

Homework 10 and 11 - CS60 - Fall 2018

(Hw10) Part 1 & 2: Due Tuesday, December 4th at 11:59pm

(Hw11) Part 3: Due Thursday, December 13 at **1:00pm**

[Homework Overview](#)

[Part 1: Big-O Review \(due 2018-12-04 at 11:59pm\)](#)

[Part A: Unrolling Recurrence Relationships](#)

[Part B: Unrolling Recurrence Relationships](#)

[Part C: \$O\(\log^2 N\)\$](#)

[Part 2: Starting Spampede \(due 2018-12-04 at 11:59pm\)](#)

[Part A: Read this overview of Spampede](#)

[Part B: Download and import the starter files](#)

[Part C: Create a quick-reference sheet for yourself for each Java class \(10 points\)](#)

[Part D: Fill in the blanks for updateGraphic \(20 points\)](#)

[Part 3: Solving Spampede \(due 2018-12-13 at 1:00pm\)](#)

[Part E: Neighbor Code \(15 points\)](#)

[Part F: Move the snake within the board \(20 points\)](#)

[Part G: Key Press \(5 points, plus the fun of playing the game\)](#)

[Part H: Write code to reverse the snake \(20 points\)](#)

[Part I: Search for Spam: Implement AI Mode! \(20 points\)](#)

[Part J: Submit!](#)

[Extra Credit: I want more!](#)

Homework Overview

This project introduces and practices a number of different techniques that are common to *software engineering*, that is the design and implementation of large software projects. Certainly this assignment can only provide a taste of this important field.

The Spampede applet is a bigger and more complex beast than you have had to deal with in the past. Before you begin, we provide you with an overview of the software design behind the *Applet* you will create. Being asked to work within an established codebase is the norm in the “real world”, so this is good practice. The only difference is that this isn’t a large codebase compared to what you’ll see in industry.

The functionality of the application is broken down into multiple classes. There are three main classes that you’ll be using

- `SpampedeData.java`: Responsible for storing all of the data for the board that represents the Board.

- `SpampedeDisplay.java`: Responsible for drawing the board on the screen.
- `SpampedeBrain.java`: Responsible for the logic of the game, e.g., deciding how to move the snake, as well as handling keystrokes and controlling the timesteps that move the snake forward.

Note - Part H and I are only worth 35 points, but constitute probably 75% of the work for this assignment. We don't want to take off 75% of the points for people who aren't able to get reverse and BFS working, so only made them 35 points. Please let us know if you have any questions! We're happy to help!!! - Colleen & your awesome grutors!

Part 1: Big-O Review (due 2018-12-04 at 11:59pm)

- Learning Goal: Practice Big-O skills: unrolling recurrence relationships, reasoning about runtimes of trees, reasoning about runtimes of $\log_2 N$, reasoning about the tradeoffs of redundancy in a data structure as a way to improve runtimes.
- Prerequisites: Big-O skills :-)
- Submission: For all of the following parts, you can do them on a computer or paper, but they should each be at most 1 page/1 image. (If this isn't enough - please post to Piazza to let us know!)
- Points: 18

When you unroll a recurrence relationship, you must:

- Show your cheat-sheet (i.e. the rewritten forms of the recurrence relationship that you'll substitute in to your recurrence relationship).
- "Unroll" the recurrence relation at least 3 times, using generic numbers (e.g., N , $N-1$, etc.)
- Demonstrate a pattern that appears in the unrolled version of the recurrence relation.
- Give the generalized result of the pattern, as a function of N
- State the Big-O runtime

Normally, you'd define any variables you use, but all of these will just use N and won't be connected to code. [You might find the example loop counting/recurrence unrolling examples helpful!](#)

Part A: Unrolling Recurrence Relationships

Unroll both of the following recurrence relationships to show that their Big-O runtimes are the same.

$T(1) = 1$ $T(N) = 1 + T(N-1)$ for all $N > 1$	$T(100) = 1$ $T(N) = 1 + T(N-1)$ for all $N > 100$
---	---

Part B: Unrolling Recurrence Relationships

Unroll both of the following recurrence relationships to show that their Big-O runtimes are the same.

$T(1) = 1$ $T(N) = N + T(N-1)$ for all $N > 1$	$T(1) = 1$ $T(N) = N/2 + T(N-1)$ for all $N > 1$
---	---

Part C: $O(\log_2 N)$

When we unroll the following recurrence relationship, we get a runtime of $O(\log_2 N)$. Write a paragraph description (or a few sentences and draw a picture) to explain why this would be. Assume that you are explaining it to someone who knows what trees are, but hasn't been introduced to the patterns of balanced binary search trees.

$$T(1) = 1$$

$$T(N) = 1 + T(N/2) \text{ for all } N > 1$$

Part 2: Starting Spampede (due 2018-12-04 at 11:59pm)









- Learning Goal: Reading code and being able to figure out how to call it. Working with 2D arrays. Debugging practice and working with multiple classes.
- Prerequisites: Java fluency
- Submission: reference.jpg, Spampede.jpg
- Points: 30

Part A: Read this overview of Spampede

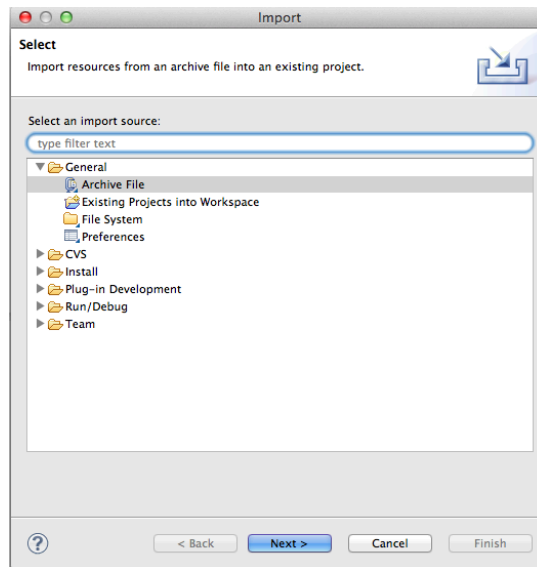
This assignment gives a user control over a spam-seeking snake. Key presses from the keyboard change the direction of the snake's movement in order to intersect snacks (spam) that appear at random places on the screen. For each snack consumed, the snake grows by one segment (a segment is simply one `BoardCell`). Variations are welcome (see the extra credit section below)!

Part B: Download and import the starter files

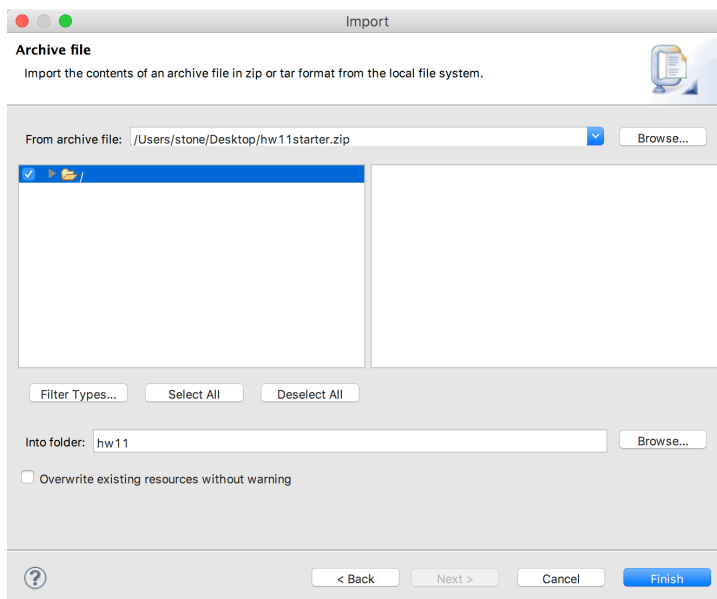
- Download the starter files in [spampede.zip](#) but do not expand this archive.
- Create a new Java project via *File - New Java Project*. Give your new project the name **hw110**
- Create a new package (*File - New Package*). Give your package the name:
com.gradescope.spampede
- Add the JUnit 4 library to the Build Path, as usual.
- Right click on the project and click "import"

New		▶
Go Into		
Open in New Window		
Open Type Hierarchy	F4	
Show In	⌘ ⌘W	▶
 Copy	⌘C	
 Copy Qualified Name		
 Paste	⌘V	
 Delete		
Build Path		▶
Source	⌘ ⌘S	▶
Refactor	⌘ ⌘T	▶
 Import...		
 Export...		
 Refresh	F5	
Close Project		

- Select “General” -> “Archive File” and click “Next”



- Click “Browse” to select the downloaded spampede.zip file, and then hit “Finish.”



- If there is a red "error" icon by the JUnit test files, that is because the JUnit 4 library is not on the build path. Go back and do that.
- If the J's next to the filename are in "outline" (rather than a solid color) like this

```

Picture.java
PictureECTest.java
PictureExplorer.java
PictureFrame.java

```

Then the files did not end up in the `com.gradescope.spampede` package inside that folder). Select all of the imported files with the mouse and manually drag them into `com.gradescope.spampede`.

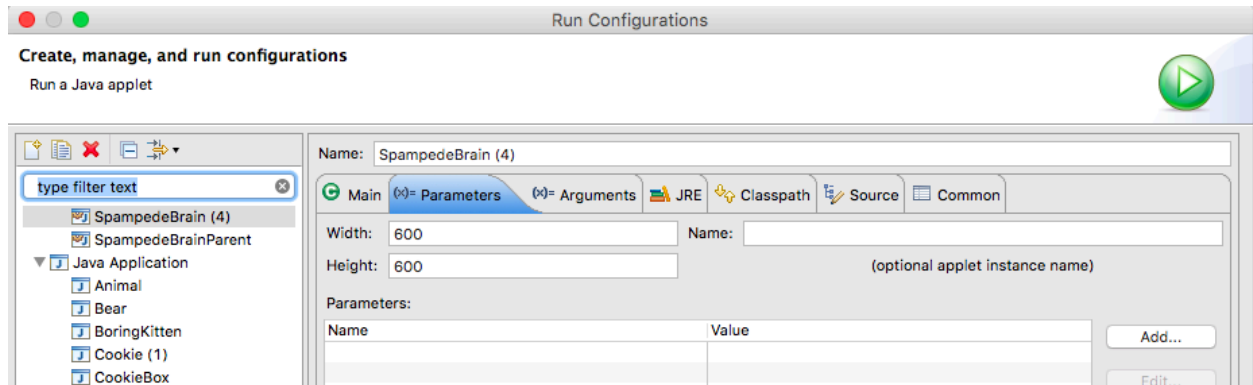
- Run `SpampedeBrain.java`
 - You should see a very small window open up.



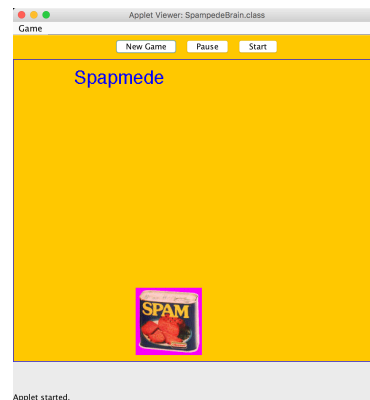
Let's fix that...

- Quit the program and go back to Eclipse. Click on the arrow next to the run button and select "Run Configurations".

On the "Parameters" tab, reset the size to be 600 x 600 (i.e. Width = 600 and Height = 600 as shown below). **Don't forget to hit Apply.**



- **Re-run `SpampedeBrain.java`.** It should now look like this (not to scale):



- If the image of spam doesn't appear, it might be because the spam image is in the incorrect package. Drag and drop the image file (`spam.gif`) outside of the `com.gradescope.spampede` package, into the `src` directory.
- Note - you'll likely see the following error in Eclipse when you play audio, but this isn't a problem (or a warning that we can easily suppress):
 - **WARNING: 140:** This application, or a library it uses, is using the deprecated Carbon Component Manager for hosting Audio Units. Support for this will be removed in a future release. Also, this makes the host incompatible with version 3 audio units. Please transition to the API's in `AudioComponent.h`.

Part C: Create a quick-reference sheet for yourself for each Java class (10 points)

(This part relies on parts A & B).

Look through `CellType`, `BoardCell`, `SpampedeData`, `SnakeMode`, `SpampedeDisplay`, and `SpampedeBrain`.

On a 8.5"x11" sheet of paper, write down some notes for yourself about

what the two enumerations are for;

what data the objects of each class contain (e.g., the instance variables), and

what data the objects of each class know how to do (i.e., the methods).

As you work on the code, if you find you need to look something up in the files more than once, make a note of it on your quick-reference sheet.

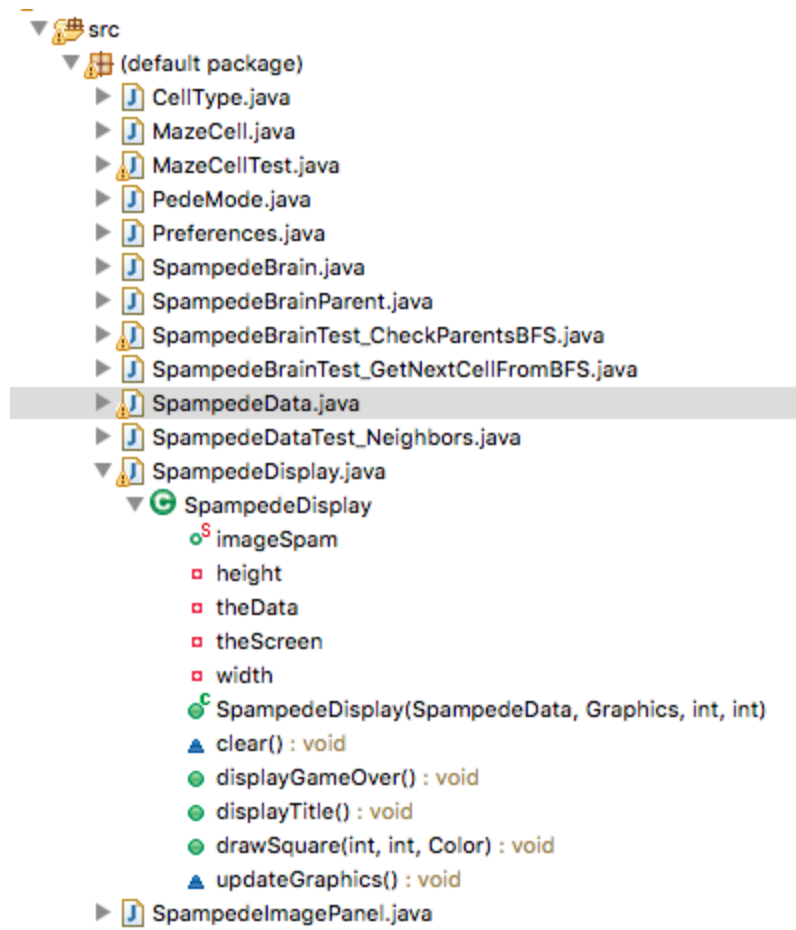
Your quick-reference sheet should be organized clearly enough that other people would be able to use it to remind themselves what a class does or in which class some functionality exists. Feel free to use pen, pencil, or a computer to make this, but we encourage you to have it on paper next to you while you work!

At the end of the assignment (not now), you should take a picture of this quick-reference sheet, name the picture reference.jpg, and submit it.

If you wait until the end of the assignment to make the reference sheet, it will just be a waste of your time. However, it will likely be very helpful to you if you make it BEFORE starting the rest of the assignment and keep updating it as needed. The more you have the “big picture” of the assignment before you get started, the easier it will be for you to write individual methods.

As you prepare your reference sheet, try to use these strategies for navigating through your Java project (to avoid things such as scrolling through many files when you want to find a particular method):

- Right-click on a method name and select “Open Declaration” to jump to where that method is defined (it might have you jump into a different file!)
- Right-click on a method name and select “Open Call Hierarchy” to see all of the methods that call the method.
- Use Ctrl+F and enter a method name in the search box to find specific methods within a file.
- Within the Package Explorer (left-hand column), you can click on the triangle icon next to the Java class name to view the public methods (green circles), methods that override a parent method (blue triangles), private instance variables (red squares) and public static variables (green circle outlines with a red S), and constructors (green circles with a green C). You can click on these to jump to the definition of that method in the file.



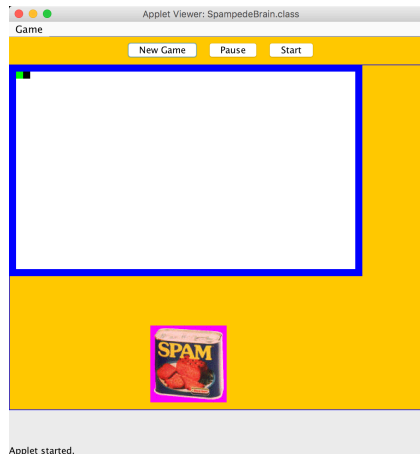
- Take a photo of your reference sheet, name it `reference.jpg`, and submit it.

Part D: Fill in the blanks for `updateGraphic` (20 points)

(This part relies on parts A, B, & C).

- **Fill in the blanks in `updateGraphics` (within `SpampedeDisplay.java`) to draw the board based on the contents of the model (`this.theData`).** This does not require very much code (10 lines or less) because a lot of useful functionality is already present in other classes (hint look at `getCellColor` in `SpampedeData.java`).

After you complete this step, you should see a board that looks like this:



- Once that works, **modify your code to shift the board 90 pixels down and to the right**. You are encouraged to do this by modifying the code in the method `drawSquare`. This will allow you to see the title and it should look something like this:



(Don't forget to quit the game (close the window) every time you're done testing. You get a brand new copy of the application running every time you hit "Run" in Eclipse, and you don't need to run 20 versions of Spampede simultaneously!)

- 90 pixels wasn't quite right; the board is too far to the right. We could play around with other numbers, but instead **use this .width and constants in the Preferences class (e.g., Preferences.CONSTANT_NAME) to compute a good offset that leaves the board horizontally centered**.



- **Fix the title of the game (in `Preferences.java`).**
- Once you have completed this part, please submit a screenshot of this as **Spampede.jpg**
- **Note about submitting image files:**
 - Sometimes your computer will hide the file extension (e.g. .jpg). This can make it harder to submit a file with the right name!
 - **View file extensions** [Windows 10](#) or [Mac](#)
 - **Convert to jpg** [there are some tools and other instructions on the internet.](#)
- If the spam picture doesn't show up! Here's a fix!
 - What solved it was moving `spam.gif` outside of the `com.gradescope.spampede` package, into the `src` directory- I think that's where `SpampedeBrainParent` expects the image to be.
 - Specifically this line
 - `URL url = this.getCodeBase();`
 - probably returns the `src` directory instead of the package directory

Part 3: Solving Spampede (due 2018-12-13 at 1:00pm)

- Learning Goal: Reading code and being able to figure out how to call it. Working with 2D arrays. Debugging practice and working with multiple classes.
- Prerequisites: Java fluency
- Submission: `hw10.zip`
- Points: 100

Part E: Neighbor Code (15 points)

(This part relies on parts A, B, & C above).

- **Complete the `get...Neighbor` methods in `SpampedeData.java` so that the test cases in `SpampedeDataTest_Neighbors.java` pass.**
 - Note: You should never make new `BoardCells`! Just return a reference to `BoardCells` that are within the `SpampedeData` object already.

- **Complete the method `getNextCellInDir()` in `SpampedeData.java` so that the test cases in `SpampedeDataTest_CellInDir.java` pass.** Your code will use `this.currentMode` and previously-completed methods.
- You might find it helpful to look at pictures of the test boards! (<http://tinyurl.com/spampedeTestBoards>).

Part F: Move the snake within the board (20 points)

(This part relies on parts A, B, C, & E).

- We provided code in `updateSnake()` within `SpampedeBrain.java`. Complete the comment above `updateSnake()` to specify which method calls `updateSnake()`. (Hint: use “Open call hierarchy” to see when `updateSnake` gets called!)
- `updateSnake()` calls the private method `advanceTheSnake`, which moves the pede to the cell (`nextCell`) provided as an argument. Fill in the rest of the method `advanceTheSnake` so that the snake moves, keeping in mind the following requirements and implementation details:
 - Requirements:
 - **You can’t modify the `snakeCells` in `SpampedeData.java` to be public. They have to stay private.**
 - You must therefore write a helper method or two in `SpampedeData.java` to modify the data for your game (e.g. `snakeCells`) as desired. `snakeCells` is of type `LinkedList<BoardCell>`, which is the Java-library version of a *double-ended queue* implemented via a linked list. You will thus have access to the methods listed and described at: <http://docs.oracle.com/javase/6/docs/api/index.html?java/util/LinkedList.html>.
 - You should not create any new `BoardCells` or modify the row or column of any existing `BoardCells`. Instead, you should use the `BoardCell` methods `becomeHead()`, `becomeBody()`, and `becomeOpen()`.
 - Implementation details:
 - If `nextCell` is part of the body or a wall, the game is over (this code is already written).
 - If `nextCell` is not spam, the snake should move (i.e., `nextCell` will be the new head and the last piece of the tail will no longer be part of the snake.)
 - If `nextCell` is spam, the head should just be moved forward (and the tail stays the same spot).
- When you complete this part, the test cases in `SpampedeBrainTest_AdvanceSnake.java` should pass.

Part G: Key Press (5 points, plus the fun of playing the game)

(This part relies on parts A, B, C, E, & F).

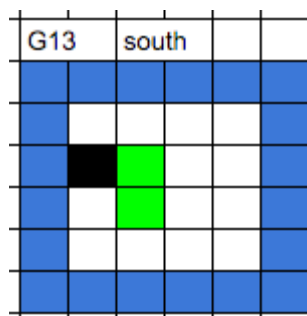
- Fill in the blanks within `keyPressed` in `SpampedeBrain.java` so that when you press the relevant key, the direction is set using the methods in `SpampedeData.java` (`setDirectionNorth()`, `setDirectionSouth()`, `setDirectionEast()`, `setDirectionWest()`). Make sure you use the private static final variables like the examples that use `AI_MODE` and `REVERSE`.
- While you can write this code using “if”, you are probably better off learning to use switch statements:
 - <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html>
 - Note - you’ll need to put the keyword `break` into each of the cases!
- After completing this step you should be able to run `SpampedeBrain.java` and play the Spampede game!

Of course, nothing will happen yet when you press the ‘a’ or ‘r’ keys (i.e., there won’t be an AI mode or reverse functionality because you haven’t written the code for that functionality yet).

Part H: Write code to reverse the snake (20 points)

(This part relies on parts A, B, C, & E).

- Write the method `reverseSnake()` in `SpampedeBrain.java`. Note, you’ll need to add methods to `SpampedeData.java` to make the necessary changes to the `snakeCells`.
- After this you should pass the test cases in the file `SpampedeBrainTest_Reverse.java` and be able to press the ‘r’ key during the spampede game and have the snake reverse. Note - these (and all) test cases are not guaranteed to be comprehensive. It is possible that your code will have errors that are caught by these test cases and we encourage you to test the functionality within the game to try to help catch additional errors.
- You can look at the boards used in the test cases to see how we calculate the new direction of the snake (<http://tinyurl.com/spampedeTestBoards>). Hint - it doesn’t involve the old direction of the snake - only the position of the new head and new neck. For example when the snake below is reversed, it will be going South.



More available at <http://tinyurl.com/spampedeTestBoards>

Part I: Search for Spam: Implement AI Mode! (20 points)

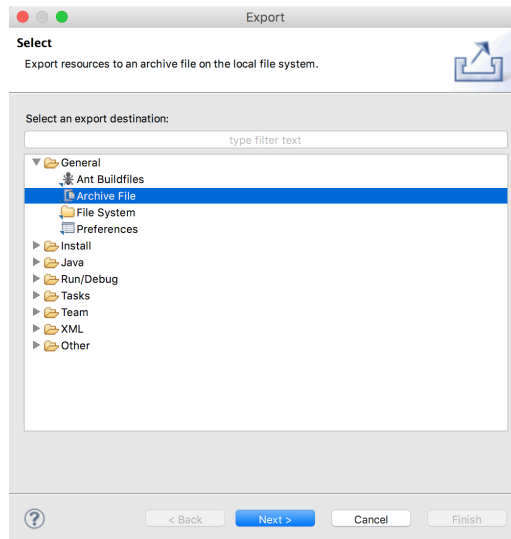
(This part relies on parts A, B, C, & E and Lecture 22 on Tuesday November 27th.)

- Write the method `getNextCellFromBFS()`, which returns the `BoardCell` on the path to the nearest spam.

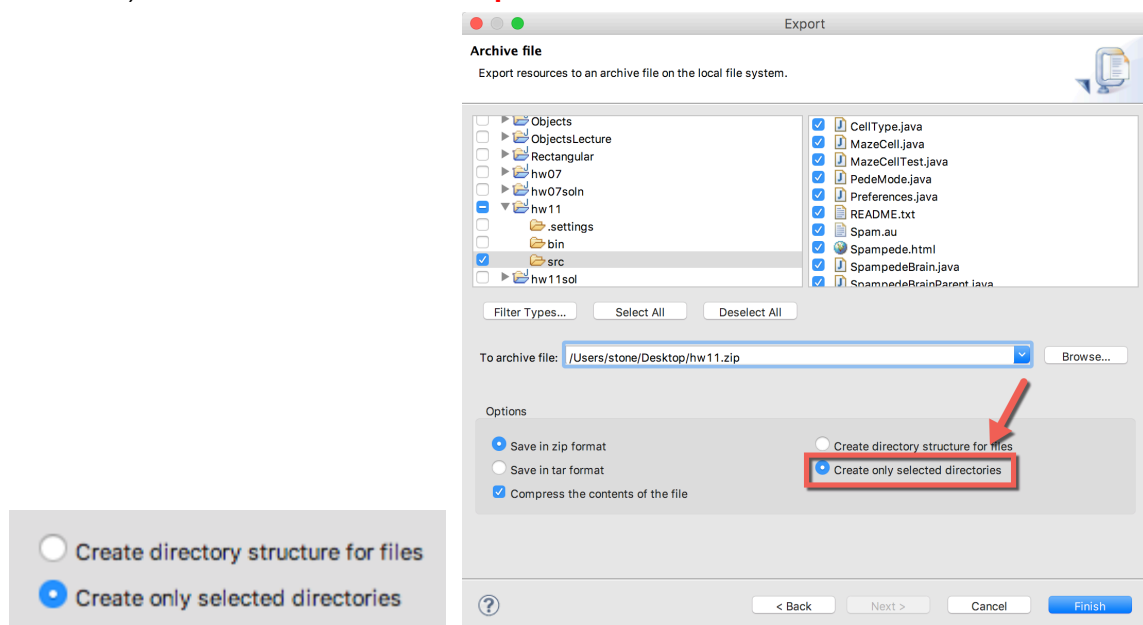
- In writing BFS you'll need to decide the order in which you add the neighbors of each cell to the Queue. You should add the neighbors:
 - North neighbor
 - South neighbor
 - East Neighbor
 - West Neighbor
- If there are two spams that are equidistant, you should return the path to the one you found first (i.e., based upon adding the neighbors to the queue in the above order).
- Read the test cases. After this you should pass the test cases in the files:
 - `SpampedeBrainTest_CheckParentsBFS.java`
 - This test file uses a method of “clear-box testing”, where we look inside the structures (here we look at the parent references of BoardCells) to make sure that the algorithm is behaving as intended.
 - These tests assume the BFS algorithm as described in class, which does not stop until a target (spam) is *dequeued*. If you prefer to implement a “more efficient” version that stops immediately when spam is first discovered (rather than waiting for that spam to make its way to the front of the queue), you will need to run `SpampedeBrainTest_CheckParentsBFS_fast.java`. *Either version is perfectly acceptable.*
 - `SpampedeBrainTest_GetNextCellFromBFS.java`
 - This test file uses a method of black-box testing, where we just test the functionality of the methods without needing to know the algorithm or data structures used.

Part J: Submit!

- Before you package up your file, make sure to **delete** (not just comment out) your debugging print statements. One of the rubric items we'll use is checking that you've done this.
- The code should be submitted as a complete **hw10.zip** file containing (only) your src folder. To do so, right-click **on the src folder** and choose Export... Select “Archive File” under general:



Select “Create only selected directories,” (making sure only src is checked in the left-hand pane as shown below) and name the file **hw10.zip**



Submit the **hw10.zip** file.

- **Gradescope seems to be unzipping the files after students upload them. I think this might be a new feature, but it makes it seem like we won't have access to the correct files. You can trust that we do! Sorry for the inconvenience!**
- If you did extra credit work (see below), you must include a text file called README.txt. In this document you should describe the extra features that you implemented.

Extra Credit: I want more!

If you'd like to extend your application, there are a couple of specific items and an open-ended invitation to improve on the applet for **optional** bonus credit. (Up to 20 points extra credit in total.)

IMPORTANT: If you add optional features, please explain them carefully in the README file. Otherwise, you won't be eligible to get credit for them.

- **Enemy Snakes!:** Allow there to be one or more "enemy" snakes that use your breadth-first-search code and/or other heuristics to play against your snake. (This is worth extra bonus points since it is a bit more challenging.)
- **Speed up:** You might want to have the rate at which the snake is moving to increase as the game progresses.
- **Scoring:** You might want to have a system of scoring with a running total displayed as a label or text field or simply drawn to the applet panel.
- **Lives:** Rather than resetting or stopping the game after a single Spampede crash, keep a text field (or label) with the number of lives remaining and decrement it after each crash. When there are no lives left, stop the game (though you might want to consider a "reset" button.)
- **Levels:** Rather than maintaining a single, static board, you may want to have the snake advance to different boards after consuming enough spam.
- **Wrapping:** Allow the snake to wrap around the game board -- either in an unlimited fashion or through small tunnels in the walls. Or you might consider a "hyperspace" square, that "sends" cells to another spot on the board.
- **General improvements:** Feel free to add additional features you think would enhance the Spampede game: different kinds of spam, sounds, images, other graphical widgets like pull-down menus or text boxes, etc.

Extra credit is awarded according to the following rubric:

- 5 points - "eh"
- 10 points - "woah"
- 15 points - "Wow! check this out!!"
- 20 points - "You've got to be KIDDING!!!! WOW!"