Pair Programming: **NOT ALLOWED**
Due: **9:59 PM CDT on WED 09/21**

P02 Walking Simulator

Overview

In this program, you will be developing a graphical implementation of a walking animation:

By cycling through the frames of this "walk cycle", you'll give the figure the appearance of walking in a window on your screen. You'll create a few figures that will walk (or not) independently, and you'll eventually make your program clickable and sensitive to key presses, as a (relatively gentle) introduction to working with a graphical user interface.

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 later, in P05.

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on **09/18**.
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
- **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.util.Random
  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 two methods from the Processing library.

- 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 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*.

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

The first few steps are similar to P01:

1. Create a new project in Eclipse, called something like **P02 Walking Sim**.
   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 Java source file within that project's src folder:
   a. WalkingSim.java (contains a main method)

But now we're going to get a little weird. Note that the following instructions are specific to Eclipse; IntelliJ and other IDE users will have similar steps but it may take a bit of fumbling. Sorry!!

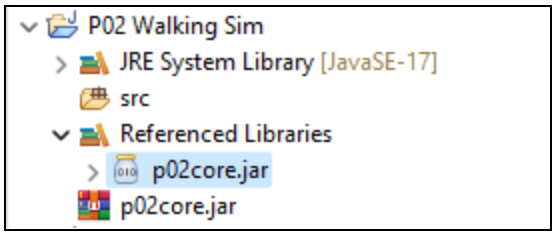
1.1 Download the Processing jar file

Download the p02core.jar file[1], which contains the core Processing library (build 4.0.1) and a custom object class we'll explore later.

Copy the jar file into your project folder, and refresh the Package Explorer panel in Eclipse. You should see the jar file there.

Right-click it and select "Build Path" and "Add to Build Path" from the menu; it should be added to your project as a Referenced Library (see the screenshot below).

[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 the Eclipse Project Explorer showing a project called P02 Walking Sim 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 P2Walker.jar file located in your project folder by clicking "Add JARs…" from the right side menu
5. Click on the "Apply" button

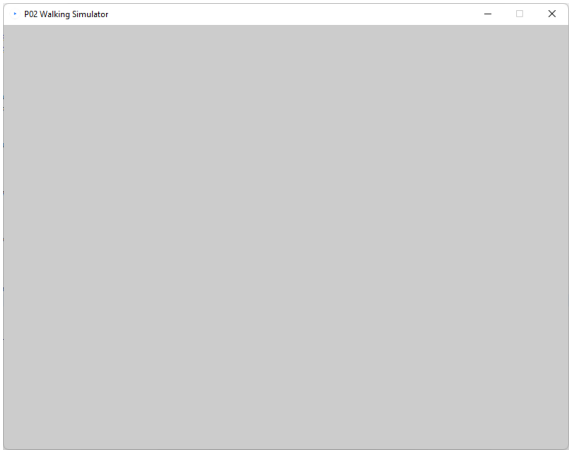
1.2 Check your setup

To test that the jar file was added correctly to your build path, find your main method in the WalkingSim 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 "P02 Walking Simulator" in the top bar as shown below, and an error message in the console that we'll resolve shortly:

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



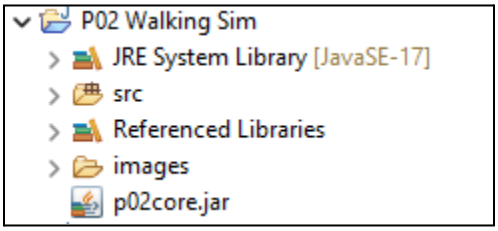
A screenshot of a blank P02 Walking Simulator window.

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 walk cycle images

Download the images.zip file and expand it; it contains 8 images of a figure walking as on the first page.

Add this 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 Walker 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, keep the javadocs for the jar file available – they contain both the Utility framework methods and the methods for the Walker class, which you'll use later.

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.
- `keyPressed()` : called automatically whenever a keyboard key is pressed.

Your code 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 Walker class overview

The Walker class is the data type for the walking figure object 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 Walking Simulator display window

In this next section, you'll begin filling out some of those callback methods in the WalkingSim 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 WalkingSim 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 WalkingSim.`
3. Solve that error by adding another **public static** method to WalkingSim named `draw` with **no parameters** and **no return value**.
4. Check out the Utility javadocs. Notice that `Utility.runApplication()` makes use of the two methods you just created – so that when they didn't exist, you got errors, even though these two methods were 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 output get printed when you run the program?
    b. Now add a print statement to draw(). How many times does THAT get printed?
    c. (Go ahead and delete those print statements now, we don't need them.)

3.2 Set the background color

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

1. Add two **private static** fields to your WalkingSim class: a Random variable called `randGen`, and an int variable called `bgColor`. These variables must be declared OUTSIDE of any method but still INSIDE the WalkingSim class. The top of the class is a good place to put them.

2. Next, initialize the `randGen` field to a new Random object within the `WalkingSim.setup()` method. You do not need to use a specific seed value at this time.

3. Use `randGen` 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.

4. Still within the `WalkingSim.setup()` method, call the `Utility.background()` method and pass your `bgColor` as an argument.

5. 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.

6. Before you continue, let's practice good code organization: the call to `Utility.background()` affects the contents of the window, so it should be in the `WalkingSim.draw()` method. Move it there now. The `setup()` method should now only initialize `randGen` and `bgColor`.

3.3 Draw one walker to the middle of the screen

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

1. Create a **private static** `PImage` field named `frame` to your WalkingSim class. For now, we're only going to load in the first frame of our walk cycle animation.

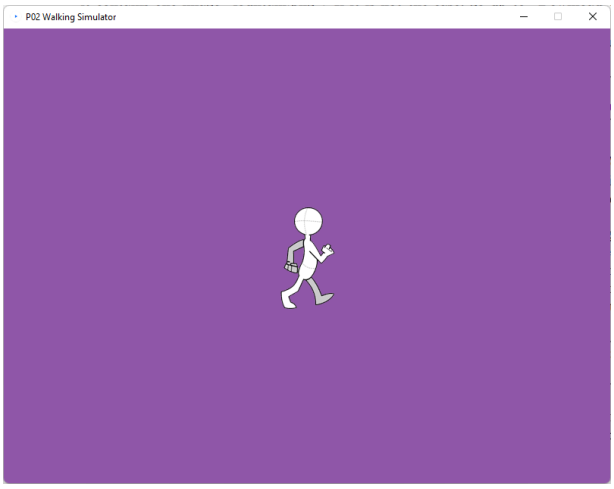
2. The documentation for Utility urges us to only ever load images in the setup method, so initialize your `frame` field there by calling
   ```
   Utility.loadImage("images" + File.separator + "walk-0.png");
   ```
   and storing the result in `frame`.

3. Make sure that you are importing **all three** of `java.util.Random`, `java.io.File`, and `processing.core.PImage` at the top of your WalkingSim class!

4. To draw this image to the screen, go into the `WalkingSim.draw()` method and add a call to the `Utility.image()` method, which 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(frame, 400, 300);
   ```

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:



A screenshot of a P02 Walking Simulator window with a purple background and a single walking figure in the center.

4. Animation

This is where it gets fun.

4.1 Load all frames of the walk cycle

You've only loaded a single animation frame so far, but we'll need the others to really make this work.

1. Replace your single PImage `frame` with a **private static** array of PImages called `frames`.
2. In the WalkingSim `setup()` method, initialize your array to the length specified by the static field `Walker.NUM_FRAMES` – this value is provided by the Walker class.
3. Initialize each element of this array in the setup method by adding its index number into the image name, as `"images"+File.separator+"walk-"+index+".png"`, and loading the image as before.
4. To test that you've done this correctly, try replacing the **first** argument of `Utility.image()` in your `WalkingSim.draw()` method with `frames[3]`. If all goes well, you should see a figure in a slightly different position than you did before!

Clean-up time: you're going to begin using the Walker objects exclusively now, so you should comment out or delete the `frame` variable if you haven't yet, as well as any calls to `Utility.image()`.

4.2 Create an array of Walkers

Your program should be able to handle multiple different, independent Walkers.

1. Create a **private static** Walker array field named `walkers` within your WalkingSim class. This will be a *perfect-size* array of Walkers (that is, we're not going to maintain a size for it).

2. Within the `WalkingSim.setup()` method, initialize the `walkers` field to a new Walker array with a capacity of 8, and add a single reference to a Walker object to the first index (you'll add more Walkers later). You can use the no-argument Walker constructor here.

3. In the `WalkingSim.draw()` method, update your call to `Utility.image()` to reference the PImage at the walker's current frame index in the frames array, as well as its current x and y coordinates (hint: check out the get methods in the Walker class - specifically `getPositionX()` and `getPositionY()`).

At this point, your window should still look mostly like the screenshot at the end of section 3.3, above.

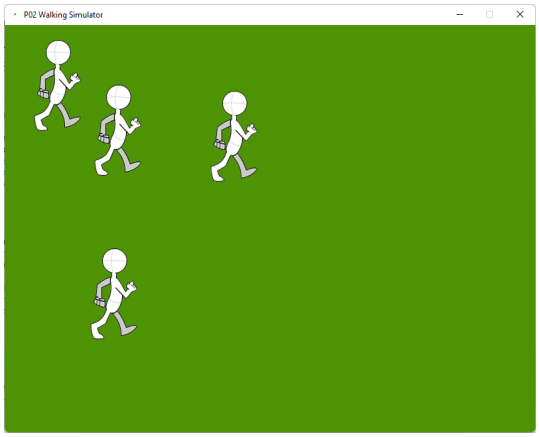
4.3 Populate your walkers array

Back to the random number generator!

1. In your `setup()` method, generate a random number between 1 and the length of the `walkers` array (inclusive). Instead of just adding one Walker, add this many to the array.

    a. Hint: use the two-argument Walker constructor with randomly-generated coordinates for the x and y values, between 0 and `Utility.width()` or `Utility.height()`.
    b. If you use the same coordinates (or the no-argument constructor), your Walkers will all end up on top of each other!

2. Your `draw()` method now needs to draw ALL of the non-null values from your `walkers` array every time it runs. Replace the call to `Utility.image()` for the single Walker with a loop that calls draw on each of the array's Walkers – and NOT on any indexes that don't have Walkers.



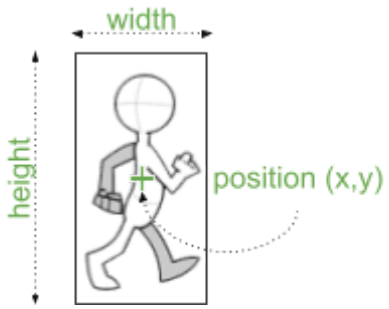
4.4 Where's the mouse?

One more step before we can get animating:

1. Recall every Walker has two methods to tell you where it is on the screen: `getPositionX()` and `getPositionY()`. These are *object* methods, which means calling them on different Walkers will get you different results (unless those two Walkers 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. All of our frames have the same width and height values.

3. Note that the position of the Walker 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: `mouseX()` and `mouseY()`.

5. Create a **public static** method named `isMouseOver` that expects a single Walker 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 Walker 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 Walker objects in your walkers array, and print the message "Mouse is over a walker!" when the method returns true. You should only see this message appear in the console when you are hovering your mouse over one of the figures.

You can comment out that last tester loop before you continue, once you are satisfied that your method works as you expect it to.

4.5 Get those walkers walking

Now you're going to define another *callback* method – this one will be called automatically whenever the user clicks the mouse.

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

2. Move your code that checks if the mouse is over any of the non-null Walkers to this method.

You're only going to care where the mouse is when it's actually being clicked.

3. If one of the Walkers IS being clicked on, use the method `setWalking(boolean)` from the Walker class to set that particular Walker's `isWalking` status to **true**. You should only change the isWalking status for the lowest-index Walker the mouse is over.

4. Back in the `WalkingSim.draw()` method, check whether each of your non-null walkers `isWalking` – and if it is, call its `update()` method. This advances its current frame index (slowly!) through the `frames` array.

5. Run your code. Try clicking on one of the figures; only that one figure should begin moving as though it is walking. If you click on another, it should start moving, too! But only in place.

6. If your walkers begin moving BEFORE they are clicked, check to make sure that you're only calling update() when a walker `isWalking`.

4.6 Traveling walkers

Time to make the walkers actually move across the window!

1. Back in your `WalkingSim.draw()` method, before you call `Utility.image()` to draw each of your non-null walkers, check to see whether that Walker is currently walking (hint: use the `isWalking()` method).

2. If that Walker IS walking, update its x-coordinate to be 3 pixels to the right – use the `getPositionX()` and `setPositionX()` methods.

3. To prevent everyone from walking off the right of the window forever, make sure to **wrap** the x-coordinate back around to zero when it goes off the edge of the screen (hint: use modulo!).

Play around with that 3-pixel value – what happens when you increase it? Make it negative? What if you were to move the y-coordinate instead? (Just make sure you put everything back to what we specified **before** submitting to Gradescope.)

5. Final touches: keyboard interaction

To finish the program, you'll add a little bit of key-pressing on the keyboard, just to get some experience with this Processing capability.

5.1 Adding more walkers

At the moment, you're stuck with the random number of walkers that your code generates to begin with. This will allow you to add some more, up to the capacity of your walkers array, using one last callback method.

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.

2. If the user types an `'a'` or an `'A'`, and there are any remaining null elements of the walkers array, add a new Walker object at a random position on the screen to the next available element in the `walkers` array. Like the others, it should not start walking until the user clicks on it.

3. If the user types an `'s'` or an `'S'`, ALL non-null Walkers in the program must STOP walking. You can use the `setWalking(boolean)` method here again, with an argument of false.

Try it out! At this time, the only testing we will have you do for this program is interacting with it yourself, and seeing if the behavior that you observe corresponds to what we've described here.

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):

- WalkingSim.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.

Copyright Notice

The walk cycle figures are sourced from Angry Animator and are the property of Dermot O Connor.

Additionally, students are not permitted to share source code for their CS 300 projects on any public site. We're only (barely) getting away with using these walk cycle figures because this is an educational application; if you wish to create your own version of this project, you should source your own walk cycle images.