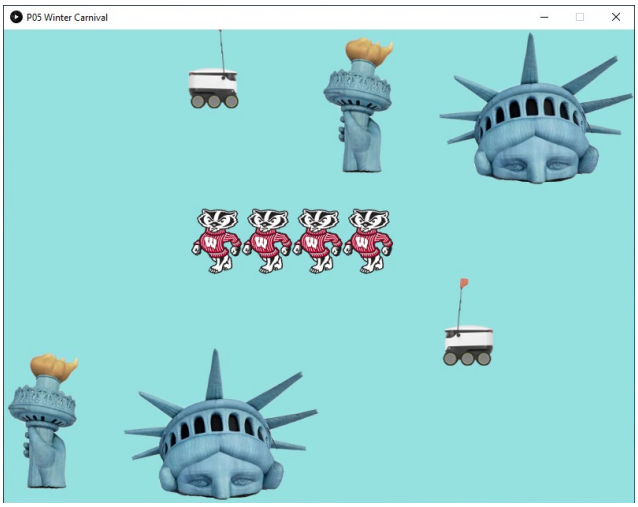
P05 Winter Carnival

Programming II (CS300) Spring 2020	**Due: 9:59PM on March 4, 2020**
Department of Computer Sciences	Pair Programming **is Allowed.**
University of Wisconsin-Madison	Register by Sunday, Mar 1 @ 9:59PM

Overview and Learning Objectives:

The winter carnival in Madison was held between February 3rd and 9th in 2020 and is the inspiration for this programming assignment. Through this assignment, you'll develop a graphical simulation with a frozen statue, starship delivery robots, and dancing badgers (as depicted in the image below).



Through the process of completing this assignment, you will have a chance to further practice organizing code across instantiable classes, and you will also use inheritance to relate classes to each other.

Grading Rubric:

5 points	**Pre-Assignment Quiz:** You should not have access to this write-up until after you have completed the corresponding pre-assignment quiz through Canvas.
15 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is correct. To become more confident in this, you must write and run additional tests of your own.
30 points	**Manual Grading and Supplemental Automated Tests:** When your final grade feedback appears on Gradescope, it will include the feedback from these additional automated grading tests, as well as feedback from human graders who review the code in your submission by hand.
50 points	**TOTAL**

Requirements and Reminders:

Everyone must complete the pair programming registration form by Sunday, Mar 1 @ 9:59PM.

Ensure that your code for every assignment is styled in conformance to the **Course Style Guide**.

0. Project Files and Setup

Create a new Java Project in Eclipse called: P05 Winter Carnival. Then create a Java class / source file within that project's src folder (inside the default package) called **WinterCarnival**. In future sections, you will also be creating the following classes, each within their own file: FrozenStatue, StarshipRobot, and DancingBadger. In order to draw graphics to a window using Java, we are providing you with a SimulationEngine class within the following P05.jar file. Save a copy of this file within your project folder, and then right-click on this file within the Package Explorer in Eclipse. From this method you should select Build Path / Add to Build Path, so that your code can begin to make use of the provided SimulationEngine class. The JavaDocs for code provided in this jar file can be found here.

1. Creating a Window and Drawing Graphics

Begin by creating a new SimulationEngine object within your main method. Running this program should create a window with a blue background titled "P05 Winter Carnival". We want to make use of the code within this class, and we want to customize it implement a winter carnival. To do this, we'll declare that our WinterCarnival class inherits from the provided SimulationEngine class. Then change your main method to create a new WinterCarnival object instead of a SimulationEngine object. You should now see the same blue window.

We can begin to customize this simulation for our needs. Create a new folder named images, inside your project folder. Then save these three files within that folder: dancingBadger.png, frozenStatue.png, starshipRobot.png. To display one of these images, we can call the instance method draw() that our WinterCarnival class inherited from the SimulationEngine. But we need to ensure that this method is called repeatedly for as long as our simulation is running. To accomplish this, we'll override the update() method (inherited from the SimulationEngine class). For testing purposes, we'll try calling the draw method with the following arguments (see JavaDocs for argument descriptions).

```
draw("images"+File.separator+"dancingBadger.png", 400, 300, false);
```

A badger should now appear in the center of your blue window. After you confirm that this is working, you can remove the draw call before moving on to Step 2.

2. Creating the Frozen Statue class

We are going to be drawing and animating a few different objects for this assignment, so we'll organize the code for these different kinds of objects into different classes. Let's start with the simplest of these classes: create a new class called FrozenStatue within the default package. Define ONLY the following fields inside this class, and use the protected access type for each of them:

- float x – the horizontal position of this object in pixels from 0-left to 800-right
- float y – the vertical position of this object in pixels from 0-top to 600-bottom
- boolean isFacingRight – used to mirror image (flip left to right) only when this field is false
- String imageName – the relative path to the image file (from the working directory)

Define one constructor for this class that takes an array of two floats as input. The first of these floats (index 0) holds the initial x position, and the second (index 1) holds the initial y position for the object being constructed. This is how positions will be stored in float[] through this assignment. The isFacingRight should be initialized to true, and the imageName String should be initialized to `"images"+File.separator+"frozenStatue.png"`.

The only instance method that should be defined within this class should have the following signature. This method should call the draw() method on the SimulationEngine that is passed to it, and this call should make use of the object's imageName, x, y, and isFacingRight fields.

```
public void update(SimulationEngine engine) {}
```

A quick test that you can perform to check whether this is working is to create a FrozenStatue object and then call its update method from your WinterCarival.update() method definition. If this works, you should see the statue image appear at the specified position within the window.

3. Creating and Updating Multiple FrozenStatue objects

Rather than creating new objects every time the simulation updates (which may waste a lot of memory), we're going to create a couple of objects when the program first starts and then only call the update method on those objects from the WinterCarnival.update() definition.

Create a new private ArrayList<FrozenStatue> field named **objects** within your WinterCarnival class. This should be the only field that is defined within this class throughout the assignment. Initialize this field to contain an ArrayList holding two FrozenStatue objects in your WinterCarnival's no-argument constructor. One frozen statue should be positioned at (600, 100), and the other at position (200,500). Then change WinterCarnival.update() to call the update() method on every object within this arraylist. This should result in two frozen statues being displayed when you run this program: one in the upper-right and one in the lower-left corner.

4. Creating the StarshipRobot class

Now create a new class called StarshipRobot to deliver food across the carnival. These robots will be similar to the FrozenStatue that we just implemented in their ability to be drawn to any position on the screen, but they will also be able to move back and forth between two positions. We'll start by having this class inherit from our existing FrozenStatue class. Define ONLY the following fields within this class, and use the protected access type for each of them:

- float[][] beginAndEnd – array of two positions, that this robot moves back and forth between the contents of this 2d array are organized as follows: { { beginX, beginY }, { endX, endY }}
- float[] destination – the position that this robot is currently moving towards
- float speed – the speed in pixels that this robot moves durring each update

Define one constructor for this class that takes a float[][] argument. This argument contains the beginAndEnd value that should be stored within this object. The new robot should start at the begin position, and should use the end position as its initial destination. It should have a speed of 6, an isFacingRight of true, and an imageName of `"images"+File.separator+"starshipRobot.png"`.

The motion for the StarshipRobot class is going to be organized across the following methods:

```
protected boolean moveTowardDestination() {}
protected void updateDestination() {}
@Override
public void update(SimulationEngine engine) {}
```

The moveTowardDestination method should update the position of the object that it is called on by moving it **speed** units closer to its **destination** (see Appendix for help calculating this position).

Additionally, this method should set this object's isFacingRight field to true, when its destination x-position is larger than its own x-position. Otherwise the isFacingRight field should be set to false. Finally, this method should return true when the initial distance (before moving) between this object and its destination is less than 2x its speed, otherwise it should return false. Note that this allow us some flexibility to move between any two points that are not necessarily a multiple of speed units apart, and ensures that the robot turns around before overshooting their target destination.

The job of the update method is to call this moveTowardDestination() method. When this method returns true, the object is close enough to its destination that its destination should be updated to the other position within beginAndEnd. The updateDestination() method should be called from update() to make this change. And the result is that the starship should move back and forth between the specified start and stop positions. The last thing that the update() method should do is call the update method that is defined within the FrozenStatue class, to draw itself to the screen at its new position.

Ensure that your StarshipRobot class is implemented correctly by creating two new StarshipRobot objects in your WinterCarnival class constructor, and add each of them to the objects ArrayList. One of these robots should move between positions (0,0)-begin and (600,100)-end, and the other should move between positions (800,300)-begin and (200,500)-end.

5. Creating the DancingBadger class

Create one more class called DancingBadger that inherits all of the movement and drawing code from your StarshipRobot class. DancingBadgers will start at a given position, and then move according to a sequence of dance steps. This class must include ONLY the following protected fields:

- DanceStep[] danceSteps – the sequence of directions / dance steps for this badger to move
- int stepIndex – the index of the next step within danceSteps for this badger to move through

Note that the provided P05.jar file contains the definition for the DanceStep enumeration, and that these JavaDocs describe the method getPositionAfter(float[]) that will be very helpful to make use of when implementing this class.

Define one constructor for this class that takes a starting position (float[]) followed by an array of DanceSteps as arguments. This constructor should initialize the new object's imageName to `"images"+File.separator+"dancingBadger.png"`, its speed to 2, its isFacingRight to true, its start position as specified, its destination to be the position that it should move toward according to the first step in the array, and its step index as 1 (since it is already setup to move according to step[0]).

The only method that needs to be defined (overridden) within this class is updateDestination(). The StarshipRobot implementation bounced the destination vector back and forth between its begin and end positions. However, the DancingBadger class will need to update this destination vector according to the next dance step (the one inside danceSteps, at index stepIndex). After updating this destination vector, be sure to increment the stepIndex by one. When the stepIndex gets larger than the largest valid array index, reset it to 0 so that it restarts the sequence of steps from the beginning.

Finally, create four DancingBadger objects and add them to the WinterCarnival's ArrayList of objects within its constructor. These Dancing Badgers should begin at positions: (304,268), (368,268), (432,268), and (496,268), and they should all make use of the same sequence of dance steps: Left, Right, Right, Left, Down, Left, Right, Right, Left, Up.

6. Assignment Submission

Congratulations on completing this CS300 assignment! The only four files that you should submit to gradescope for this assignment are: WinterCarnival.java, FrozenStatue.java, StarshipRobot.java, and DancingBadger.java. Your score for this assignment is based on your submission marked "active" and made prior to the hard deadline. If you are working with a partner, be sure to review and follow the course Pair Programming Policy for linking your final submission to both your and your partner's accounts.

Appendix I: Implementation Tips

a. Tip for calculating the distance between two points

The distance between two points with (x,y) coordinates (x1,y2) and (x2,y2) can be calculated as:

$$distance = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

b. Tip for calculating a position that is a specific distance closer to a given destination.

This formula makes use of a given position "old" to move from, a "destination" position to move toward, and a scalar moveDistance describing the distance to move in that direction. The x and y components of the resulting position can be computed as follows:

$$newX = oldX + \frac{moveDistance * (destinationX - oldX)}{distance\ between\ destination\ and\ old, before\ moving}$$

$$newY = oldY + \frac{moveDistance * (destinationY - oldY)}{distance\ between\ destination\ and\ old, before\ moving}$$