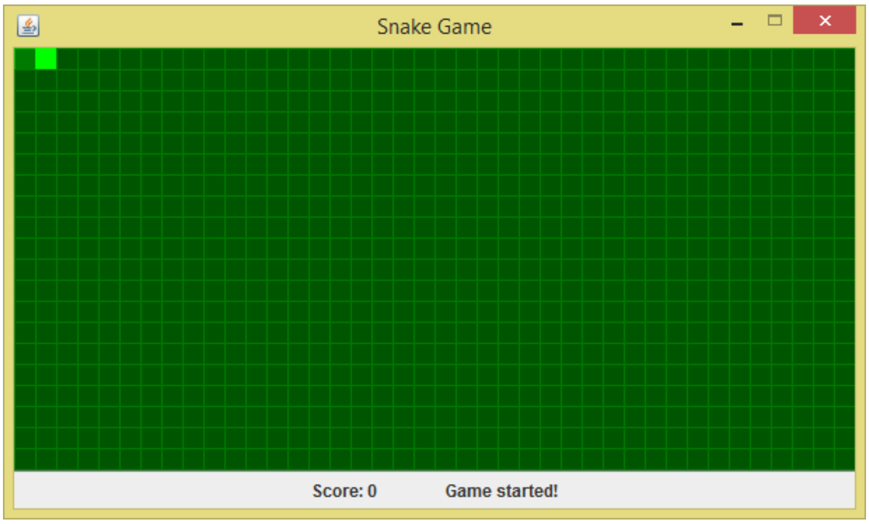


Score: 0 Game started!

Introduction

Now that the Snake game is starting to get interactive (it responds to keyboard input), it's time to start polishing the game so it looks more finished. In this worksheet we'll cover some simple Game Over scenarios (leaving the board, eating another piece of Snake), moving all of the Snake, eating food and growing the Snake once it eats food.

Tasks

3.1 Stop the game when the Snake leaves the board

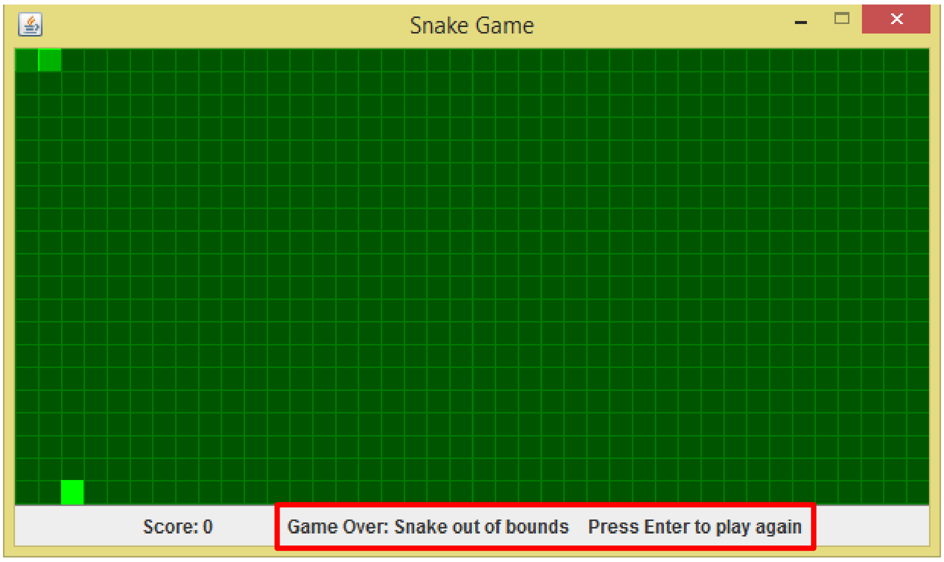
When the Snake reaches the end of the board, we want that to cause Game Over. One of the simpler ways to do this is using Java exceptions. Exceptions are basically errors that are caused when the program runs. One type of exception that we're interested in is called an ArrayIndexOutOfBounds exception, meaning that we have tried to access an index in the array that doesn't exist, e.g -1 or any number larger than the size of the array. In java, if we know that an exception will occur, we can tell java to try and do something, then specify what to do if we "catch" the exception. In this case, we know that if we try to move the Snake outside the board, we will get an ArrayIndexOutOfBoundsException because we use the 2D grid array to store each cell.

In the TimerListener class:

- Inside the moveSnake() method
    - Replace your call to snake.move() with the following code:

```
try {
   snake.move();
} catch (ArrayIndexOutOfBoundsException ex) {
   throw new GameOverException(null, "Snake out of bounds");
}
```

You should now run the Snake game, and move the Snake's head outside the board, which will allow us to catch the ArrayIndexOutOfBoundsException mentioned above. If you've succeeded, the message on the StatusBar will be changed:

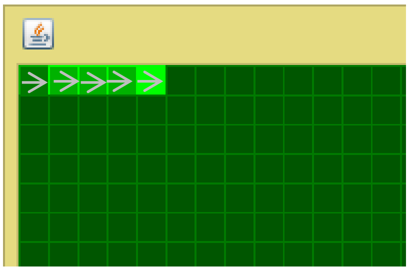


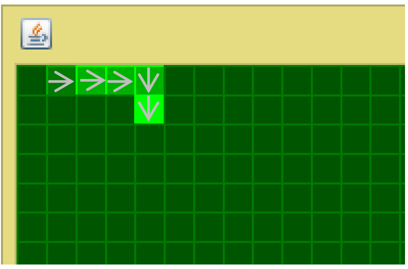
Game Over: Snake out of bounds   Press Enter to play again

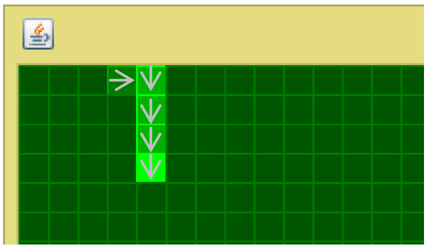
3.2 Create a method to help us change the Snake's direction

The Snake's head already responds to keyboard input, but now we want to change it so that all of the Snake moves. We're going to do this by changing the direction of each of the Snake's body pieces, starting from the tail and moving up, copying the direction of the (Snake) cell in front of it. This is so that the Snake will 'remember' where you have turned, and remember to change direction at that Cell. The code for this one is quite complex, so I will include a full solution in the hint, but try to complete it without looking at the hint first.

There's still another couple of steps before we get to see the Snake move when we run the program, but it hopefully the image below will explain a bit more about what is meant about the Snake remembering which Cell it turned at. The first image below is my initial Snake (I added a couple of extra body cells to make it clearer). The next is the result of pressing the down arrow before the Snake moved (and waiting 1 game tick), so the head Cell moves down one row, and the next body cell remembers that the Snake changed direction here. The final image is after a few more game ticks, all but the tail Cell are now facing south.







In the Snake class:

- Create a new public and void method called updateDirections()
    - public void updateDirections()
- Inside the updateDirections() method:
    - Set the tail direction to the direction of the last body piece. The index of the last body piece is 'body.size() – 1'.
        - Create a new Cell variable called lastBodyCell
        - Assign that (using '=') to `body.get(body.size()-1)`
        - E.g '`Cell lastBodyCell = body.get(body.size() -1);`'
        - Now use the following to set the tail's direction:
            - `tail.setDirection(lastBodyCell.getDirection());`
    - Loop through between body.size() – 1 and 1 (code shown below)
        - To change the direction, we want to set each body Cell's new direction to the direction of the Cell before it

        -
          ```
          for (int i = body.size() - 1; i > 0; i--) {
          }
          ```
        - Inside the for loop (i.e between the { and } brackets) we want to do the same thing as before, but for the body cells:

            -
              ```
              Cell currentBodyCell = body.get(i);
              currentBodyCell.setDirection(body.get(i-
              1).getDirection());
              ```
    - As the for loop doesn't include the body cell at index 0 (i.e the first body Cell), so we want to set this to the head cell's direction (after the loop):
        - `body.get(0).setDirection(head.getDirection());`
    - And finally set the direction of the head Cell to the value in the dir field:
        - `head.setDirection(dir);`

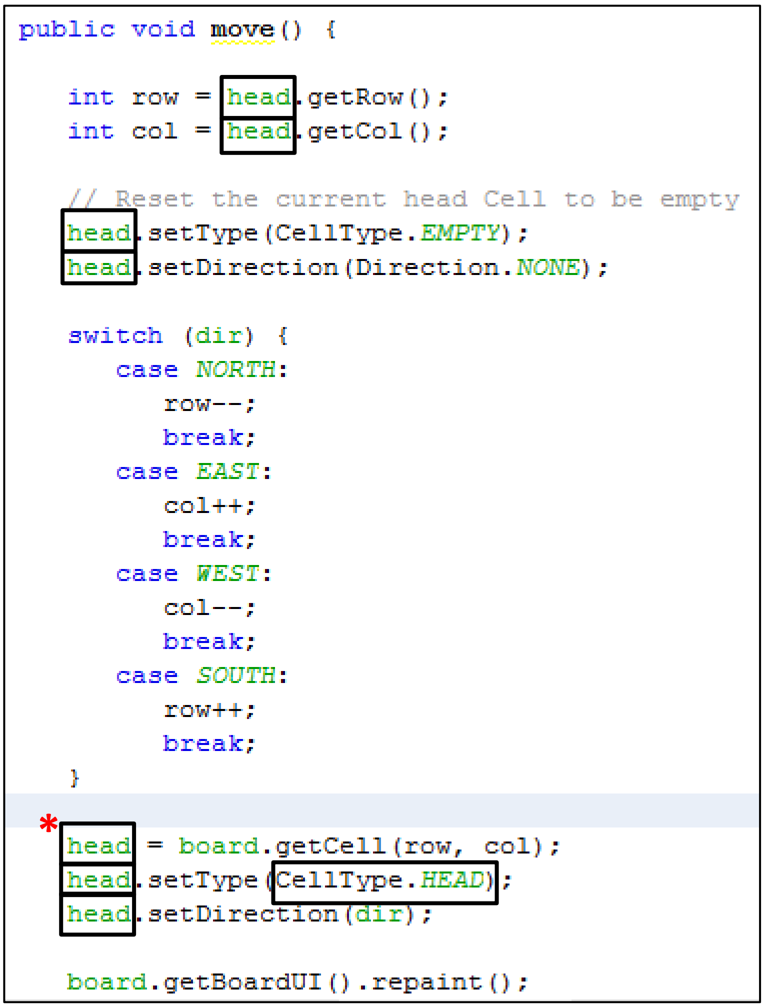
In the **TimerListener** class:

- In the moveSnake() method:
    - On the line before the call to 'snake.move()':
    - Call '`snake.updateDirections()`'

3.3 Move ALL of the Snake

Now that we're using the method to update the direction of all the Cell objects in the Snake, it's time to actually move them! This will require some refactoring work, which means changing the code to give it better structure; in this case by extracting some code from the move method out in to a new method. For this task, the instructions will be in this description rather than in an instruction box, as the changes are more complicated than previous ones.

At the minute, your move() method should look something like the following:



```
public void move() {

    int row = head.getRow();
    int col = head.getCol();

    // Reset the current head Cell to be empty
    head.setType(CellType.EMPTY);
    head.setDirection(Direction.NONE);

    switch (dir) {
        case NORTH:
            row--;
            break;
        case EAST:
            col++;
            break;
        case WEST:
            col--;
            break;
        case SOUTH:
            row++;
            break;
    }

   *head = board.getCell(row, col);
    head.setType(CellType.HEAD);
    head.setDirection(dir);

    board.getBoardUI().repaint();
```

This move method at the minute just moves the head Cell, but we want to be able to do this same action for all Cells, so we want to extract this into a new method.

To do this, we can create a new method, called moveCell(), that takes 2 parameters: a Cell object as a parameter, called cell; and a CellType parameter called type. To make things easier, it should also return the Cell object at the new co-ordinates.

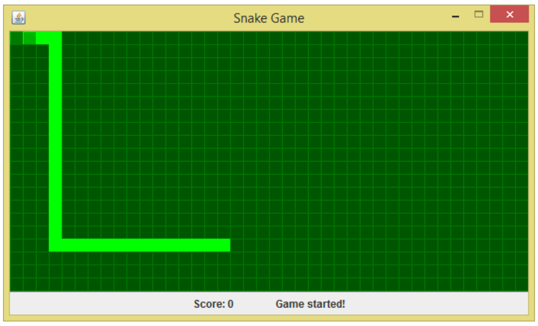
Inside moveCell(), you can copy all the code inside move(), except for the last line (`board.getBoardUI().repaint();`).

For the first 4 times you see the field ‘head’ being referenced (highlighted in red), you need to change it to ‘cell’ (the name of the Cell parameter). This is all the red highlights before the one with the asterisk. For the last 3 lines here, head should be changed to a new variable, such as newCell. In the line with the asterisk, you **also need to declare the type**, i.e ‘`Cell newCell = board.getCell(row, col);`’. Where you see ‘CellType.HEAD’, change it to ‘`type`’ (the name of the CellType parameter).

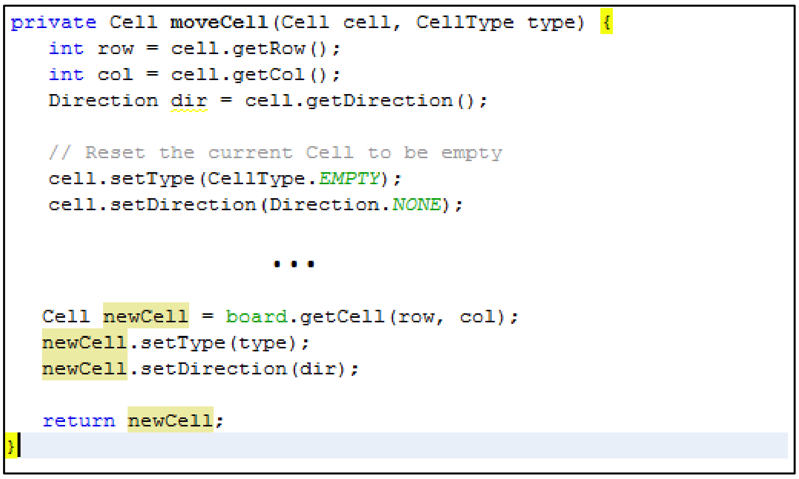
Also, make sure that after the first two lines (starting with int row…, int col…), you will need to create a Direction variable called dir, and assign it to cell.getDirection();. At the end of your moveCell() method, you should return newCell.

Now, to test that it works, go back into the original move() method (**not** moveCell()), and at the start, call your new moveCell method(). I.e, in the move() method, the first line should be ‘head = moveCell(head, CellType.HEAD);’

If you forget to change all of the occurrences of ‘head.’, you will start to get strange behaviour:



Your moveCell() method should look something like the following (I've omitted the switch statement, which should also be in there):



```
private Cell moveCell(Cell cell, CellType type) {
    int row = cell.getRow();
    int col = cell.getCol();
    Direction dir = cell.getDirection();

    // Reset the current Cell to be empty
    cell.setType(CellType.EMPTY);
    cell.setDirection(Direction.NONE);

                     ...

   Cell newCell = board.getCell(row, col);
   newCell.setType(type);
   newCell.setDirection(dir);

   return newCell;
}
```

3.4 Update the move method

Now that we've created the moveCell() method to help us move the Snake, the next thing to do is to actually move the Snake! We do this by updating the move method in the Snake class. For each part of the Snake, we want to do 2 things:

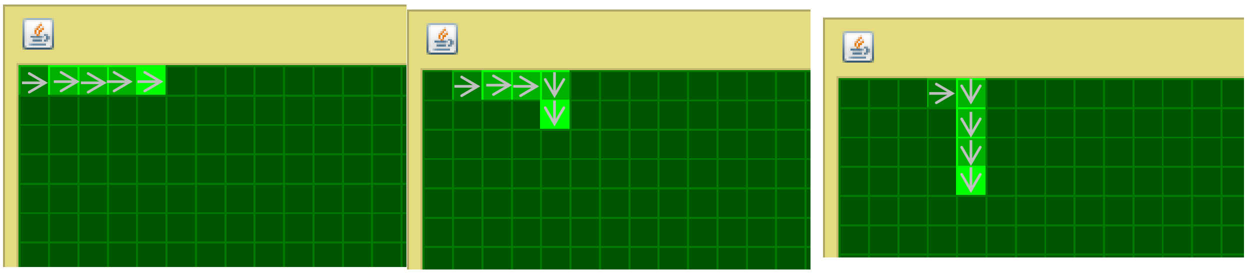
1. Move the body part to the next Cell
2. Make sure the head, body and tail fields still hold the correct Cell objects.

This second part is why we return the Cell after we have moved it, to make it easier to update the fields.

In the Snake class:

- In the move() method:
- First, move the Snake's head:
    - Call the moveCell() method on the Snake's head (held in the 'head' field), passing in CellType.HEAD as the second parameter.
    - Assign this to the head field (so the head field contains the new Cell object representing the head)
    - E.g head = moveCell(head, CellType.HEAD);
- Next, the body – as this is a list it's a bit more complex
    - Write a for loop, looping between i= 0, and I < body.size()
    - Inside the for loop, create a new Cell variable called bodyCell and assign it to 'body.get(i)'
    - Then (still inside the for loop) create a Cell variable called newBodyCell, and assign it to 'moveCell(bodyCell, CellType.BODY)'
    - Finally (still inside the loop) call body.set(i, newBodyCell);
        - This last bit replaces the old body cell (before it was moved) with the new one
- Finally, the tail:
    - Same as for the head, but remember to update the **tail** field instead, passing in the tail field and CellType.HEAD.

Now try running your Snake game, if all goes well then your Snake should move around correctly. By this I mean that when you change direction, the game should remember which Cell you turned at, and always turn at that Cell (until the Snake has passed over it). The arrows on the image below show the direction each Cell is facing at that point.



3.5 Getting hungry? Let's add food!

Now that the Snake moves around, it's time to start adding food onto the board for the Snake to eat. We can split this task into 4 general steps:

- Placing the food
- Collision detection, to tell if the Snake has hit any food (or itself!)
- If the Snake has hit any food, update the Score and then…
- Place more food!

Notice that the first and fourth steps are exactly the same! This means that it's a good idea for us to put this in a separate method, that way once we have written it once we can then just call the method when we need to place more food. For this task we'll just create this method.

We are going to randomly place the food in an empty Cell, by using the 'random()' method in the Math class. This generates a random number between 0 and 1, so we can multiply this by the rows and cols fields to get a random row index and column index. This is a static method, meaning that we don't have to create a 'Math' object to call the method, we can just use 'Math.random()'.

Another thing that we haven't previously used that we will need is a different type of loop. We've already seen for loops, but this time we're going to look at a **while loop**. The syntax for a while loop looks like this:

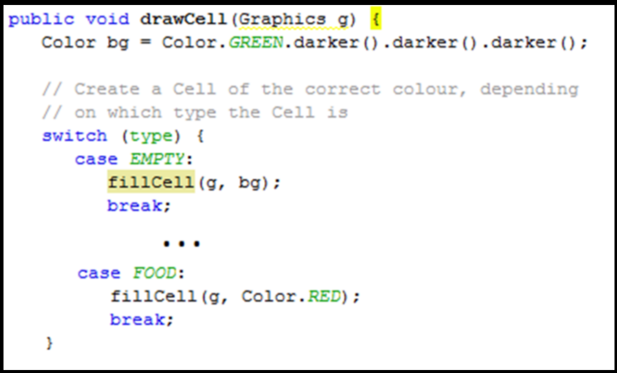
```java
boolean condition = false;
while (!condition) {
   // Do something
   // Remember that at some point you need to change the condition to
   // true otherwise you'll get an infinite loop.
   condition = true;
}
```

- In the **Board** class:

  Create a new public and void method called placeFood():
- Inside the placeFood method:
  - Create a new Boolean variable called emptySpaceFound, and set it false
    - We are g oing to use this variable as a condition for a while loop
  - Create 2 new int variables, called row and col, and set them both to -1
  - Write a while loop, e.g while (!emptySpaceFound) {

    }
  - Inside the while loop:
    - Assign the row variable to ‘Math.random() * rows’
      - This gives you a random row index between 0 and rows
    - Now do the same for the col variable, and assign it to ‘Math.random() * cols’
    - We still need to update the ‘emptySpaceFound’ condition:
      - emptySpaceFound = getCell(row, col).getType() == CellType.EMPTY;
  - After the while loop we want to set the cell as a food cell… but first we need to add ‘FOOD’ to the CellType enum (in CellType.java)
  - Now , use the following line to set the Cell to be of type ‘FOOD’
    - getCell(row, col).setType(CellType.FOOD);
  - In the start() method (at the end):
    - The last thing we need to do now, in the Board class , is to call placeFood() so that the food actually gets placed!

Remember the drawCell() method (**in the Cell class**) that we use to draw each different type of Cell? We still need to update this to cope with FOOD, by adding a new FOOD case statement. It should look similar to the following example:



```
public void drawCell(Graphics g) {
   Color bg = Color.GREEN.darker().darker().darker();

   // Create a Cell of the correct colour, depending
   // on which type the Cell is
   switch (type) {
      case EMPTY:
         fillCell(g, bg);
         break;

              ...

      case FOOD:
         fillCell(g, Color.RED);
         break;
   }
```

3.6 Check for collisions and update score!

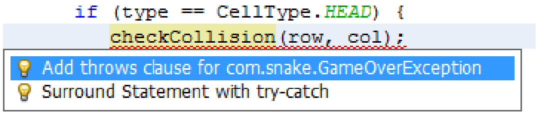
Now we’ve got the food appearing, but without checking for collisions the food doesn’t get replaced, and our score doesn’t get updated! To check for collisions, we will check the new head location before actually moving the Cell. If the Cell is food, then we want to update the score, and (in future) extend the Snake by 1 piece. However while we're checking for collisions we may as well check to make sure that the Cell is empty (using an else if attached to the first if statement):

```
if (cell.getType() == CellType.FOOD) {
   // Update score and place food
} else if (cell.getType() != CellType.EMPTY) {
  throw new GameOverException(board, “Collision detected!”); }
```

In the **Snake** class:

- Create a new private int field called score, and assign it to 0.
- Create a new public and void method called checkCollision()
    - It also throws a GameOverException:
    - public void checkCollision(int row, int col) throws GameOverException {
- It should take 2 int parameters, one called row and one called col
- Inside the method:
    - Create a new Cell variable called cell, and assign it to the Cell held at the row and col passed in:
        - Cell cell = board.getCell(row, col);
    - Use an if statement to check if the Cell at that location is food (check the example above to see the if / else if statement syntax
        - If (cell.getType() == CellType.FOOD) {
          }
    - In the if statement (between the curly brackets), do the following:
        - Update the score field by a number, e.g 10
        - Update the score bar:
            - ScoreBar.getInstance().setScoreLabels(score);
        - Call board.placeFood() to place the next bit of food
    - In the **else if** (cell.getType() != CellType.EMPTY) statement, do the following:
        - This means it's Game Over! Luckily we have a GameOverException that we can throw, which will stop the game if the Snake runs into itself:
        - throw new GameOverException(board, "Collision detected!");
- Now it's time to use this method inside the moveCell() method
    - Put the following lines of code just after the switch code – this checks that it is the Head before checking for collisions with the new row and col indexes.
    - If (type == CellType.HEAD) {
         checkCollisions(row, col);
      }

Note: because the checkCollision method throws an error, we're going to have to change the moveCell() and move() methods to throw the exception as well. This will lead to the call to checkCollision() being underlined in red. To easily solve this, press CTRL + ENTER, and select 'Add throws cause for com.snake.GameOverException'. You will also need to do the same thing in the move() method.



```
if (type == CellType.HEAD) {
    checkCollision(row, col);
```

Add throws clause for com.snake.GameOverException
Surround Statement with try-catch

3.7 Make the Snake grow when it eats food

Now that the Snake can move around and eat food, the next thing to do is to make the Snake grow by 1 piece every time it eats some food. This is quite easy to do as it turns out, all we need to do is modify the checkCollision() method to add the current tail Cell to the body, and set the type of the Cell to BODY.

In the **Snake** class:

- In the checkCollision() method:
  - Inside the 'if (cell.getType() == CellType.FOOD) {' statement, after setting the score label (but before placing more food), do the following:
    - Add the current tail cell to the body, using body.add(tail);
    - Now we want to set that cell as a body cell, for ease, we can do this as follows:
      - tail.setType(CellType.BODY);
      - This works because (at the moment) the tail and last body piece are the same Cell, though this will change next time the snake moves.