Poll Everywhere

PollEv.com/javabear text **javabear** to 22333



Which of the following should the client call to update the appearance of an application window?

`frame.setVisible(true)`	(A)
`frame.pack()`	(B)
`frame.paintComponent()`	(C)
`frame.repaint()`	(D)

called during set-up

never call this yourself



Lecture 25: Event-Driven Programming

CS 2110

November 19, 2026

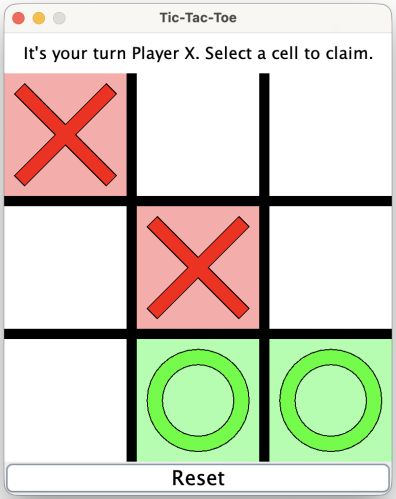
Today's Learning Outcomes

86. Categorize aspects of a GUI application as part of its *model*, *view*, or *controller*.

91. Describe the *observer pattern* and use it to write event listeners for graphical applications.

92. Explain the execution of a Swing application and its use of the event dispatch thread.

Where We Left Off...



GUI Applications can be organized into 3 aspects:

- **Model:** "back end" state representation
- **View:** "front end" windows and widgets
  - component hierarchy
  - layout managers
  - custom painting
- **Controller:** application logic (today's topic)

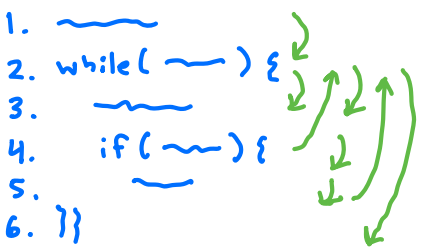
Still need to

- add symbols to board when clicked
- make reset button work
- make turn label refresh
- add end game messages

Programming Paradigms

Imperative

- write out sequence of instructions our code should follow
- "lower level"
- most code you've written



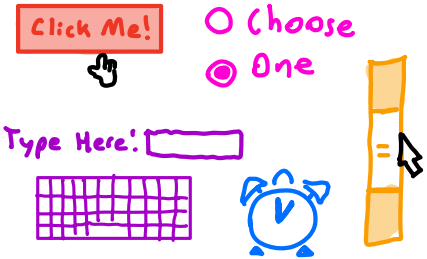
Declarative

- describe what you want the code to do, let it figure out how to make it happen
- "higher level"
- frameworks enable this



Event-Driven (subset of declarative)

- describe what a program should do in response to different "events"
  - mouse clicks
  - keyboard input
  - timers
  - state changes



The Observer Pattern

Interactive programs "invert control"

- We write code that describes what should happen *in response* to an event
- The running application determines when our code is actually executed

callbacks: "you'll need this code later"

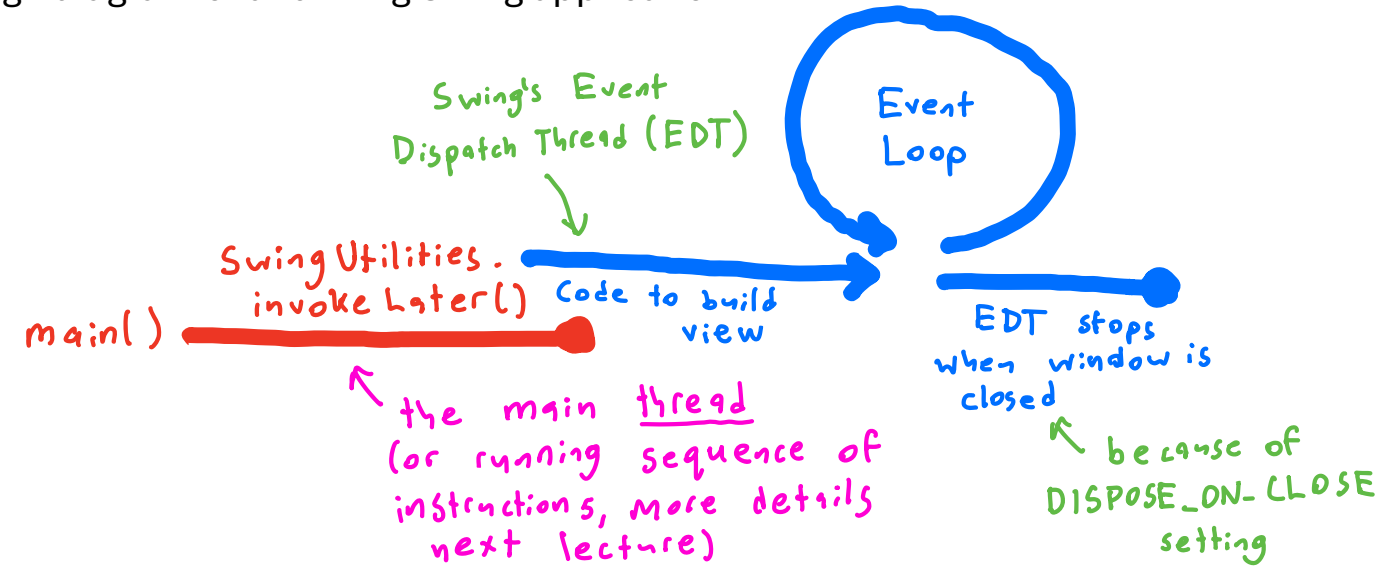
This inversion of control realizes the **observer pattern.**

many different objects (called observers or listeners) that know how to respond to different events will register with a central entity (called the subject) that promises to alert them when the event takes place

The Swing Event Loop

The subject in Swing's realization of the observer pattern is its **event loop.**

Rough diagram of a running Swing application:



Events and Listeners

An **event** is an object that models a scenario that can trigger a response in our program.

- ActionEvent (click, timer) - KeyEvent (typing) - MouseEvent

Swing interacts with your operating system to detect and create events

The event source is the context where the event takes place

- clicking on a button or checkbox, entering text in a box, mousing over a panel

An event listener specifies the response action

- ActionListener, MouseListener, etc.

We use the add___Listener() methods to tell Swing what to do when a particular source becomes the context of a particular event.

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333



Which of these references the **source** of the ActionEvent in the following code?

```
class App extends JFrame implements ActionListener {
  public App() {
    JButton b = new JButton("B");
    add(b);
    b.addActionListener(this);
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    print(e.getSource() + " activated!" );
  }
}
```

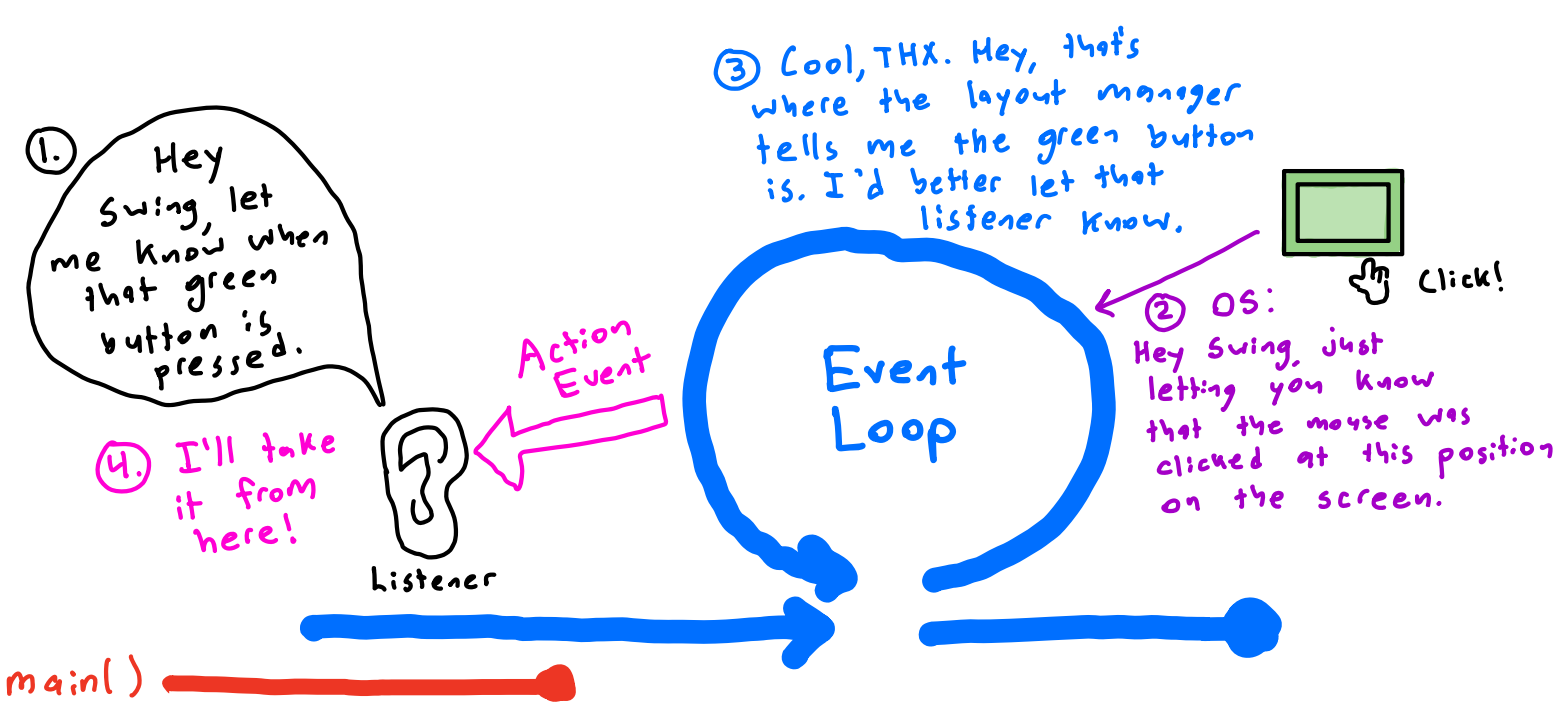
App (A)

b source (B)

e event (C)

this listener (D)

Visualizing Events





Coding Demo: The Reset Button



Simplified Listener Creation

1. Nest the listener class inside of one class where it's used

2. Use an anonymous class defined within method body

syntax:

interface or superclass name

superconstructor args (empty for interface)

```
new ActionListener() {
    // method definitions
}
```

* anonymous classes don't have names, so they can only be instantiated on line where they're declared

can capture static variables, fields and "final" local variables in scope

3. Use a lambda expression
   (event listeners are really packaging up behaviors)

Poll Everywhere

PollEv.com/javabear text `javabear` to 22333



In which of the following cases could we not use a lambda expression to define a listener?

need a functional interface

When the listener's body references a field. **(A)**

When the listener defines multiple methods. **(B)**

When the listener's body must call a helper method. **(C)**

We can always do this! **(D)**



Coding Demo: The Cell Buttons



Property Changes

Sometimes, we need a state update in one class to trigger an update in another class.

Directly linking (or *coupling*) all the classes to enable this (through an instance method call) becomes messy.

Solution: In the observer pattern, we can delegate this "message passing" to the central subject (event loop)

Property changes are artificial events that signal some named property has been changed (firePropertyChange())

Other classes can listen for changes to that named property (addPropertyChangeListener())



Coding Demo: The Turn Label

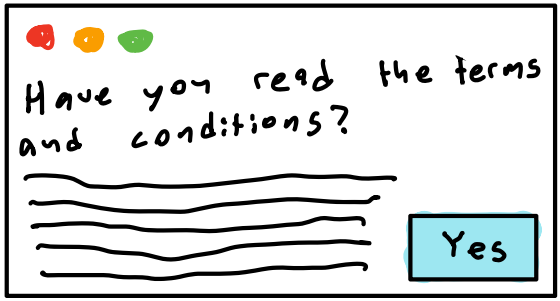


Dialog Boxes

= Temporary, secondary windows that prompt users at particular points in the application.

They are modal windows: they block execution and interaction with other windows until they are closed.

JOptionPanes are a good option for informational messages.



Handling Many Events

So far, our illustrations of the event loop have been simplified. In reality, hundreds of events may be queued in the event loop per second.

Event loop hands control of EDT over to listener method to respond, which blocks event loop (lag)

* Keep event listeners quick and simple

or

* delegate work to another thread to happen concurrently with EDT

