CSE 331
Software Design & Implementation

Hal Perkins

Autumn 2012

Event-Driven Programming

The plan

- User events and callbacks
  - Event objects
  - Event listeners
  - Registering listeners to handle events
- Anonymous inner classes
- Proper interaction between UI and program threads

Event-driven programming

Modern 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, batch simulation

Kinds of events

Typical events handled by a GUI program:

- Mouse move/drag/button press/button release/click
- Keyboard: key press or release, sometimes with modifiers like shift/control/alt
- 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 observer/observable pattern:

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

Event objects

A Java event is represented by an *event object*

Parent class is **`AWTEvent`**. Some subclasses:

- **`ActionEvent`** – 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), …

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 often handles several buttons. How to tell which is which?

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

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

Similar mechanisms to decode other events

Listener classes

ButtonDemo1.java defines a class that is only used 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-scale class when all we want is to create a method to be called after a button click?

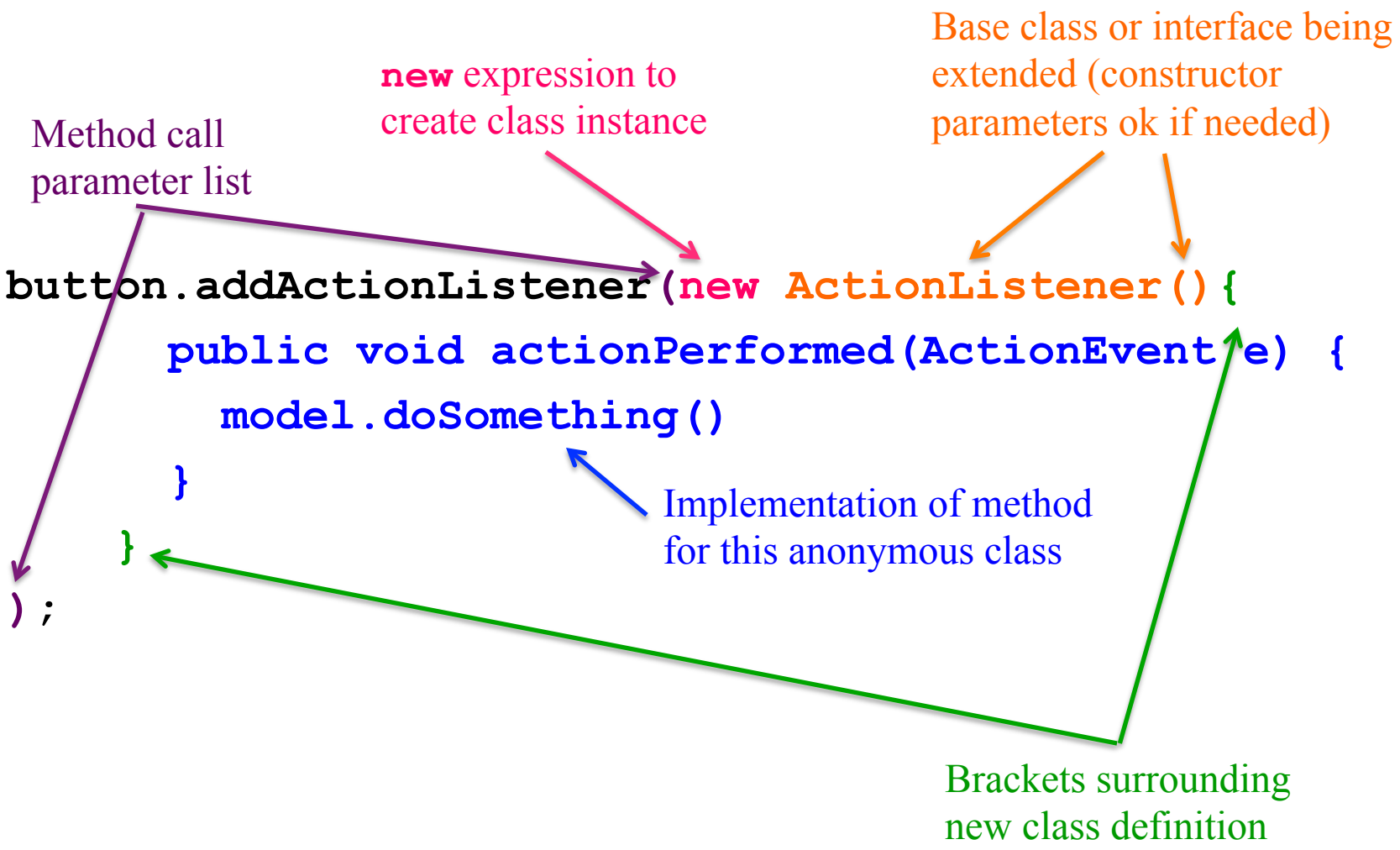
Solution: anonymous inner classes

- Warning: ghastly syntax ahead

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 base class 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)

Example


Method call parameter list
new expression to create class instance
Base class or interface being extended (constructor parameters ok if needed)
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

Program thread and UI thread

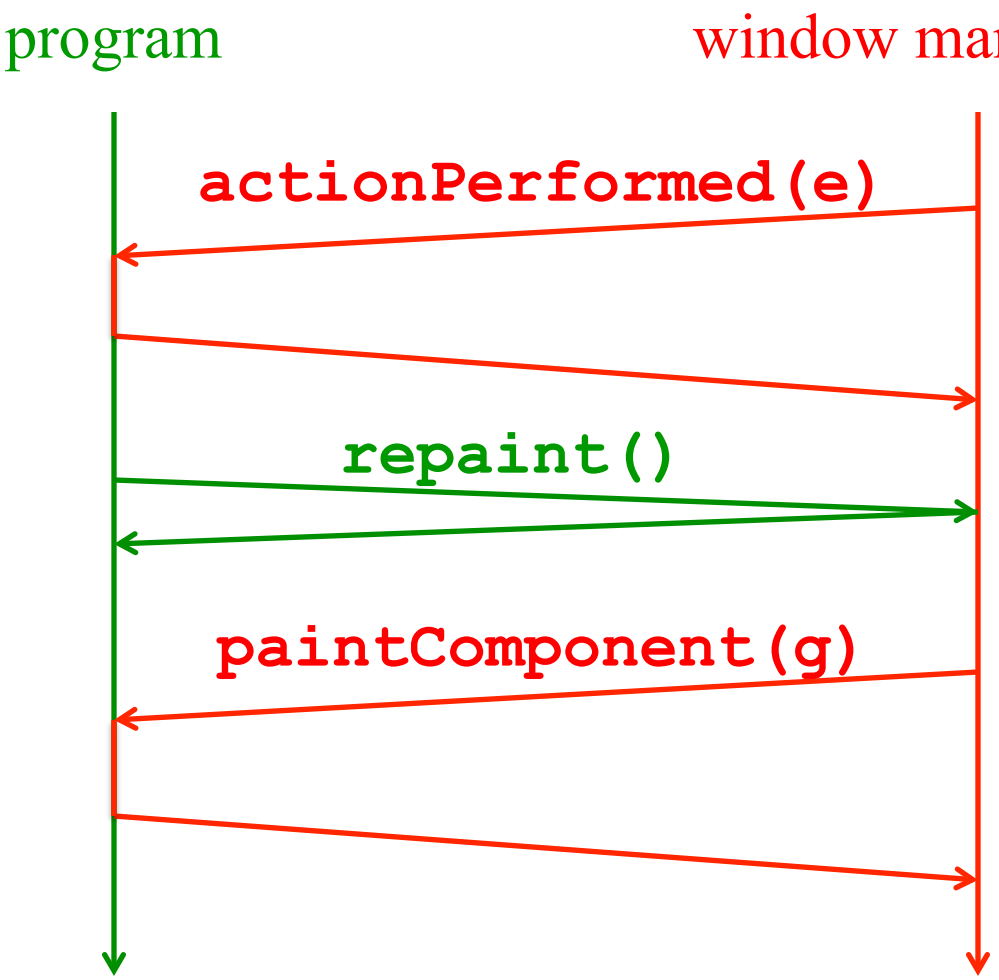
Recall that the program and user interface are running in concurrent threads

All UI actions happen in the UI thread – *even when* they execute callbacks to code like **`actionListener`**, etc. defined in your program

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 by reading data that is available whenever the window manager calls it

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 usually should not do a lot of work

If the event handler does a lot of computing the UI will appear to freeze up

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
- Not usually an issue in well-behaved programs, but can happen if you work at it (deliberately or not)
- Some advice:
  - Keep event handling short
  - Call repaint when data is ready, not when partially updated
  - Don't update data in the UI and program threads at the same time (particularly for complex data)
  - Never ever call **`paintComponent`** directly

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:

- Might not be the very bestest design
- Unlikely to be directly appropriate for your project
- Use it for ideas and inspiration, and feel free to steal small bits if they *really* fit

Have fun!