R·I·T

Computer Science II (4003-232-01)

Week 7: Event-Driven Programming in Java

Richard Zanibbi

Rochester Institute of Technology

Event-Driven Programming

Event-Driven Programming

Programs (or parts of programs) wait for messages from an **event loop** representing system events that have occurred at **run-time**.

Handler (or *Listener*) algorithms are **registered** for specific events and then executed when those events are received by the **event loop**

- **Example events:** pressed keys, mouse moves/clicks, connecting a USB device to a personal computer, time stamp

Event Creator: The Operating System

Operating system detects/defines system events and passes them onto programs (including Java programs)

Event-Driven Programming in Java:

- The JVM receives event messages from the OS, and then sends messages (I.e. invokes methods) to objects registered for each event.
- Java provides a set of interfaces defining the message interface for components (widgets) registered for each type of event (see page 467 of Liang). These methods are invoked by the JVM for all objects registered for an event, when an event is received.

Comparison of Event and Exception Messages in Java

Event Messages as Objects

1. Type of event (object type, e.g. ActionEvent)
2. Which object created (was the source of) the event message
3. When the event occurred
4. Data specific to the event type (e.g. the item selected from a list)

The Java event loop, handler interfaces, and handler registration methods are used for handling event messages

Run-Time Error Messages (Exceptions) as Objects

1. Type of exception (object type, e.g. IOException)
2. Which object/method threw (was the source of) the exception message, and at what statement
3. State of the call stack when the exception was thrown
4. Data specific to the exception type (e.g. bad array index value)

The try-catch statement and 'throwing' protocol are used for handling exception messages

Registering Handlers for Events

Java GUI Components

Have registration methods for creating a list of objects implementing handler interfaces for each possible event

- e.g. addActionListener() for associating handlers with mouse clicks on JButton objects (*ActionListener* objects)

When a component is notified of an event, a message is created and then sent to every listener object in the appropriate event “listener” list

....a bit like being on the ‘mailing list’ for each event.

Inner Class Example ('SimpleEventDemoInnerClass.java')

Inner Class:

– Declared within the scope of a class, and has access to its data members and methods (including for instances).

– Useful for defining objects that will be used within a class, but not elsewhere.

```
public class SimpleEventDemoInnerClass extends JFrame {
  public SimpleEventDemoInnerClass() {
    JButton jbtOK = new JButton("OK");
    setLayout(new FlowLayout());
    add(jbtOK);

    ActionListener listener = new OKListener();
    jbtOK.addActionListener(listener);
  }

  ....

  private class OKListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("It is OK");
    }
  }
}
```

Anonymous Classes

Anonymous Class

Within a method call, both:

- Defines a new class based on an existing class or interface (overriding methods)
- Creates an instance of this new class.

Example of an Anonymous Class:

```
window.addWindowListener(new WindowAdapter() {
            public void windowActivated(WindowEvent event) {
                        System.out.println(“Window Activated”);
            }
} );
```

For Java Swing

–Programmers use inner and anonymous classes extensively to reduce the number of separate classes and files needed for GUI programs.

–In particular, inner and anonymous classes get used to define *handlers* for interface events (e.g. mouse, keyboard, timer).

Important Note:

javac output for inner and anonymous classes differs from "normal classes"

(e.g. *OuterClass$InnerClass.class, Outerclass$1.class)*

MVC Architecture (A high-level organization for GUIs)

Purpose

High-level organization for programs with interfaces.

Model: stores and manipulates "core" program data

- Can be thought of as the "essential" data and operations for a program, independent of how this is shown to the user
  - » e.g. a single chess game model object might be hooked up to a text-based or GUI based interface.

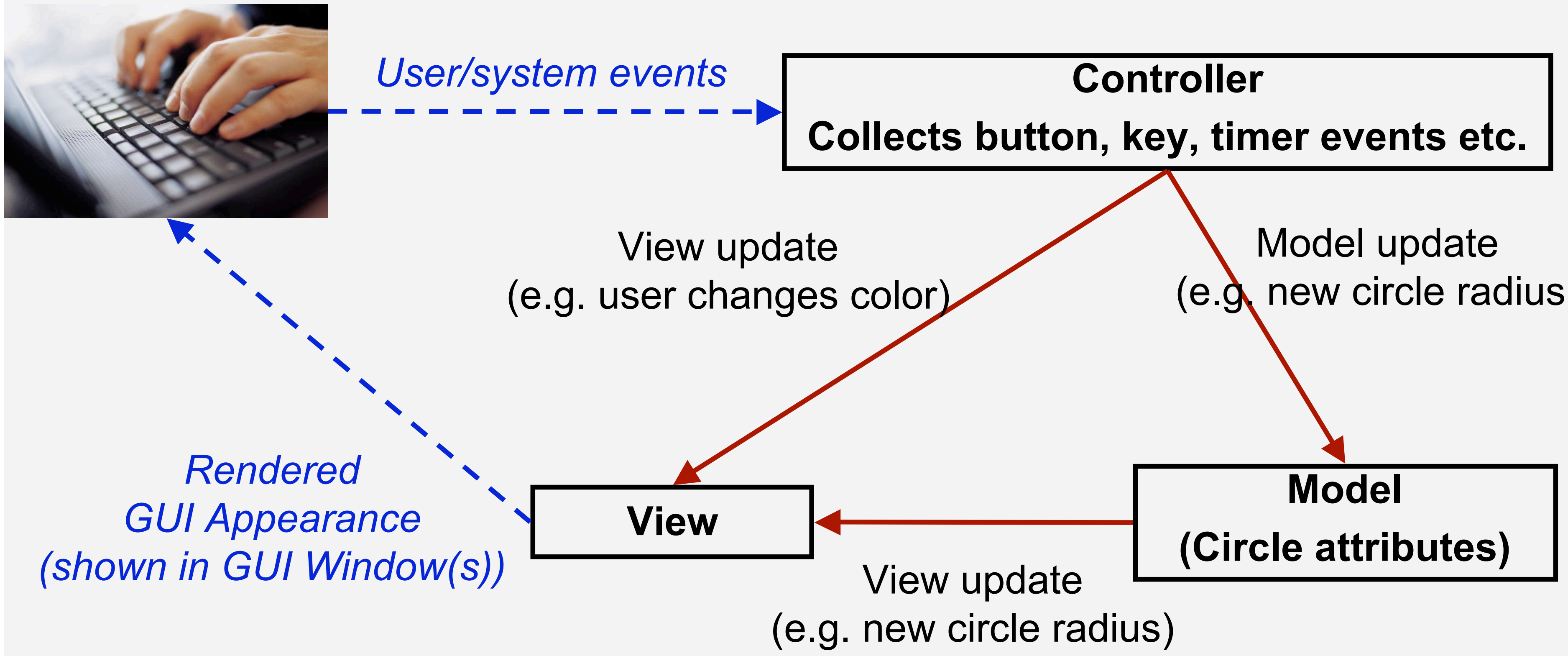
View: visualizes state of the model

- Filtering: what parts of the model are visualized?
- Visualization: how the chosen elements are drawn
- A single model may be used with *multiple* views

Controller: collects user and system events

- Interaction with the view and/or model
  - » e.g. user selects what to show (e.g. selecting a tab)
  - » e.g. user replaces a value in a spreadsheet (updated model as well as view)

MVC Diagram: Manipulating a circle viewed in a GUI



Note: In Java Swing (and many other GUI toolkits), View/Controller are often merged

MVC in Swing

Model

Implemented using an object from a class with event and event listener registration methods

View(s)

Implemented as listeners registered with a model object; notified when the model (state of the “core” program) is updated.

Controller

Implemented using widgets with registered listeners that invoke model and/or view operations as needed.

Model-View-Controller (MVC) Example

CircleModel.java

CircleView.java

CircleController.java

MVCDemo.java

(Liang pages 1008-1014)