CS 121 - Intro to Programming - Lecture 23

Announcements

CS 187 - now two discs on Wed, 9:05, and 10:10
Program 7 due next Wednesday
4/16 - embedded 11 due
4/22 OWL assignment #11 due
4/23 - embedded 12 due
4/28 - embedded 13 due
Final - May 11, 130, Totman
Wed is Mon
The concept of an interface:

• A mechanism for specification

• We’ve seen them before: Java API

• Encapsulation: working on a “need-to-know” basis
• An interface is a kind of contract

• Big principles -> reuse; hygiene
Classic example of an interface at work

• An editor’s copy & paste feature

• You almost surely only understand its functionality - that is, its interface

• Its implementation is opaque

• You don’t need to know how it’s implemented

• You would rather not know how it’s implemented

• An implementor may alter the implementation - and you would never know
Interfaces also solve the “generalized sorting problem”: How can we sort

An array of Infants (by age)
An array of Strings
An array of words by frequency

With a single method?

```java
Arrays.sort(kids);
Arrays.sort(someStrings);
Arrays.sort(words);
```
Answer: we handled this using the library interface Comparable, with its single method

```java
compareTo(Object obj)
```

We can also write our own Interfaces...
Here is the “Scoring” interface - it’s just two methods -- and not even that: the methods involved are “disembodied”

```java
public interface Scoring{
    public double getScore();
    public void setScore(double newScore);
}
```
Here is the “Scoring” interface - it’s just two methods -- and not even that: the methods involved are “disembodied”

```java
public interface Scoring{
    public double getScore();
    public void setScore(double newScore);
}
```

Note the syntax:
Which classes might implement Scoring?

Baseball players - runs scored
Golfers - money earned or shots taken
Employees - days arriving on time
Union members - seniority

And here’s how we could exploit Scoring...
public class Scorefns {
    // has methods that exploit Scoring interface

    public static int scoreMax(Scoring[] theArray) {
        // returns array position of highest score ele
        int highPos = 0;
        for (int j = 1; j < theArray.length; j++) {
            if (theArray[j].getScore() > theArray[highPos].getScore()) {
                highPos = j;
            }
        }
        return highPos;
    }
}
A related idea...

An Abstract Class...

Interface  Abstract class  (Concrete)class
Another notion: an **abstract class**

At one extreme -> full-blown, concrete classes
Other extreme -> interfaces: everything disembodied

**In the middle**: abstract classes: classes that

1) Cannot be instantiated

2) Generally have at least one method marked “abstract”

3) You make “real” classes out of them by extending them, providing bodies for the abstract methods.
One way to think of an abstract class:

it’s a fancy sound system, all ready to go, except that the component that provides the sound isn’t there, although the wire to the sound component (an iPod?) is ready to be plugged in.

You build a “concrete” system by extending what you have - you add a sound source.
Recall the machinery for the Echo class, which addresses file io (reading line-oriented data from an external text file)

The method `processLine` was a sort of “blank”, which we filled in via overriding...

Using the abstract class mechanism, we can make `processLine` a true blank or “empty” component..
import java.util.Scanner;
import java.io.*;

public abstract class LineReader{

String fileName; // external file name
Scanner scan; // for reading from file

public LineReader(String f) throws IOException{
    fileName = f;
    scan = new Scanner(new FileReader(fileName));
}
}
public void readLines()
{
    while(scan.hasNext()){
        processLineLine(scan.nextLine());
    }
    scan.close();
}

public abstract void processLineLine(String line);
import java.io.*;
import java.util.*;

public class NewEcho extends LineReader {

    public NewEcho(String f) throws IOException {
        super(f);
    }

    public void processLine(String line) {
        System.out.println(line);
    }
}

import java.util.*;
import java.io.*;
public class LineDriver{
    public static void main(String[] args){
        try{
            Scanner scan = new Scanner(System.in);
            System.out.println("Enter name of text file");
            String fileName = scan.next();
            NewEcho r = new NewEcho(fileName);
            r.readLine();
        }
        catch(Exception e){e.printStackTrace();}    }
}
You cannot instantiate LineReader; you can only instantiate concrete derived classes, such as NewEcho
public abstract class JobTimer {

public abstract void doJob();
    // does some job, to be named later in a subclass

public void runJob() {
    // garbage collector to make more memory available
    System.gc();
    long s1 = System.currentTimeMillis();
    doJob();
    doJob();
    long s2 = System.currentTimeMillis();
    long runTime = (s2 - s1);
    System.out.println("running time in milliseconds: " +
                        runTime);
}
}
public class AddTimer extends JobTimer {
    //how many times operation performed
    private long numOperations = 10000000;
    //implementation of the abstract method

    public void doJob(){
        long k = 0;
        int result = 0, operand = 12345;
        while(k<numOperations){
            result = operand + operand;
            k++;
        }
    }
}
public static void main(String[] args){
    AddTimer a = new AddTimer();
    a.runJob();
}

running time in milliseconds: 186
Graphics: our next/last big topic. Fun, dramatic, useful, etc.

Graphical Opoly

Elements

• program execution advances by clicking
• also by menu selection
• picture updated after every click
• labels updated after every click
A **JFrame** is like a window frame.

It comes with a content pane, a container that holds things (surfaces..)

We’ll place on the surface something we can draw on /write on / add hardware to: a **JPanel**

Then we do just that: we write on it, draw on it, color it.
JFrame (frame)

cContentPane

JPanel (panel - is “affixed” to contentPane)
import java.awt.*;
import javax.swing.*;

public class FirstGraphics{

    public static void main(String[] args){
        JFrame frame = new JFrame("Getting Started");
        Container c = frame.getContentPane();
        BabyGeoPanel p = new BabyGeoPanel(Color.green);
        c.add(p); // add panel to frame's container
        frame.pack(); // prepares frame for display
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

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

public class BabyGeoPanel extends JPanel{

    public BabyGeoPanel(Color g){
        setPreferredSize(new Dimension(700,300));
        setBackground(g);
    }

    public void paintComponent(Graphics g){
        super.paintComponent(g);
        g.setColor(Color.red);
        g.fillRect(10,20,100,100);
        g.setColor(Color.blue);
        g.drawOval(200,20,300,100);
        g.setColor(Color.black);
        g.drawRect(200,20,300,100);
        g.drawLine(0,0,200,20);
    }
}

increasing $x$

- $(10, 30)$

increasing $y$

- $(22, 105)$
Two big themes, revisited

Factor out the general framework, use inheritance (good news)

“Write once, run everywhere” principle in Java starts to break down when graphics gets complicated (bad news)
Inheritance and the frame/panel machinery

We’ll split off the frame code and make a general purpose “display” window.

We’ll develop a simple, general mechanism for adding panels to the code

This will shift the work of GUI development to a panel or panels
import java.awt.*; import javax.swing.*;
public class DisplayWindow extends JFrame{
   private Container c;
   public DisplayWindow(){
      super("Display"); c = this.getContentPane(); }
   
   public void addPanel(JPanel p){
      p.setPreferredSize(new Dimension(500,400));
      c.add(p); }
   
   public void showFrame(){
      this.pack();
      this.setVisible(true);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```java
import java.awt.*; import javax.swing.*;
import java.util.*;

public class BabyGraphics2{

    public static void main(String[] args){
        DisplayWindow d = new DisplayWindow();
        PositionPanel p = new PositionPanel(Color.red);

        Scanner s = new Scanner(System.in);
        int x = s.nextInt(); int y = s.nextInt();
        p.setPt(x,y);
        d.add(p);
        d.showFrame();
    }
}
```
import java.awt.*;
import javax.swing.*;

public class PositionPanel extends JPanel{

    private int x, y;
    private Color c;

    public PositionPanel(Color c){
        setPreferredSize(new Dimension(500,500));
        this.c = c;
    }

    public void setPt(int a, int b){x =a; y = b;}
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(c);
    g.drawOval(x, y, 2, 2);
    g.setColor(Color.black);
    String s = "\(x+y\)";
    g.drawString(s, x, y);
}
}
Java’s event model

What Scanner-based interactions are like.. (a script)

What modern computing is like..

Your program needs to know:

1) What event to listen for
2) Clarify who’s listening
3) Provide a mechanism for event reaction