Com S 227
Spring 2024
Assignment 3
300 points

Due Date: Friday, April 5, 11:59 pm (midnight)
5% bonus for submitting 1 day early (by 11:59 pm April 4)
10% penalty for submitting late (by 11:59 pm April 7)
No submissions accepted after April 7, 11:59 pm

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, 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, try office hours, the schedule is on Canvas. Lots of help is also available through the Piazza discussions. Please start the assignment as soon as possible to get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

This is a "regular" assignment so we are going to read your code. Your score will be based partly on the specchecker's functional tests and partly on the grader's assessment of the quality of your code. See the "More about grading" section.

Contents

Introduction

The purpose of this assignment is to give you 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: **`LizardGame, Lizard, and GameFileUtil`**. As always, your primary responsibility is to implement these classes according to the specification and test them carefully. The three classes can be used, along with some other components, to create an implementation of a game we call Lizards, which is a mix between the classic snake game[1] and a sliding blocks puzzle game.

The game is played on a 2-dimensional grid of "cells". Each cell may contain a wall, an exit, or the body segment of a lizard. Cells are located by column and row.

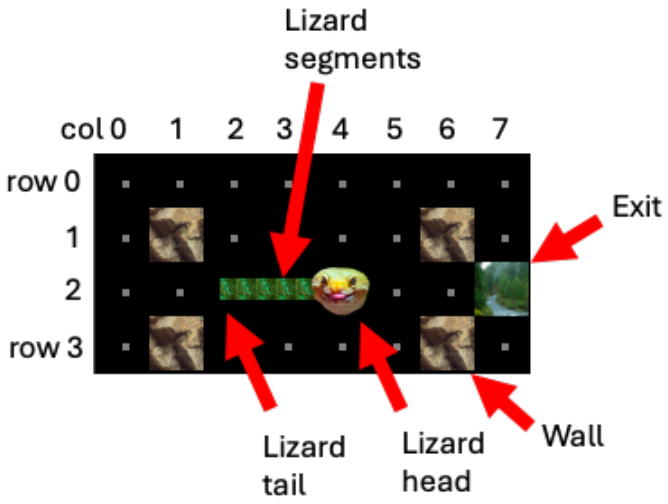

Figure 1

The user presses down the left mouse button and drags lizards to move them around with the goal of getting all the lizards to the exit. A lizard's body is multi-segmented and the user can press and drag any segment. The lizard moves in a snake-like fashion, that is, each segment must follow in the path of the segments in front or behind of them when the lizard moves forward or backward respectively. The only exception is when the user clicks and drags the head or tail

[1] https://en.wikipedia.org/wiki/Snake_(video_game_genre)

segments, which can move in any direction. There are only four possible directions of movement: up, down, right, left (never diagonally).

To illustrate the movement, Figure 2 shows four different scenarios: 1) the user selects the head segment and drags it in some forward direction 2) the user selects the tail segment and drags it in some backward direction 3) the user selects one of the inner segments and drags it in a forward direction 4) the user selects an inner segment and drags it in a backward direction. It is also possible, but not shown, that the user pushes the head segment in a backward direction or the tail segment in a forward direction.

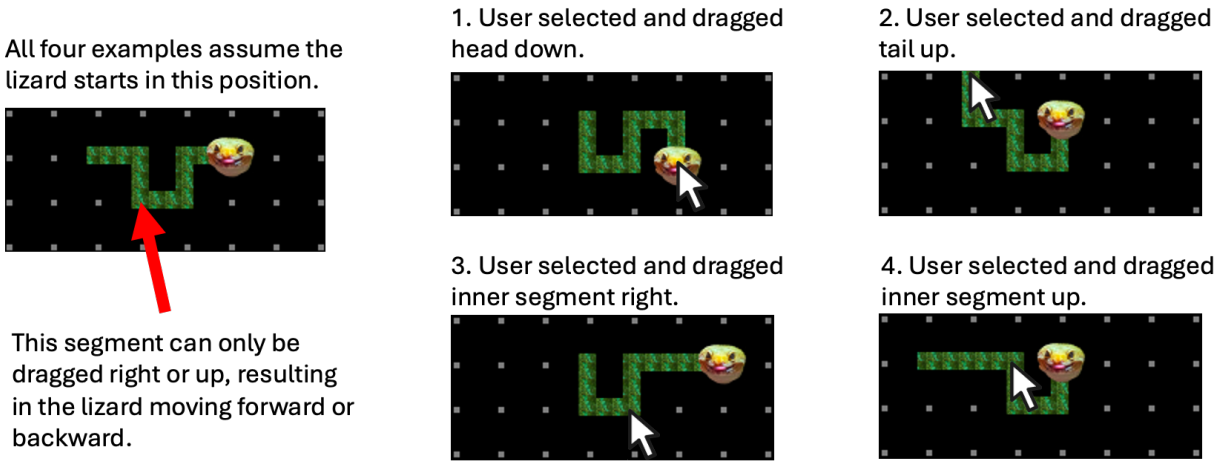

Figure 2

The lizard may not move off the grid and may not move onto a wall. When it moves onto an exit, the lizard is removed from the grid. When all of the lizards have been removed, the player wins.

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.

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. The **`api`** package contains some classes for representing data in the game. There are some simple tests for you to start with located in the default package.

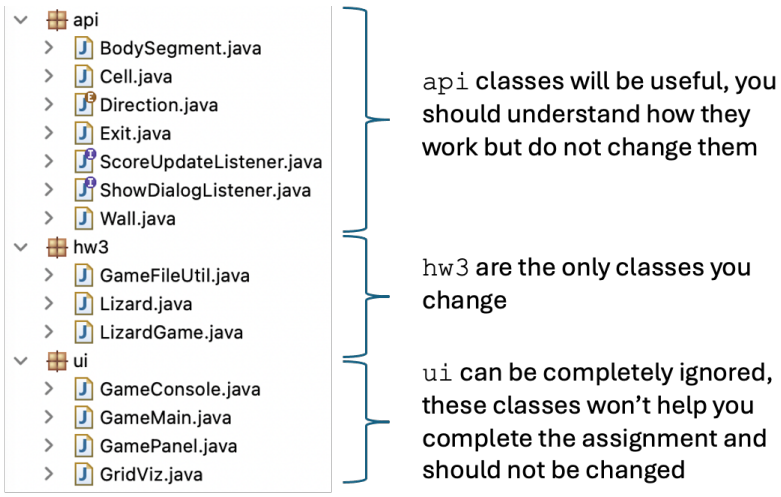

Figure 3

You may **not** modify the code in the **`ui`** or **`api`** packages.

Specification

The complete specification for this assignment includes

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

Overview of the GameFileUtil class

The **`GameFileUtil`** class is a simple utility class with a single static method to load a game file.

```
public static void load(String filePath, LizardGame game);
```

Game files are in plain text and have a particular format. There are three main sections to the file: the dimensions of the grid, a representation of the grid's cells, and a description of the lizards. The table below gives an example of these three segments.

8x8	Number of columns by number of rows, can be any integers larger than 0.
W WW . WWW W. . WWWWWW . E. WWWWW W. W W. W W .	A representation of the grid's cells, showing the location of walls "W" and exits "E". Each row ends with a dot.
L 5,1 6,1 6,2 5,2 4,2 3,2 2,2 L 1,2 0,2 0,3 0,4 1,4 2,4 3,4 4,4 5,4 6,4 6,5 6,6 5,6 4,6	Each line represents a lizard. The pairs of numbers represent the location of segments, starting from the tail (left most pair) of the lizard and ending with the head (right most).

Coordinates are specified as (column, row) pairs. As shown in Figure 1, the columns are numbered from left to right starting at zero and the rows are numbered from top to bottom starting at zero. We use column before row consistently throughout this document and the API because it matches with the convention of specifying coordinates as (x, y), however, it conflicts with the convention of arrays being rows of columns. How you implement internal data structures such as arrays does not need to match with the API, all that matters is that you are consistent in your usage of the data structures.

Finally, note that load does not return anything. It is passed a LizardGame object and it is expected that the method calls mutator methods on the object to update the state of the game.

Overview of the Lizard class

The **`Lizard`** class models a lizard as a collection of body segments (see the class **`api.BodySegment`**). The class contains several accessor and mutator method that can be used by other parts of the program, for example **`LizardGame`**. The segments are provided to the object with a call to **`setSegments(ArrayList<BodySegment> segments)`**, where the segments are expected to be ordered from tail to head. In other words, you can assume the first element of the list is always the tail.

Overview of the LizardGame class

The **`LizardGame`** class models the game and its methods are called by classes in the **`ui`** package to create a complete game. For example, the UI calls **`move()`** every time the user presses and drags the mouse over the graphical representation of the gird.

Also, **`LizardGame`** is responsible for keeping track of the grid of cells. Each cell is represented as a **`Cell`** object. The grid is specified by (column, row), as shown in Figure 1 and described in **`GameFileUtil`** above, however you can store the cells any way you want as long as the public methods produce the correct results.

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 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 classes **`GamePanel`** and **`GridViz`** contain most of the UI code.

Importing the sample code

The sample code includes a complete skeleton of the three 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 (SimpleTest.java, etc) into the `src` folder.

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 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 the UI, we are going to test that each method works according to its specification.***

We will provide a SpecChecker, it will contain many tests to help you get started, but **do not expect that it will test for every possible mistake you might make.** 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 tests. The SpecChecker we use for grading may have more tests added.

You can find the SpecChecker on Canvas. Import and run the SpecChecker just as you practiced in Lab 1. It will run a number of functional tests and then bring up a dialog offering to create a zip file to submit. Remember that error messages will appear in the *console* output. There are many test cases so there may be an overwhelming number of error messages. ***Always start reading the errors at the top and make incremental corrections in the code to fix them.*** When you are happy with your results, click "Yes" at the dialog to create the zip file. See the document "SpecChecker HOWTO", if you are not sure what to do.

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.

A good starting point is the **`Lizard`** class. A lizard is modeled as a collection of body segments (see the class **`api.BodySegment`**). A lizard can be constructed for test like this.

```
Lizard liz = new Lizard();
Cell cell0 = new Cell(1, 1);
Cell cell1 = new Cell(2, 1);
Cell cell2 = new Cell(2, 2);
ArrayList<BodySegment> segments = new ArrayList<BodySegment>();
segments.add(new BodySegment(liz, cell0));
segments.add(new BodySegment(liz, cell1));
segments.add(new BodySegment(liz, cell2));
```

There are several methods to implement in **`Lizard`**. Right now, these methods may not seem to be that important. But when you implement **`LizardGame`**, you may find many uses for these methods, so take note of what they do. Here are some simple tests for the methods in **`Lizard`**.

```
liz.setSegments(segments);
System.out.println("The lizard has " + liz.getSegments().size() + "
segments, expected 3.");

BodySegment headSegment = liz.getHeadSegment();
Cell headCell = headSegment.getCell();
System.out.println("The head segment is at " + headCell + ", expected
(2,2,Ground,Lizard).");

BodySegment tailSegment = liz.getTailSegment();
Cell tailCell = tailSegment.getCell();
System.out.println("The tail segment is at " + tailCell + ", expected
(1,1,Ground,Lizard).");

BodySegment middleSegment = liz.getSegmentAt(cell1);
Cell middleCell = middleSegment.getCell();
System.out.println("The middle segment is at " + middleCell + ",
expected (2,1,Ground,Lizard).");

BodySegment aheadSegment = liz.getSegmentAhead(middleSegment);
Cell aheadCell = aheadSegment.getCell();
System.out.println("The segment ahead of the middel is at " + aheadCell
+ ", expected (2,2,Ground,Lizard).");

BodySegment behindSegment = liz.getSegmentBehind(middleSegment);
Cell behindCell = behindSegment.getCell();
System.out.println("The segment behind the middel is at " + behindCell
+ ", expected (1,1,Ground,Lizard).");

Direction aheadDir = liz.getDirectionToSegmentAhead(middleSegment);
System.out.println("From the middle segment, ahead is " + aheadDir + ",
expected DOWN.");

Direction behindDir = liz.getDirectionToSegmentBehind(middleSegment);
```

```
      System.out.println("From the middle segment, behind is " + behindDir +
", expected LEFT.");

      Direction headDir = liz.getHeadDirection();
      System.out.println("The head is pointing " + headDir + ", expected
DOWN.");

      Direction tailDir = liz.getTailDirection();
      System.out.println("The tail is pointing " + tailDir + ", expected
LEFT.");
```

The class **GameFileUtil** is used to load saved game files. You are provided with a couple of sample files in the **examples** directory.

```
      // Example tests for GameFileUtil class
      // (requires some implementation of LizardGame)
      LizardGame game = new LizardGame(0, 0);
      GameConsole gc = new GameConsole();
      game.setListeners(gc, gc);

      System.out.println();
      GameFileUtil.load("examples/game1.txt", game);
      System.out.println("Expected a message saying the number of lizards is
now 1.");
      System.out.println("DO NOT print this message in GameFileUtil, the
ScoreListener needs to be called in LizardGame.");

      System.out.println();
      System.out.println("The grid with is " + game.getWidth() + ", expected
8.");
      System.out.println("The grid height is " + game.getHeight() + ",
expected 4.");
      System.out.println("The cell at (0,0) is empty (" + game.getCell(0,
0).isEmpty() + "), expected true.");
      System.out.println("The cell at (1,1) has a wall (" + (game.getCell(1,
1).getWall() != null) + "), expected true.");
      System.out.println("The cell at (7,2) has an exit (" + (game.getCell(7,
2).getExit() != null) + "), expected true.");
      System.out.println("The cell at (2,2) has a lizard (" +
(game.getCell(2, 2).getLizard() != null) + "), expected true.");
```

The **LizardGame** class models the game and its methods are typically called by the UI. Write simple tests to take place of the UI. The main interaction with the user happens when the user selects and drags the body segment of a lizard. These mouse events result in calls to the method **move()**.

Without a doubt, **move()** is the most complex of the methods you need to implement. Read the javadocs, review Figures 1 and 2 of this document, review the many methods available to help with the logic in **Lizard** and **LizardGame** and finally, break the problem into smaller parts and possibly create helper methods of you own. Many of the other methods in **LizardGame** are designed to help **move()** do its job, but you will also probably want to implement your own helper methods, which is allowed as long as they are **private**. Here are some simple tests for **LizardGame**.

```
      // Example tests for LizardGame
      // (assuming previous tests worked and the game is loaded)
      segments = new ArrayList<BodySegment>();
      segments.add(new BodySegment(liz, game.getCell(1, 0)));
      segments.add(new BodySegment(liz, game.getCell(2, 0)));
      segments.add(new BodySegment(liz, game.getCell(3, 0)));
      liz = new Lizard();
      liz.setSegments(segments);
      System.out.println();
      game.addLizard(liz);
      System.out.println("Expected a message saying the number of lizards is
now 2.");

      System.out.println();
      game.removeLizard(liz);
      System.out.println("Expected a message saying the number of lizards is
now 1.");

      System.out.println();
      liz = game.getLizards().get(0);
      Cell adjCell = game.getAdjacentCell(1, 1, RIGHT);
      System.out.println("Right of cell (1,1) is " + adjCell + ", expected
cell (2,1,Ground,Empty)");
      System.out.println("Cell (5,2) is available (" + game.getCell(5, 2) +
"), expected true.");
      System.out.println("Moving head of lizard one RIGHT.");
      game.move(4, 2, RIGHT);
      System.out.println("Cell (5,2) is available (" + game.isAvailable(5, 2)
+ "), expected false.");
      System.out.println("The head of the lizard is in cell (5,2) ("
            + (liz.getHeadSegment().getCell() == game.getCell(5, 2)) + "),
expected true.");
```

More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a two thirds) on the specchecker's functional tests and partly on the TA's assessment of the quality of your code. This means you can get partial credit even if you have errors, and it also means that even if you pass all the specchecker tests you can still lose points. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Are you using a conditional statement when you could just be using `%`? Are you using a loop for something that can be done with integer division?

Additionally, you should not take this to mean that your only job is it pass the specchecker's test by any means possible, such as hard-coding return values. The version of the specchecker we use for grading may be modified to use different inputs and test the methods in different ways than the speccecker you are given for testing. Bottom line, you job is to correctly implement algorithms that meet the specifications.

Some specific criteria that are important for this assignment are:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

See the "Style and documentation" section below for additional guidelines.

Restrictions on Java Features For this Assignment

There are no restrictions on the Java features you can use to implement this assignment. However, see above the above section about using good programming practices. You may not use any outside libraries (i.e., external libraries such as Apache Commons; otherwise, we won't be able to compile your code when grading.

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 (in this case the classes in the hw3 package).
- 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 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 four files, **`LizardGame.java`**, **`GameFileUtil.java`**, and **`Lizard.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.

> 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.