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 **09/15**.
+5%	**Bonus Points**: students whose ***final*** submission to Gradescope is before **5:00 PM Central Time** on **WED 09/18** *and* who pass ALL immediate tests 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.
50 points	**MAXIMUM TOTAL SCORE**

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 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/427315/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!

Getting Started

1. Create a new project in Eclipse, called something like **P02 Walking Simulator**.
   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. **WalkingSim.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.

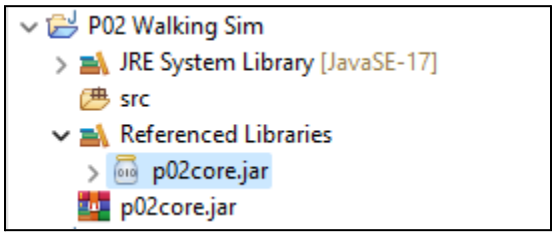
1.1 Download the Processing jar file

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

1. the core Processing library (this one's using build 4.0.1; we'll use the current 4.3 build later this semester),
2. a custom Utility class to make Processing easier for you to use right now,
3. 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 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 p02core.jar file located in your project folder by clicking "Add JARs…" from the right side menu
5. Click on the "Apply" button

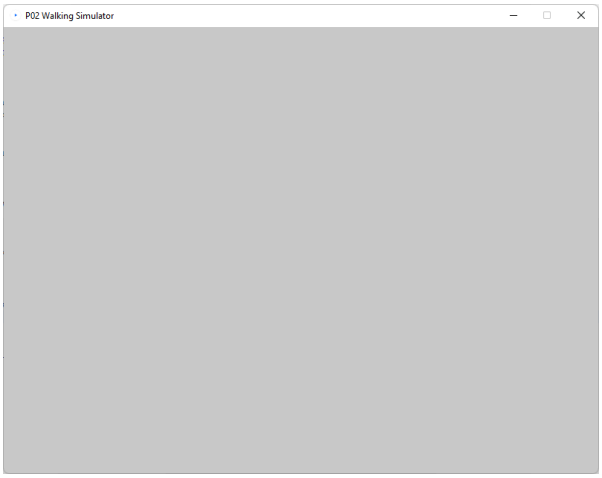
1.2 Check your setup

To test that the jar file was added correctly to your build path, 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:

[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 Walking Simulator window.

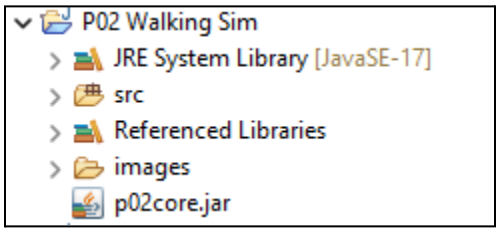
```
ERROR: Could not find method named setup that can take arguments [] in class WalkingSim.
```

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

Finally, download the images.zip file ***and unzip it***; it contains 8 images of a figure walking 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 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, 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 Walker 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.
- `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 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. Note that `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** 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'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

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. In `setup()`, initialize the *`randGen`* field to a new Random object. Then 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.

3. Move to the `WalkingSim.draw()` method, and call `Utility.background()` method 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 `WalkingSim.draw()` method. The `setup()` method only initializes variables like *`bgColor`* and shouldn't affect the display window at all.

3.3 Draw one walker to the middle of the screen

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

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

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

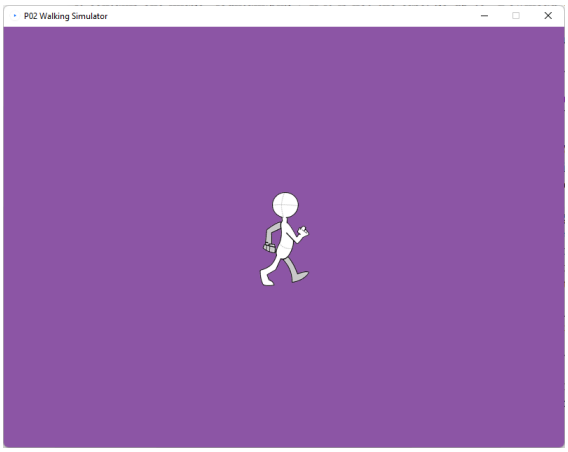
2. Create a **private static** `PImage` field named *`frame`* to your WalkingSim class. For now, we're only going to load in the first frame of our walk cycle animation.

3. Since we're initializing variables in `setup()`, initialize your *`frame`* field there by calling
    `Utility.loadImage("images" + File.separator + "walk-0.png");`
    and storing the result in *`frame`*.

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 `Utility.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 frozen walker centered on a randomly-generated purple background (your background will probably be a different color):



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

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. Where your *`frame`* was, add a **private static** array of PImages called *`frames`*. This will be a *perfect-size* array of images that we'll index into "circularly" – that is, the indexes will be accessed in the order 0, 1, … *`frames`*`.length` - 1, 0, 1, ...

2. In the `setup()` method, initialize the *`frames`* 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 the *`frames`* 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!

4.2 Create an array of Walkers

Let's set up the program for multiple independent Walkers.

1. Create a **private static** Walker array field named *`walkers`* right below your *`frames`* array. This will also be a *perfect-size* array (that is, we're not going to maintain a size for it).

2. In the `setup()` method, initialize the *`walkers`* field to a new array of Walker objects with a capacity of 8 (you may hard-code this value). 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 `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()`).

Verify that your window still looks mostly like the screenshot at the end of section 3.3, above. That's boring. Let's change that.

4.3 Populate your walkers array

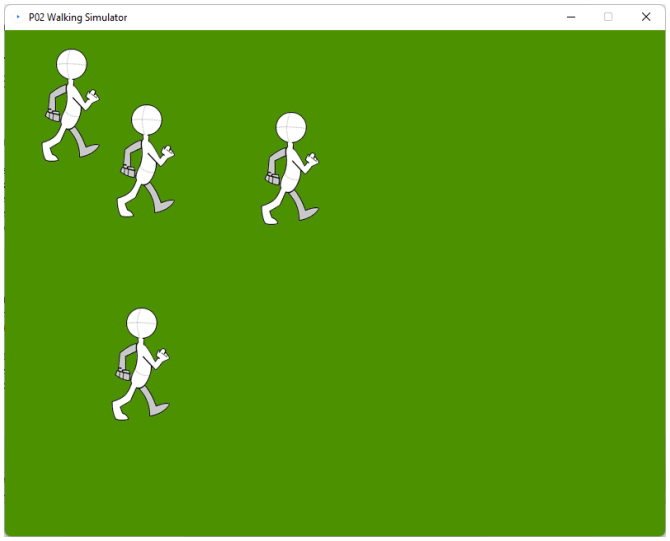
Back to the random number generator!

1. In your `setup()` method, generate a random number between 1 and the length of the *`walkers`* array (inclusive). Remove that single Walker from the first index, and add the randomly generated number of Walkers to the array instead. (Leave the rest of the values null.)

    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 for each call (or the ***no***-argument constructor), your Walkers will all end up on top of each other! Make sure to generate new positions for each Walker object you add.

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.

Now you should see something a little more interesting, like this randomly-generated green background with FOUR walkers scattered around the window:



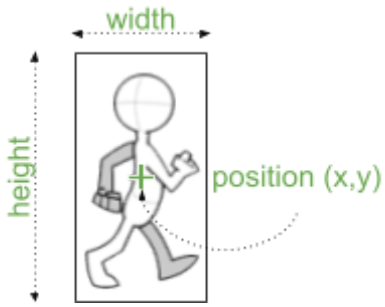
4.4 Where's the mouse?

One more step before we can get animating:

1. 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 reported (x,y) 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: `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 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. For extra debugging help, you might want to print out the index of the Walker 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 one of the figures.

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

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. In `mousePressed()`, add a loop to check whether the mouse is over any of the non-null objects in your *`walkers`* array. (Use that method you wrote in 4.4!) For *only* the **lowest-index** Walker

that the mouse is over, call the `setWalking(boolean)` object method from the Walker class to set that particular Walker's `isWalking` status to **true**.

3. 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.

4. 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.

If your walkers begin moving right away, 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. In your `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'`, *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'`, ALL non-null Walkers in the program must STOP walking. You can use the `setWalking(boolean)` method here again, this time with an argument of **`false`**.

Try it out! The only testing we will have you do for this program is interacting with it yourself, to see whether the behavior 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, make a final submission of your source code to Gradescope.

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

- **WalkingSim.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.

Students whose final submission (which must pass ALL immediate tests) 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 Central Time on the due date with no penalty.

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
- `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/walk-2.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
- `int width()` – returns the width of the application window in pixels