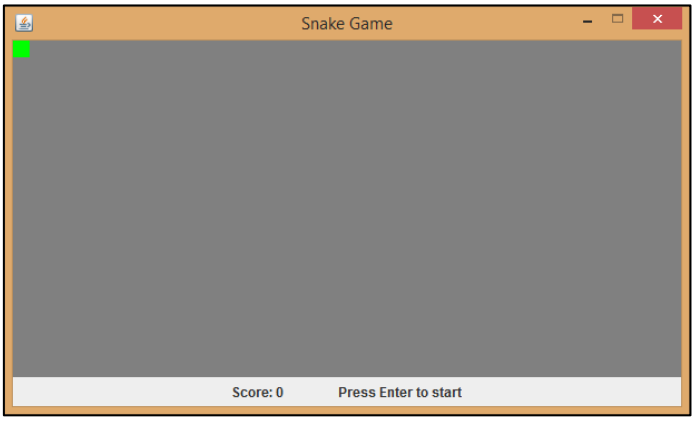
This time we'll need to change the `BoardUI` class, which is responsible for displaying the board (i.e the grid).

In the BoardUI class:

- Create a new Cell object in the paintComponent() method
  - For parameters you can just use 15, 0, 0 for now (meaning the cell is 15 x 15px, and is in row 0 and column 0)
- Call the method drawCell(g) on the cell object you just created
  -

```
Cell cell = new Cell(15, 0, 0);
cell.drawCell(g);
```

Then if you run the code it should look like this. If it doesn't then have a look at your code and see if you can figure out what's wrong, but if not then just ask!



4.4 Draw ALL the Cells

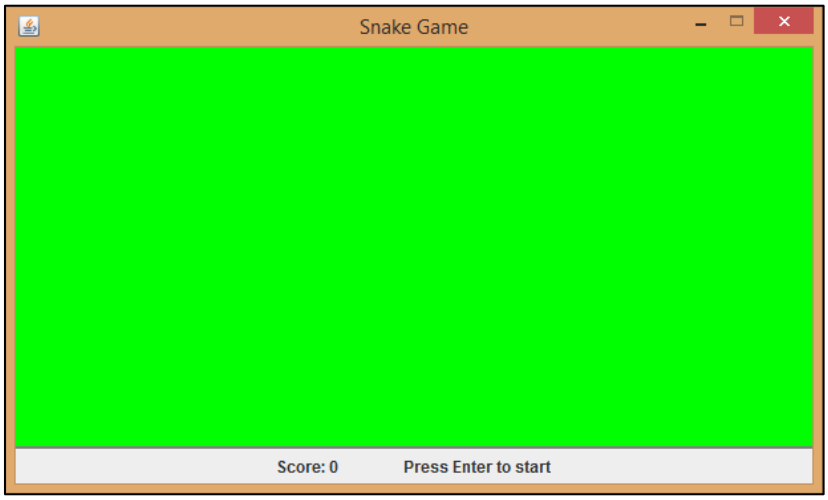
Now, instead of just drawing one Cell object that we create, we want to draw all the cells in the grid (which is in the Board class). If you look at the constructor method for BoardUI, you can see that it is passed a Board object, and the number of rows and columns in the board. You should create fields for these values, as you did in the Cell class.

In the BoardUI class:

- Create a new field for each parameter in the BoardUI constructor
- Assign the value of the parameter to the field
- Comment out your old code in the paintComponent method
- Write nested for loops, like you did earlier
  - Remember we want to loop from 0 to rows in the outer loop…
  - … and 0 to cols in the inner loop
- Inside the for loops:
  - Call the method getCell(row, col) on the board field you created earlier and assign it to a new Cell variable
    - E.g Cell cell = board.getCell(row, col);
  - Call the drawCell method on the Cell as you did before

Remember the method you wrote earlier to get a Cell from the Board? Now it's time to use it.

When you're done, it should look like this:



Optional: You could also change your drawCell() method to draw a border around each cell as well. To draw a border you can use the method g.drawRect(), which will take the same parameters as before, but draw just the outline of a rectangle instead. Before doing this you will want to set the colour to something else (e.g `Color.GREEN.darker()` gives you a darker green!).

• Create CellType enum in Cell class

Since the Cell class is responsible for drawing the Cell onto the screen, we want the Cell to know what type of Cell it is… i.e is it empty, or is it part of the Snake, i.e the head, body, or tail? One way of

doing this is assigning a number to each type, e.g 0 is empty, 1 is the Snake head, 2 is the Snake body, etc.

However, this would mean we would need to remember which number corresponded to which. Java (and many other languages) have the concept of an **enum** or enumerated type, which gives us a way to limit the possible values of a variable to those contained in the enum. A common example is compass directions, which would be declared as follows (it is Java convention to uppercase all your enum values):

You have been given the CellType.java enum, it is up to you to fill in the values!

```
/**
 * This is an enum file representing the different types of Cell. You will fill this in
 * as part of an exercise.
 */
public enum CellType {
   DECLARE, DIFFERENT, TYPES, IN, HERE, SEPARATED, BY, COMMAS
}
```

In the CellType class:

- Add the following values (separated by commas, and typed in upper case):
  - EMPTY, HEAD, BODY, TAIL

In the Cell class:

- Create a field of type CellType (and called type), and assign it to EMPTY
  - E.g: private CellType type = CellType.EMPTY
- Create a new method called getType that will return the value of the type field
- Create a new method called setType that sets the type field to a new value
  - This new value should be assigned to the CellType parameter.

- **Change the Cell class to draw different CellTypes**

6.1 Refactor the drawCell method

Now that you have an enum, you can use this to allow the drawCell method to draw different types of CellType. First, you should **refactor** the drawCell method. Refactoring is the process of restructuring your code without changing the behaviour, and is an important part of developing good software, and being able to maintain it in the future.

In the Cell class:

- Create a new **private** method called fillCell
  - It should have a void return type
  - It should take a Graphics, and a Color object as arguments
- Move your code inside drawCell to fillCell
  - Remember to change the call to setColour to use the new Color argument!
- Call your new method from drawCell (e.g)

```
37          */
            public void drawCell(Graphics g) {
39
40              fillCell(g, Color.GREEN);
41
42          }
```

- Run the program again to make sure the Cell is still displayed correctly
- Run the program again, but pass a different color to the fillCell method
  - This is to make sure the Color parameter is actually being used!

6.2 Write a Switch statement

A Switch statement (or case statement in some languages), is an alternative to having many if / else if… / else branches, and is well suited to being used with enum types.

```
Direction dir = getDir();
switch (dir) {
    case NORTH:
        // Do something
        break;
    case EAST:
        // Do something
        break;
    case SOUTH:
        // Do something
        break;
    case WEST:
        // Do something
        break;
    default:
        // Do something
}
```

The syntax for a switch statement is shown below (using the Direction example from earlier):

You can see that we 'switch' on a variable, and then write code for different cases, and also a default case.

The '`break;`' instruction is very important, as if you don't have this then the switch statement will just continue executing until it hits a break statement, or the end of the switch statement. (e.g if you wanted to do something for the case NORTH, but didn't have a break then it would also execute whatever code is in the EAST case.

In the Cell class,:

- Create a switch statement for the different types of CellType
- You should 'switch' on the CellType field you created earlier.
- Inside each case, call fillCell with a different colour, so you can see the different types of CellType!

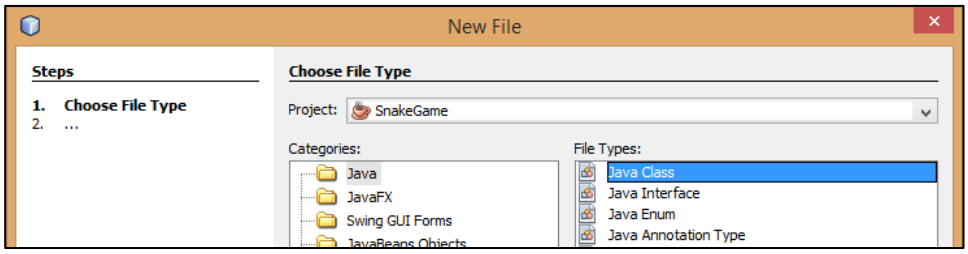
```
switch (type) {
    case EMPTY:
        fillCell(g, Color.GREEN);
    case HEAD:
        fillCell(g, Color.RED);
        break;
    // Rest of cases
}
```

-

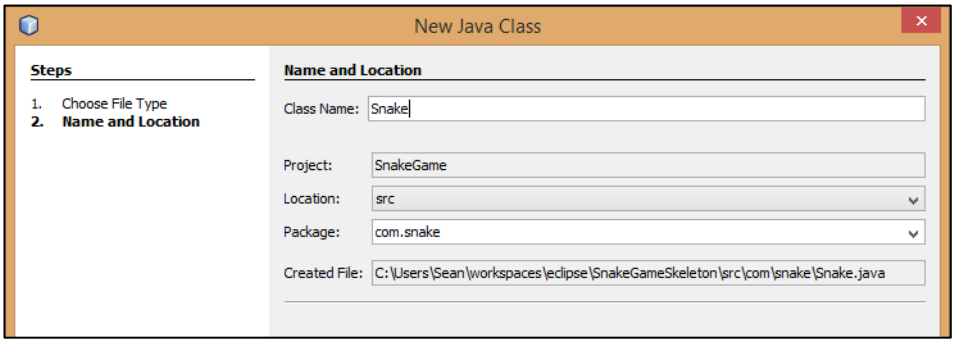
- **Place the Snake on the board**

7.1 Create a Snake class

To create a new class in Netbeans, go to File > New File… and select Java class:



On the next screen, call the new class Snake, and make sure package is com.snake, not com.snake.ui



In the Snake class:

- Change the top line to read 'public class Snake extends AbstractSnake {'
  - This is to allow the testing program to run
  - You might need to put the cursor over AbstractSnake and press ALT + ENTER, then import the AbstractSnake class
- Add a constructor method that has a Board parameter
- In the constructor:
  - Assign the value of this Board parameter to a new field

7.2 Create a new Snake object

In the Board class:

- In the Board class:
  - Create a new field with the type 'Snake', but don't assign anything to it yet.
    - This is so the board class always has access to the Snake object.
  - In the constructor method (the method called Board):
    - Create a new Snake object
  - Your code should look similar to this:

```
private Snake snake;

public Board(int cellSize, int rows, int cols) {
    snake = new Snake(this);
}
```

7.3 Create a 'getSnake()' method

This allows other classes to access the Snake object!

In the Board class:

- Create a new method called getSnake()
  - It should return a Snake object
- Inside the method, return the snake field you just created

7.4 Set the Snake head

Now it's time to place the Snake's head on the board, and then we can test to see if the code has worked! You'll want to think of row, column co-ordinates to place the Snake's head, and make sure that there are another 2 spaces free next to it (one for the body, and one for the tail).

In the Snake class:

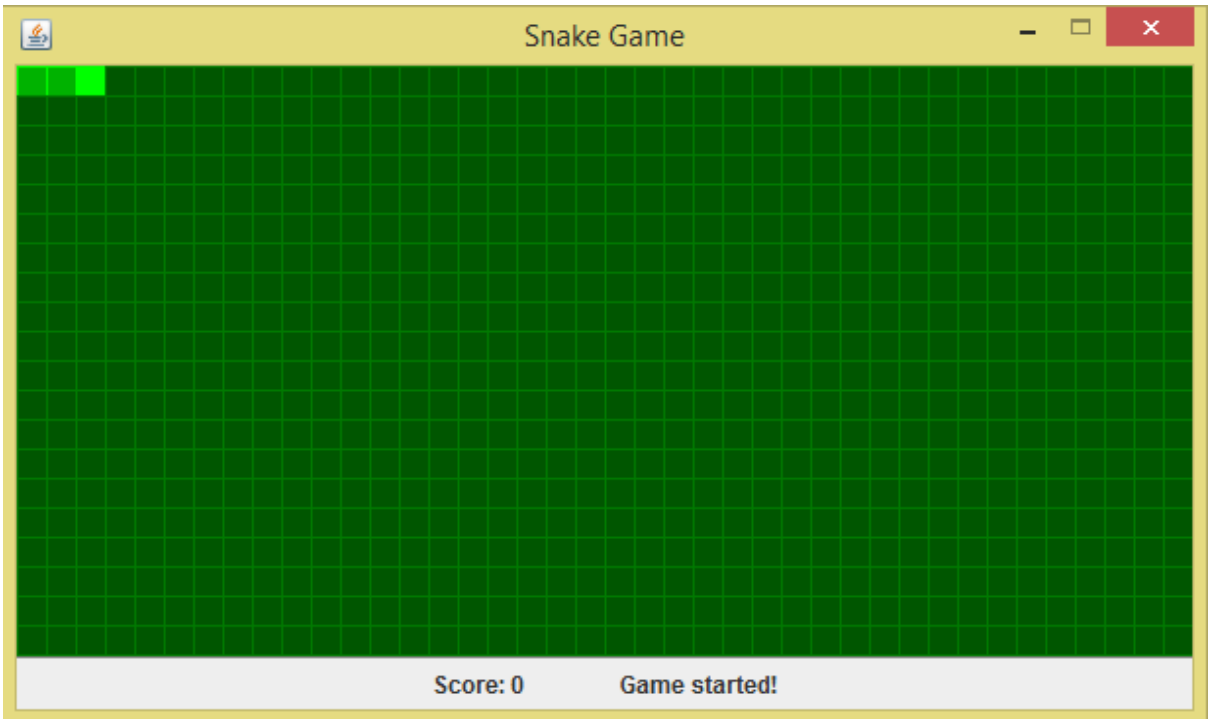
- Create a new method called place()
  - The return type will be void
- Inside the place method:
  - Create a Cell variable, and use board.getCell(row, col) to get the Cell at your co-ordinates
  - Use the setType method on the Cell object to set it to CellType.HEAD

7.5 Call the place() method!

Now that we've filled in the place method, it's time to actually call it!

In the Board class:

- Inside the start() method:
  - Call the place() method on the snake field
  - This calls the method you just wrote!
- Run your code to test it – you should be able to see the Snake's head appearing as a different colour!



8. Now do the same for the Snake's body and tail!