Interface Types & Polymorphism

& introduction to graphics programming in Java
Interfaces in Java

• We often refer to the set of public methods in a class as the class interface:
  – Consider the term “user interface” – this is the set of tools by which a user interacts with an operating system and application set, and hence a computer
  – The public methods form the interface for a class: this is the set of tools by which a client can interact with an object
Interfaces in Java

• In Java, the term “interface” is also used to refer to a data type that includes a set of method signatures without implementation

• Looks like a C++ class

• Java’s way of doing multiple inheritance; a class can only extend one parent class, but can implement multiple interfaces
Polymorphism

• Because Java interfaces specify, but do not implement methods, the implementing classes must provide method bodies
  – This means several different classes might contain methods with the same signatures
  – Thus several *objects of different types* could respond to the *same messages*
  – This behavior is known as *polymorphism*
Implementing an interface: generic syntax

public class MyClass implements SomeInterface, SomeOtherInterface {

    // implementations for all methods specified in
    // SomeInterface and SomeOtherInterface must
    // be provided here; can be as simple as:

    public void aMethod() { }

    // can also have additional methods not specified
    // by the interface(s)

}
Example of an Interface: javax.swing.Icon

public interface Icon
{
    public int getIconHeight();
    public int getIconWidth();
    public void paintIcon(Component c, Graphics g, int x, int y);
}

To implement this interface, a class would:
  - have *implements Icon* at the end of its heading
  - implement the 3 functions specified in the interface

An example of an implementation appears on the next slide
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class HolyIconBatman implements Icon {
    private int iSize;
    public HolyIconBatman (int mySize) {
        iSize = mySize;
    }

    public int getIconWidth() {return iSize;}
    public int getIconHeight() {return (int)(iSize * (2.0/3.0));}
    public void paintIcon(Component c, Graphics g, int x, int y) {
        Graphics2D context = (Graphics2D)g;
        Ellipse2D.Double background = new Ellipse2D.Double(x, y, iSize, iSize*(2.0/3.0));
        context.setColor(Color.BLACK);
        context.fill(background);
        context.setColor(Color.YELLOW);
        context.drawString("R",x+17,y+18);
    }
}

Interface types

• There is no such thing as an Icon object, since Icon is not a class, it’s an interface
• However, you can find many examples in the Java API of methods that take Icon parameters
• To call such a method, you need an argument object that implements the Icon interface – such as the one on the previous slide
Interfaces as parameters

• The static method `showMessageDialog()` of the `JOptionPane` class (part of `javax.swing`) is an example of a method that expects an `Icon` as one of its arguments.

• The method displays a window with:
  – A message
  – An (optional) icon
  – An OK button
JOptionPane.showMessageDialog()

• The method signature is:
  public static void showMessageDialog (
      Component parent, Object message,
      String title, int messageType, Icon anIcon)
  • parent is usually either some kind of window (e.g. JFrame or JPanel) or may be null;
  • message is usually a String (although it could be some other kind of object, or even null)
  • title is the window title
  • messageType is an int value indicating a message type; actual argument is usually one of the constants defined for this purpose in the JOptionPane class
  • anIcon is an instance of a class that implements Icon
Example

public class BatCave {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Holy icon Batman!", "BatCave",
        JOptionPane.INFORMATION_MESSAGE,
        new HolyIconBatman(40));
    }
}

OUTPUT:

Note that the Icon object passed to showMessageDialog in main is constructed on the fly and unnamed – this is an example of an anonymous object (more on this later)
The ImageIcon class

- Although an Icon, in and of itself, cannot be constructed, the API contains a class, ImageIcon, which implements the Icon interface
- An ImageIcon object can incorporate an existing picture (gif or jpg), rather than drawing an original
- The next example illustrates this
public class HiClass {
    public static void main(String[] args) {
        ImageIcon icon = new ImageIcon("c:/course notes/pdd/sp2007/InterfacesNPolymorphism/me2.gif");
        JOptionPane.showMessageDialog(null, "Hi Class!", "My Very Own Window", JOptionPane.INFORMATION_MESSAGE, icon);
    }
}
Polymorphism in Action

• Both code examples employ calls to showMessageDialog, which displays:
  – An icon
  – A message
  – An OK button

• showMessageDialog must compute size of dialog:
  \[ \text{width} = \text{icon width} + \text{message size} + \text{blank size} \]

• How do we know the icon width?
  \[ \text{int width} = \text{anIcon}.\text{getIconWidth}(); \]
Polymorphism in action

- `showMessageDialog` doesn't know which icon is passed; could be `ImageIcon`, `HolyIconBatman`, or some other such class
- The actual type of parameter `anIcon` is not `Icon`; there are no objects of type `Icon`
- `anIcon` belongs to a class that implements `Icon`
- That class defines a `getIconWidth` method, which `showMessageDialog` calls to help determine the size of its window
Polymorphism in action

• If a class implements an interface type, its objects can be assigned to variables of the interface type
• There’s no such thing as an interface object, but interface parameters (and other variables) are possible
• Such a variable contains a reference to an object whose class implements the interface
Benefits of Polymorphism

• Polymorphism refers to the ability to select different methods according to the actual type of an object

• Provides for *loose coupling*:
  – `showMessageDialog` decoupled from `ImageIcon`
  – Doesn't need to know about image processing

• Polymorphism also provides *extensibility* - for example, client programmer can supply new icon types
Drawing Shapes

• The paintIcon method of the Icon interface is responsible for drawing the icon

• One of the parameters to the method is a graphics context of type Graphics:
  – Carries out drawing operations
  – Includes methods for drawing shapes, changing colors and fonts
In case last semester is only a dim memory …

• In CS1, we looked at drawing various shapes and using layout managers, as well as event-driven programming and GUIs in general
• The current textbook uses a slightly different approach, with Graphics2D as the main drawing class instead of Graphics
• Either approach is valid for most applications; I am following the current text in these notes for continuity with other material, but I have posted the GUI notes from CS1 on the web site for your reference
Drawing shapes

- Graphics2D class provides more powerful drawing operations, but most API methods still use Graphics parameters for historical reasons.
- Since Graphics2D is a child class of Graphics, a Graphics2D object can be passed as a Graphics parameter.
- Alternatively, a Graphics object can be cast as a Graphics2D, as was done in the HolyIconBatman example:
  
  ```java
  Graphics2D context = (Graphics2D)g;
  ```
Drawing shapes

- Drawable objects are objects of those classes that implement the Shape interface
- Such objects include rectangles, ellipses and line segments
Drawing Rectangles

• Rectangle2D.Double constructed with
  – top left corner (x,y coordinates)
  – width
  – height

• Example: draws hollow rectangle using current foreground color
  Graphics2D g2;
  Shape rectangle = new Rectangle2D(x,y,w,h);
  g2.draw(rectangle);
Drawing Ellipses

- Can draw Ellipse2D objects to represent ellipses (including circles)
- Need to specify bounding box: rectangle in which ellipse appears – so constructor arguments are the same as Rectangle2D:
  
  ```java
  Shape ellipse = new Ellipse2D.Double(x,y,w,h);
  g2.draw(ellipse);
  ```
Drawing Ellipses
Drawing Solid Shapes

• The Graphics2D draw method draws the outline of the requested shape in the current color
• Can draw a solid rectangle or ellipse by calling the fill method instead
• Specify color by calling the setColor method on the graphics context
Colors in Java

- There are 13 predefined colors in the Java Color class.
- In addition, colors can be constructed using 3 integer arguments in the range 0-255, indicating the amount of saturation of red, green and blue color components.
- The predefined colors are: black, blue, cyan, dark gray, gray, green, light gray, magenta, orange, pink, red, white and yellow.
Drawing Lines

• Construct an object of type Line2D.Double using two Point2D.Double objects
  – Point2D.Double is a point in the plane
  – Line2D.Double joins 2 points

• Example:
  Point2D.Double end1 = new Point2D.Double(x1,y1);
  Point2D.Double end2 = new Point2D.Double(x2,y2);
  Shape line = new Line2D.Double(end1, end2);
  g2.draw(line);
Relationship between shape classes
Drawing text

• The Graphics2D drawString method draws a text string using the current font and foreground color; arguments specify:
  – Text to be drawn
  – Base point coordinates (x and y)