P02 Party Hopping

Overview

Everyone needs a little variety in their lives. In this program, you'll be creating a visual management system for moving people (represented as 😊 in the program) between three different destinations:



Our Agents (😊) will enter the program in an "idle" state, bobbing around in the middle of the screen. You'll then be able to click on them (😳) and send them to one of the three destinations with a key press!

This is a **very long** writeup, but it is intended to be followed as a walkthrough. You will add and remove code over the course of creating this program – pay close attention to how each of these operations modifies your code! We'll be revisiting GUIs in P05, and having a clear understanding of how these components contribute to the program (whether we ultimately keep them or not in P02) will help you out a LOT later.

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 10:00 PM on **1/31**.
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.
50 points	**MAXIMUM TOTAL SCORE**

Table of Contents

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
- **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.

- NOTE: 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. This program can be completed successfully using ONLY these methods.

- 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** 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.

- All other sources must be cited explicitly in your program comments, in accordance with the Appropriate Academic Conduct guidelines.

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

- Any use of ChatGPT or other large language models ***must be cited*** AND your submission MUST include **screenshots** of your interactions with the tool ***clearly showing all prompts and responses in full***. Failure to cite or include your logs is considered academic misconduct and will be handled accordingly.

Need More Help?

Check out the resources available to CS 300 students here:
https://canvas.wisc.edu/courses/447785/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:

- Appropriate Academic Conduct, which addresses such questions as:
  - How much can you talk to your classmates?
  - How much can you look up on the internet?
  - How do I cite my sources?
  - 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

1. Create a new project in Eclipse, called something like **P02 Party Hopping**.
   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(s) within that project's src folder:
   a. **PartyHopping.java** (includes a main method)

All methods in this GUI program will be **static** methods, as this program focuses on procedural programming; in future GUI programs we'll be using the graphics library more like it is intended to be used (that is, in an Object Oriented way!).

Note that the following instructions are Eclipse-specific, but IntelliJ and other IDE users should be able to extrapolate.

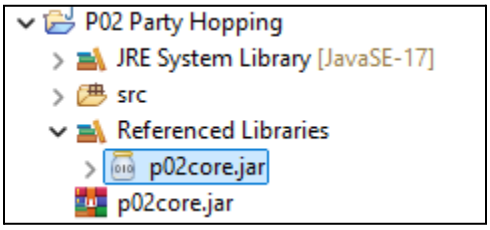
1.1 Download the Processing jar file

Download the `p02core.jar` file from Canvas[1], which contains:

1. the core Processing library,
2. a custom Utility class to make Processing easier for you to use right now,
3. and two custom object classes 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 Project Explorer showing a project called
P02 Party Hopping 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 located in your project folder by clicking "Add JARs…" from the right side menu
5. Click on the "Apply" button

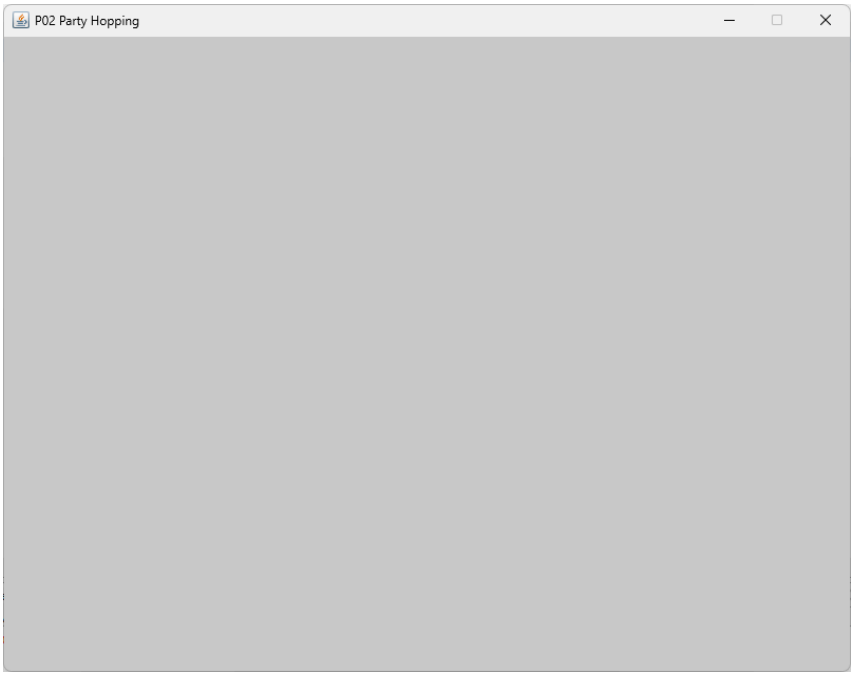
1.2 Check your setup

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

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

If everything is working properly, you should see a blank window with the text "P02 Party Hopping" in the top bar as shown below, and an error message in the console that we'll resolve shortly:

[1] **For Mac users with Chrome**: this download may be blocked. If you're opposed to switching to Firefox (omg *please* switch) go to "chrome://downloads/" and click on "Show in folder" to open the folder where the jar file is located.



A screenshot of a blank P02 Party Hopping window.

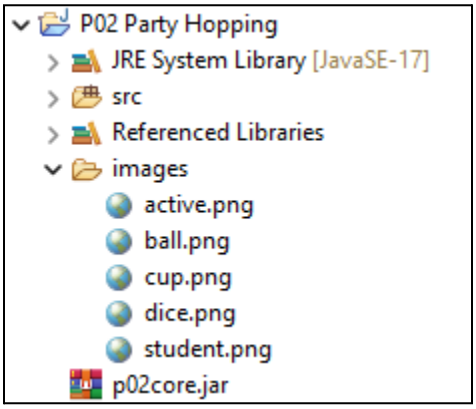
```
ERROR: Could not find method named setup that can take arguments [] in class PartyHopping.
```

If you have any questions or difficulties with this setup, 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 fix that now.

1.3 Download the images

Finally, download the `images.zip` file ***and unzip it***; it contains the three (3) location images shown on the first page as well as two images of 😊 and 😳 for the little guys we're going to move around. (Make SURE you **unzip** the file before continuing; if you try to use the zip file directly, it won't work.)

Drop this folder into your project folder in Eclipse:



2. Utility framework and overview of Agent 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 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 Agent class javadocs. (Inquisitive students may notice there's another provided class in the JAR file – we'll get to that one later on.)

2.1 GUI programming overview

A graphical user interface (GUI) program works slightly differently from the text-based programs you're probably used to: Processing's PApplet class maintains a program that runs in the background, constantly checking for things called "events". These events, which are things like "a key on the keyboard is pressed" or "a mouse button was clicked", will cause PApplet to talk to your program.

This conversation relies on what we refer to as **callback** methods. You probably won't ever call them from your own code; they're just there to communicate with the Processing libraries.

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

- `setup()` : called *once*, when the program begins. All data field initialization should happen here, any program configuration actions, etc. It should *set up* everything your program needs to run.
- `draw()` : called in a continuous loop, as long as the program is running. Repeatedly draws the application window and the current state of its contents to the screen.
- `mousePressed()` : called automatically whenever the (left) mouse button is pressed.
- `keyPressed()` : called automatically whenever a keyboard key is pressed.

Again: the code that you write for this program 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 Agent class overview

The Agent class objects will be represented as 😊 in your application window; you'll be creating them and calling their *instance methods* to make them do things on the screen. Read the full descriptions of the methods – not just the summaries at the top of the page – to understand how they 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 Party Hopping display window

In this next section, you'll begin filling out the callback methods in the PartyHopping 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 PartyHopping named `setup`, with **no parameters** and **no return value**. You can leave it empty for now.

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

3. Solve that error by adding another **public static** method to PartyHopping named `draw` with **no parameters** and **no return value**. You can leave it empty for now, too.

4. `Utility.runApplication()` causes both `setup()` and `draw()` to be called – 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** or comment out the print statements from 4a and 4b now, we don't need them (and they'll probably get annoying if you leave them in).

Logically, we want to organize our code so that `setup()` contains *only* code initializing variables we need for the program, and `draw()` contains *only* code affecting the application window display.

3.2 Set the background color

To begin, let's make the background color of your application window something other than grey.

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

2. In `setup()`, initialize *`bgColor`* to an integer representing a nice desaturated blue. The RGB combination (81, 125, 168) makes a good color; to translate this into a single integer, use the `Utility.color(r,g,b)` method with those values.

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

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

3.3 Draw some exciting destinations

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

1. Add the following import statements to your PartyHopping class:
    a. `import java.io.File;`
    b. `import processing.core.PImage;`

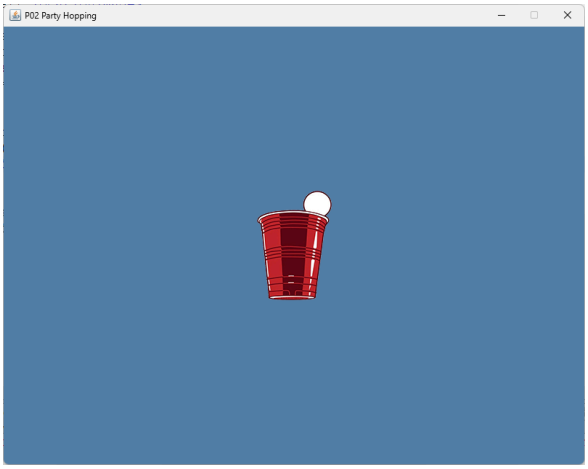
2. Create a **private static** `PImage` array named `locationImages` to your PartyHopping class.

3. Since we're initializing variables in `setup()`, initialize your `locationImages` field there by initializing it to an array of length 3, then calling
    `Utility.loadImage("images" + File.separator + "cup.png");`
    and storing the result in `locationImages[0]`. This call creates a PImage object that contains the picture in the file located at the specified path (in this case, our red Solo cup).

4. To draw this image to the screen, add a call to the `Utility.image()` method in `PartyHopping.draw()`. This method draws a PImage object at a given (x,y) position on the screen. (See the Utility class documentation for more information, but notably, the top-left corner is (0,0) and the bottom right is (800,600).) To drop the image at the center of the screen:
    `Utility.image(locationImages[0], 400, 300);`

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

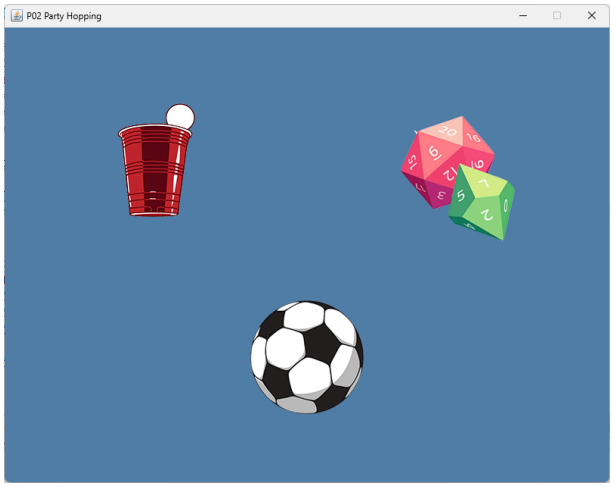
If you run your program now, it should look something like this:



Now load ALL of those images from page 1 into the correct indexes of `locationImages` and draw them in the following locations on the screen:

0. `cup.png`, location (200, 175) (Note: this is DIFFERENT from what I just told you!)
1. `dice.png`, location (600, 200)
2. `ball.png`, location (400, 435)

✅ CHECKPOINT 1: STATIC IMAGES
When you run the program now, you should see:



4. Adding Agents

So far, all you've done is draw a few static (as in, not-moving) images to a game window. Nothing moves and the only objects you've really used are PImages – and even then, you haven't actually called methods ON them, you've just passed them to Utility methods as arguments.

The point of this program is to get some practice with USING OBJECTS, and that's what we're going to do now!

4.1 Create an array of Agents

Your program will, eventually, have the ability to manage many agents! Let's start small, though.

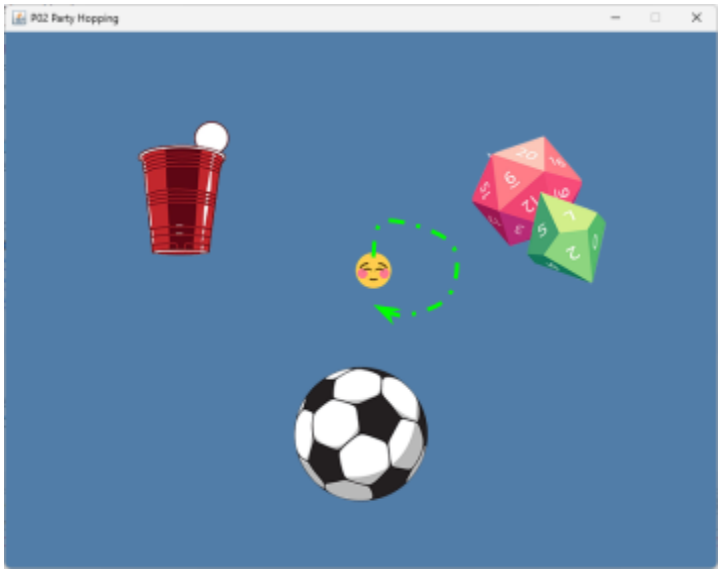
1. Add a **private static** Agent array field named `agents` to your growing collection of fields (there should be three now). This will be a *perfect-size* array (that is, we're not going to maintain a size for it – if an element is null, it's empty; if it's not null, it must be a valid Agent).

2. In the `setup()` method, initialize the `agents` field to a new array of Agent objects with a capacity of 15 (you may hard-code this value). Add a single reference to an Agent object to the first index (you'll add more Agents later). You'll need to load in the image for a student as the argument to this constructor – this is the `student.png` image.

3. In the `draw()` method, ***after*** you've drawn the background and all three static images, add a call to your new Agent's `draw()` method – not a recursive call to the method you're in (which has the same name!), but a call to the *instance* method `draw()` from the Agent class.

✅ CHECKPOINT 2: ONE AGENT

When you run the program, there is one agent bobbing around in the middle of the screen:



The Agent's idle animation (circling around a point) is handled entirely by its `draw()` method. Every time you call the method, it updates its location a little and changes the exact position where it's being drawn.

Because the Processing library repeatedly calls `draw()` from your PartyHopping class, this results in a (literal!) loop!

4.2 Wake the Agent up

To make this application work, we need the user to be able to communicate with specific Agents on the screen. The easiest way to do that is by clicking on the one you want to talk to.

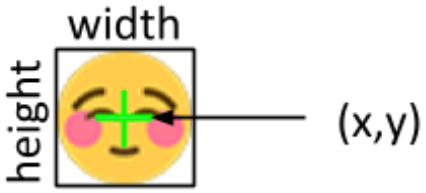
4.2.1 Where is the mouse?

Step one: figure out whether the mouse is over an Agent.

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

2. Every Agent also has two methods to tell you the dimensions of its image: `width()` and `height()`. These will report the number of pixels across and from top to bottom the Agent's image is, respectively.

3. Note that the (x,y) position of every 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 Agent parameter and returns a boolean value. Use the methods and attributes defined above to implement this method so that this method returns true if and only if the mouse is currently hovering over *any* part of one of the Agent images, including the blank space in the corners.

    You can include the exact edge of the image in this calculation or not; we won't be testing that.

6. In your `draw()` method, before drawing your one Agent, add a call to the `isMouseOver()` method and print "YES" if the method returns true, and "NO" if it returns false.

✅ CHECKPOINT 3: MOUSEOVER
The program correctly prints YES or NO depending on whether your mouse is currently over an Agent.

4.2.2 Agent Activated

Now that you can tell whether the mouse is OVER an Agent, let's add another callback method to handle the event that you're actually CLICKING on that Agent.

1. Create a **public static** method named `mousePressed` with **no parameters** and **no return value**.

2. In `mousePressed()`, add a loop to check – ONCE – whether the mouse is over any of the non-null objects in your `agents` array. For *only* the **highest-index** Agent that the mouse is over, call the `activate()` instance method from the Agent class.

    This causes an internal boolean in that Agent object, `isActive`, to be flipped to **true**. You can access the current value of this boolean using the Agent's `isActive()` method!

If you test your code now, you should see the YES/NO output while you're hovering over the Agent, but if you CLICK on the Agent… **the program crashes**.

```
java.lang.RuntimeException: Encountered trouble running
PartyHopping.draw with arguments: [].
```

Counter to the stated reason, this is actually because the *Agent's* `draw()` method is failing! It's trying to draw an active Agent to the screen using a different image that we haven't actually provided yet, and calling `Utility.image()` with a null PImage value is bad news.

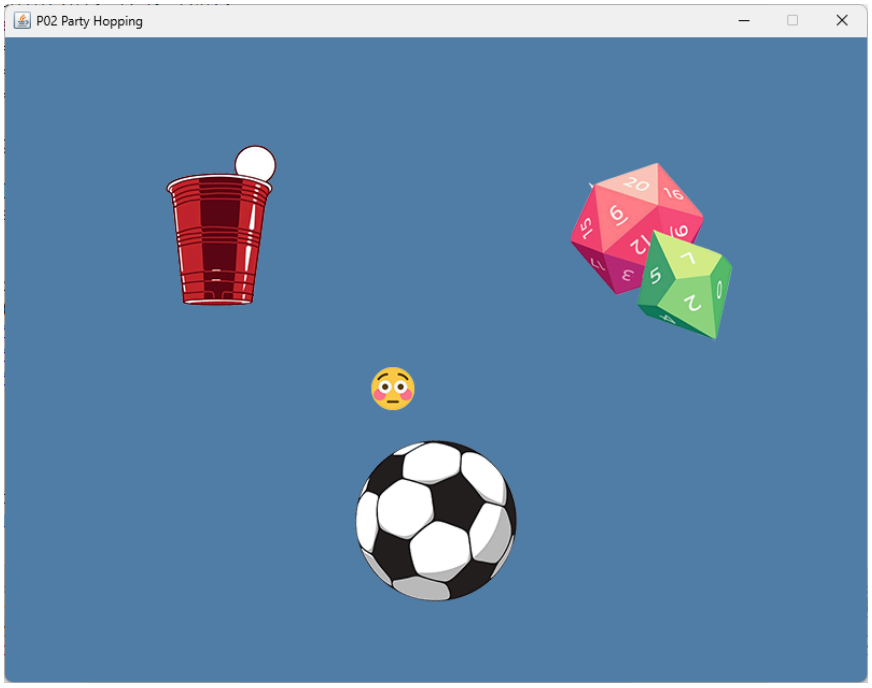
4.2.3 Active Agent image

For this program, all Agents will share a single PImage that they'll use for their "active" state. (We could have done this for the normal image too, but… *design decision*.)

1. Back in `setup()`, load the image from `active.png` and pass the resulting PImage as an argument to the ***class*** method `setActiveImage()` from the Agent class.

You can get rid of the loop causing the YES/NO output in `draw()` now, since we'll only care about the mouse location when it's actually being clicked.

✅ CHECKPOINT 4: ACTIVATE
Clicking on an Agent causes it to wake up!



4.3 Send the Agent on a mission

Active agents need places to go and things to do. We COULD have the user click on a destination for the Agent, but let's experiment with another form of GUI interaction: keyboard input.

This requires another callback method for PApplet to call:

1. Create a **public static** method named `keyPressed` with a single char parameter and **no return value**. The value of the char parameter will be the character corresponding to the key on the keyboard that was pressed, provided by PApplet whenever it detects the event of someone hitting a key on the keyboard.

    a. You can **test** this function now by adding a print statement for that character parameter and running the program. If I type "asdf" (slowly), the characters `a s d f` show up in the console!

2. If the user types an `'a'`, *and* there is an active Agent in the `agents` array, use the Agent instance method `setDestination()` with the class constant `Agent.A` to set the ***highest-index*** active Agent's destination to the red solo cup. (Note that you'll be able to activate multiple Agents at the same time, once we have multiple Agents.)

    Incidentally, the `setDestination()` method in Agent also causes its `isActive` boolean to go back to false, so if you want to give the Agent another destination, you'll need to click on it again.

3. There are also class constants in Agent corresponding to the dice and soccer ball; these are `Agent.B` and `Agent.C` respectively. These should be used if the user types `'b'` or `'c'`.

4. If the user types an `'i'`, you can send the Agent back to its idle spot in the middle of the screen using the destination code -1. You can ignore all other input for now.

✅ CHECKPOINT 5: MISSIONS
You can activate an Agent and send it to hover on all three static images,
and also back to the middle of the screen.

4.4 *FRIENDS!*

Our one Agent is very lonely. Let's populate the rest of that array.

4.4.1 Add an Agent

1. In `keyPressed()`, if the user types a `'.'` , check the `agents` array for a null element and add a new Agent with the same `student.png` image as the original *at the lowest index possible*. Note that this adds only ONE new Agent at a time.

2. If there is no room to add a new Agent, simply do nothing.

3. In `draw()`, update the method to draw ***every*** *non-null element* of the `agents` array, not just the one at index 0. (Draw them in standard, increasing order.)

Verify that pressing `'.'` adds ONLY ONE Agent, that BOTH Agents are drawn to the screen, and that you can click on one Agent and send them to a new destination without affecting the other one.

✅ **CHECKPOINT 6: WE ARE ALL INDIVIDUALS**
You can activate ONE of multiple Agents and send them around the screen without the other Agents being affected.

4.4.2 Remove an Agent

1. In `keyPressed()`, if a user types an `'x'` *and an Agent is active*, remove the highest-index active Agent by setting its index to null in the `agents` array.

2. If no Agents are active, simply do nothing.

This may cause gaps between Agents in your array! Make sure that yourx `draw()` method can handle this by adding a LOT of Agents as in 4.4.1, and then activate and remove a few.

✅ **CHECKPOINT 7: GOODBYE**
Add the maximum number of Agents, activate at least three, and verify that pressing `'x'` removes *only* the TOP one.

5. Final touches: object-oriented design

You may have noticed that our locations aren't exactly under the center of where the Agents are floating (the soccer ball in particular). The expected centers of those images are hard-coded into the Agent class, and to be honest: I've adjusted where those images are a little bit since I first wrote that class. Aesthetics, you understand.

I could have gone into the Agent class and fixed the center coordinates, but then I'd have to change those values in multiple places whenever I decided to adjust them, and that's not only bad practice with respect to code maintenance, it's just plain annoying.

There is a better way!!

5.1 The Party class

Rather than just drawing static images to predefined locations in the window, we've provided the custom Party class to contain the image information, an identifier for that location, and where its image is on the screen.

This isn't a particularly complicated object, but it's an example of the object-oriented principle of *encapsulation*: keeping related information together.

1. Add one more **private static** array to your group of class variables in PartyHopping, this time an array of Party objects called `locations`.

2. Initialize it in `setup()` to have length 3, and to contain three new Party objects that correspond to the static images we created back in section 3.3. For example, the red Solo cup would be created as follows:

   ```
   locations[0] = new Party('a', 200, 175, locationImages[0]);
   ```

   The first argument corresponds to the **key** you press to send an Agent to that location, then the (x,y) **coordinates** of the image, and then the **PImage** object for that location.

5.2 Refactoring

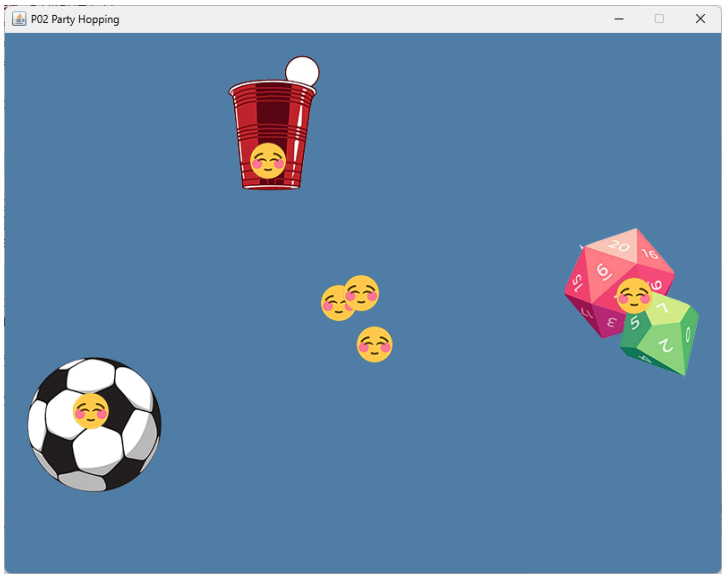
Unfortunately we now have a little more work before this is fully incorporated into our program, but it will be worth it.

1. In `draw()`, remove the calls to `Utility.image()` with the static images and instead loop through the `locations` array and call the Party instance method `draw()` on each object. No messing with images or locations or anything, just… `draw()`. Nice, huh?

2. In `keyPressed()`, remove the parts where you're detecting a hard-coded a, b, c, or i and then setting the active Agent's destinati..on using a class constant.

   Instead, check to see if the provided key argument matches any of the ID values for your `locations` (use the Party instance method `getID()` to check this) and if so, pass that entire Party object to the active Agent's `setDestination()` method instead! (**Note**: sending an Agent back to the idle spot at the center of the screen ***no longer works***. Don't worry about it.)

✅ FINAL CHECKPOINT: PARTY TIME
Verify that you can modify the locations of the Party objects (in the setup method)!



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, make a final submission of your source code to Gradescope.

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

- **PartyHopping.java**

Additionally, if you used generative AI at any point during your development, ***you must include screenshots*** showing your FULL interaction with the tool(s).

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.

Copyright notice

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 **static** methods:

- `void background(int)` – sets the background color of the window
- `int height()` – returns the height of the application window in pixels. This is different from the Agent's `height()` method.
- `int width()` – returns the width of the application window in pixels. This is different from the Agent's `width()` method.

- `void image(PImage, float, float)` – draws an image to the application window, centered at the given (x,y) position
- `char key()` – returns the char representation of the key being pressed on the keyboard
- `PImage loadImage(String)` – creates and returns a PImage representation of an image file at the provided relative path location (e.g. `"images/active.png"`)
- `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
- `void runApplication()` – begins the GUI program's execution