Small Assignment #6
Due: Wednesday, 3/26/2025 by 11:59 PM

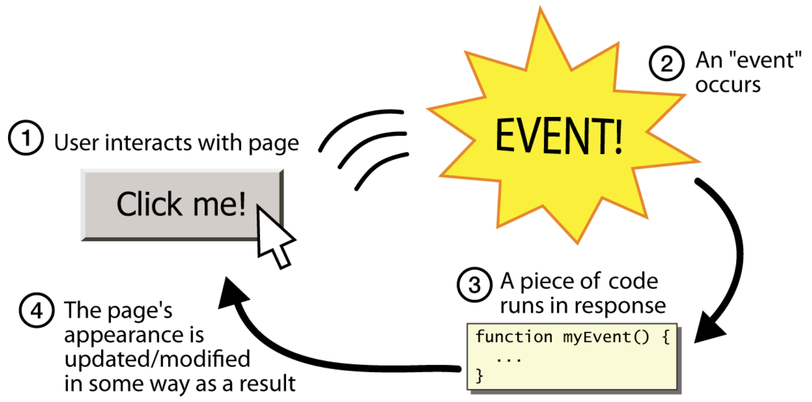
Submission. Submit this as a single PDF on Gradescope. Assign pages to questions.

Note: This is not actually as long as it seems. There's just a lot of reading.

Graphical User Interfaces
A graphical user interface (GUI) is a program that provides user interactions using graphical elements (which is very different than the text-based user interface you were supposed to implement for LA 1). Often GUI development is done using frameworks that are built for it, and Java has gone through a few iterations of GUI frameworks including *awt*, *swing*, and *javafx.* Of the three, *javafx* is the best in terms of what it offers. The downside is that it is no longer part of Java, which means that setting it up and figuring out how to get it running can be tricky. *Swing*, while less versatile and less visually appealing, is still maintained in Java and while it replaced *awt*, it does still use pieces of that framework.

For the purposes of this class, you are welcome to use whatever framework you prefer, as long as your code is in Java. Note that my sample code uses *swing* because it is simpler to get going and is sufficient for my purposes.

Event-Driven Programming
GUI development is closely related to the idea of *event-driven programming*. In a text-based UI, the program typically prompts the user for a command and then waits for the command. A GUI is different because the flow of execution in the program is based on responding to *events*. Events can include things like *a button click* or a *key being pressed*, which are ways for the user to provide input. When an event happens, there are parts of the code, called *event handlers*, that specify how the program responds to the event. The picture below gives a visualization of this process.


1 User interacts with page
Click me!
2 An "event" occurs
EVENT!
3 A piece of code runs in response
4 The page's appearance is updated/modified in some way as a result

```
function myEvent() {
  ...
}
```

Question 1. The following guides you through parts of the code demo. Follow this guide and provide your answers to each question.

- Start with the View class. This is the class that runs. Don't run it yet. First notice that this class extends JFrame, which is a type of container in the swing framework. The JFrame can hold various items in the GUI. Before running the program, make a note of the following:
    - the instance variables
    - Line 19 – note the title
    - LInes 37-47 – the buttons
- Now run the program, and answer the following questions with respect to the actual GUI frame.
    (a) What does Line 19 do?
    (b) What does LIne 20 do?
    (c) What do the two labels do?
    (d) What do the two buttons do?

- Back to the code, notice that each button has an ActionListener.
    (e) What class is the Listener?

- Now move to the Controller. Notice that it implements the `ActionListener` interface, which requires it to implement that `actionPerformed` method.
    (f) Explain what this method does and how it relates to the Button setup in the View (Lines 37-47).

To summarize so far, the View sets up the GUI and waits for user input (through events like button clicks).
When an event happens, the Controller handles it … by calling a method on the Model, which is where the data is stored.

(g) What does the Controller ask the Model to do if the "increment" button is clicked?
(h) What does the Controller ask the Model to do if the "decrement" button is clicked?

- Now take a look at the Model. This is where the number itself is stored, and there is only one, even though it shows up in two ways in the View.
    (i) When the Model's number is incremented, what happens to the two labels in the View?
    (j) When the Model's number is decremented, what happens to the two labels in the View?

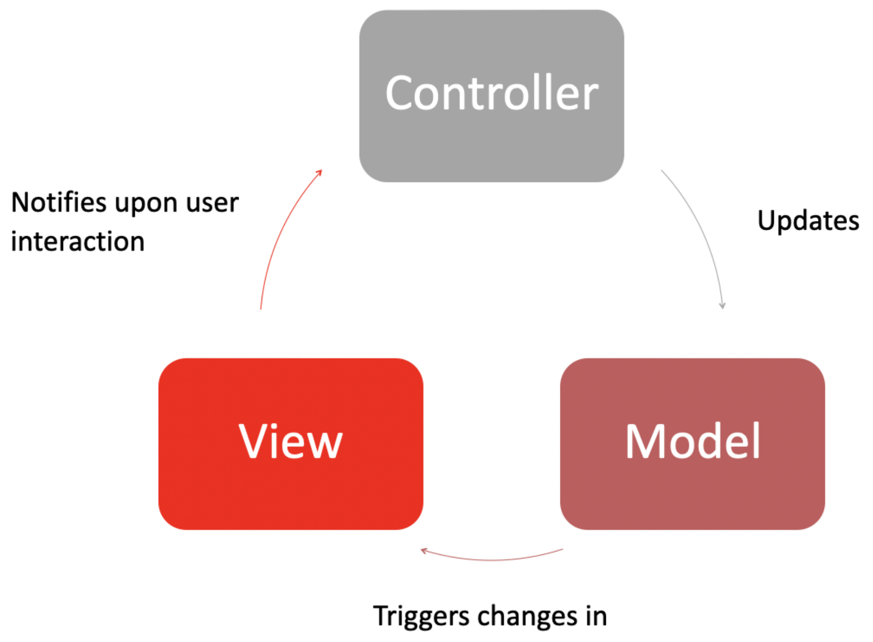
MVC: Model-View-Controller
What we have here is a simple example of the Model-View-Controller design pattern, which is a tricky design pattern because it has so many forms. This is just one possible form. But the main idea is that the Model, View, and Controller are clearly separate entities with very different jobs.

The *Model* stores the data.

The *View* interacts with the user.
The *Controller* handles events (and updates the Model accordingly).

This version of MVC also uses a design pattern called OBSERVER, which we will discuss next.

But first, take a look at the diagram below and answer the question.



(k) In the code example, when the user presses the increment button, explain what follows with reference to the diagram above.

OBSERVER Design Pattern.
The OBSERVER design pattern is very useful with GUIs because GUIs often have so many different components that need to work together according to the *same* information from the *same* Model. In this case, we have two Labels in our JFrame that both need to respond to changes in the number that is stored in the Model. We don't want too many dependencies, so this is a case where OBSERVER can help us.

A naive approach would be for both labels to query the Model periodically to see if the number has changed. But instead, we *invert the control*. In the diagram above, notice the final arrow connecting Model to View. When the model changes, it triggers a change in the View. This is part of the OBSERVER pattern. We will discuss this pattern in more detail in class, but for now, take a look at the code and answer the following questions.

(l) When the Model `increments` or `decrements` the number, it also calls another method. What method is that, and what does it do?

(m) Notice that the `notifyObservers` method loops through a list of Observers. Look in the other classes (not the Model) to find out where specific Observers are being added to that list. What are the Observers and where exactly in the code are they registered?

(n) Notice that in the `notifyObservers` method, the type of each object is the general `Observer`, not specific observers. This is an example of polymorphism because `Observer` is an interface type. What classes in this code demo implement Observer? What method are they required to implement because of that? How does that relate to the `notifyObservers` method in the `Model`.