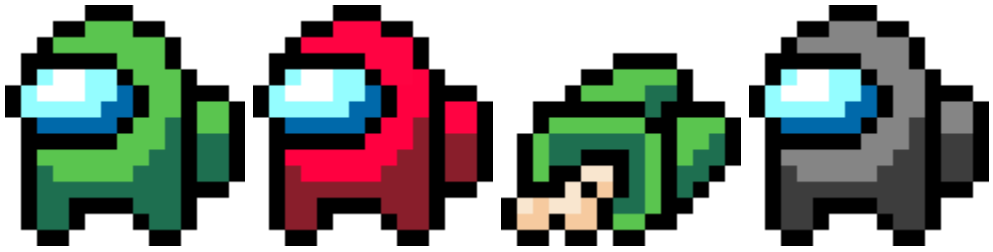
P02 Among Us

Overview

There is an impostor among us…



This program is an introduction to using the graphical user interface (GUI) library Processing. We won't be using it *directly* just yet, but between our home-brewed Utility class and this writeup, you should get a good sense of how the library works in preparation for using it directly in later assignments.

So no, we're not going to write the whole Among Us game, but… well, you'll see.

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on **09/17**.
+2.5 points	**5% Bonus Points**: students whose ***final*** submission to Gradescope has a timestamp earlier than **5:00 PM CDT** on **WED 09/20** will receive an additional 2.5 points toward this assignment, up to a maximum total of 50 points.
25 points	**Immediate Automated Tests**: accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. Passing all immediate automated tests does **not** guarantee full credit for the assignment.
20 points	**Additional Automated Tests**: these will also run on submission to Gradescope, but you will not receive feedback from these tests until after the submission deadline.

Learning Objectives

After completing this assignment, you should be able to:

- **Initialize** (create) and **use** custom objects and their methods in Java
- **Describe** how null references can be detected in a perfect-size array of objects
- **Create** a simple program with a graphical user interface using our Utility frontend for the Processing library
- **Explain** how and when the code in GUI "callback" methods runs

Additional Assignment Requirements and Notes

Keep in mind:

- Pair programming is **NOT ALLOWED** for this assignment. You must complete and submit P02 individually.

- The ONLY external libraries you may use in your program are:

  ```
  java.io.File
  processing.core.PImage
  ```

  Use of *any* other packages (outside of java.lang) is NOT permitted in your SpaceStation class.

- IMPORTANT: The automated tests in Gradescope do not have access to the full Processing library. If you use any methods in your program besides those provided in the Utility class, your code may work on your local machine but **FAIL** the automated tests. See the Reference section at the end of this writeup for a full list of the Processing methods Gradescope will have.

- You are allowed to define any **local** variables you may need to implement the methods in this specification (inside methods). You are NOT allowed to define any additional instance or static variables or constants beyond those specified in the write-up.

- All methods must be static. You are allowed to define additional **private** static helper methods.

- All classes and methods must have their own Javadoc-style method header comments in accordance with the CS 300 Course Style Guide.

- Any source code provided in this specification may be included verbatim in your program without attribution.

- **Run your program locally before you submit to Gradescope**. If it doesn't work on your computer, *it will not work on Gradescope*.

Need More Help?

Check out the resources available to CS 300 students here:
https://canvas.wisc.edu/courses/375321/pages/resources

CS 300 Assignment Requirements

You are responsible for following the requirements listed on both of these pages on all CS 300 assignments, whether you've read them recently or not. Take a moment to review them if it's been a while:

- Academic Conduct Expectations and Advice, which addresses such questions as:
  - How much can you talk to your classmates?
  - How much can you look up on the internet?
  - What do I do about hardware problems?
  - and more!
- Course Style Guide, which addresses such questions as:
  - What should my source code look like?
  - How much should I comment?
  - and more!

1. Getting Started

Your first few steps will be familiar from P01:

1. Create a new project in Eclipse, called something like **P02 Among Us**.
   a. Ensure this project uses Java 17. Select "JavaSE-17" under "Use an execution environment JRE" in the New Java Project dialog box.
   b. Do **not** create a project-specific package; use the default package.
2. Create one (1) Java source file within that project's src folder:
   a. **SpaceStation.java** (contains a main method)

But now we're going to add a twist - we've provided some code for you in a JAR[1] file, rather than as source code. The instructions in this writeup will focus on **Eclipse** procedures; there is a pinned post on the CS 300 Piazza with additional help for **IntelliJ** users.

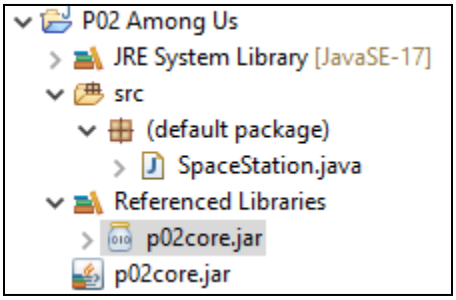
[1] JAR stands for **J**ava **AR**chive. It's a convenient way to distribute compiled Java code.

1.1 Download the Processing JAR file

Download the p02core.jar file[2], which contains the core Processing library (build 4.3), our Utility.java interface to that library, and a custom object class we'll explore later.

Copy the JAR file into your project folder or drag it into the project in the sidebar in Eclipse, and refresh the Package Explorer panel in Eclipse if you don't see it there yet.

Right click the JAR file in the project and select Build Path > Add to Build Path from the menu. The JAR should now appear as a Referenced Library in your project:



A screenshot of the Eclipse Package Explorer pane showing a project called P02 Among Us, set up with a build path that references p02core.jar

If the "Build Path" entry is missing when you right click on the JAR file in the Package Explorer:

1. Right-click on the project and choose "Properties"
2. Click on the "Java Build Path" option in the left side menu
3. From the Java Build Path window, click on the "Libraries" tab
4. Add the `p02core.jar` file by clicking "Add JARs…" from the right side menu and navigating to the file in your system
5. Click on the "Apply" button

1.2 Check your setup

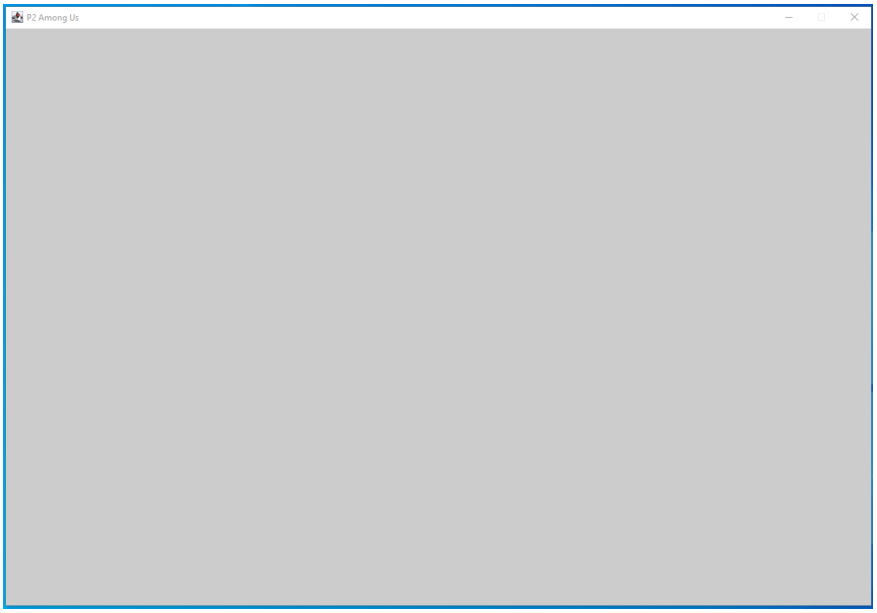
To test that the jar file was added correctly to your build path, add a main method to the SpaceStation class and add the following method call:

```
Utility.runApplication(); // starts the application
```

If everything is working properly, you should see a blank window with the text "P2 Among Us" in the top bar as shown on the next page, and an error message in the console that we'll resolve shortly:

[2] **For Mac users with Chrome**: this download may be blocked. If you're opposed to switching to Firefox (your friendly neighborhood CS professor says *SWITCH*) go to "chrome://downloads/" in the browser and click on "Show in folder" to open the folder where the JAR file is located.

```
ERROR: Could not find method named setup that can take arguments [] in class SpaceStation.
```



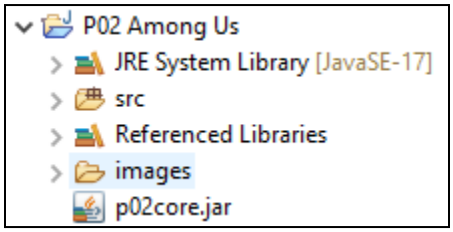
A screenshot of a blank P2 Among Us window.

If you have any questions or difficulties with this setup, please check Piazza or talk to a TA or peer mentor before continuing. Note that the provided jar file will ONLY work with Java 17, so if you're working with another version of Java, you'll need to switch now.

1.3 Download the sprite images

Finally, download the images.zip file ***and unzip it***; it contains 6 images of our Amogus friends as on the first page. (Make SURE you unzip the file before continuing; if you try to use the zip file directly, it won't work.)

Add the unzipped folder to your project folder in Eclipse, either by importing it or drag-and-dropping it into the Package Explorer directly:



2. Utility framework and overview of Amogus class

Using the Processing library in its pure form requires a bit more understanding of Java and Object-Oriented Programming than you have right now, so we've provided an interface called Utility so it's a bit easier to just jump into. This allows you to set up a GUI program in a way that's very similar to the text-based programs you've been writing so far.

Later this semester, you'll use the real thing! But for now: think of this as training wheels.

As you're writing your code for this program, you may want to refer to the Appendix at the end of this document for a full listing of the methods available to you in the Utility class, as well as the Amogus class javadocs.

2.1 Callback methods overview

A graphical user interface still works slightly differently – it relies on **callback** methods. These are methods that another method calls; you won't call them from your code, the GUI library will.

Later in this program, you'll implement the following *callback* methods:

- `setup()` : called automatically when the program begins. All data field initialization should happen here, any program configuration actions, etc. This method is only ever called *once*.
- `draw()` : runs continuously as long as the application window is open. Draws the window and the current state of its contents to the screen.
- `mousePressed()` : called automatically whenever the mouse button is pressed.
- `mouseReleased()`: called automatically whenever the mouse button stops being pressed.
- `keyPressed()` : called automatically whenever a keyboard key is pressed.

The code that YOU write won't ever call these methods; you're just going to set them up so that Processing can use them while your program is running.

2.2 Amogus class overview

The Amogus class is the data type for the Amogus sprites that you'll create and use in your application. Make sure to read the descriptions of the methods – not just the summaries at the top of the page – to understand how these methods work.

You will ***not*** be implementing any of these methods. They are provided for you in their entirety in the jar file you've already downloaded and added to your code. All you need to do is use them!

3. Adding to the Space Station display window

In this next section, you'll begin filling out some of those callback methods in the SpaceStation class.

3.1 Define the `setup()` and `draw()` callback methods

When you created your blank window, we noted an error related to the lack of a `setup()` method. Let's take care of that error next.

1. Create a **public static** method in SpaceStation named `setup`, with **no parameters** and **no return value**.

2. Next, run your program. The error message should now read: `ERROR: Could not find method named draw that can take arguments [] in class SpaceStation.`

3. Solve that error by adding another **public static** method to SpaceStation named `draw` with **no parameters** and **no return value**. Run the program again: no more errors!

4. **Note**: `Utility.runApplication()` calls `setup()` once and then keeps calling `draw()` repeatedly until the application ends – so when those methods don't exist, you get errors in your console, even though these two methods are never called within your code.

    a. Add a print statement (`System.out.println()`) with some test output to the `setup()` method. How many times does that get printed when you run the program?
    b. Now add a print statement to `draw()`. How many times does THAT get printed?
    c. **Delete** the print statements from 4a and 4b now, we don't need them and they'll be annoying if you leave them in.

Logically, we'll be organizing our code so that the `setup()` method contains only code initializing variables we'll need for the program, and the `draw()` method contains only code affecting what's being displayed in the application window.

3.2 Set the background color

Let's make the background color of your application window a different randomly-generated color every time you run the program.

1. Add a **private static** field to your SpaceStation class: an int variable called *`bgColor`*. This variable must be declared OUTSIDE of any method but still INSIDE the SpaceStation class. The top of the class is a good place to put it.

2. In `setup()`, use the provided `Utility.randGen` variable (of type java.util.Random) to generate a random integer value with no enforced bounds, and store the value in `bgColor`. This way we're only generating ONE random color every time we run the program.

3. Move to the `SpaceStation.draw()` method, and call `Utility.background()` with your `bgColor` static variable as the single argument.

4. Now, every time you run the program, the background will be a different color! Try running it a few times and see what kinds of colors you get.

We're practicing good code organization: the call to `Utility.background()` affects the contents of the window, so it belongs in the `SpaceStation.draw()` method. The `setup()` method only initializes variables like `bgColor` and shouldn't affect the display window at all.

3.3 Draw one amogus to the middle of the screen

Now let's start adding some objects to our application.

1. Add the following import statements to your SpaceStation class:

   a. `import java.io.File;`
   b. `import processing.core.PImage;`

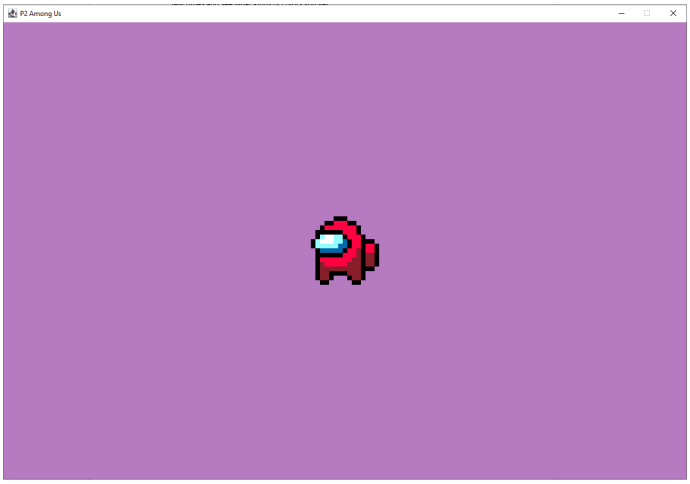
2. Create a **private static** `PImage` field named `sprite` in your SpaceStation class. For now, we're going to deal directly with the image, rather than using the Amogus class.

3. Since we're initializing variables in setup, Initialize your `sprite` field there by calling
   `Utility.loadImage("images"+File.separator+"sprite1.png");`
   and storing the result in `sprite`.

4. To draw this image to the screen, go into `SpaceStation.draw()` and add a call to the `Utility.image()` method, which draws a PImage object at a given (x,y) position on the screen. (The top left corner of the window is considered (0,0) and the bottom right is (1200,800).) To drop the image at the center of the screen:
   `Utility.image(sprite, 600, 400);`

5. Notice the importance of adding this line AFTER you call `Utility.background()` – if you call it *before* calling background, the background color will simply get drawn over your image and you won't be able to see it.

If you run your program now, it should look something like the screenshot below, which shows our single red sprite centered on a randomly generated light purple background.



3.4 Set up the real background

While the single random color is fun, we can really set the atmosphere for a game by using an appropriate background image.

1. Create another **private static** `PImage` field named *`background`* in your SpaceStation class.
2. Initialize it by using `Utility.`*`loadImage`*`()` on the "background.jp**e**g" file (yes, with the **e**) from the images directory. In ***which method*** should this command be placed?
3. Display the background by replacing the call to `Utility.`*`background`*`()` with another call to `Utility.`*`image`*`()` as in section 3.3, this time with the background image as an argument and the (x,y) coordinates (600,**500**). (This is slightly off-center for a reason.)
4. Delete or comment out the code declaring or using the *`bgColor`* field; we will not be using it.

Run the program again, and…

3.5 Cleanup

Before you continue, **delete** or comment out the code declaring or using the *sprite* field. It was just there to be educational.

4. Adding more friends to the space station

Just one Amogus stuck in the center of the screen is boring. Let's spice things up.

4.1 Using the Amogus class

Aside from the background, all other image loading and management is actually done behind the scenes in the Amogus class. One less thing for you to worry about!

1. Where your *sprite* field was, add a **private static** array of Amogus objects called *players*. This will be a *perfect-size* array of Amoguses (that is, we're not going to maintain a size for it - this is NOT a guarantee that every space will always be filled!)

2. Define a **private static** constant integer field in your SpaceStation class called ***NUM_PLAYERS***, and set it to 8. As this is a constant, we must initialize it immediately, rather than in `setup()`.

3. In the `SpaceStation.setup()` method, initialize the *players* array to a new array with a capacity of ***NUM_PLAYERS***.

4. Where before you'd drawn your *sprite* image to the application window, now you'll be using the power of the Amogus class:

    a. Add a single reference to an Amogus object in the first index of the *players* array in the `SpaceStation.setup()` method (we'll add more later, in another method). You can use the single-argument Amogus constructor here, to generate an Amogus at the center of the screen in one of the three possible colors.

    To properly initialize your Amogus, you'll need to generate one integer value randomly - a value in {1,2,3}. Be aware that `Utility.randGen.nextInt(bound)` generates an integer between `0` and *bound*`-1`; how might you move that to be between `1` and *bound*?

    b. In the `SpaceStation.draw()` method, rather than calling any Utility methods, add a call to the draw() method of the Amogus at the first index of *players*. (Be sure to do this AFTER you draw the background image, or you'll just draw that on top of your poor Amogus.)

At this point, if you run your code, the result should be mostly unchanged from the screenshot in 3.4 - though your Amogus might be a different color now.

4.2 Define the keyPressed() callback method

That *players* array is looking very empty right now. Let's populate it.

1. Add a **public static** method named `keyPressed`, with **no parameters** and **no return value**, to your SpaceStation class. This method is called automatically by Processing if a key is pressed on the keyboard.

2. When a key is pressed on the keyboard, the Utility class will tell you what char it is if you call the `Utility.key()` method. We're only going to handle the case where the user typed `'a'` in this application.

3. When the user presses the `'a'` key, we want to add a new Amogus to the players array if there's space available. However, we don't want to keep adding them to the center of the screen, we want to add them to random locations! And in random colors!

    a. To create a random location, generate a random integer between 0 and the width of the screen for its x coordinate, and 0 and the height of the screen for its y coordinate. (Note that these coordinates should be generated SEPARATELY. Also, be sure to use `Utility`'s width and height, not the background image's (which is rather larger than the window).)

    b. To create a random color, use the same approach that you used in step 4a in the previous section.

    c. Use the four-argument constructor to make a new Amogus at the next available index in the players array. For now, just set the last argument to false.

4. Update your `SpaceStation.draw()` method to draw ALL non-null elements of the *players* array!

If you run your code now, you should be able to press the `'a'` key 7 times and have a new Amogus appear at a random location in the window. If you press it any more than that, *nothing should happen*.

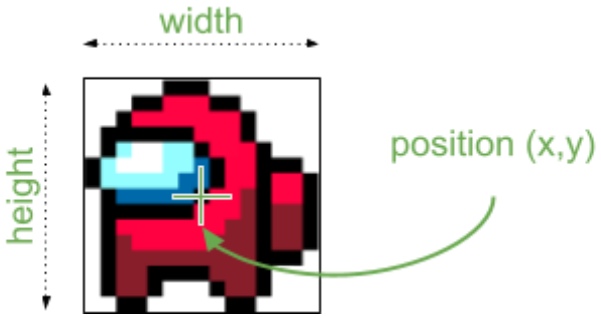
4.3 Where's the mouse?

We'll be making our Amoguses draggable, rather than keyboard-controllable. Before we can do that, though, we need to figure out whether the cursor is over a sprite!

1. Every Amogus has two methods to tell you where it is on the screen: `getX()` and `getY()`. These are *object* methods, which means calling them on different Amoguses will get you different results (unless those two Amoguses are in exactly the same location).

2. Every PImage has two attributes (public fields), `.width` and `.height`, to give you the dimensions of the image. Every Amogus will give you a reference to its image if you call the `image()` object method on it.

3. Note that the reported (x,y) position of the Amogus object corresponds to the **center** of the image within the display window:



4. The Utility class provides two methods to tell you where the mouse is relative to the application window at any given time: `Utility.mouseX()` and `Utility.mouseY()`.

    a. **IMPORTANT**: the (x,y) coordinates of the application window probably don't align to your expectations for how (x,y) coordinates work. Try adding a print statement to your draw() method to display `Utility.mouseX()` and `Utility.mouseY()`, and watch how they change as you move your mouse around the screen. Where is (0,0)?

5. Create a **public static** method named `isMouseOver` that expects a single Amogus parameter and returns a boolean value. Use the methods and attributes defined above to implement this method so that it returns true if and only if the mouse is currently hovering over *any* part of one of the Amogus images (**not** including its edges!)

6. To test your implementation, add a loop to your `draw()` method to check whether the mouse is over any of the non-null Amogus objects in your *`players`* array, and print the message "Mouse is over amogus" when the method returns true. For extra debugging help, you might want to print out the index of the Amogus the mouse is currently over, to verify that it's the one you expect! You should *only* see this message appear in the console when you are hovering your mouse over an Amogus.

You can keep that last bit of test output or not, as you like. You'll need the loop going forward, though.

4.4 Define the `mousePressed()` and `mouseReleased()` callback methods

Now that you know when the mouse is over an Amogus, let's do something with that information.

1. Add two new callback methods to your SpaceStation class: a **public static** method named `mousePressed`, with **no parameters** and **no return value**, and a **public static** method named `mouseReleased`, with **no parameters** and **no return value**.

2. In `mousePressed()`, add a loop to check whether the mouse is over any of the non-null objects in your *`players`* array. (Use that method you wrote in 4.3!) For *only* the **lowest-index** Amogus that the mouse is over, call the `startDragging()` object method.

3. In `mouseReleased()`, add a loop to call the `stopDragging()` object method on all non-null objects in the *`players`* array.

If you run your program now, you should be able to drag an Amogus all around the window! The exact mechanism behind this is handled in the Amogus class; you'll implement it yourself in a later assignment.

5. There is an impostor among us

You didn't think we were going to do an Among Us program without impostors, did you?

5.1 Select an impostor

We'll select an impostor randomly from the players added by pressing the `'a'` key (so never our very first guy).

1. Add a **private static** integer field to your SpaceStation class, called *`impostorIndex`*. Initialize it in the `setup()` method to a random value between 1 and the maximum number of Amoguses your *`players`* array can store.

2. Add a print statement to `setup()` to print `"Impostor index: "` and the generated value to the console, so you can verify that it's working properly.

3. When you add an Amogus to that index of the *`players`* array in your `keyPressed()` method, that Amogus should be designated an impostor. You do this by setting the final argument of the four-argument Amogus constructor to true.

Run your program and verify that the impostor is added at the correct position - an impostor Amogus will face *the other direction*!

5.2 Okay but what do they *do*

In the game Among Us, the impostor goes around trying to sneakily kill all of the other players and pretend they're completely innocent. Our impostor isn't sneaky, but they do get to go on a killing spree!

Here's the idea: if you drag the impostor over another Amogus (or another Amogus over the impostor), that other Amogus should… not be alive anymore, to put it delicately. The challenge here will be to detect WHEN two Amoguses' images are over each other!

You've already written code to detect when a POINT (the current cursor location) is over an image. Reference that method as you go!

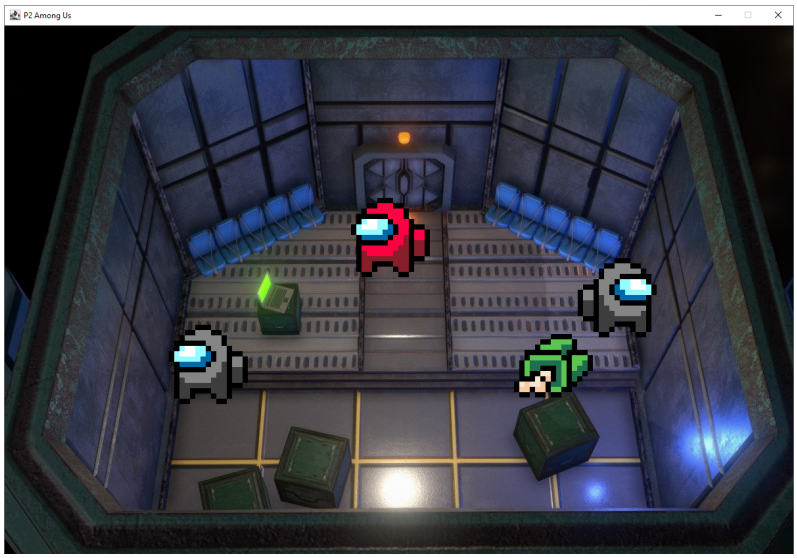
1. Create a **public static** method named `overlap` that expects two Amogus references in any order and returns a boolean value.

2. Implement a logical test within this method to determine if the first Amogus overlaps in any way with the second Amogus. It does not matter which one has a lower index in the array.

3. Feel free to reference this GeeksForGeeks article if you need help with the logic! Note that the article has (0,0) centered at the bottom right vs. our graphical application which has (0,0) centered at the top left. Be sure to adjust the visual diagram accordingly when calculating the points.

4. In the `draw()` method, before you draw an Amogus to the window, add a loop to verify whether that Amogus is currently overlapping any of the other non-null Amoguses. If it is AND the overlapping Amogus is an impostor (see the `isImpostor()` method)... call the Amogus method `unalive()` on the other Amogus 😈

And now, when you drag your impostor friend around… 👀



6. Assignment Submission

Hooray, you've finished this CS 300 programming assignment!

Once you're satisfied with your work, both in terms of adherence to this specification and the academic conduct and style guide requirements, submit your source code through Gradescope.

For full credit, please submit ONLY the following file (source code, *not* .class files):

- SpaceStation.java

Your score for this assignment will be based on the submission marked "**active**" prior to the deadline. You may select which submission to mark active at any time, but by default this will be your most recent submission.

Students whose final submission is made before 5pm on the Wednesday before the due date will receive an additional 5% bonus toward this assignment. Submissions made after this time are NOT eligible for this bonus, but you may continue to make submissions until 10:00PM CDT on the due date with no penalty.

Copyright notice

7. Appendix: The Utility class

Listed here are the Processing variables and methods available to you on this programming assignment in Gradescope. While you are encouraged to explore the Processing library on your own as we continue with the semester, be aware that Processing is huge and we will only be able to make a small portion of its functionality available on Gradescope for any given assignment.

In P02, you MUST interface with Processing by way of the Utility class.

Gradescope's `Utility` class includes the following variables:

- *`randGen`* – a Random variable for generating random values

Gradescope's `Utility` class includes the following **static** methods:

- `void runApplication()` – begins the GUI program's execution
- `void background(int)` – sets the background color of the window
- `PImage loadImage(String)` – creates and returns a PImage representation of an image file at the provided relative path location (e.g. `"images/sprite2.png"`)
- `void image(PImage, float, float)` – draws an image to the application window, centered at the given (x,y) position
- `int mouseX()` – returns the current x coordinate of the cursor in the application window
- `int mouseY()` – returns the current y coordinate of the cursor in the application window
- `int width()` – returns the width of the application window in pixels
- `int height()` – returns the height of the application window in pixels
- `char key()` – returns the char representation of the key being pressed on the keyboard

There are several other methods included in the p02core.jar distribution of the Utility class, but they are not relevant for this assignment and will not be included in Gradescope's version of the class.