Interfaces & Polymorphism
part 2:

Collections, Comparators, and
More fun with Java graphics
Collections (from the Java tutorial)*

• A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit
• Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another
• Collections typically represent data items that form a natural group

Java’s Collections class

• Consists of a set of methods that operate on collections (which are classes that implement the Collection interface or one of its descendants or their implementing classes, such as ArrayList)

• One of these methods is static method sort

• The sort method works on objects that implement the Comparable interface
Comparable interface

```java
public interface Comparable {
    int compareTo(Object other);
}
```

Suppose object1 is an object of a class that implements Comparable; then the message `object1.compareTo(object2)` should return:

- a negative number if object1<object2
- zero if they are equal
- a positive number (if object1>object2)
Using Collections.sort

• The String class implements the Comparable interface; thus, to sort an array of Strings:

  ```java
  ArrayList words = new ArrayList();
  words.add("fossil");
  words.add("burgeoning");
  words.add("aspersin");
  Collections.sort(words);
  ```
Using Collections.sort

• If you have an ArrayList of objects of a new class that you wish to sort, that class should implement the Comparable interface; otherwise, Collections.sort throws an exception

• Your class should:
  – have `implements Comparable` in its heading
  – implement the compareTo method
The Comparator interface

- Objects of a class that implements Comparable can be sorted using Collections.sort() based on the implementing class’s compareTo method.
- If more than one type of sorting is desired, it is more convenient to implement the Comparator interface rather than the Comparable interface.
- An alternative version of Collections.sort() works with implementors of Comparator.
Comparator

public interface Comparator
{
    int compare(Object a, Object b);
}

Return values from compare() are as follows:
• returns a negative number if a < b
• returns 0 if a == b
• returns a positive number if a > b
Using sort() with Comparator

- The sort() method that works with the Comparator interface expects two arguments:
  - a list to be sorted (e.g. an ArrayList - a descendant of List, which is a descendant of Collections)
  - a Comparator (in other words, an object of an implementing class)
Implementing Comparator

• A key fact about implementing Comparator: the class that does the implementing need not be the basis for the objects to be sorted
• To create multiple sorting methods, can create multiple Comparator classes, each of which implements compare() in a different way
• The next several slides exemplify these ideas
public class Thing implements Comparable
{
    private String name;
    private int size;
    public Thing (String aName, int aSize) {
        name = aName;
        size = aSize;
    }
    public String getName()    {
        return name;
    }
    public double getSize()    {
        return size;
    }
    public int compareTo(Object otherObject)     {
        Thing other = (Thing) otherObject;
        if (this.size < other.size) return -1;
        if (this.size > other.size) return 1;
        return 0;
    }
}

Implementation of compareTo is based on the value of size; no way to compare two Things based on name (or any other attribute we might define for a Thing)
import java.util.*;
public class ThingByName implements Comparator
{
    public int compare(Object object1, Object object2)
    {
        Thing t1 = (Thing) object1;
        Thing t2 = (Thing) object2;
        return t1.getName().compareTo(t2.getName());
    }
}
import java.util.*;

public class ThingCompTest {
    public static void main(String[] args) {
        ArrayList bunchOStuff = new ArrayList();
bunchOStuff.add(new Thing("ambergris", 87));
bunchOStuff.add(new Thing("gummy bear", 4));
bunchOStuff.add(new Thing("Belgium", 30510));
Comparator comp = new ThingByName();
Collections.sort(bunchOStuff, comp);
System.out.println("things sorted by name:");
for (int i = 0; i < bunchOStuff.size(); i++) {
    Thing t = (Thing) bunchOStuff.get(i);
    System.out.println(t.getName() + " " + t.getSize());
}
Collections.sort(bunchOStuff);
System.out.println("things sorted by size:");
for (int i = 0; i < bunchOStuff.size(); i++) {
    Thing t = (Thing) bunchOStuff.get(i);
    System.out.println(t.getName() + " " + t.getSize());
}
}
}

Exhibit C: a test class, illustrating both sorting methods

OUTPUT:

things sorted by name:
Belgium 30510.0
ambergris 87.0
gummy bear 4.0

things sorted by size:
gummy bear 4.0
ambergris 87.0
Belgium 30510.0
Anonymous objects

• An anonymous object is one that is created on the fly in a method call; we saw an example of this in the HolyIconBatman class:

```java
JOptionPane.showMessageDialog(null, "Holy icon Batman!", "BatCave",
    JOptionPane.INFORMATION_MESSAGE, new HolyIconBatman(40));
```

• An anonymous object is simply an object reference that is not stored in a variable
Anonymous objects

• In the ThingCompTest class, the following lines of code:

  Comparator comp = new ThingByName();
  Collections.sort(bunchOStuff, comp);

could be replaced by a single line, using an anonymous object instead of variable comp:

  Collections.sort(bunchOStuff, new ThingByName());
Anonymous classes

• Anonymous objects are essentially literal values; since they aren’t stored anywhere, they can be used once, but would have to be reconstructed to be used again
• Like an anonymous object, an anonymous class is an unnamed class, defined on the fly
• When defining an anonymous class, you must also construct an anonymous object
Anonymous classes

• No need to name objects that are used only once:
  Collections.sort(bunchOStuff, new ThingByName());
• By the same taken, no need to name classes that will only be used once
• The code below creates an anonymous Comparator class to compare things (could use this instead of creating class ThingByName)

    Comparator comp = new Comparator() {
        public int compare(Object obj1, Object obj2) {
            Thing t1 = (Thing)obj1;
            Thing t2 = (Thing)obj2;
            return t1.getName().compareTo(t2.getName());
        }
    };

    NOTE THE SEMICOLON
Anonymous classes

• The expression:
  
  ```java
  Comparator comp = new Comparator(){ … };
  ```
  
  – defines a class that implements the Comparator interface type
  
  – defines method compare()
  
  – constructs an object of the new (unnamed) class

• An anonymous class is a special case of an inner class (a class defined inside another)
Anonymous classes

• Commonly used in factory methods:

```java
public class Thing { …
    public static Comparator ThingByName() {
        return new Comparator() {
            public int compare(Object o1, Object o2) { . . . }
        }
    }
}
```

• Eliminates need to create separate class whose only purpose is to facilitate sorting of primary class
Anonymous classes

- Can now sort ArrayList a of Thing objects by calling:
  \[
  \text{Collections.sort(a, Thing.ThingByName());}
  \]
- Neat arrangement if multiple comparators make sense (by name, by size, by ranking, etc.)
- Gives both types of comparison equal preference, rather than arbitrarily choosing one to implement using compareTo()
Frames

• We have already seen limited use of graphics programming associated with methods of the JOptionPane class

• A JOptionPane object is one type of GUI window; the generic window type is the frame

• Frame window has decorations:
  – title bar
  – borders
  – close box
Constructing and displaying a frame window

JFrame frame = new JFrame();
frame.pack(); // automatically sizes window
frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
// exits program when window closes
frame.setVisible(true); // displays frame

Note: requires import statements:
    import java.awt.*;
    import javax.swing.*;
Can also set size of frame using frame.setSize(w, h);
Drawing in a frame

- The part of the frame below the title bar and between the borders is the content pane of the window; you can add components to this by assigning it to a Container object, then adding elements to this object.
- One such element that can be added is a JLabel object, which can be used for display of text or images (or both).
- The next slide illustrates this.
import java.awt.*;
import javax.swing.*;

public class Me2 {
    public static void main (String[] args) {
        JFrame frame = new JFrame();
        ImageIcon pic1 = new ImageIcon("me2.gif");
        JLabel picture1 = new JLabel(pic1);
        ImageIcon pic2 = new ImageIcon("3x2x2connector.jpg");
        JLabel picture2 = new JLabel(pic2);
        Container windowContents = frame.getContentPane();
        windowContents.setLayout(new FlowLayout());
        windowContents.add(picture1);
        windowContents.add(picture2);
        frame.pack();
        frame setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
Notes on example

• As before, an Icon interface variable holds an ImageIcon reference, which is constructed using the name of a graphics file:
  
  ```java
  Icon pic1 = new ImageIcon("me2.gif");
  ```

• The JLabel objects picture1 and picture2 are constructed with icons inside them:
  
  ```java
  JLabel picture2 = new JLabel(pic2);
  ```

• These objects are then placed in the JFrame’s content pane and displayed
More notes on example

• Note the line:

  contentPane.setLayout(new FlowLayout());

  – this generates a **layout manager** to control how the multiple components will line up within the content pane

  – a FlowLayout lines up components side by side
Adding user interface components

• User interface components such as buttons (instances of JButton) and text fields (instances of JTextField) can also be added to the content pane of a frame
• The FrameTest example (next slide) illustrates this
import java.awt.*;
import javax.swing.*;

public class FrameTest {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        JButton helloButton = new JButton("Say Hello");
        JButton goodbyeButton = new JButton("Say Goodbye");
        final int FIELD_WIDTH = 20;
        JTextField textField = new JTextField(FIELD_WIDTH);
        textField.setText("Click a button!");
        Container contentPane = frame.getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(helloButton);
        contentPane.add(goodbyeButton);
        contentPane.add(textField);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
User interface components

• The FrameTest program displays a window that looks like this:

  ![FrameTest Window]

• But the buttons don’t do anything; the window merely displays them.

• To make the buttons active, we need to incorporate ActionListeners.
ActionListener interface

• An ActionListener object is one that implements the ActionListener interface and its single method, actionPerformed

• The actionPerformed method requires an ActionEvent parameter; such a parameter is supplied by a user action, such as a mouse click
Making buttons work

• Construct a listener object and add it to the button; this can be accomplished using an anonymous class:

```java
helloButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        textField.setText("hello world!");
    }
});
```
Accessing variables from enclosing scope

- The anonymous class constructed on the previous slide is, as previously noted, an example of an inner class.
- Note that the inner class accesses the textField object, which is a variable of the outer class.
- When a local variable of an enclosing scope is to be accessed by an inner class, the variable must be declared final.
Constructing multiple instances of anonymous ActionListener

- Write helper method that constructs objects
- Pass variable information as parameters
- Declare parameters final

public static ActionListener createGreetingButtonListener(final String message)
{
    return new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            textField.setText(message);
        }
    };
}
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ActionTest {
    private static JTextField textField;

    public static void main(String[] args){
        JFrame frame = new JFrame();
        final int FIELD_WIDTH = 20;
        textField = new JTextField(FIELD_WIDTH);
        textField.setText("Click a button!");
        JButton helloButton = new JButton("Say Hello");
        helloButton.addActionListener(createGreetingButtonListener("Hello, World!"));
        JButton goodbyeButton = new JButton("Say Goodbye");
        goodbyeButton.addActionListener(createGreetingButtonListener("Goodbye, World!"));  
        Container contentPane = frame.getContentPane();
        contentPane.setLayout(new FlowLayout());
    }

    private static ActionListener createGreetingButtonListener(String greeting) {
        return e -> textField.setText(greeting);
    }
}
contentPane.add(helloButton);
contentPane.add(goodbyeButton);
contentPane.add(textField);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.pack();
frame.setVisible(true);
}

public static ActionListener createGreetingButtonListener(final String message) {
    return new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            textField.setText(message);
        }
    };
} // ends definition of new ActionListener (& return statement)
} // ends createGreetingButtonListener method
} // ends class