CSE 331
Software Design & Implementation

Hal Perkins

Spring 2017

GUI Event-Driven Programming

The plan

- User events and callbacks
  - Event objects
  - Event listeners
  - Registering listeners to handle events

- Anonymous inner classes
  - (and a quick look at Java 8 lambdas)

- Proper interaction between UI and program threads

Event-driven programming

Many applications are *event-driven* GUI programs:

- Program initializes itself on startup then enters an *event loop*
- Abstractly:

```
do {
    e = getNextEvent();
    process event e;
} while (e != quit);
```

Contrast with application- or algorithm-driven control where program expects input data in a particular order

- Typical of large non-GUI applications like web crawling, payroll, simulation, optimization, …

Kinds of GUI events

Typical *events* handled by a GUI program:

- Keyboard: key press or release, sometimes with modifiers like shift/control/alt/etc.
- Mouse move/drag/click, button press, button release – also can have modifiers like shift/control/alt/etc.
- Finger tap or drag on a touchscreen
- Joystick, drawing tablet, other device inputs
- Window resize/minimize/restore/close
- Network activity or file I/O (start, done, error)
- Timer interrupt (including animations)

Events in Java AWT/Swing

Many (most?) of the GUI widgets can generate events (button clicks, menu picks, key press, etc.)

Handled using the Observer Pattern:

- Objects wishing to handle events register as observers with the objects that generate them
- When an event happens, appropriate method in each observer is called
- As expected, multiple observers can watch for and be notified of an event generated by an object

Event objects

A Java GUI event is represented by an *event object*

- Superclass is **`AWTEvent`**
- Some subclasses:

  `ActionEvent` – GUI-button press

  `KeyEvent` – keyboard

  `MouseEvent` – mouse move/drag/click/button

Event objects contain information about the event

- UI object that triggered the event
- Other information depending on event. Examples:

  `ActionEvent` – text string from a button

  `MouseEvent` – mouse coordinates

Event listeners

Event listeners must implement the proper interface: **`KeyListener`**, **`ActionListener`**, **`MouseListener`** (buttons), **`MouseMotionListener`** (move/drag), …

- Or extend the appropriate library *abstract class* that provides empty implementations of the *interface* methods

When an event occurs, the appropriate method specified in the interface is called: **`actionPerformed`**, **`keyPressed`**, **`mouseClicked`**, **`mouseDragged`**, …

An event object is passed as a parameter to the event listener method

Example: button

Create a **`JButton`** and add it to a window

Create an object that implements **`ActionListener`**

- (containing an **`actionPerformed`** method)

Add the listener object to the button's listeners

`ButtonDemo1.java`

Which button is which?

Q: A single button listener object often handles several buttons. How to tell which button generated the event?

A: an **`ActionEvent`** has a **`getActionCommand`** method that returns (for a button) the "action command" string

- Default is the button name (text), but usually better to set it to some string that will remain the same inside the program code even if the UI is changed or button name is translated. See button example.

Similar mechanisms to decode other events

Listener classes

`ButtonDemo1.java` defines a class that is used only once to create a listener for a single button

- Could have been a top-level class, but in this example it was an inner class since it wasn't needed elsewhere
- But why a full-blown class when all we want is to house a method to be called after a button click?

A more convenient(?) Java shortcut: *anonymous inner classes*

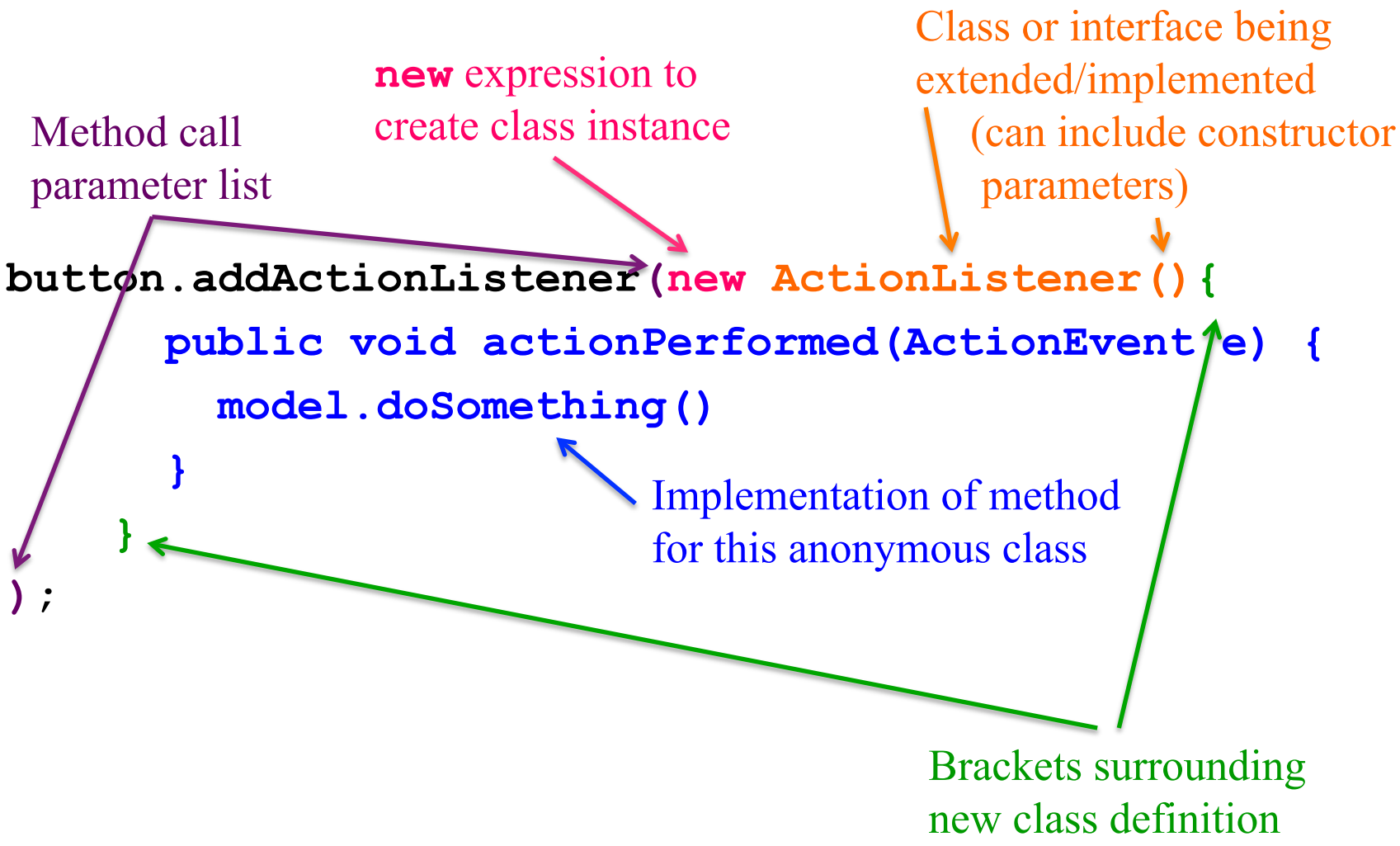
Anonymous inner classes

Idea: define a new class directly in the **`new`** expression that creates an object of the (new) anonymous inner class

- Specify the superclass to be extended or interface to be implemented
- Override or implement methods needed in the anonymous class instance
- Can have methods, fields, etc., but not constructors
- But if it starts to get complex, use an ordinary class for clarity (nested inner class if appropriate)

Warning: ghastly syntax ahead

Example


Class or interface being extended/implemented (can include constructor parameters)
new expression to create class instance
Method call parameter list
button.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e) {
model.doSomething()
}
});
Implementation of method for this anonymous class
Brackets surrounding new class definition

Example

`ButtonDemo2.java`

Lambdas (Java 8) [optional]

Why create a complete class (anonymous or otherwise) if you just want to define a method to be called when a button is clicked?

Java 8 provides *lambdas* – anonymous methods – for situations like this

- Limitation: a lambda is not a complete object, so if you want private state, constructors, etc., you want an anonymous or named class instead
- Many other uses, especially with container classes
- Feel free to use in your code *if* you understand what's happening

`ButtonDemo3.java`

Program thread and UI thread

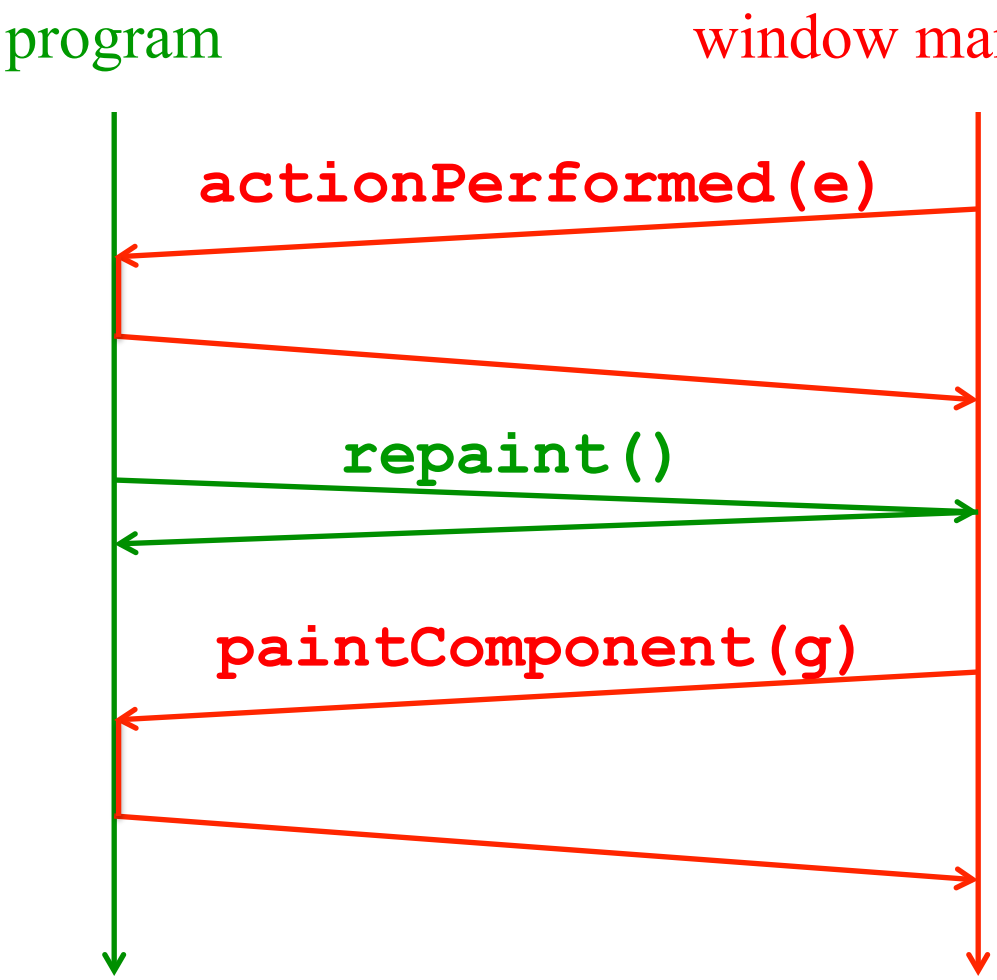
Recall that the program and user interface are running in separate, concurrent threads

All UI actions happen in the UI thread – *including callbacks* like **`actionListener`** or **`paintComponent`**, etc. defined in your code

After event handling and related work, call **`repaint()`** if **`paintComponent()`** needs to run. **Don't** try to draw anything from inside the event handler itself (as in ***you must not do this!!!***)

> Remember that **`paintComponent`** must be able to do its job whenever the window manager calls it – so any data it needs to render must be prepared in advance

Event handling and repainting



Remember: your program and the window manager are running concurrently:

- Program thread
- User Interface thread

It's ok to call **`repaint`** from an event handler, but ***never*** **call `paintComponent` yourself** from either thread.

Working in the UI thread

Event handlers should not do a lot of work

- If the event handler does a lot of computing, the user interface will appear to freeze up
  - (Why?)

- If there's lots to do, the event handler should set a bit that the program thread will notice. Do the heavy work back in the program thread.
  - (Don't worry – finding a path for campus maps should be fast enough to do in the UI thread)

Synchronization issues?

Yes, there can be synchronization problems

- (cf. CSE332, CSE451, CSE452, …)

Not generally an issue in well-behaved programs, but can happen

Advice:

- Keep event handling short
- Call **`repaint`** when data is ready, not when only partially updated
- Don't update data in the UI and program threads at the same time (particularly for complex data)
- Never call **`paintComponent`** directly
  - (Have we mentioned you should never ever call **`paintComponent`**? And don't create a new **`Graphics`** object either.)

If you are building industrial-strength UIs, learn more about threads and Swing and how to avoid potential problems by scheduling computations to be run by the UI thread (Swing tutorial, *Core Java*, …)

Larger example – bouncing balls

A hand-crafted MVC application. Origin is somewhere back in the CSE142/3 mists. Illustrates how some swing GUI components can be put to use.

Disclaimers:

- Not the very best design (maybe not even particularly good)
- Unlikely to be directly appropriate for your project
- Use it for ideas and inspiration, and feel free to steal small parts if they *really* fit

Enjoy!