Interfaces, GUIs, and Applets

Course Topics

Elements of the Java Platform

The Java Language

Java Arrays, Objects, Methods

Java's Object Orientation and I/O

▶ *Interfaces, Graphical User Interfaces, and Applets*

Topics this week:

- "In our last episode..."
- OO Highlights
- Abstract Windowing Toolkit (AWT)
- Interfaces
- Examples of Interfaces from various java.awt.* classes
- Event Driven Programming
- A Typical Event Driven Application
- JDK 1.1 Event Model - Different from JDK 1.0
- Java GUI Programs
- Java Enabled - How *browsers* use Java
- Applets
- What Does an Applet Look Like?
- A Bigger Applet
- Assignment for next time

Interfaces, GUIs, and Applets

"In our last episode..."

Overloading

Two or more methods may have the same *name* but take different *arguments* (number, type, order); they have different *signatures*.

Question: *In what way may two method signatures* NOT *differ to achieve overloading*

Garbage Collection

Reclaims the memory used by objects for which no more references exists.

Question: *What does that mean?; Why is it a good thing?*

Input / Output

- Input / Output using Reader / Writer classes from java.io package.
- You can 'stack' different I/O classes to get additional capabilities. e.g., the BufferedReader class.
- Generally, all of the additional xxxReader and yyyWriter classes, e.g., InputStreamReader, BufferedReader, FileWriter, *inherit* the capabilities of the *abstract* parent Reader and Writer classes.
- Input (and Output) of text is different from I/O of Java primitive data.

OO Highlights

Encapsulation

A mousetrap, student, roster, bank account, professor, _____ should contain just what is needed to accomplish the problem task and support the required interface.

Implementation details should *not* be accessible outside of an object: objects should only expose **`public`**ly the methods and fields they are willing to maintain.

Inheritance

An object type that inherits from another can provide additional details (specialization) as well as extended capabilities provided by the more generic parent (**`super`**) class.

Creating **`abstract`** and generic classes and then sub classes that inherit from them can be useful, but ...

- When you are just starting, don't get hung up on creating an elaborate inheritance hierarchy.
- Concentrate on your immediate problem, use inheritance only if it helps you achieve your ultimate goal.
- You might do better to think of designing in terms of **`interfaces`** if you want to maximize re-usability.

Abstract Windowing Toolkit (AWT)

A least common denominator set of GUI components supported by the native windowing environments of the platforms Java was originally built to support: Windows, Unix X Window System, Macintosh

Basic Controls

- Button
- Checkbox, Choice, List - for handling selectable items
- Label
- MenuItem - for presenting menubars
- TextField

Other User Input Controls

- Scrollbar
- TextArea

Custom

- Canvas - for arbitrary drawing

Containers

- Panel, ScrollPane, Dialog
- LayoutManagers

AWT being supplanted by *Swing* in JDK 1.1 and Java 2

Swing provides a superset of GUI capabilities.

Better, faster, more portable

Built on top of AWT

SimpleGUI.java

Here is an example fo a Java application that creates a simple graphical user interface: one button displayed in a top level window with a title and borders.
application

```
/** A very simple example of a Java GUI application
 **/

// AWT components and Event classes used:

import java.awt.Button;
import java.awt.Frame;

public class SimpleGUI {

  private Frame appFrame = null;
  private Button doSomethingButton = null;

  public static void main(String[] args) {

    SimpleGUI app = new SimpleGUI();
    app.run();
  }

  /** Constructor: make a frame to hold the GUI components and the
   ** GUI components themselves.
   **/
  public SimpleGUI() {
    appFrame = new Frame("Simple Java GUI");
    doSomethingButton = new Button("Do Nothing");

    appFrame.add(doSomethingButton);
  }

  /** run the application
   **/
  public void run() {
    appFrame.validate();
    appFrame.setVisible(true);
  }
}
```

Note that, other than displaying the button, this application does *nothing*.

Interfaces

- In Java, an **interface** is a way of defining a template for *some* of the methods that a class *must* implement.
- Interfaces may only contain method *signatures* (and **static final** variables: Java's version of constants.)
  An interface can not contain any instance variables or method definitions.
- Think of an interface as if it were a small contract.
- Classes agree to abide by the contract by saying they **implement** the interface.
- The compiler will make sure that classes that say they **implement** an interface abide by the contract.

Interfaces are similar to abstract classes except ...

- abstract classes can have default implementations of methods
- A Java class can only **`extend`** one class (abstract or otherwise)
- A Java class can **`implement`** many interfaces

Examples of Interfaces from various java.awt.* classes

Interface	**Methods**
ActionListener	actionPerformed()
ItemListener	itemStateChanged()
KeyListener	keyPressed()
	keyReleased()
MouseListener	mouseClicked()
	mouseEntered()
	mousePressed()
WindowListener	windowIconified()
	windowClosed()

Event Driven Programming

Most computer applications being developed today, especially ones with a *Graphical User Interface (GUI)* are based on a programming model known as "*Event Driven*".

The life cycle of these applications all follow the same pattern:

- Create and initialize the various user interface components: menus, buttons, display areas, etc. and register each one's interest in certain events.
- Sit in an infinite loop, wait for events to occur, and dispatch messages to the user interface components that registered interest in the event type.

A Typical Event Driven Application

Somewhere inside most modern applications that use a graphical user interface (and in many other types of applications using the Event Driven model) is what amounts to an *infinite loop*.

The infinite loop may be hidden away inside an application framework you are using.

Whether hidden or not, those infinite loops look something like the following pseudo code:

```
while (true) {

  event = waitForSomethingInterestingToHappen();

  callThingThatRegisteredInterest( event );

}
```

Event Driven Programming is all about writing the methods that are the thingThatRegisteredInterest in handling some event, e.g.,

- methods to call when Buttons are clicked on
- methods to call when Menu items are selected
- Text display that allows you to type, delete, cut, paste, ...
- (Actor, Plane, Ship, Car) that gets ’hit’, shot, attacked, ...

Question: *How is event driven programming different from what we’ve done so far?*

JDK 1.1 Event Model - Different from JDK 1.0

In JDK 1.1 and beyond,

> *Every time the user types a character or pushes a mouse button, an event occurs. Any object can be notified of the event. All it has to do is implement the appropriate* ***interface*** *and be registered as an* ***event listener*** *on the appropriate* ***event source****.*
>
> from: *The Java Tutorial*, Campione & Walrath, 1999

Typical Java GUI Application

```
// create user interface

...

// tell user interface components which objects are interested in things
// that happen to them

...

while (true) {

  letThingsHappen();

  // user interface components will tell interested objects about the
  // happenings

}
```

Java GUI Programs

In Java applications that use a graphical user interface (GUI) or other event driven styles you see lots of:

```
...
Something s = new Something();
...
UserInterfaceComponent c = new UserInterfaceComponent();
c.addSomethingListener(s);
...
```

and, in the implementation of the Something.java class source, you would see

```
public class Something
    implements SomethingListener {
   ...
   public void somethingDone() {
   ...
   }
...
}
```

where *SomethingListener* is one of a number of interfaces and *somethingDone* is a method that the *somethingListener* **interface** defines.

The SomethingListener interface is a contract that classes agree to abide by by saying they implement the interface.

The UserInterfaceComponents don't have code that refers to the SomeThing class (or any other application specific class that you write); they simply treat all such classes as SomethingListener objects.

This allows the UserInterfaceComponent classes to be independent of application specific classes.

JavaGUI2.java

```
/** A very simple example of a Java GUI application that responds
 ** to button events.
 **/

// AWT components and Event classes used:

import java.awt.Button;
import java.awt.Frame;

import java.awt.event.ActionListener;   // interface
import java.awt.event.ActionEvent;      // events that interface relates to

public class SimpleGUI2 implements ActionListener {

  private Frame appFrame = null;
  private Button doSomethingButton = null;

  public static void main(String[] args) {

    SimpleGUI2 app = new SimpleGUI2();
    app.run();
  }

  /** Constructor: make a frame to hold the GUI components and the
   ** GUI components themselves.
   **/
  public SimpleGUI2() {
    appFrame = new Frame("Simple Java GUI");
    doSomethingButton = new Button("Do Something");

    // This object that is being constructed agrees to listen for
    // events the button will generate.
    doSomethingButton.addActionListener(this);

    appFrame.add(doSomethingButton);
  }

  /** run the application
   **/
  public void run() {
    appFrame.validate();
    appFrame.setVisible(true);
  }

  public void actionPerformed(ActionEvent e) {
    System.out.println("You pressed the button");
  }
}
```

Java Enabled - How *browsers* use Java

- References to a Java class is embedded in a web page, much as an image is embedded:

```
<html>
 <head>
  <title>The Hello Class Applet</title>
 </head>
 <body>
  <applet code="HelloClassApplet.class" width=150 height=100>
   Either your browser doesn’t support Java or you haven’t enabled Java.
   </applet>
 </body>
</html>
```

- The JVM your browser uses downloads the class file referenced and starts executing the Java class bytecodes.

Applets

Java code run by a browser must be an *applet*, not an application.

- The browser JVM expects the class mentioned in the web page to

```
extend java.applet.Applet
```

- java.applet.Applet has four methods the browser calls to control the applet:
  - public void init() - Called by the browser to inform the applet that it has been loaded into the system. initializes the applet
  - public void start() - Called by the browser to inform the applet that it should start its execution.
  - public void stop() - Called by the browser to inform the applet that it should stop its execution.
  - public void destroy() - Called by the browser to inform the applet that it is being reclaimed and that it should destroy any resources that it has allocated.

  These are referred to as the applet *life-cycle* methods

- The applet class does not have a main method; if it has one it is ignored by the browser.
- Instead of starting with the class's main method, browser JVM's start by calling the referenced applet's *init* method.
- As bytecodes in the currently executing class reference other Java classes, those classes are downloaded or accessed, and the JVM executes them.

What Does an Applet Look Like?

```
/** A very simple expamle of a Java GUI applet
 **/

// AWT components and Event classes used:

import java.applet.Applet;
import java.awt.Button;
import java.awt.Frame;

import java.awt.event.ActionListener;   // interface
import java.awt.event.ActionEvent;      // events that interface relates to

public class SimpleApplet extends Applet implements ActionListener {

  // we don't need a frame; the applet itself is a kind of
  // (extends a) GUI component container called a Panel.
  private Button doSomethingButton = null;

  /** Constructor: make a frame to hold the GUI components and the
   ** GUI components themselves.
   **/
  public void init() {
    doSomethingButton = new Button("Do Something");

    // This object that is being constructed agrees to listen for
    // events the button will generate.
    doSomethingButton.addActionListener(this);

    // add the button to the applet
    add(doSomethingButton);
  }

  /** run the application
   **/
  public void start() {
    // the browser will make the components visible
  }

  public void stop() {
  }

  public void destroy() {
  }

  public void actionPerformed(ActionEvent e) {
    // Where do you think thhis output will appear?
    System.out.println("You pressed the button");
  }
}
```

Either your browser doesn’t support Java or you haven’t enabled it.

A Bigger Applet

```
/** Adapted from Java in a Nutshell
 **/

import java.applet.Applet;
import java.awt.Button;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;

public class ScribbleApplet extends Applet
  implements ActionListener, MouseListener, MouseMotionListener {

  private int last_x, last_y;

  public void init() {
    addMouseListener(this);
    addMouseMotionListener(this);

    // Create a clear button
    Button b = new Button("Clear");

    // Tell the button which object (this applet) has the
    // actionPerformed() method to call when button is clicked.
    b.addActionListener(this);
    add(b);
  }

  /** From ActionListener interface: clear the applet area when
   ** button is pressed
   **/
  public void actionPerformed( ActionEvent e) {
    Graphics g = getGraphics();
    g.setColor(getBackground());
    Dimension d = getSize();
    g.fillRect(0, 0, d.width, d.height);
  }

  /** From MouseListener interface: remember where mouse was ehen
   ** mouse button is pressed.
   **/
  public void mousePressed( MouseEvent e) {
    last_x = e.getX();
    last_y = e.getY();
  }

  /** From MouseMotionListener interface: determine where mouse is
   ** while button is down; draw a line from last position to this
   ** one, remember where mouse is fo next time.
   **/
  public void mouseDragged(MouseEvent e) {
    Graphics g = getGraphics();
    int x = e.getX();
    int y = e.getY();
    g.drawLine(last_x, last_y, x, y);
    last_x = x;
    last_y = y;
  }

  /** Both MouseListener and MouseMotionListener interfaces define
   ** other methods that must be implemented.  These do nothing
   ** implementations satisfy the MouseListener interface
   ** "contract".
   **/
  public void mouseReleased(MouseEvent e) {
  }
  public void mouseClicked(MouseEvent e) {
  }
  public void mouseEntered(MouseEvent e) {
  }
  public void mouseExited(MouseEvent e) {
  }
  /** And this do nothing implementation satisfies the
   ** MouseMotionListener interface "contract".
   **/
  public void mouseMoved(MouseEvent e) {
  }

  }
```

Either your browser doesn’t support Java or you haven’t enabled it.

Assignment for next time

Review

- Textbook
  - Chapter 1 - Introduction and a Taste of Java
  - Chapter 2 - Primitive Types and Strings
  - Chapter 3 - Flow of Control
  - Chapter 4 - Classes, Object, and Methods
  - Chapter 5 - Programming with Classes and Methods
  - Chapter 6 - Inheritance - through the section "Constructors in Derived Classes" that ends on page 301 (you may skip the remainder of Chapter 6)
  - Chapter 7 - Event Driven Programming Using the AWT
  - Chapter 9 - Streams and File I/O - Up to but not including the Programming Exampel that starts at the bottom of page 495.
- Class Notes
- Have questions

Next week will be a review session - on *Monday, 29 November*

Final Exam will be in two weeks, on Monday, 6 December