CSE 331
Software Design & Implementation

Kevin Zatloukal

Summer 2016

Event-Driven Programming: Servers & GUIs

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Review

- Module Design
  - reducing coupling
  - supporting reuse (where easy)

- Events & Callbacks
  - pass in code to be called when a certain **event** occurs
  - can be synchronous or asynchronous (immediate or delayed)

- Today: event-driven programming
  - most programs in nature are event-driven
    - they spend most of the time waiting
  - examples: (web) servers and GUIs (client programs)
    - web server is waiting for client connections
    - GUI client is waiting for a user event (mouse, key, touch, etc.)

Event-driven programming

An *event-driven* program is designed to wait for events:

- program initializes then enters the *event loop*
- abstractly:

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

Contrast with most programs we have written so far

- they perform specified steps in order and then exit
- that style is still used, just not as frequently
    - example: computing Page Rank or other Big Data work

Server Programming

- Servers sit around waiting for events like:
  - new client connections
  - new data from the client (large scale servers)

- Simple version (normal scale):

```
while (true) {
    wait for a client to connect
    process the request; send a response back
}
```

  - (might want to use a new thread for processing)
  - web servers usually look like this

Sockets (*off topic*)

- Each client connection is represented by a "socket"

- A socket is like a **file**
  - can be read from and written to
  - (in Unix, sockets and files are nearly identical)

- Client and server each have "half" of the socket
  - what the client writes is read by the server
  - what the server writes is read by the client

| client socket | read ← write | write read | server socket |

Example: Chat Server

`ChatServer.java`

(don't freak out…)

Advanced Server Programming

- Large scale servers usually do not have one thread per client
  - it would be hard to scale that past hundreds of clients

- Instead, they have a small number (1?) of threads that simultaneously wait on events from all sockets
  - new connections on the server socket
  - new data to read on any client socket
  - finish writing to any client socket
    - (can now by write more)
  - handlers do not make any calls that might wait for something

- These servers look much more like GUI clients…

GUI Client Programming

- Clients sit around waiting for events like:

  - mouse move/drag/click, button press, button release

  - keyboard: key press or release, sometimes with modifiers like shift/control/alt/etc.

  - finger tap or drag on a touchscreen

  - window resize/minimize/restore/close

  - timer interrupt (including animations)

  - network activity or file I/O (start, done, error)

    - (we will see an example of this shortly)

Events in Java AWT/Swing

AWT & Swing are the Java libraries for writing GUIs

Most of the GUI widgets can generate events
– button clicks, menu picks, key press, etc.

Events are 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

Likewise, advanced servers register handlers on each socket

Event objects

GUI event is represented by an *event object*

   – passes information often needed by the handler

In Java, the superclass is `AWTEvent`. Some subclasses are:

      `ActionEvent` – GUI-button press

      `KeyEvent` – keyboard

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

Event objects contain

   – UI object that triggered the event

   – other information depending on event. Examples:

      `ActionEvent` – text string from a button

      `MouseEvent` – mouse coordinates

Event listeners / handlers

Event listeners must implement the proper interface:

        `KeyListener` – handle key press

        `ActionListener` – handle button press

        `MouseListener` – handle mouse clicks

        `MouseMotionListener` – handle mouse move/drag

When an event occurs

- the appropriate method specified in the interface is called: `actionPerformed`, `keyPressed`, `mouseClicked`, `mouseDragged`, …

- an event object is passed to the listener method

Example: button

Create a **JButton** and add it to a window
- – (we will talk about windows next time)

Create an object that implements **ActionListener**
- – contains an **actionPerformed** method

Add the listener object to the button's listeners
- – then it will be called when the button is pressed

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

For advanced servers, you may want to use one handler for all "data to read" events on all client sockets.

Listener classes

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

Not ideal in a couple of respects:
- – listener code is far away from where it's used
  - • that makes it a little harder to understand
- – it's a lot of code for just one listener
  - • imagine doing this in a UI with thousands of components
  - • (that's not a stretch)

A more convenient shortcut: *anonymous inner classes*
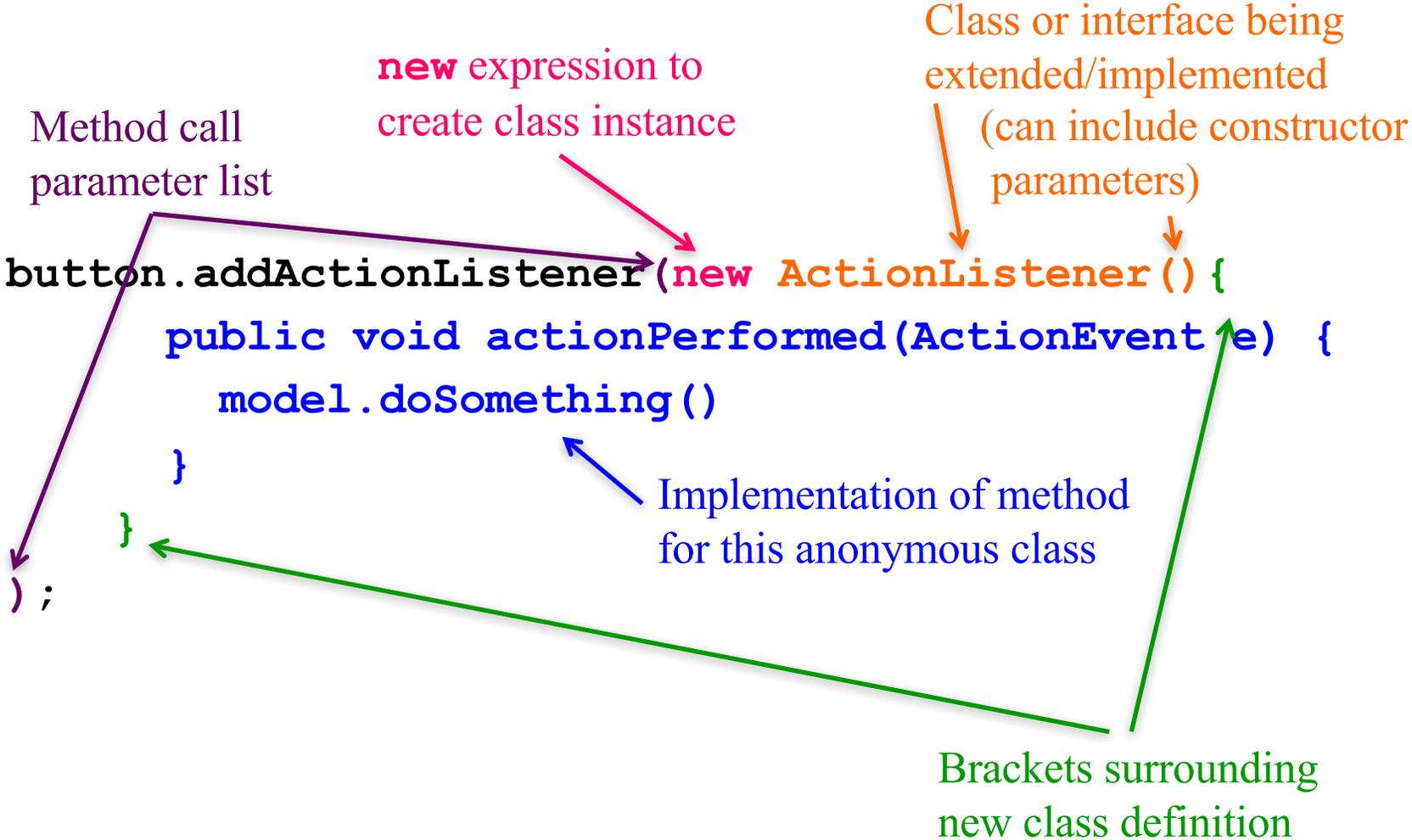
Anonymous inner classes

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

Tip: if it starts to get complex, use an ordinary class for clarity

Warning: ghastly syntax ahead

Example: anonymous inner class

Method call
parameter list

new expression to
create class instance

Class or interface being
extended/implemented
(can include constructor
parameters)

```
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        model.doSomething()
    }
  }
);
```

Implementation of method
for this anonymous class

Brackets surrounding
new class definition

Example: button

ButtonDemo2.java

Lambdas (*off topic*)

Idea: why write a class at all when we just want to create a method to be called after a button click?

An even more convenient shortcut: *lambdas*

- – in Java 8, you can use lambdas to create anonymous methods instead of creating an anonymous class that only exists to house a method.

Example: button

ButtonDemo3.java

Example: GUI + sockets

Most modern client applications have to both

- display a GUI

- communicate with one or more servers

- (doing both creates additional difficulties…)

We can make an example by writing a GUI chat client

`ChatClientGUI.java`

(don't freak out…)

UI thread

- The event loop of a GUI program is run on the "UI thread"

- Often have need of additional threads
  - example: chat UI needed one to listen for new messages
  - any work that may take > 250ms should be done elsewhere

- **Warning**: most UI frameworks are not multi-thread safe
  - this will **not** be an issue *in this class* but will be IRL
  - very few UI API methods can be called from other threads
  - instead, they provide ways to push work onto the UI thread
    - pass a callback to be called from the UI thread
    - then perform the UI changes you need there