GUI Event-Driven Programming

CSE 331

Software Design & Implementation

Slides contain content by Hal Perkins and Michael Hotan

Outline

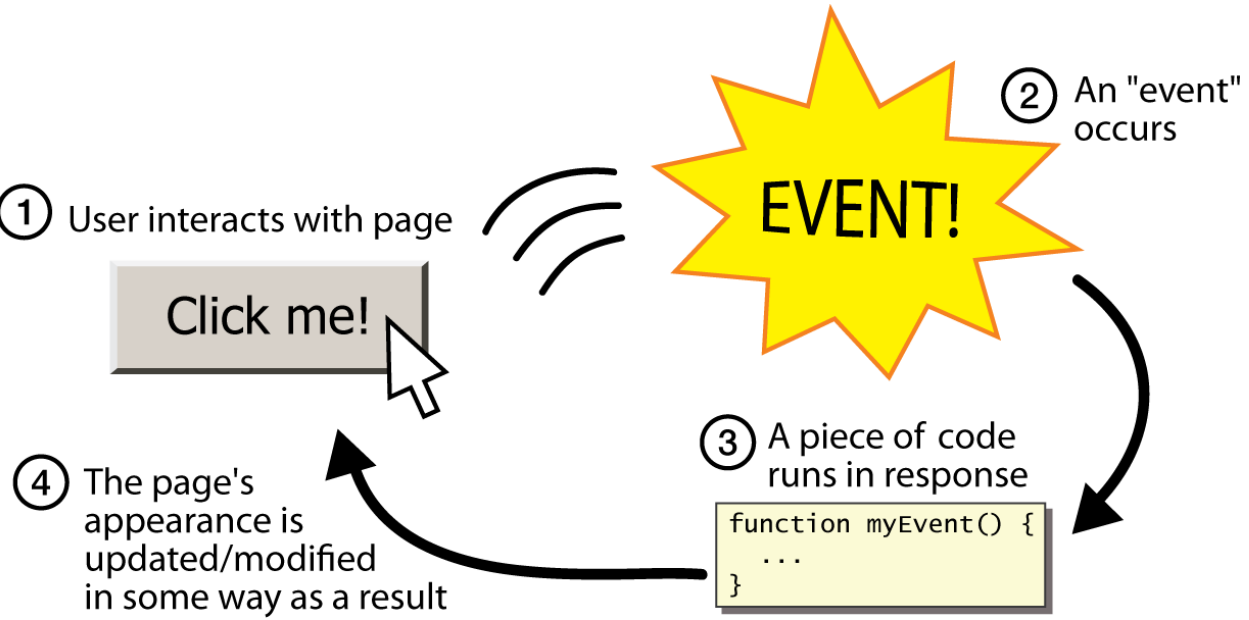
User events and callbacks

- Event objects
- Event listeners
- Registering listeners to handle events

Anonymous inner classes

Proper interaction between UI and program threads

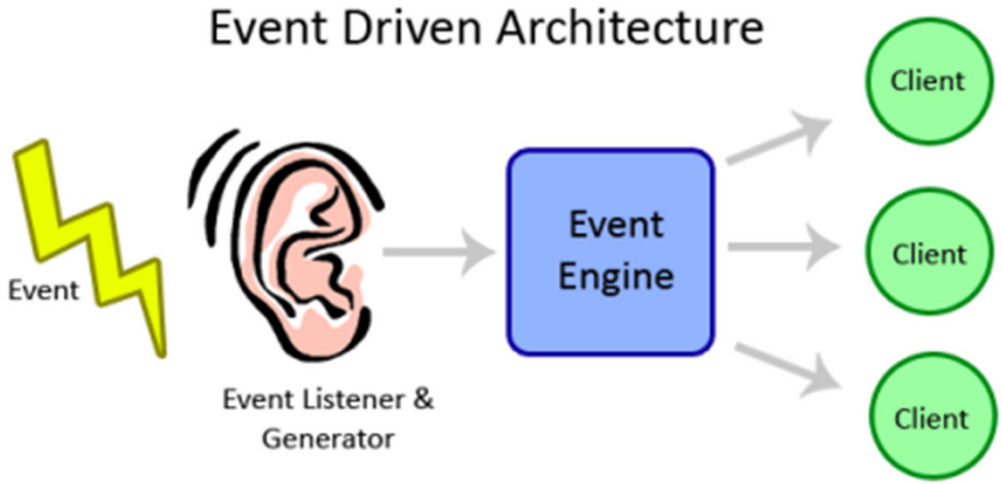
Event-driven programming



The main body of the program is an event loop.
Abstractly:

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

Event-driven programming



Event-driven programming:
A style of coding where a program's overall flow of execution is dictated by events.

- The program loads, then waits for user input events.
- As each event occurs, the program runs particular code to respond.
- The overall flow of what code is executed is determined by the series of events that occur
- Contrast with application- or algorithm-driven control where program expects input data in a pre-determined order and timing
  - Typical of large non-GUI applications like web crawling, payroll, batch simulation

Graphical events

- **event**: An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.

- **listener**: An object that waits for events and responds to them.
  - To handle an event, attach a *listener* to a component.
  - The listener will be notified when the event occurs (e.g. button click).

Kinds of GUI events

- Mouse move/drag/click, mouse button press/release
- Keyboard: key press/release, sometimes with modifiers like shift/control/alt/meta/cokebottle
- Touchscreen finger tap/drag
- Joystick, drawing tablet, other device inputs
- Window resize/minimize/restore/close
- Network activity or file I/O (start, done, error)
- Timer interrupt (including animations)

EventObject represents an event

```
import java.awt.event.*;
```

- EventObject
  - AWTEvent (AWT)
    - **ActionEvent**
    - TextEvent
    - ComponentEvent
      - FocusEvent
      - WindowEvent
      - InputEvent
        - KeyEvent
        - MouseEvent

An action that has occurred on a GUI component

button/menu/ checkbox/text box/…

keyboard

mouse move/drag/ click/wheel

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

Action events

- **action event**: An action that has occurred on a GUI component.
  - The most common, general event type in Swing. Caused by:
    - button or menu clicks,
    - check box checking / unchecking,
    - pressing Enter in a text field, ...
  - Represented by a class named `ActionEvent`
  - Handled by objects that implement interface `ActionListener`

Handling events in Java Swing/AWT

GUI widgets can generate events (in response to button clicks, menu picks, key press, etc.)

Handled using observer pattern.

Standard observer pattern:

- Object wishing to handle event is an **`Observer`**
- Object that generates events is an **`Observable`**
- Observer registers with the observable
- When an event happens, observable calls a method (**`update`**) in each observer
  - may notify multiple observers

Implementing a listener

```
public class MyClass implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        code to handle the event;
    }
}
```

`JButton` and other graphical components have this method:

```
/** Attaches the given listener to be notified of clicks and events that occur on this component. */
public void addActionListener(ActionListener al)
```

EventListener type hierarchy

```
import java.awt.event.*;
```

- EventListener
  - AWTEventListener
  - **ActionListener** has method **actionPerformed()**
  - TextListener
  - ComponentListener
  - FocusListener
  - WindowListener

  - KeyListener has method **keyPressed()**
  - MouseListener has method **mouseClicked()**
  - MouseMotionListener has method **mouseDragged()**

When an event, occurs the appropriate method specified in the interface is called

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

JButton

a clickable region for causing actions to occur

- `public JButton(String text)`
  Creates a new button with the given string as its text.

- `public String getText()`
  Returns the text showing on the button.

- `public void setText(String text)`
  Sets button's text to be the given string.

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.

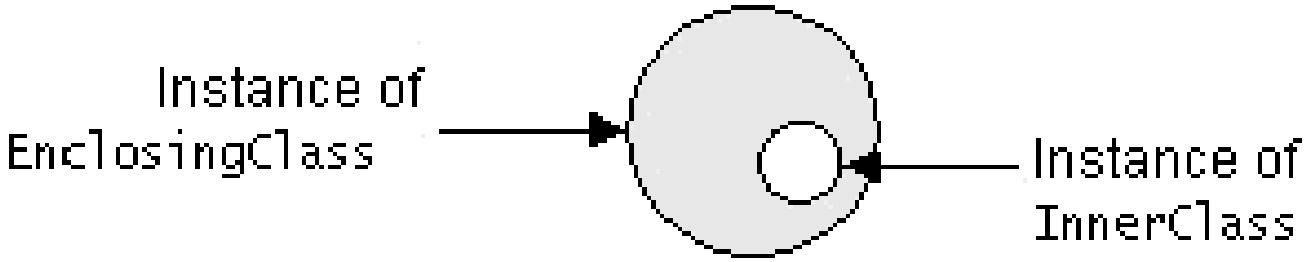
Similar mechanisms to decode other events

Inner classes

- **nested class**: A class defined inside of another class.
  - Two varieties: static nested class, and inner classes

- Usefulness:
  - Inner classes are hidden from other classes (encapsulated).
  - Inner objects can access/modify the fields of their outer object.

- Event listeners are often defined as nested classes inside a GUI.



Nested class syntax

```
// Enclosing outer class
public class OuterName {
   ...

   // Instance nested class (= inner class)
   // A different class for each instance of OuterName
   private class InnerName {
      ...
   }

   // Static nested class
   // One class, shared by all instances of OuterName
   // (syntactic sugar for a top-level class)
   static private class NestedStaticName { ... }
}
```

- If private, only the outer class can see the nested class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterName .** `this`

Use-once 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

Anonymous inner classes

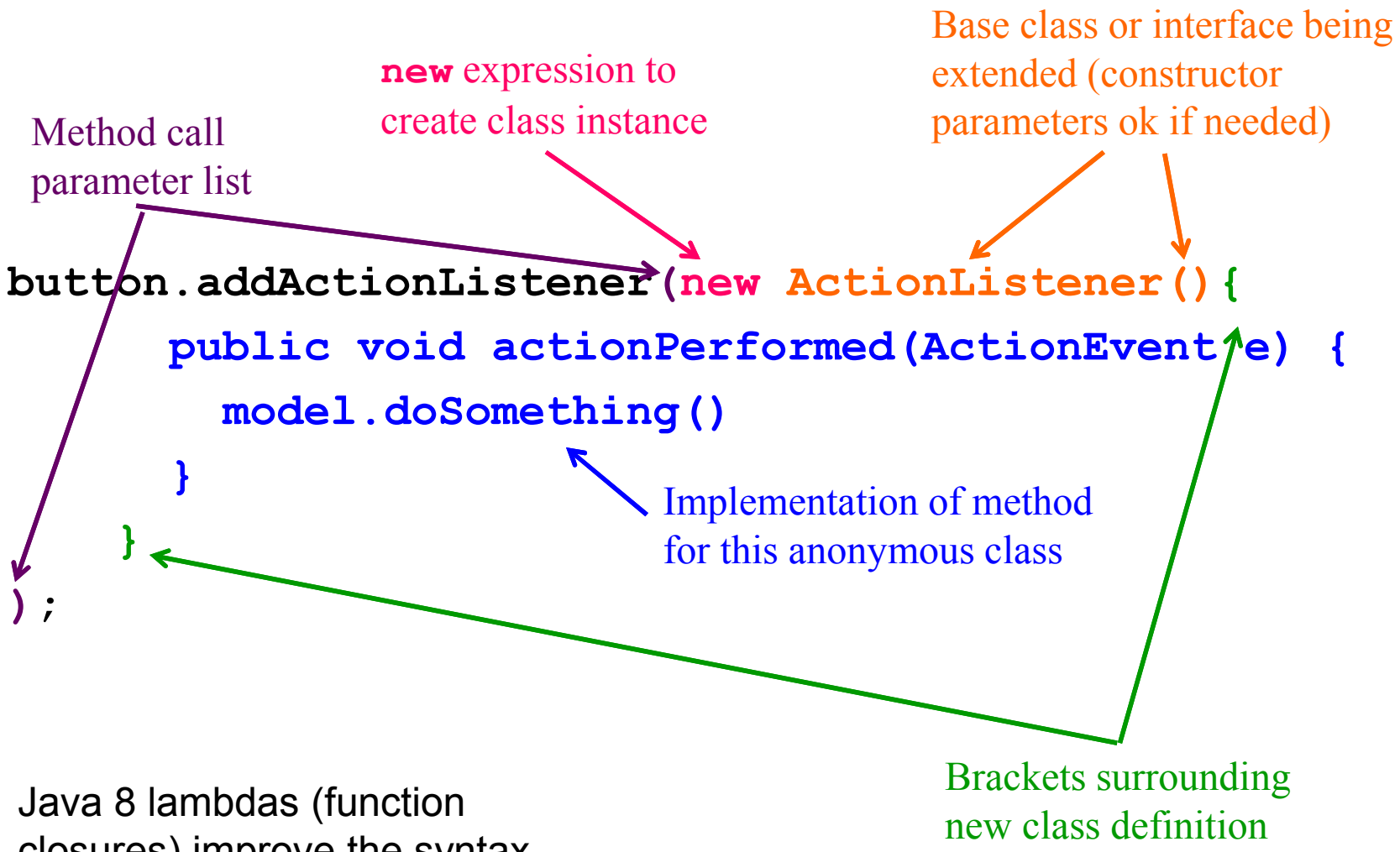
Idea: define a new class directly in the **`new`** expression that creates an object of the (new) anonymous inner class

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)

Anonymous class syntax



Java 8 lambdas (function closures) improve the syntax

Example

ButtonDemo2.java

Program thread and UI thread

The program and user interface run in concurrent threads

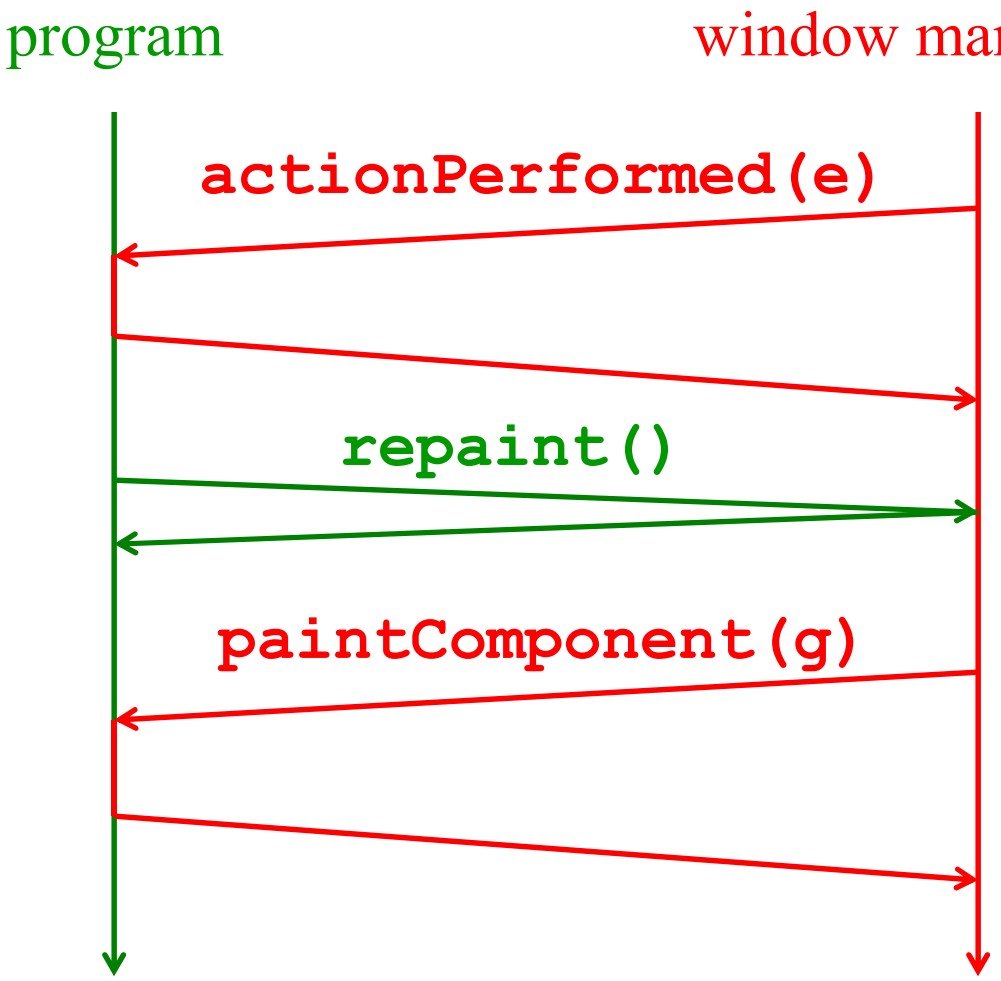
All UI actions happen in the UI thread – *even when* they execute callbacks to code like **`actionListener`**, etc. defined in your program

After event handling and related work, you may call **`repaint()`** if **`paintComponent()`** needs to run.

Do not 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 user interface 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
  - (Have we mentioned you should never call **`paintComponent`**? And don't create a new **`Graphics`** object either.)

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:

- Poor 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!