Com S 227
Fall 2022
Assignment 4 v3
275 points

Due Date: Friday, December 9, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Dec 8)

NO LATE SUBMISSIONS! All work must be in Friday night.

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.

Table of Contents

Introduction

The purpose of this assignment is to give you some experience working with inheritance and abstract classes in a realistic setting. In this project you will complete the implementation of a small hierarchy of classes comprising the active entities in the video game "Pac-man". Your task is ultimately to implement the four concrete classes representing the four "ghosts" or enemies in the game. The classes are named according to the common nicknames for the ghosts:

`Blinky.java`
`Pinky.java`
`Inky.java`
`Clyde.java`

All of them must directly or indirectly implement the interface **`api.Actor`**. In implementing them, you will also need to define one or more abstract classes in order to:

- a) avoid duplicating common code between the four types, and
- b) avoid having redundant code or attributes between your code and the **`Pacman`** class, which is already implemented and also implements the **`api.Actor`** interface.

All your code goes in the **`hw4`** package. The **`hw4`** package initially includes just the given **`Pacman`** class, which you will eventually need to modify in order to move common code into an abstract superclass that is also extended, directly or indirectly, by your ghosts.

(Note: you should probably *not* plan to move or modify the more complex methods of **`Pacman`**, namely **`tryTurn`** and **`update`**. Although the ghost classes also have an **`update`** method, you'll find that it is significantly different from the one in **`Pacman`** due to the different strategies for motion. You might find it useful to read the **`update`** code, however, since some of the calculations will be similar.)

Using inheritance

Part of your grade for this assignment (e.g. 20-25%) will be based on how effectively you are able to implement everything with a minimum of code duplication. Here are some specific restrictions; see the later section " Requirements and guidelines regarding inheritance" for more details.

- You may not use **`public`**, **`protected`** or package-private variables. (*You can provide protected accessor methods or constructors when needed.*)

- You may not modify anything in the **`api`** package, other than uncommenting the lines in the **`PacmanGame`** constructor that call the constructors for **`Blinky`**, **`Pinky`**, **`Inky`**, and **`Clyde`**.

Overview

Pacman is a simple and iconic video game dating from 1980. If you are not familiar at all with the game, don't worry. To get a feel for it, try the "Google doodle" that was created for the game's 30th anniversary:

https://www.google.com/logos/2010/pacman10-i.html

(If it's not obvious: use the arrow keys, eat all the dots before you run into a ghost. But if you eat an "energizer" (one of the larger dots) then you get extra points for catching a "frightened" ghost.)
There many, many internet sites where you can play it too, most of which have a lot of annoying ads. At the time this document was written, this one seems relatively benign and faithful to the original:

https://www.retrogames.cz/play_017-DOS.php?language=EN

For a conceptual explanation of the game, read the first 20% or so of this article:

https://gameinternals.com/understanding-pac-man-ghost-behavior

Overview of ghost behavior

The ghost AI

For our purposes, the really interesting part is the logic for the ghost movement. There are four ghosts, and they all have slightly different behavior. In playing the game it seems like sometimes they are all after you, and sometimes they back off; sometimes they seem like they are chasing you from behind, sometimes they are trying to get ahead and cut you off. None of this is an illusion; it's all designed into the game. What's amazing is that this seemingly complex behavior is a consequence of some very simple rules that you are about to implement.

The best place to begin is to read the rest of the article mentioned above:

https://gameinternals.com/understanding-pac-man-ghost-behavior

The main idea is that in most cases the ghosts move by choosing a "target" cell, and then selecting the next cell based on minimizing the straight-line distance to the target. We refer to this as the *next-cell algorithm*. There is a detailed explanation and some diagrams in the gameinternals article. There are five possible "modes" in which a given ghost can be, and the target in use depends on the mode. We identify modes using the enum type **`Mode`**:

`INACTIVE` - ghost is at its home location (i.e., the "ghost house") and does not move until the mode is changed
`DEAD` - uses the next-cell algorithm where the target is the home location
`FRIGHTENED` - there is no target; ghost just chooses a neighboring cell at random, using a pseudorandom generator provided on construction
`SCATTER` - uses the next-cell algorithm where the target is a fixed pre-defined cell outside one of the corners of the maze
`CHASE` - uses the next-cell algorithm where the target is selected according to a strategy that is different for each of the ghosts
We will implement the targeting strategies pretty faithfully based on their description in the gameinternals article. However, we don't try to reproduce the "overflow bug" for Pinky and Inky that is mentioned there. Also note that the target cell may be outside the maze, or have negative coordinates; this is not a problem.

Cells and exact coordinates and the MazeMap

The maze is a rectangular grid of cells (similar to the maze from miniassignment 3). At any given time, an actor has an *exact position* within this grid, a floating-point value in units of grid cells giving the horizontal and vertical distance to the actor's center from the upper-left corner of the maze. For example, if an actor's **`rowExact`** is 2.5 and **`colExact`** is 3.1, then its center is near the left edge of the cell with location (2, 3). This is considered the actor's "current cell", and the location of the current cell can always be found as

```
((int) a.getRowExact(), (int) a.getColExact())
```

The calculations for targeting are all based on the current cell location, not the floating point values.

In determining where it can move, a ghost needs to know where the walls are, but does not need access to all details of the game or the 2d grid. On construction we provide each ghost with an instance of **`api.MazeMap`**. A **`MazeMap`** encapsulates the game but provides only a few limited accessor methods for detecting walls and the boundaries of the grid. (The latter are needed for when there is a tunnel that wraps around from one side to the other.)

How motion works; the update() method

The idea is that each actor has a *current increment* (i.e., a speed) and a *direction*, and periodically the game will invoke the actor's **`update()`** method which in most cases just adds or subtracts the increment to/from the exact row or column coordinate, depending on the direction. Each actor is constructed with a *base increment* (originally chosen based on the difficulty level of the game and the game's frame rate), which it uses as the current increment most of the time. The current increment is two-thirds of the base increment when in FRIGHTENED mode, and is double the base increment when in DEAD mode. (The game originally reduced the ghost speed when wrapping around in the tunnel, but we are ignoring that detail.)

Descriptors

The argument to the **`update`** method is of type **`api.Descriptor`**. It is just a simple, immutable data container that contains the current position of the player (Pacman), the direction of the player, and the position of the 0th ghost (i.e. Blinky). These pieces of information are generally needed for finding the target and therefore for calculating the next cell.

Increment adjustments for an approaching turn

Each time a ghost's center crosses into a new cell, it calculates (based on the **`calculateNextCell`** method) which will be the next cell after that, so it is always looking one cell ahead. Therefore, as it moves across a cell, it knows in advance whether it needs to change direction when it reaches the center of the cell. This is significant, because the distance to the center of the cell may be smaller than a full increment, and if changing direction, it can't go past the center.

Wrapping around

Another minor special case is when the column index is 0 or is equal to the width - 1. This can only happen when the actor is at the end of the tunnel and may need to wrap around to the other side (depending on the direction). We assume that tunnels, if any, will only go horizontally.

The calculateNextCell and getNextCell methods

These are **required** public methods of all four ghost classes and are not part of the **`Actor`** interface. This is where the ghost AI happens!

```
public void calculateNextCell(Descriptor desc)
public Location getNextCell()
```

The `calculateNextCell` method does not return anything, but is expected to update some aspects of the ghost's internal state, in particular the value returned by `getNextCell`. You may also find it convenient to maintain an instance variable for the *next direction* as well, to make it easy to detect an upcoming turn. The behavior depends on the mode, as described above under "The Ghost AI". The `getNextCell` method is a one-line accessor that just returns the current value of the next cell.

Note that in INACTIVE mode, `calculateNextCell` does nothing and the result of `getNextCell` is undefined.

Overriding the setMode method

The override of `setMode` for the ghosts needs to update the mode, duh, and also:

1. invoke calculateNextCell, and
2. adjust the speed, if necessary

Periodically the game will invoke `setMode` on a ghost. The game maintains a "global" mode that alternates between SCATTER and CHASE based on an internal timetable depending on the level of difficulty, and most of the time all the ghosts will be set with the global mode. However, when the player eats an energizer, all active ghosts will be set to FRIGHTENED, and then set back to the global mode after a period of time depending on the game level. However, if the player catches a ghost in FRIGHTENED mode, it is changed to DEAD and returns itself to its home location. When a dead ghost reaches home, the game changes it to INACTIVE mode for a level-dependent period of time.

(Note: the original game also included a complex set of rules for determining when an inactive ghost was able to become active again, based on several counters for many dots had been consumed at which point in the game. We are ignoring all of those rules: inactive ghosts are changed to an active mode based solely on a timer maintained by the game. Similarly, the original game included a rule that a ghost reverses direction upon most mode changes. We are ignoring that one too.)

Overriding reset()

The reset method needs to set the mode to INACTIVE, set the location to the home location, set the current speed to the base speed, and set the current direction to the home direction.

Constructors

Each ghost has a constructor with the following parameters, in this order:

`MazeMap maze`
a read-only representation of the maze, for detecting walls and edges
`Location home`
initial location for the ghost
`double baseSpeed`
base increment
`Direction homeDirection`
initial direction
`Location scatterTarget`
fixed target for use during SCATTER mode
`Random rand`
predefined global random generator for use during FRIGHTENED mode

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 demo code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. The UI code is complex and 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 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 everything in your hw4 package. **Please check this carefully. In this assignment you may be including one or more abstract classes of your own, in addition to the four required classes and your modified Pacman class, so make sure they have been included in the zip file.**

Importing the sample code

The sample code is distributed as a complete Eclipse project that you can import.

1. Download the zip file **`hw4_skeleton.zip`**. 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 for some reason you have problems with the process above, 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 **`hw4`** and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

Out of the box, the sample code should compile and you should be able to run the main class **`ui.RunGame`** and control the player icon with the arrow keys. You can edit the main method to change the maze. Note that the maze is constructed based on a string array, as for miniassignment 3, in which the letters S, B, P, I, C represent for the initial locations of the player, Blinky, Pinky, Inky, and Clyde, respectively. However, in the game class **`api.PacmanGame`**, the lines of code that actually construct the ghosts is commented out. (See roughly lines 195 - 230.) As you implement ghost classes you can uncomment the relevant lines.

To slow down (or speed up) the game play, adjust the constants **`PLAYER_SPEED_FACTOR`** and **`ENEMY_SPEED_FACTOR`** defined at the top of the **`PacmanGame`** class. For testing it is sometimes useful to construct a game with a smaller frame rate (see the SimpleTest examples below).

Getting started

Sometimes the best way to start on an inheritance hierarchy is, initially, to forget about inheritance. That is, if you have two classes A and B that might be related, just implement both of them completely from scratch. Then look at what they have in common. Does it make logical sense for one to extend the other? Should the common code be factored into a common, abstract superclass? **Be sure you have done and understood the second checkpoint of Lab 8 before you start making decisions about inheritance.**

1. So one reasonable place to start is just to define the **`Blinky`** class, declare it to implement the **`Actor`** interface, and add all the required methods. You will find that much of the code for the simpler methods can be copied directly from **`Pacman.java`**. (If you are already comfortable with inheritance, you could consider moving those common methods into an abstract superclass that can be extended by **`Pacman`** and by **`Blinky`**. But this can always be done later.) Note that nothing related to "turning mode" will be used in **`Blinky`**; the motion for the ghosts is actually much simpler to implement since it is programmatically controlled rather than user controlled.

2. There are two somewhat complex methods to implement in **`Blinky`**: **`calculateNextCell`** and **`update`**. They are closely interrelated and a bit tricky to develop independently. But here is an idea. First, create a completely bogus implementation of **`calculateNextCell`**, like this:

```
public void calculateNextCell(Descriptor d)
{
  Location currentLoc = getCurrentLocation();
  if (currentLoc.row() > 2)
  {
    nextLoc = new Location(currentLoc.row() - 1, currentLoc.col());
    nextDir = UP;
  }
  else
  {
    nextLoc = new Location(currentLoc.row(), currentLoc.col() + 1);
    nextDir = RIGHT;
  }
}
```

(This assumes that there are two instance variables **`nextLoc`** and **`nextDir`**, representing the next cell and the direction towards that cell; the latter is convenient but not mandatory.) That is, the next cell is always the one above until it gets to row 2, and then the next cell is always to the right. Then, uncomment the lines in **`PacmanGame.java`** where **`Blinky`** is instantiated, probably around lines 195 - 200. You can then make a simple test like this (see SimpleTest.java in the default package).

```
public static final String[] SIMPLE1 = {
  "#######",
  "#.....#",
  "#.....#",
  "#.....#",
  "#.....#",
  "#..B..#",
  "#S....#",
  "#######",
};
```

```
public static void main(String[] args)
{
  // using a frame rate of 10, the speed increment will be 0.4
  PacmanGame game = new PacmanGame(SIMPLE1, 10);

  // Blinky is always at index 0 in the enemies array
  Blinky b = (Blinky) game.getEnemies()[0];

  System.out.println(b.getCurrentLocation()); // expected (5, 3)
  System.out.println(b.getCurrentIncrement()); // expected 0.4
  System.out.println(b.getCurrentDirection()); // expected UP
  System.out.println(b.getRowExact() + ", " + b.getColExact());//(5.5, 3.5)
  System.out.println(b.getNextCell());  // expected null
  System.out.println();
```

(That default initial direction of UP is selected by the game when instantiating Blinky.)
Next add a call to **`calculateNextCell`**:

```
  // this should update value of getNextCell()
  b.calculateNextCell(makeDescriptor(game));
  System.out.println(b.getNextCell()); // expected (4, 3)
  System.out.println();
```

where **`makeDescriptor`** is just a helper method to create a descriptor for the current game state:

```
private static Descriptor makeDescriptor(PacmanGame game)
{
  Location enemyLoc = game.getEnemies()[0].getCurrentLocation();
  Location playerLoc = game.getPlayer().getCurrentLocation();
  Direction playerDir = game.getPlayer().getCurrentDirection();
  return new Descriptor(playerLoc, playerDir, enemyLoc);
}
```

3. After this setup, you can start experimenting with the implementation of **`update`**. The first thing to do is, of course, update the position based on the current direction and current increment. (Note: eventually, you'll add a check to make sure that update does nothing when the current mode is INACTIVE, but let us worry about that later.) Since the current direction is UP, we expect the row coordinate to decrease by 0.4 with each update (subject to floating-point roundoff errors, of course):

```
  // now some updates
  b.update(makeDescriptor(game));
  System.out.println(b.getRowExact() + ", " + b.getColExact());
  // expected ~5.1, 3.5
  b.update(makeDescriptor(game));
  System.out.println(b.getRowExact() + ", " + b.getColExact());
  // expected ~4.7, 3.5
```

4. If we do a bunch more updates, you'll need to implement a bit more into your **`update`** method. If we add these lines to the test,

```
for (int i = 0; i < 10; ++i)
{
  b.update(makeDescriptor(game));
  System.out.println(b.getRowExact() + ", " + b.getColExact());
  System.out.println(b.getCurrentDirection());
  System.out.println(b.getNextCell());
}
```

we eventually expect to see the following output:

```
4.299999999999999, 3.5
UP
(3, 3)
3.899999999999999, 3.5        <-- (1) crossing into next cell should trigger
UP                                    call to calculateNextCell
(2, 3)                        <-- new value of getNextCell is cell above
3.499999999999999, 3.5
UP
(2, 3)
3.099999999999999, 3.5
UP
(2, 3)
2.6999999999999993, 3.5       <-- (2) crossing into next cell should trigger
UP                                    call to calculateNextCell
(2, 4)                        <-- new value of getNextCell is to the right now
2.5, 3.5                      <-- (3) since there is a change in direction, we
RIGHT                             can't go past the center of current cell, so
(2, 4)                            only subtract 0.2 instead of full increment.
2.5, 3.9                          Current direction changes to RIGHT when we
RIGHT                             reach the center
(2, 4)
2.5, 4.3
RIGHT
(2, 5)
2.5, 4.7
RIGHT
(2, 5)
2.5, 5.1000000000000005
RIGHT
(2, 6)
```

See the appendix at the end of this document for an illustrated version of the above.
The annotations above illustrate the key pieces of the method. One thing is probably not obvious, though: be sure that when you check whether you are close to the center of the cell, as in note (3) above, that you allow for the fact that you might already *be* at the center (or within some margin of error). This happens in particular when a ghost transitions from INACTIVE mode after returning to its home location. It will frequently occur that, say, its initial direction

might be UP, but the first call to **`calculateNextCell`** selects a cell to the left. In that case, the first **`update`** call should recognize that the distance to the center of the current cell is (approximately) zero and that the new direction is something other than UP, so it should just subtract zero units from the row coordinate and then the change the current direction to LEFT.

You can see that a frequently needed value is the distance from the current position to the center of the current cell, measured in the direction of travel. You may wish to reuse the helper method **`distanceToCenter`**, found at the bottom of **`Pacman.java`**.

5. You can more or less independently work on the targeting calculation in **`calculateNextCell`**. Remember that the **`setMode`** method should always invoke **`calculateNextCell`** when setting the mode.

Here is a simple test (see SimpleTest2.java in the default package).

```
  public static final String[] SIMPLE2 = {
    "############",
    "#..........#",
    "#.B........#",
    "#..........#",
    "#S.........#",
    "############",
  };

public static void main(String[] args)
{
  PacmanGame game = new PacmanGame(SIMPLE2, 10);
  Blinky b = (Blinky) game.getEnemies()[0];
  b.setMode(SCATTER, SimpleTest.makeDescriptor(game));
  System.out.println(b.getNextCell()); // expected (2, 3)


  System.out.println(b.getCurrentDirection());  // UP
  b.update(SimpleTest.makeDescriptor(game));
  System.out.println(b.getCurrentDirection());  // RIGHT
  System.out.println(b.getRowExact() + ", " + b.getColExact());
     // still 2.5, 2.5
  System.out.println();
```

The default scatter target for Blinky is at (-3, width - 3) or (-3, 9) in this case, as set in the constructor. If we look at the three cells around (2, 2), the cell (2, 3) at right has the closest straight-line distance to the scatter target. Note also that since the direction will change from UP to RIGHT, and the ghost is already at the center of the current cell, it does not actually change

position in the call to update, effectively moving zero distance to the center and then changing the current direction to right.

An important and non-obvious point: when two of the neighboring cells are the "same" distance from the target, they need to be prioritized in the order UP --> LEFT --> DOWN --> RIGHT, as described in the gameinternals article. Because of roundoff errors, you need to consider two distances to be the "same" if they differ only by some small margin of error, e.g. the ERR constant defined in **`Pacman.java`**. Or put another way, you should not, for example, prefer a LEFT direction over an UP direction unless the distance is shorter by an amount that is more than ERR.

Another subtlety: remember that a ghost can't change direction when it is already past the center of its current cell. So what happens if there is a mode change in this case? Since it is too late to change direction, the solution is to simply do nothing. When the ghost crosses into its next cell, it will trigger another call to **`calculateNextCell`** that will use the now-current mode. (See SimpleTest3.java for an example.) This same rule - a ghost can only change direction when it reaches the center of its current cell - also applies when a ghost is FRIGHTENED and is reversing direction.

6. Up to this point, you have not actually needed the descriptor that is always passed in to update and setMode. For CHASE mode, you do, since the target in this case is the current location of Pacman. (Remember, we use the integer row and column index for these calculations, not the "exact" values.) We could continue the test above using the same maze:

```
game = new PacmanGame(SIMPLE2, 10);
b = (Blinky) game.getEnemies()[0];
b.setMode(CHASE, SimpleTest.makeDescriptor(game));
System.out.println(b.getNextCell()); // expected (2, 1)
```

This time, the next cell should be the one closest to Pacman at (4, 1), which is obviously the cell below at (3, 2). However, the current direction is UP and ghosts are not allowed to reverse direction except in FRIGHTENED mode, so the cell below is off-limits. Among the three possible cells (1, 2), (2, 1), and (2, 3), going left to (2, 1) minimizes the distance to target.

Hopefully you have noticed that once you know the target cell, which depends on the current mode, the algorithm to identify a list of possible cells and minimize the distance should always be the same. Try DEAD and FRIGHTENED mode too. Don't forget to update the current increment (speed) appropriately in your **`setMode`** method: two-thirds speed when FRIGHTENED, double-speed when DEAD. The examples above should give you some ideas for testing everything.

7. Next you might consider how to implement another ghost. What code is the same? What is different? This is where you want to define an abstract class for the code that is common to the four ghost types.

8. If you haven't already done so, take a look at the common code between **`Pacman`** and the ghosts. You are expected to refactor this common code into a base class shared by **`Pacman`** and all the ghosts. Your modifications to the **`Pacman`** class not change its behavior. Note that it should NOT be necessary to modify the **`tryTurn`** or update methods of **`Pacman`**. (And yes, it is perfectly ok for one abstract class to extend another.)

Requirements and guidelines regarding inheritance

A generous portion of your grade (20-25%) will be based on how well you have used inheritance, and abstract classes, to create a clean design with a minimum of duplicated code. Please note that there is no one, absolute, correct answer for where every method and piece of data belongs – you have design choices to make. In general we can say, however that is NOT appropriate to make any of the ghosts a subclass of Pacman, which includes a great deal of code that is inapplicable to the ghosts along with some code that is common to the ghosts. Similarly, it is not appropriate to make any of the ghosts a subclass of another ghost; rather, they should all extend a common supertype.

Specific requirements

> **You are not allowed to use non-private variables**. Call the superclass constructor to initialize its attributes, and use superclass accessor methods to access them. If your subclass needs some kind of access that isn't already provided by public methods, you are allowed to define your own **`protected`** methods or constructors.

Other general guidelines

- You should not use **`instanceof`** or **`getClass()`** in your code, or do any other type of runtime type-checking, to implement correct behavior. Rely on polymorphism.

- No class should contain extra attributes or methods that it doesn't need to satisfy its own specification (exception: the getMode/setMode methods are intentionally implemented as do-nothing methods in Pacman class. Long story.)

- Do not ever write code like this:

```
public void foo()
{
  super.foo()
}
```

There is almost never any reason to declare and implement a method that is already implemented in the superclass, unless you need to override it to change its behavior. *That is the whole point of inheritance!*

Style and documentation

Special documentation requirement: write a few sentences, as part of the class Javadoc for Blinky, explaining your design choices for how you organized the class hierarchy.

Roughly 10 to 15% of the points will be for documentation and code style.

Some restrictions and guidelines for using inheritance are described above.

When you are overriding a superclass or interface method, **it is usually NOT necessary to re-write the Javadoc**, unless you are really, really changing its behavior. Just include the @Override annotation. (The Javadoc tool automatically copies the superclass Javadoc where needed.)

The other general guidelines are the same as in homework 3. Remember the following:

- You must add an @author tag with your name to the class javadoc at the top of each of the classes you write.
- You must javadoc each instance variable and helper method that you add, as well as specified methods that are not already documented in the Actor interface. Anything you add that is not in the public API must be **`private`** or **`protected`**.
- 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 **`hw4`**. 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.

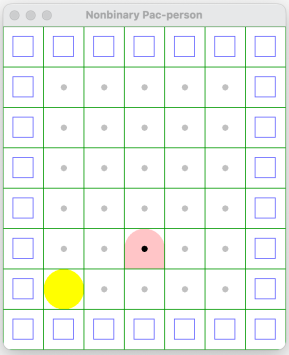
Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw4.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains the four required classes, the Pacman class, and any others you defined in the **`hw4`** package. **Please LOOK at the file you upload and make sure it is the right one and contains everything needed!**

Submit the zip file to Canvas using the Assignment 4 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 linked on our Canvas home page.

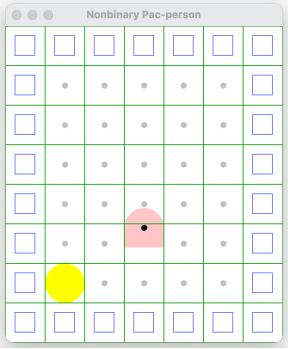
> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw4**, which in turn should contain everything in your hw4 directory. You can accomplish this by zipping up the **hw4** 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.

Appendix: SimpleTest illustrated

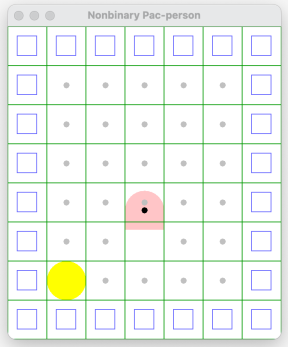
Remember this is using a simplified version of calculateNextCell that always sets the next cell to be the one above until it gets to row 2, after which it sets the next cell to the one at right. This is after one initial call to calculateNextCell:



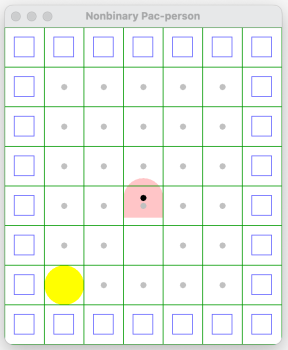
Current exact position (center of ghost): 5.5, 3.5
Current direction: UP
Next cell: (4, 3)



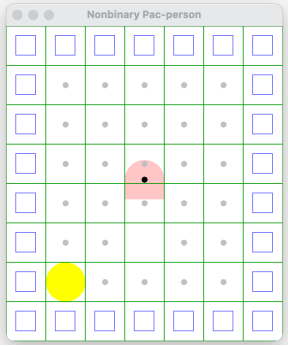
1st update():
Next cell is above, no direction change, subtract increment from row coordinate
Current exact position: ~5.1, 3.5
Current direction: UP
Next cell: (4, 3)



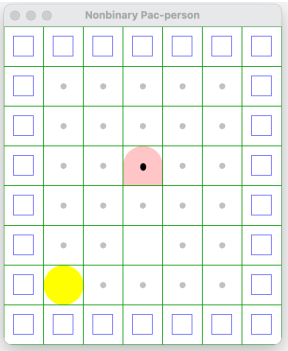
2nd update():
Next cell is above, no direction change, subtract increment
Current exact position: ~4.7, 3.5
Current direction: UP
Crossing into next cell triggers call to calculateNextCell
Next cell is now: (3, 3)



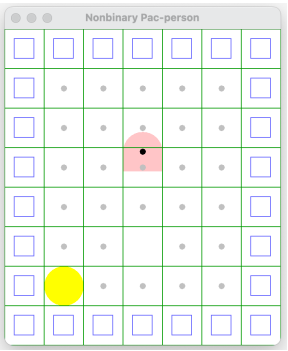
3rd update():
Next cell is above, no direction change, subtract increment
Current exact position: ~4.3, 3.5
Current direction: UP
Next cell: (3, 3)



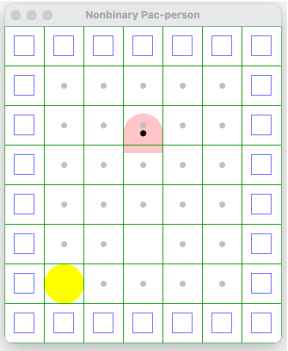
4th update():
Next cell is above, no direction change, subtract increment
Current exact position: ~3.9, 3.5
Current direction: UP
Crossing into next cell triggers call to calculateNextCell
Next cell is now: (2, 3)



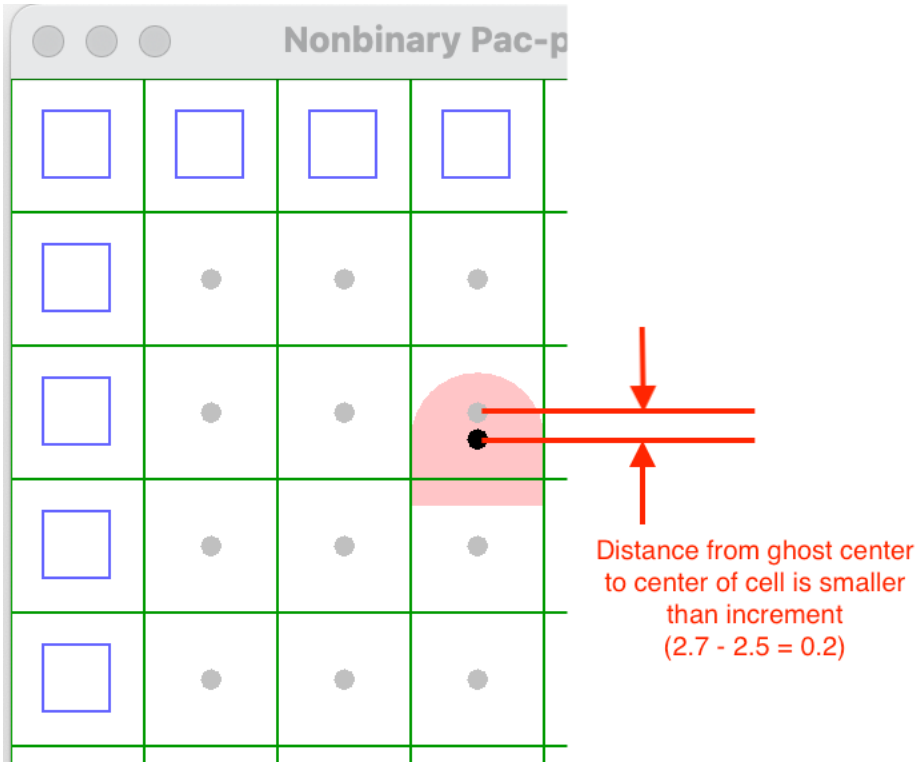
5th update():
Next cell is above, no direction change, subtract increment
Current exact position: ~3.5, 3.5
Current direction: UP
Next cell: (2, 3)

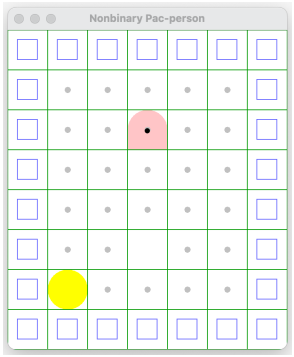


6th update():
Next cell is above, no direction change, subtract increment
Current exact position: ~3.1, 3.5
Current direction: UP
Next cell: (2, 3)

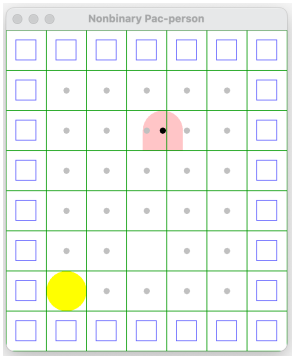


7th update():
Next cell is above, no direction change, subtract increment
Current exact position: ~2.7, 3.5
Current direction: UP
Crossing into next cell triggers call to calculateNextCell
Next cell is now: (2, 4)

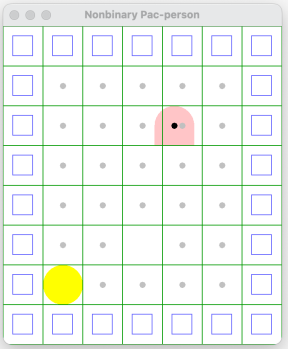




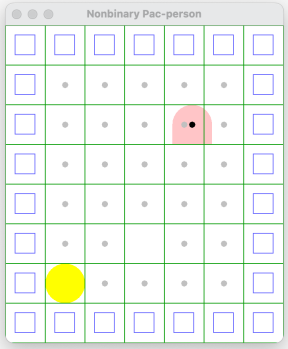
8th update():
Next cell (2, 4) is to the **right** and our current direction is UP. *Since there is a direction change, we can't go past center of current cell. Since distance to center is only 0.2, which is less than increment, we can't subtract a full increment; subtract 0.2 instead*
Current exact position: 2.5, 3.5
Current direction is now: RIGHT
Next cell: (2, 4)



9th update():
Next cell to right, no direction change. Since we are going right now, we **add** the increment to the column position
Current exact position: 2.5, ~3.9
Current direction: RIGHT
Next cell: (2, 4)



10th update():
Next cell to right, no direction change, add increment
Current exact position: 2.5, ~4.3
Current direction: RIGHT
Crossing into next cell triggers call to calculateNextCell
Next cell is now: (2, 5)



11th update():
Next cell to right, no direction change, add increment
Current exact position: 2.5, ~4.7
Current direction: RIGHT
Next cell: (2, 5)