Lecture 32

We look at how to write *Graphical User Interfaces (GUIs).*

We see that we have to write some class definitions of our own, but there are numerous useful class definitions in the Java class library that we will also use.

Most importantly, we see that GUIs are *event-driven programs*.

In constructing a simple GUI, we see we need three kinds of objects: *components*, *events* and *listeners*.

Java versions/GUI Toolkit versions

- To write GUIs, need a GUI Toolkit (i.e. library classes):
  - AWT (Abstract Windowing Toolkit): for low functionality, inefficient GUIs with native look-and-feel
  - Swing: for high functionality, more efficient GUIs with pluggable look-and-feel

Java	Development kit	GUI Toolkit
Java 1	JDK 1.0 library of about 200 classes/interfaces	AWT (with inheritance-based event model)
	JDK 1.1 library of about 500 classes/interfaces	AWT (with delegation-based event model) or Swing (downloaded separately)
Java 2	SDK 1.2, 1.3 & 1.4 library of about 3000 classes/interfaces	Swing

Event-driven programming

- GUIs are examples of *event-driven programs*: events cause methods to be executed
- *Components* (widgets, window gadgets, controls): Objects we create which have a screen presence and with which the user interacts (e.g. buttons). When a user interacts with a component, that component is the *event source*
- *Events:* Java automatically creates an event object when the user interacts with a component. The event object has an instance variable recording the event source
- *Listeners:* Objects we create and to which we delegate the task of responding to events. A listener object will have an instance method, called a *handler*, that is invoked by Java to take appropriate action to respond to the event

Components

- Components include buttons, textfields, scrollbars, windows, pop-up menus, menu items, etc.
- E.g. `javax.swing.JButton`
  - Constructor, e.g.:

    ```
    JButton btn = new JButton("Press me");
    ```

  - Instance methods, e.g.:

    ```
    String str = btn.getText();
    btn.setText("Press me harder!");
    ```

    Also `addActionListener` and `removeActionLi`
    — see later
  - If you have, e.g., a window, then the window object will offer methods that allow you to add components (such as buttons) to the window — see the example program

Event objects

- There are different classes of event to represent different user actions (some quite low-level, others quite high-level)
- E.g. `java.awt.event.ActionEvent`
  - Unlikely we would call the constructors
  - Created by Java when, e.g., a user clicks on a button
  - Instance methods for an `ActionEvent ae`:
    * `ae.getSource();` — returns a reference to the event source, i.e. the button/textfield/etc. that generated this event
    * `ae.getActionCommand();` — returns a `String` associated with the source (e.g. the button's text)

Listener objects

- Listener objects are delegated the task of responding to certain events for certain components
- Listener objects contain *handler methods* that run in response to events. Unlikely that we would invoke the handlers ourselves: Java does it for us when events occur
- Listeners can be any class of object we like but must implement certain interfaces to guarantee inclusion of appropriately named handlers
- E.g. we could write a listener for button clicks
  - We would state: `implements ActionListener` (`java.awt.event.ActionListener`)
  - So we must include definition of `actionPerformed` method

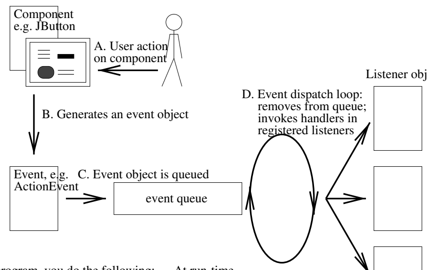
Registering listeners

- How does Java know which listener object will respond to a component's events?
- If a component's events are to be responded to, we must register one or more listeners with the component
- E.g. if `btn` is a `JButton` and `lstnr1` and `lstnr2` are listeners (`implements ActionListener`):

```
btn.addActionListener(lstnr1);
btn.addActionListener(lstnr2);
...
btn.removeActionListener(lstnr2);
```

- When the user interacts with the button, Java invokes the handler methods in each registered listener

A button-click pic



In the program, you do the following:
1. Write a class that creates a window, creates a JButton, adds the JButton to the window & makes the window visible;
2. Write listener classes (implements ActionListener), contains handler method (actionPerformed)
3. Create listener object(s) and register them with the JButton

At run-time,
1. The user clicks on the button
2. Java creates an ActionEvent object
3. Java adds the event object to a queue of events
4. Event dispatch loop: Java repeatedly
   4.1 takes events from the queue
   4.2 determines their listeners
   4.3 invokes each listener's handler method witl the event as a parameter
   4.4. executes each handler method

Exercise: Colour this picture in using wax crayons.

Our first GUI

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

public class SimplePanel
   extends JPanel
{
   public SimplePanel()
   {




   }

   private JButton btn;
```



```
   public static void main(String[] args)
   {  JPanel pnl = new SimplePanel();
      JFrame frm =
            new JFrame("A simple GUI");
      Container contentPane =
                  frm.getContentPane();
      contentPane.add(pnl);
      frm.setBounds(100, 300, 400, 200);
      frm.setVisible(true);
   }
}
```


Our first GUI's listener class

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

public class SimpleButtonListener
   implements ActionListener
{
  public SimpleButtonListener()
  {
  }

  public void actionPerformed(ActionEvent ae
  {


  }
}
```


Observations

- We didn't have to code up any of the low-level details (detection of user interaction, creation of event object, enqueuing of event object, the event dispatch loop)
- A component may have zero, one or more listener objects registered with it
- Our code doesn't send a message to invoke the handler, `actionPerformed`: Java invokes it for us
- Handlers shouldn't involve a lot of computation. Why?