Beyond Interfaces - Delegation & Multiple Inheritance in Java

- There are many widgets whose job is to show a value from a range of integer values, like progress bars, sliders, scrollbars, etc.
- The common functionality of storing one value from a specified range is described by an interface:

```java
public interface BoundedRangeInterace {
    int getValue();
    int getMinimum();
    int getMaximum();
    void setValue(int newValue);    // lots of more methods
}
```
- The widgets that need to use this are JSlider and JProgressBar. However, they do not implement the interface, though they provide all the methods.
- Instead, there is a DefaultBoundedRangeModel class which implements this interface.
public class DefaultBoundedRangeModel implements BoundedRangeInterface, Serializable
{
    private int value = 0;
    private int min = 0;
    private int max = 100;

    public int getValue() {
        return value;
    }

    public int getMinimum() {
        return min;
    }

    public int getMaximum() {
        return max;
    }
}

Both classes JSlider and JProgressBar have an instance of the BoundedRangeModel class as a member. This lets them use the methods as implemented in the DefaultBoundedModelClass.
Delegation as Alternative to Inheritance

```java
public class JSlider extends JComponent implements SwingConstants, Accessible {
    protected BoundedRangeModel sliderModel;

    public JSlider(int orientation, int min, int max, int value) {
        checkOrientation(orientation);
        this.orientation = orientation;
        sliderModel = new DefaultBoundedRangeModel(value, 0, min, max);
        sliderModel.addChangeListener(changeListener);
        updateUI();
    }

    public BoundedRangeModel getModel() { return sliderModel; }

    public int getValue() { return getModel().getValue(); }

    public void setValue(int n) {
        BoundedRangeModel m = getModel();
        int oldValue = m.getValue();
        m.setValue(n); // more stuff, to update slider
    }

    public int getMinimum() { return getModel().getMinimum(); }
}
```

Matthew Morgenstern
Similar use of Delegation in JProgressBar

```java
public class JProgressBar extends JComponent implements SwingConstants, Accessible {
    protected BoundedRangeModel model;
    public JProgressBar(int orient, int min, int max) {
        setModel(new DefaultBoundedRangeModel(min, 0, min, max));
        updateUI();
    }
    public JProgressBar(BoundedRangeModel newModel) {
        setModel(newModel);
        updateUI();
    }
    public BoundedRangeModel getModel() { return model; }
    public int getValue() { return getModel().getValue(); }
    public int getMinimum() { return getModel().getMinimum(); }
    public int getMaximum() { return getModel().getMaximum(); }
    public void setValue(int n) {
        BoundedRangeModel brm = getModel();
        int oldValue = brm.getValue();
        brm.setValue(n);
    }
}
```
Window Events

Window events are associated with:
- Iconifying / Deiconifying a window (Minimizing a window)
- Activating / Deactivating a window (Selecting a window by clicking inside)
- Closing a window (Typically to exit program)
  ... and others.

We'd like to be able write listeners for these events when needed.

The listeners are described by the interface WindowListener:

```java
public interface WindowListener extends EventListener {
    // Invoked the first time a window is made visible:
    public void windowOpened(WindowEvent e);
    // Invoked when a window has been closed:
    public void windowClosed(WindowEvent e);
    // Invoked when a window is minimized:
    public void windowIconified(WindowEvent e);
    // Invoked when a window is expanded to a normal state:
    public void windowDeiconified(WindowEvent e);  //......... etc.
}
```
Adapters - Another Approach for Inheritance

- Provides ad hoc selective adaptation of each class instance created
- We don't want to implement this interface each time we write a class for a window. The solution: Sun provides an (abstract) class windowAdapter, that implements the interface WindowListener.

```java
public abstract class WindowAdapter
    extends Object
    implements WindowListener
```

- An abstract adapter class for receiving window events..
- Extend this class to create a WindowEvent listener and override the methods for the events of interest.
- If you implement the WindowListener interface directly, you have to define all of the methods in it. This abstract class defines null methods for them all, so you can only have to define methods for events you care about.
- Create a listener object using the extended class and then register it with a Window using the window's addWindowListener method.
The Adapter Class

```java
public abstract class WindowAdapter implements WindowListener {
    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    .............
}
```
Using an Adapter Class

- Add a WindowListener to our Windows and Frames, just by defining an anonymous extension of WindowAdapter. The JSlider demo attached works this way:

```java
public class SliderDemo extends JFrame implements ChangeListener {
    JSlider mySlider;
    JLabel myValue;

    // The constructor
    public SliderDemo(String windowTitle) {
        super(windowTitle);

        // Add a listener for window events:
        addWindowListener ( new WindowAdapter() {
            public void windowClosed(WindowEvent e) { System.exit(0); }
        });
    }
}
```
**Dynamic Instance Inheritance**

```java
import java.awt.Color;

class Length {
    int pts; // assume that pts are universal
    Length(int pts) { this(pts); }

    public String toString()
    { return String.valueOf(pts) + " pts"; }
}

class Element {
    final static Color DEFAULT_COLOR = Color.black;
    final static Length DEFAULT_PADDING = new Length(0);

    Element container; // ...a root element has container == null

    // For the following fields, a a null value at Access Time
    // means "no local value," i.e., "inherit from container"...
    Color    color;
    Length   paddingLeft;

    Element()
    { container = null; color = null; }

    // Much elided and simplified
    // visibility modifiers dropped.
    /* instances inherit values from 
     * other instances
     * dynamic property values
     * dynamic inheritance hierarchy */
```
Dynamic Instance Inheritance  -- cont’d

Element getContainer()
    { return container; }

void setContainer(Element container)
    { this.container = container; }

Color getColor() {
    if (color != null)
        // override container...
        return color;

    if (container == null)
        // no value defined...
        return DEFAULT_COLOR;

    // inherit value from container...
    return container.getColor();
}

    void setColor(Color color)
    { this.color = color; }

Length getPaddingLeft() // ...no chaining
    { return (paddingLeft != null ? paddingLeft : DEFAULT_PADDING; }

    void setPaddingLeft(Length padding)
    { paddingLeft = padding; }
}
Document Structure - Instance Inheritance

class BlockElement extends Element { /* ... */ }
class InlineElement extends Element { /* ... */ }

class HTMLElement extends Element { /* ... */ }
class BodyElement extends Element { /* ... */ }

class ParagraphElement extends BlockElement { /* ... */ }
class UnorderedListElement extends BlockElement { /* ... */ }
class ListItemElement extends BlockElement { /* ... */ }
class BoldElement extends InlineElement { /* ... */ }

class ElementTest {
  public static void main(String[] as) {
    // <html> <!-- html -->
    // <body> <!-- body -->
    //  <p style="color: red; padding-left: 10pt"> <!-- p0 -->
    //   <b>bold and red</b> <!-- b0 -->
    //  </p>
    // </body>
    // </html>
  }
}
HTMLElement html = new HTMLElement();
BodyElement body = new BodyElement();
ParagraphElement p0 = new ParagraphElement();
BoldElement b0 = new BoldElement();
UnorderedListElement ul0 = new UnorderedListElement();
ListItemElement li0 = new ListItemElement();
BoldElement b1 = new BoldElement();

body.setContainer(html);
p0.setContainer(body);
b0.setContainer(p0);
ul0.setContainer(p0);
li0.setContainer(ul0);
b1.setContainer(li0);

p0.setColor(Color.red);
p0.setPaddingLeft(new Length(10));

ul0.setColor(Color.green);

System.out.print("b0.color has-value ");
System.out.println(b0.getColor());  // red  (inherited from  p0)
System.out.print("b1.color has-value ");
System.out.println(b1.getColor());  // green (inherited from ul0)

System.out.print("p0.padding-left has-value ");
System.out.println(p0.getPaddingLeft());  // 10 pts  (local)
System.out.print("ul0.padding-left has-value ");
System.out.println(ul0.getPaddingLeft());  // 0 pts  (not inherited)
// PROPERTIES CAN BE DYNAMICALLY MODIFIED...
System.out.println();
System.out.println("p0.color = blue");
p0.setColor(Color.blue);
System.out.print("b0.color has-value ");
System.out.println(b0.getColor()); // blue (inherited from p0)
// ...i.e., "bold and red" is now in blue.

// AND PROPERTIES CAN LIKewise BE DYNAMICALLY OVERRIDDEN
System.out.println("b0.color = red");
b0.setColor(Color.red);
System.out.print("b0.color has-value ");
System.out.println(b0.getColor()); // red (local value overrides p0)
// ...i.e., "bold and red" is back to red.
System.out.println("unset b0.color");
b0.setColor(null);
System.out.print("b0.color has-value ");
System.out.println(b0.getColor()); // blue (inherited from p0)
// ...i.e., "bold and red" is blue again.
System.out.println("p0.color = red");
p0.setColor(Color.red);
System.out.print("b0.color has-value ");
System.out.println(b0.getColor()); // red (inherited from p0)
   // ...i.e., "bold and red" is back to red.

// THE INHERITANCE HIERARCHY ITSELF IS ALSO DYNAMIC...

System.out.println();
System.out.println("b0.container = li0");
b0.setContainer(li0);
System.out.println("b1.container = p0");
b1.setContainer(p0);

System.out.print("b0.color has-value ");
System.out.println(b0.getColor()); // green (inherited from ul0)
System.out.print("b1.color has-value ");
System.out.println(b1.getColor()); // red (inherited from p0)
   // ...i.e., "bold and green" is now in red, and "bold and red"
   // is in green, :).

} // actually, System.out.println(b0.getColor()) prints...
   // java.awt.Color[r=255,g=0,b=0]
Extra Slides
Employee - base object

The only difference between the simple Java and JavaScript Employee definitions:

- specify the type for each property in Java but not in JavaScript
- create an explicit constructor method for the Java class

<table>
<thead>
<tr>
<th>JavaScript</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>function Employee () {</td>
<td>public class Employee {</td>
</tr>
<tr>
<td>this.name = &quot;&quot;;</td>
<td>public String name;</td>
</tr>
<tr>
<td>this.dept = &quot;general&quot;;</td>
<td>public String dept;</td>
</tr>
<tr>
<td>}</td>
<td>public Employee () {</td>
</tr>
<tr>
<td></td>
<td>this.name = &quot;&quot;;</td>
</tr>
<tr>
<td></td>
<td>this.dept = &quot;general&quot;;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Matthew Morgenstern 17 CS211 Accel/Proj - Sept. 20 & 21, 2000
The Manager and WorkerBee definitions show the difference in how you specify the next object higher in the inheritance chain.

- **JavaScript**: add a prototypical instance as the value of the prototype property of the constructor function. You can do so at any time after you define the constructor.
- **Java**: specify the superclass within the class definition. You cannot change the superclass outside the class definition.

<table>
<thead>
<tr>
<th>JavaScript</th>
<th>Java</th>
</tr>
</thead>
</table>
| function Manager () {  
  this.reports = [];  
}  
Manager.prototype = new Employee;  
| public class Manager extends Employee {  
  public Employee[] reports;  
  public Manager () {  
    this.reports = new Employee[0];  
  }  
}  
| function WorkerBee () {  
  this.projects = [];  
}  
WorkerBee.prototype = new Employee;  
| public class WorkerBee extends Employee {  
  public String[] projects;  
  public WorkerBee () {  
    this.projects = new String[0];  
  }  
}  

Matthew Morgenstern
Engineer & SalesPerson definitions create objects that descend from WorkerBee & Employee.
- An object of these types has properties of ancestor objects above it in the linear (single) chain.
- In addition, the constructor definitions override the inherited value of the dept property with new values specific to these objects.

<table>
<thead>
<tr>
<th>JavaScript</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>function SalesPerson () {</td>
<td>public class SalesPerson extends WorkerBee {</td>
</tr>
<tr>
<td>this.dept = &quot;sales&quot;;</td>
<td>public double quota;</td>
</tr>
<tr>
<td>this.quota = 100;</td>
<td>public SalesPerson () {</td>
</tr>
<tr>
<td>}</td>
<td>this.dept = &quot;sales&quot;;</td>
</tr>
<tr>
<td>SalesPerson.prototype = new WorkerBee;</td>
<td>this.quota = 100.0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>public class Engineer extends WorkerBee {</td>
</tr>
<tr>
<td>function Engineer () {</td>
<td>public String machine;</td>
</tr>
<tr>
<td>this.dept = &quot;engineering&quot;;</td>
<td>public Engineer () {</td>
</tr>
<tr>
<td>this.machine = &quot;&quot;;</td>
<td>this.dept = &quot;engineering&quot;;</td>
</tr>
<tr>
<td>}</td>
<td>this.machine = &quot;&quot;;</td>
</tr>
<tr>
<td>Engineer.prototype = new WorkerBee;</td>
<td>}</td>
</tr>
</tbody>
</table>
### Specifying Properties in Constructors - 1

<table>
<thead>
<tr>
<th>JavaScript</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>function Employee (name, dept) {</td>
<td></td>
</tr>
<tr>
<td>this.name = name</td>
<td></td>
</tr>
</tbody>
</table>
|   this.dept = dept || "general";
| }                   | public class Employee {  |
|                     |   public String name;  |
|                     |   public String dept;  |
|                     |   public Employee () {  |
|                     |     this("", "general");  |
|                     | }  |
|                     |   public Employee (name) {  |
|                     |     this(name, "general");  |
|                     | }  |
|                     |   public Employee (name, dept) {  |
|                     |     this.name = name;  |
|                     |     this.dept = dept;  |
|                     | }  |
### Specifying Properties in Constructors - 2

#### JavaScript

```javascript
function WorkerBee (projs) {
    this.projects = projs || []; 
}
WorkerBee.prototype = new Employee;
```

```javascript
function Engineer (mach) {
    this.dept = "engineering";
    this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
```

#### Java

```java
public class WorkerBee extends Employee {
    public String[] projects;
    public WorkerBee () {
        this(new String[0]);
    }
    public WorkerBee (String[] projs) {
        this.projects = projs;
    }
}
```

```java
public class Engineer extends WorkerBee {
    public String machine;
    public WorkerBee () {
        this.dept = "engineering";
        this.machine = "";
    }
    public WorkerBee (mach) {
        this.dept = "engineering";
        this.machine = mach;
    }
}
```