1 Introduction

In this Project, you’ll be implementing a game known as “Robots”. I played this a long time ago on Linux... but it’s quite hard to find now. The best video I found was this one:  
https://www.youtube.com/watch?v=K7L55s9sf4k
I’ll also be posting a video to Panopto giving my own introduction to this project, which will be much more interesting. But if you want to see the original, it’s there for you to see.

The point of this project is twofold. First, it’s another chance to play with arrays; we’ll be allocating them, duplicating them, moving data around - and
as always, iterating over them. Second, it’s a chance to play with objects in an asynchronous model.

Asynchronous programming basically means (this is a massive oversimplification, but run with me) “wait for things to happen” instead of “go do stuff.” What does that mean for you? In this project, I will provide the complete main() method - you’re not allowed to change it! This main() method will create an object of type Proj09GameState, and then will call some methods in that object. You must implement Proj09GameState, and all of the methods that main() will call.

Now, in the real world, asynchronous programming models are much more complex. You have things like GUIs and network connections and other interesting events. This is a very simplified project. But it’s a good place to start.

2 What is “Robots?”

In Robots, there is a single player, who moves around on a grid. Also on the grid are many different robots, which are trying to kill the player. Each time that the player moves one square, every robot moves one square as well. The robots always move towards the player, and if they reach the player, he dies.

The player has no weapon. The only way to defeat the robots is to force them to crash into each other. Each time that they do so, they are destroyed, and leave a bit of rubble behind. When other robots crash into the rubble, they are destroyed as well. (Unfortunately, the rubble piles never grow larger. They always fill up only a single square.)

The object of the game, of course, is to survive as long as you can. For Project 9, you won’t survive very long, since we won’t implement teleports. But in Project 10, we’ll implement teleports (which allow you to escape when you’re trapped), multiple levels, and few other features.
Here’s a picture of the game right **before** two robots crash into each other:

Here’s the same game, after they crashed (the player chose not to move):
2.1 How the Robots Move

The robots move one square at a time; they can move left, right, up, down, or diagonal. The logic for their movement is simple: if they are above the player, they move down, and vice-versa for up; if they are to the left of the player, they move right, and vice-versa for left.

Because of this, the robots typically move in a direct diagonal lines, until they are directly above, below, to the left, or to the right of the player. Then they move straight toward the player. This means that if the player stands still, the robots tend to line up in a giant “plus” around them.

2.2 How the Player Moves

The player has the same 8 basic moves as the robots, but also can choose not to move for one turn. This is often useful when hiding behind a pile of rubble: if you find the right location, you can simply wait while the robots all crash into the rubble.

(See the program requirements below to see what types of moves are required for partial credit, and for full credit.)

3 main()

You will not write the main() method; instead, download the class Proj09Robots from http://www.cs.arizona.edu/classes/cs127a/spring16/projects/proj09_code. Take a look at my main() method. It explains how the program works, and points out the key methods that you need to implement.

You must not change this file. If you turn in an alternate version of the file, we’ll ignore it. This means that you must design the class Proj09GameState so that it works well with this class.

4 class Proj09GameState

You must implement this class. It has 6 mandatory methods - although you can add additional “helper” methods if you’d like. You will definitely need to include some instance data fields - the game state must include something to store the location of all of the robots (and rubble), something to store the location of the player, and (if you’d like) some additional variables.

You may use any variables that you want inside this class; I don’t care what they are, their types, or how they are named. However, I’ll offer suggestions. In my class, I had the following variables:

- A 2D array, which stored the locations of all robots and rubble.
  One value represented a robot; another represented rubble; and a third represented an empty square.
  I did not include the player in this grid.
• The \((x, y)\) coordinates of the player
• A boolean - can you guess what this might be for???

NOTE: You may make the grid any size you want - provided that it’s no smaller than 5x5.

4.1 static vs. non

You must not declare any static variables or methods inside Proj09GameState. Every one of your variables must be an instance variable; every method must be an instance method.

4.2 public vs. private

Every variable and every method must be marked either public or private; if you don’t include either marker, you’ll lose some points.

I haven’t taught the class about the difference between public and private yet - so it’s perfectly OK to mark everything public. But if you know about private already - or if you just want to experiment - it’s OK to use private for some things.

4.3 init()

The \texttt{init()} method initializes the object. You should create storage for the board, place the player in a good starting position, and initialize any other variables you may have added. Do not place any robots on the board yet.

I have not taught the class about constructors yet, but if you know about them, it’s OK to do all of this work in the constructor, and to make \texttt{init()} a NOP.

This method should not do any drawing.

4.4 addRobots(int)

The \texttt{addRobots(int)} method adds robots (the parameter tells you how many) to the board. Make sure that you don’t add two robots in the same location; also don’t add them at the same location as the player. (It’s OK to have robots right next to the player, though.)

Make sure that your method adds as many robots as the parameter calls for; although you know that \texttt{main()} will only ask for 10 robots, use the parameter to control your loop. That way, if we change \texttt{main()}, your method will still do the right thing.

This method should not do any drawing.

4.5 boolean isGameOver()

This method returns true if the game is over, false if not. Does this give you any ideas for how you might use a boolean instance variable?
4.6 boolean handleKeyTyped(char)

This method takes a single character which has just been typed on the keyboard and checks to see if it is a valid move. If the key is recognized (and if the move was possible), then update the player position and then return true. If the key is not recognized, or if the move is impossible (because the player is against the edge of the board), then return false.

You must support at least the following keys:

- w - Move up
- a - Move left
- s - Don’t move, but let the robots move one step closer
- d - Move right
- x - Move down

Additional keys will also need to be supported if you want to earn all of the points for this assignment.

The player may never move out of the board. So if the player is on the left edge of the board and tries to move left, the move is ignored. Return false (which means that the robots won’t move, either).

I encourage you to add (limited) debug messages to the System.out. However, they are not required.

This method should not do any drawing.

4.7 moveRobots()

This method will move all of the robots toward the player. If two robots crash, they will be replaced with rubble; if a robot crashes into rubble, it will also be removed (but the rubble will stay).

This method should update all of the robots, not just some of them (even if one of them catches the player).

This method should not do any drawing.

4.7.1 Implementation Hints

You may implement this method however you want, however I wanted to give you a hint. In my code, each time that this method is called, I allocate a new array, and write my updates into that. When I’m done, I simply save the new array as the game state, and forget the old one. This method is a lot easier than trying to move the robots around in the same array. Of course, it’s also pretty inefficient - but that’s OK for this assignment.

Here’s way-too-high-level pseudocode for my implementation of this method (feel free to use an expanded version of this for your own pseudocode):
allocate new board

for each element in the (old) 2D array
  if it’s rubble
    copy it to the new

  if it’s a robot
    figure out where it would move
    figure out if it crashes into rubble or another robot
    update the new array
    check to see if it has caught the player

save new array back as the new game state

4.8 draw()

This method should draw the current state of the game to the screen. Draw some sort of grid; draw the robots; draw any rubble that exists; draw the player.

The pictures that you use for the player, robots, and rubble can be quite trivial (as you have seen from my example solution). If you want to make your program look like my example, that’s OK. But you are encouraged to use your creativity, and make the pictures better.

You should use `StdDraw.show(int)` to show the board to the screen. It’s OK if the first time you draw the screen, it appears slowly; however, it must redraw quickly after that.

5 Pseudocode Requirements

You must write pseudocode for the following methods in the `Proj09GameState` class:

- `addRobots(int)`
- `moveRobots()`

Spend some time on your pseudocode. Remember, the point of pseudocode is not to come up with a perfect algorithm - so it’s OK if it doesn’t match your Java code. What matters is that you demonstrate that you took some time to think it all through. Are you considering the inputs and the outputs? Do you have a rough idea of the loops that will be required? Etc.

You will be graded on the quality of your pseudocode. Again, we’re not looking for perfect algorithms - but evidence of serious thought about them.
6 Program Requirements

To receive most (but not all) of the points for this assignment, your program must:

- Create a class named Proj09GameState.
- Implement all of the required methods. Methods should be defined such that the provided main() method will compile and run; they should do what the spec says above.
- Implement logic which will cause the game to end when the player is in the same space as a robot or a pile of rubble.
- Support the keys w,a,s,d,x as detailed in the description of the handleKeyTyped() method.

To receive full credit, your program must additionally do the following:

- Implement diagonal moves, using the keys q,e,z,c.
- Support capital versions of all keys, as well as the numeric keypad (1-9). (That is, '7' should do the same thing as 'q', and so on.)
- Improve the picture of the robots, rubble, and player. Each picture must include at least three drawing elements (squares, rectangles, lines, etc.).

7 Pseudocode

Download the latest pseudocode requirements from:
http://www.cs.arizona.edu/classes/cs127a/spring16/projects/
Project 9 will be graded based on v1.3 of the requirements.

8 Java code

After you complete the pseudocode (and turn it in!), start working on translating it to Java. (For a list of steps for converting the pseudocode to Java, see the Project 1 spec.)

Your class must have exactly the name given above: Proj09GameState. Make sure that your Java file is named to match.

8.1 Header Comment

Every Java file that you submit must have a Header Comment - which is a simple comment, at the top of your file, which gives basic information about it. A header comment must include:

- The name of the Java class
Late Day Clarifications

Pseudocode

The syllabus says that you can use Late Days for late projects - however, the syllabus does not explicitly say how this works with pseudocode. The SLs and I talked it over, and we decided on the following policy:

- Late days cannot be spent on pseudocode. Turn it in on time, or you'll just miss that part of the project grade.
- You must turn in your pseudocode using turnin; email will no longer be accepted.

(We were flexible about both of these points in the first two projects - but starting with Project 3, we're going to enforce this.)

Java code

Starting with Project 3, if you email us your code (instead of using turnin), it will count as a Late Day, even if you do it before the deadline. Please use turnin. And remember: if you are having trouble connecting from home, it's always possible to come to the 228 lab, and upload your file using a thumbdrive.
9 Turning in Your Programs

See the class website for information about how to upload your files to lectura, and how to use the ‘turnin’ utility.

9.1 Pseudocode

Your pseudocode file must be either TXT or PDF files, but you may name them whatever you want. You must turn them in using the assignment name

\[ \text{cs127a_s16_sX_proj09_pseudocode} \]

(replace the \( X \) with your section letter). **REMEMBER:** The pseudocode is due two days before the Java code!

9.2 Java Code

You must turn exactly one file, named exactly this:

\[ \text{Proj09GameState.java} \]

You must turn in the Java file(s) using the assignment name

\[ \text{cs127a_s16_sX_proj09} \]

(replace the \( X \) with your section letter).