Com S 227
Fall 2021
Assignment 3
300 points

Due Date: Friday, November 5, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Nov 4)

10% penalty for submitting 1 day late (by 11:59 pm Nov 6)

No submissions accepted after November 6, 11:59 pm

(Remember that Exam 2 is MONDAY, November 8.)

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

*Note: Our second exam is Monday, November 8, which is just two days after the late due date for this assignment. **It will not be possible to have the assignments graded and returned to you prior to the exam**.*

Please start the assignment as soon as possible and get your questions answered right away!

Introduction

The purpose of this assignment is to give you some practice writing loops, using arrays and lists, and, most importantly, to get some experience putting together a working application involving several interacting Java classes.

There are three classes for you to implement: **`Snake, SnakeBasket, and SnakeUtil`**. As always, your primary responsibility is to implement these classes according to the specification and test them carefully.

These two classes can be used, along with some other components, to create an implementation of our version of a puzzle game called "Snake Escape."

The game is played on a 2-dimensional grid of cells. Each cell may be empty, be a wall, or may be occupied by a snake. One of the snakes is designated as the "green" snake. One cell is designated as an exit. The goal is to move the green snake through empty cells to the exit. Naturally, other snakes may be blocking the path and may need to be moved. A snake can be moved forward by grabbing its head, or backwards by grabbing its tail.

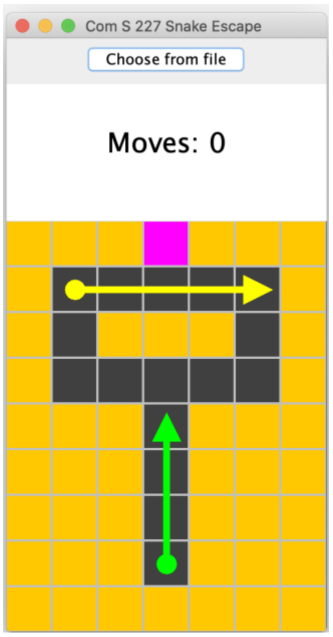

Figure 1 - a game instance as rendered by the sample GUI

We also may have designated cells containing either *apples* or *mushrooms*. A snake can move headfirst into a cell containing an apple, but the length of the snake increases by 1. A snake can also move headfirst into a cell containing a mushroom, and its length decreases by 1.

As of the time of this writing, there is an implementation of a similar game called "Snake Slider" available online here, if you want to try it out.
http://www.bdrgames.nl/html5/snakeslider/index.html
Disclaimer: when you go to implement your code, be careful to follow ***this*** *specification. We do not claim that our specification exactly matches the game linked above or any other online version, or anything else in the universe.*

The three classes you implement will provide the "backend" or core logic for the game. In the interest of having some fun with it, we will provide you with code for a GUI (graphical user interface), based on the Java Swing libraries, as well as a simple text-based user interface. More details below.

The sample code includes a documented skeleton for the classes you are to create in the package **`hw3`**. The additional code is in the packages **`ui`** and **`api`**. The **`ui`** package is the code for the GUI, described in more detail in a later section. The **`api`** package contains some relatively boring types for representing data in the game, along with a utility class with methods for creating the game grid and printing out the game in text format. There are a few examples of test cases for you to start with located in the default package, along with a simple text-based user interface.

You should not modify the code in the **`api`** package.

Specification

The complete specification for this assignment includes

- this pdf,
- the online Javadoc, and
- any "official" clarifications posted on Canvas and/or Piazza

The enumerated type api.Direction

To describe motion of a snake head or tail in the grid, we use the four constants **`Direction.LEFT`**, **`Direction.RIGHT`**, **`Direction.UP`**, and **`Direction.DOWN`**. These are defined as an "enumerated" type or **`enum`**. You can basically use them as though they had been defined as **`public static final`** constants. You can't construct instances, you just use the four names as literal values. Use **`==`** to when checking whether two values of this type are the same, e.g. if **`dir`** is a variable of type **`Direction`**, you can write things like

```
if (dir == Direction.UP)
{
  // do cool stuff
}
```

An **`enum`** type can also be used as the switch expression in a **`switch`** statement.

Tip: add the line

```
import static api.Direction.*;
```

to the top of your file. Then you can refer to the four constants without having to type "**`Direction.`**" in front of them.

Overview of the Snake class

A **`Snake`** object is basically just a list of grid positions. Each position is an object of type **`api.SnakePiece`**, which just encapsulates a row and column. Each snake also has a unique "id" which is a single character.

A snake's head or tail can be moved in one of four directions, and logically the rest of the pieces need to follow along. See the javadoc for details; also, there are some detailed examples in the Getting Started section.

Overview of the api.BasketCell class

(This class is already implemented and you may not modify it.) The grid on which the game is played is a 2D array of **`BasketCell`** objects. This is a simple type and you should familiarize yourself with the available methods. Basically, each cell has a status which is either *empty*, a *wall*, an *exit*, an *apple*, or a *mushroom*. Once constructed, the only way the status can change is that an apple or mushroom can be changed to empty via the **`removeFood()`** method. The actual status is represented internally by the **`enum`** type **`Status`**, but after construction this value is not directly accessible, so you should never actually need to use the **`Status`** type directly. The **`BasketCell`** class has many accessor methods such as **`isEmpty()`**, **`isExit()`**, etc. for determining its state.

In addition, a cell may be occupied by a snake, provided that the cell is empty or is an exit. When occupied, the cell has a reference to the **`Snake`** object that is "in" it.

Overview of the SnakeBasket class

The **`SnakeBasket`** class encapsulates the state of the game, including, in particular, a 2D grid of **`BasketCell`** and a list of **`Snake`** objects. The key operations in the game are

`void grabSnake(int row, int col)`

> If the given row/col represent either the head or tail of a snake, then that becomes the "current" snake, and subsequent **`move`** operations will apply to that snake's head or tail. (Whether it's the head or the tail that was grabbed is something you'll have to keep track of.)

`void releaseSnake()`

> Releases the current snake, if any, after which a different snake can be grabbed.

`void move(Direction dir)`

Attempts to move the current snake, by the head or tail, in the given direction. A move is only possible under the following conditions. (Below, where it says "adjacent cell", that means adjacent to the current snake's head or tail, as appropriate, in the direction indicated by **`dir`**.)

- The adjacent cell is *empty*; then the snake (head or tail) moves into the cell
- The adjacent cell is the *exit* and the current snake is the green one; then the snake (head or tail) moves into the exit and the game ends
- The current snake was grabbed by the head, and the adjacent cell has the row/column of the current snake's tail; then the snake (head) moves into the cell ("chasing its tail", so to speak)
- The current snake was grabbed by the tail, and the adjacent cell has the row/column of the current snake's head; then the snake (tail) moves into the cell
- The current snake was grabbed by the head and the adjacent cell is an *apple*; then the apple is removed and the snake (head) moves into the cell, increasing its size by one piece
- The current snake was grabbed by the head, consists of at least three pieces, and the adjacent cell is a *mushroom*; then the mushroom is removed and the snake (head) moves into the cell, reducing its size by one piece

Each completed move adds 1 to a move counter, accessible via the method **`getMoves`**, which is how the game is scored.

It is important to note that there is a bit of redundancy in the game state: each **`SnakePiece`** contains a row/column that is used to indicate its position in the grid, but each grid cell also includes a reference to the **`Snake`** containing that **`SnakePiece`**. The redundancy is intentional, since it dramatically simplifies the implementation. However, it is crucial to ensure that the information in the snake pieces and the information in the grid cells is always 100% consistent upon construction and after each move. That is the purpose of the method **`resetAllSnakes`**. Basically what this method should do is:

- call **`clearSnake()`** on every cell
- iterate over each snake and for each snake piece, call **`setSnake()`** on the cell at that piece's row/column

Note that this method should be called from **`move()`** after each successful move and can also be used to set up the snakes when initially constructing the game.

Construction of SnakeBasket instances.

A **`SnakeBasket`** is constructed using a string array descriptor. As an example, a descriptor for the game pictured in Figure 1 would look like this:

```
public static final String[] test1 = {
"#  #  #  E  #  #  #",
"#  y4 y3 y2 y1 y0 #",
"#  .  #  #  #  .  #",
"#  .  .  .  .  .  #",
"#  #  #  g0 #  #  #",
"#  #  #  g1 #  #  #",
"#  #  #  g2 #  #  #",
"#  #  #  g3 #  #  #",
"#  #  #  #  #  #  #"
};
```

Each string in the array consists of an equal number of whitespace-separated tokens (nonempty substrings). Tokens consisting of a *single character* have the following interpretations:

'#' – a wall
'.' – an empty cell
'@' – an apple
'$' – a mushroom
'E' – an exit

Tokens consisting of *two or more characters* represent pieces of snakes. The first character is the snake's id, and the number that follows is the index of the piece within the snake. For example, in the descriptor above, "g0" is the green snake's head, and "g3" is its tail.

The **`api.GridUtil`** class (*already implemented*) includes a static method **`createGridFromDescriptor`** that constructs a 2D array of **`BasketCell`** from a descriptor such as the one above. A call to this method is the first step in each of the **`SnakeBasket`** constructors. **This method does NOT, however, do anything with the snake information in the descriptor, it just sets up the grid**. In the first iterations of your implementation, you should use the two-argument constructor, for which you manually create and pass in a list of snakes. Your **`resetAllSnakes`** method can then add them to the grid. See SimpleBasketTest.java for an example.

Part of your assignment will be to implement the method **`SnakeUtil.findSnakes`**, which will read a descriptor and from it construct a list of snakes. Once you have this method done, you'll be able to construct a **`SnakeBasket`** using the one-argument constructor. This saves a lot of trouble in creating test cases, and also enables you to use the "Choose from file" option in the GUI. *Please note that you can get everything else working without this method!*

More about the api.GridUtil class and api.SnakeLayoutException

In addition to the **`createGridFromDescriptor`** method described in the previous section, **`GridUtil`** contains two additional public static methods. The **`validateDescriptor`** method is used to verify that a given string array has the required format for a valid descriptor. It is invoked at the beginning of **`createGridFromDescriptor`** and you should also invoke it at the beginning of your **`findSnakes`** method.

api.SnakeLayoutException

Notice that the **`validateDescriptor`** method returns **`void`**. If a descriptor is found to be invalid, the method throws a **`SnakeLayoutException`**. This is a custom exception type defined for this project. Thus, since your **`findSnakes`** method will start out by calling **`validateDescriptor`**, it may end up throwing a **`SnakeLayoutException`** as well, but it is not a "checked" type of exception so it does not require a **`throws`** declaration on the method. In addition, the **`findSnakes`** method is supposed to throw a **`SnakeLayoutException`** if any of the snakes is invalid, as defined by the **`Snake`** class **`isValid()`** method. We will study the exception mechanism in detail towards the end of the course; for now, all we need to know is that to generate an exception, you construct one and precede it with the **`throw`** keyword. For example, at the bottom of **`findSnakes`** you will need some code such as this:

```
for (Snake snake : mySnakeList)
{
  if (!snake.isValid())
  {
    throw new SnakeLayoutException("Invalid snake '" + snake.getId() + "'.");
  }
}
```

The GridUtil.display() method

The remaining method of **`GridUtil`** is the **`display()`** method, which takes a **`SnakeBasket`** object and prints out the current state of the grid using the conventions with which the descriptors are defined for representing the status of each cell and the snakes. This is useful for debugging and is used for the TextUI.

The method SnakeUtil.createDescriptorsFromFile

In addition to the **`findSnakes`** method, you'll implement this utility method. The idea is that you would have a file containing a number of text blocks representing descriptors, such as

```
#  #  #  E  #  #  #
#  y4 y3 y2 y1 y0 #
#  @  #  #  #  @  #
#  .  $  .  $  .  #
#  #  #  g0 #  #  #
#  #  #  g1 #  #  #
#  #  #  g2 #  #  #
#  #  #  g3 #  #  #
#  #  #  #  #  #  #
```

where each such block is preceded and followed by blank lines. The **`createDescriptorsFromFile`** reads the file and returns a list of string arrays, one for each such block of text, where each string in the array is one line of text. Note, however, that the method is actually quite a bit simpler than it sounds, since it is not required to do any validation of the file format. It just reads *any* text file by putting consecutive lines of text into a string array until a blank line is encountered. (In practice, the validation happens in the methods **`GridUtil.createGridFromDescriptor`** and **`SnakeUtil.findSnakes`**.)

The text-based UI

The default package includes the class **`TextUI`**, a text-based user interface for the game. It has a **`main`** method and you can run it anytime. In order to start playing the game, however, you'll need at a minimum:

- Enough of Snake implemented to add pieces and move the head or tail (does not depend on isValid)
- Enough of SnakeBasket implemented to do resetAllSnakes, grabSnake, releaseSnake, and parts of move()

The code for TextUI is not complex and you should be able to read it without any trouble. It is provided for you to illustrate how the classes you are implementing might be used to create a complete application. Although this user interface is clunky, it has the advantage that it is easy to read and understand how it is calling the methods of your code. It will be much more useful for debugging than the GUI.

The GUI

There is also a graphical UI in the `ui` package. The GUI is built on the Java Swing libraries. This code is complex and specialized, and is somewhat outside the scope of the course. You are not expected to be able to read and understand it. However, you might be interested in exploring how it works at some point. In particular it is sometimes helpful to look at how the UI is calling the methods of the classes you are writing.

The controls are simple: You can use the mouse to grab a snake head or tail, and move it.

The main method is in **`ui.GameMain`**. You can try running it, and you'll see the initial window, but until you start implementing the required classes you'll just get errors. All that the main class does is to initialize the components and start up the UI machinery. The class **`GamePanel`** contains most of the UI code and defines the "main" panel, and there is also a much simpler class **`ScorePanel`** that contains the display of the score.

You can edit **`GameMain`** to use a different descriptor. Alternatively, once you have **`SnakeUtil`** fully implemented, you can use the "Choose from file" button in the game to select one of the games from a file of descriptors.

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it.

Do not rely on the UI code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. The code for the UI itself is ten times more complex than the code you are implementing, and it is not guaranteed to be free of bugs**. *In particular, when we grade your work we are NOT going to run either of UIs, we are going to test that each method works according to its specification.***

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specifications, ask questions when things require clarification, and write your own unit tests.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the two required classes. Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** as well.

Importing the sample code

The sample code includes a complete skeleton of the two classes you are writing. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box.

1. Download the zip file. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

If you have an older version of Java or if for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Extract the zip file containing the sample code (*Note: on Windows, you have to right click and select "Extract All" – it will not work to just double-click it*)
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Eclipse Package Explorer, navigate to the src folder of the new project.
5. Drag the **`hw3`**, **`ui`**, and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.
6. Copy the remaining top-level files (TextUI.java, etc) into the **`src`** folder

Getting started

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification, so this getting started section will not be quite as detailed as for the previous assignments.

You can find the example test cases below in the default package of the sample code. *Don't try to copy/paste from the pdf.*

Please remember that these are just examples to get you started and you will need to write many, many more tests to be sure your code is working correctly. (You can expect that the functional tests we run when grading your work will have something like 100 test cases, for example.)

Don't rely on the GUI for testing your code. Doing so is difficult and frustrating because the GUI itself is so complex. Rely on simple, focused test cases that you can easily run in the debugger.

0. `SnakeBasket` is largely dependent on `Snake`, so it makes sense to start on `Snake` first. However, you won't need the `isValid` method until you are finishing up with `SnakeUtil.findSnakes`. The method `SnakeUtil.createDescriptorsFromFile` doesn't depend on anything else, so you can work on it anytime.

1. Familiarize yourself with the `SnakePiece` and `BasketCell` classes. The code is simple, and you'll need to use them a lot.

2. A `Snake` essentially has a list of `SnakePiece`s, and an id of type char. Be sure you can construct an empty Snake and get its list of pieces and id:

```
Snake s = new Snake('g');
System.out.println(s.getId()); // expected 'g'
System.out.println(s.getPieces()); // expected [] (empty ArrayList)
```

3. Then try adding some pieces and make sure they get added to the list:

```
s.addPiece(2, 3);
s.addPiece(2, 4);
s.addPiece(3, 4);
s.addPiece(4, 4);
System.out.println(s.getPieces());
// expected [(2, 3), (2, 4), (3, 4), (4, 4)]
```

3. There are some simple accessor methods you can implement now…

```
System.out.println(s.isGreen()); // expected true
System.out.println(s.getHead()); // (2, 3)
System.out.println(s.isHead(2, 3)); // expected true
System.out.println(s.getTail()); // (4, 4)
System.out.println(s.isTail(4, 4)); // expected true
```

4. Then think about `moveHead.` If the head is at (2, 3) and the direction is UP, then the new head should be at (1, 3):

```
s.moveHead(Direction.UP);
System.out.println(s.getPieces());
// expected [(1, 3), (2, 3), (2, 4), (3, 4)]
```

You might think you have to iterate over the whole list and create new pieces. But another way to think about this: all you really have to do is create a new piece for the head, stick it at the beginning, and delete the piece for the tail.

5. Similar for **`moveTail`**:

```
s.moveTail(Direction.RIGHT);
System.out.println(s.getPieces());
// expected [(2, 3), (2, 4), (3, 4), (3, 5)]
```

Hopefully you can identify common code in the two above methods, and factor it into a helper method! The reasoning will be similar for **`moveHeadAndGrow`** and **`moveHeadAndShrink`** (though you will not need these methods until you start using apples and mushrooms in your game grid).

6. There is a second **`addPiece`** method, that you will not actually need until you implement **`SnakeUtil.findSnakes`**, which has an additional argument representing the *index* for the new piece. You can implement it and test it now, or later:

```
s = new Snake('x');
s.addPiece(1, 2, 4); // put (2, 4) at index 1
s.addPiece(3, 4, 4); // put (4, 4) at index 3
System.out.println(s.getPieces());
// expected [null, (2, 4), null, (4, 4)]
```

7. For **`SnakeBasket`**, the first thing to do is implement **`resetAllSnakes`**, so that you can at least construct one in a correct initial state. Try something like this:

```
String test0[] = {
  "#  E  #",
  "#  .  #",
  "#  .  #",
  "#  .  #",
  "#  #  #"
};

ArrayList<Snake> snakes = new ArrayList<>();
Snake s = new Snake('g');
s.addPiece(2, 1);
s.addPiece(3, 1);
snakes.add(s);
SnakeBasket b = new SnakeBasket(test0, snakes);
```

```
    System.out.println(b.getCell(2, 1).hasSnake()); // expected true
    System.out.println(b.getCell(3, 1).hasSnake()); // expected true

    GridUtil.display(b);
    // expected:
//       0  1  2
//    0  #  E  #
//    1  #  .  #
//    2  #  g0 #
//    3  #  g1 #
//    4  #  #  #
```

7. If that much works, try a more interesting example:

```
    s = new Snake('g');
    s.addPiece(4, 3);
    s.addPiece(5, 3);
    s.addPiece(6, 3);
    s.addPiece(7, 3);
    snakes.add(s);
    s = new Snake('y');
    s.addPiece(1, 5);
    s.addPiece(1, 4);
    s.addPiece(1, 3);
    s.addPiece(1, 2);
    s.addPiece(1, 1);
    snakes.add(s);

    b = new SnakeBasket(test1, snakes);
    GridUtil.display(b);
    // expected:
//   0  1  2  3  4  5  6
//0  #  #  #  E  #  #  #
//1  #  y4 y3 y2 y1 y0 #
//2  #  .  #  #  #  .  #
//3  #  .  .  .  .  .  #
//4  #  #  #  g0 #  #  #
//5  #  #  #  g1 #  #  #
//6  #  #  #  g2 #  #  #
//7  #  #  #  g3 #  #  #
//8  #  #  #  #  #  #  #
```

8. Then work on the method to grab a snake by the head or tail and make it the "current" snake:

```
    b.grabSnake(1, 5);
    System.out.println(b.currentSnake() != null); // expected true
    System.out.println(b.currentSnake().getId()); // expected 'y'
    System.out.println(b.currentWasGrabbedByHead());  // expected true

    b.releaseSnake();
    System.out.println(b.currentSnake() != null); // expected false
    // if you grab someplace that isn't a head or tail,
    // it doesn't become the current snake
    b.grabSnake(1, 4);
    System.out.println(b.currentSnake() != null); // expected false
    System.out.println();
```

9. Next, start on `move()`. A good place to begin is just to move a snake only if the adjacent cell is empty (according to the `isEmpty()` method). Once you have that much working, you can think about the other cases (moving onto the snake's own head or tail, moving onto cells with food, moving to the exit).

```
    b.grabSnake(4, 3); // green snake head
    System.out.println(b.currentSnake().getPieces());
    // expected [(4, 3), (5, 3), (6, 3), (7, 3)]

    b.move(Direction.UP);
    System.out.println(b.currentSnake().getPieces());
    // expected [(3, 3), (4, 3), (5, 3), (6, 3)]
```

After a move, be sure you are calling `resetAllSnakes` to update the grid:

```
    GridUtil.display(b);
    // should look like this
//       0  1  2  3  4  5  6
//    0  #  #  #  E  #  #  #
//    1  #  y4 y3 y2 y1 y0 #
//    2  #  .  #  #  #  .  #
//    3  #  .  .  g0 .  .  #
//    4  #  #  #  g1 #  #  #
//    5  #  #  #  g2 #  #  #
//    6  #  #  #  g3 #  #  #
//    7  #  #  #  .  #  #  #
//    8  #  #  #  #  #  #  #
```

9. The method `SnakeUtil.findSnakes` is likely the trickiest part of the assignment. You need to iterate over rows and columns of the descriptor, and when you find a token that represents part of a snake, add a piece with that row and column to the correct snake. *But the piece has to be added at the correct index within the snake.* That is the reason for the second Snake addPiece method: `public void addPiece(int index, int row, int col)`.
Make sure you have that method working; see the bottom of SimpleSnakeTests.java for an example.

Once that is done the logic is more or less like this:

```
create an empty list of snakes
iterate over the rows and columns of the descriptor
    if a token at (row, col) is a string with length greater than 1
        the first character is an id
        the rest is an index
        if we have already created a snake with that id
            add a piece to that snake at the index
        otherwise
            construct a new snake with that id and add to the list
            add a piece to the snake at that index
```

Read the code for **`GridUtil.createGridFromDescriptor`** for an example of an easy way to iterate over the descriptor rows and columns. The only potentially confusing part is this line:

```
String[] tokens = firstRow.split("\\s+");
```

The String **`split()`** method returns an array of strings obtained by dividing up a string at a given delimiter. In the Java "regular expression" syntax, the delimiter **`"\\s+"`** means "any amount of whitespace". So, for example, if we have string **`s = "foo    bar baz"`** then **`s.split("\\s+")`** returns the array **`["foo", "bar", "baz"]`**. (This could also be done with a Scanner, of course.)

10. The last step in the **`findSnakes`** method is to verify that all snakes are valid (see the subsection "api.SnakeLayoutException" on page 7). At this point you'll need to go back and implement the **`isValid`** method of the **`Snake`** class if you haven't done that already.

11. As noted above the method **`SnakeUtil.createDescriptorsFromFile`** can be done independently of anything else. See SimpleFileTests.java for a sample test case. The sample code includes two text files in the project directory. The short file testfile.txt is referred to in the simple test case, and the file games.txt contains descriptors for a number of sample games. When you have **`findSnakes`** and **`createDescriptorsFromFile`** working, you'll be able to use the "Choose from file" button in the GUI to select a game from the ones in the file.

Style and documentation

Roughly 10% of the points will be for documentation and code style. The general guidelines are the same as in homework 2. However, since the skeleton code has the public methods fully documented, there is not quite as much to do. Remember the following:

- You must add an @author tag with your name to the javadoc at the top of each of the classes you write.

- You must javadoc each instance variable and helper method that you add. Anything you add must be `private`.
- Since the code includes some potentially tricky loops to write, **you ARE expected to add internal (//-style) comments, where appropriate**, to explain what you are doing inside the longer methods. A good rule of thumb is: if you had to think for a few minutes to figure out how to do something, you should probably include a comment explaining how it works. Internal comments always *precede* the code they describe and are indented to the same level.
- Keep your formatting clean and consistent.
- Avoid redundant instance variables
- Accessor methods should not modify instance variables

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder `hw3`. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag `hw3`. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any announcements from the that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw3.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw3`**, which in turn contains three files, **`Snake.java`**, **`SnakeBasket.java`**, and **`SnakeUtil.java`**. Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 3 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found as link #9 on our Canvas home page.

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw3**, which in turn should contain the three required files. You can accomplish this by zipping up the **src** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.