Java Programming Week 8
Containers and Classes

Orange Coast College
Computer Science 170
Spring 2004

The Plan For Week 8
- Containers and Layout: how to use layout managers
- Panel and GridLayout: how to create nested layouts
- Inheritance: how to create new classes from existing ones
- Constructors: how to write methods to initialize objects
- Sizing and Metrics: writing methods that produce values
- Lab: Meet the Ovenator
- Homework: The Ovenator II

Lesson A: Containers & Layouts
- Containers and Components
  - The Container class
  - Container methods
- Layout Management
  - Changing the layout manager
  - The FlowLayout class
  - The BorderLayout class
- Exercise 8A
Object, Component and Container

- **Object** is Java's root class
- **Component** is superclass of most AWT GUI objects
  - An abstract class: cannot create Component objects
  - To create a component you must use subclasses
- **Container**: subclass that can hold other components
  - Two main branches:
    - **Panel**: superclass of Applet
    - **Window**: superclass of Frame and Dialog

The Component Family Tree

```
Object
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Button</td>
</tr>
<tr>
<td>Choice</td>
</tr>
<tr>
<td>GridLayout</td>
</tr>
<tr>
<td>List</td>
</tr>
<tr>
<td>TextComponent</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Container</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Panel</td>
</tr>
<tr>
<td>Applet</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Window</td>
</tr>
<tr>
<td>Frame</td>
</tr>
<tr>
<td>Dialog</td>
</tr>
</tbody>
</table>
```

The Container Class

- Like Component, Container is an abstract class
- Means you can't create Container objects
- You can have Container variables
- These must refer to one of the concrete subclasses
- Container methods: complete list on page 110-112
  - The most used method in the class is `add()`
  - Places a component "inside" a container
  - Component can exist in only one container at a time
  - Use `getParent()` to retrieve a component's container
  - May "re-seat" components by calling `doLayout()`
Layout Management

- Most Java programs don't explicitly size their widgets
  - Common in VB world, but restricted to Windows
  - Java programs should run OK on different platforms
  - Different platforms have different widget sizes
  - Should be more like HTML: platform independent
- Java uses a set of classes called "layout managers"
  - "Intelligent" agents arrange and size your components
  - Each one employs a different "strategy"
  - You can also create your own layout managers

The FlowLayout Class

- Layout manager automatically used in applets
- The strategy of the FlowLayout class works like this:
  - Steps through component list, starting with first added
  - "Asks" each component it's "preferred size"
  - Send the setSize() message to each component
  - Positions each component starting on the top row
  - If a component won't fit, then it is moved to the next row
- FlowLayout works like your word processor
  - Instead of "word wrap" it performs "widget wrap"

Creating a FlowLayout Object

- Use one of these constructors to create a FlowLayout
  - FlowLayout()
  - FlowLayout(int alignment)
  - FlowLayout(int alignment, int hgap, int vgap)
- The hgap/vgap controls space between components
- The alignment parameter can be one of these:
  - FlowLayout.LEFT
  - FlowLayout.RIGHT
  - FlowLayout.CENTER
Using FlowLayout

- To use a layout manager, you call `setLayout()`
  - Pass the layout manager as an argument
  - Normally, this code goes into the `init()` method
- Here's an example that changes to a left-aligned version
  ```java
  FlowLayout mgr = new FlowLayout(FlowLayout.LEFT);
  this.setLayout(mgr);
  ```

- Example: `SimpleFlowRight.java` [run the applet]

The BorderLayout Class

- A very popular layout manager
  - Default in `Frame` and in Java 2 `JApplets`
- Screen space is divided into 5 regions
  - North, South, East, West, and Center
  - Each region can hold only a single component
- Layout manager's strategy:
  - Uses control's width preference for East/West
  - Uses control's height preference for North/South
  - Center and remaining direction “swells” to fit space

Creating BorderLayout Objects

- `BorderLayout` has only two constructors
  ```java
  BorderLayout()
  BorderLayout(int hgap, int vgap)
  ```
- Second allows you to control space between components
- Set your layout manager in `init()` like this:
  ```java
  BorderLayout blMgr = new BorderLayout();
  setLayout(blMgr);
  ```
  // or like this
  ```java
  setLayout( new BorderLayout(10, 15) );
  ```
Adding Components

- Use one of these special forms of `add()`:  
  ```java
  add(String constraint, Component widget)
  add(Component widget, Object constraint)
  ```
- First of these, (used in your book), is **obsolete**
- The constraint can be one of these five values:
  ```java
  BorderLayout.NORTH, BorderLayout.SOUTH
  BorderLayout.EAST, BorderLayout.WEST
  BorderLayout.CENTER
  ```

A BorderLayout Example

- Construct a `BorderLayout` with `hgap = 10, vgap = 20`
  ```java
  setLayout(new BorderLayout(10, 20));
  ```
- Add component to each position like this:
  ```java
  add( new Button("North"), BorderLayout.NORTH);
  add( new Button("South"), BorderLayout.SOUTH);
  ...
  ```
- Example: [SimpleBorderLayoutPanel.java](#) | [run the applet]

Layout Manager Methods

- You can retrieve the horizontal and vertical gaps like this:
  ```java
  int hgap = mgr.getHgap();
  int vgap = mgr.getVgap();
  ```
- You can change the gaps like this:
  ```java
  mgr.setHgap(8);
  mgr.setVgap( mgr.getHgap() );
  ```
- Introduction to Exercise 8A
Lesson B: Panels and GridLayout
- Using the GridLayout class
  - Different GridLayout constructors
  - Passing 0 as the row or column
- Using the Panel class
  - Creating nested layouts with panels
- Exercise 8B

The GridLayout Class I
- Divides space into rows and columns
  - Components swell to fill all available space
- Three constructors
  - GridLayout() // like GridLayout(1, 0)
  - GridLayout(int rows, int cols)
  - GridLayout(int rows, int cols, int hgap, int vgap)
- Default is 1 row, 0 means any number of columns
- No special version of add(): GL1.java [run it]

The GridLayout Class II
- Two-argument constructor is often used incorrectly
  - The number 0 means "any number"
  - You may specify rows or columns, but not both
  - If you specify both, then columns is ignored
- Both of these are identical:
  - setLayout(new GridLayout(3, 4));
  - setLayout(new GridLayout(3, 0));
Panels

- An "empty" container class
  - Each one can have its own layout manager
  - The default is FlowLayout.CENTER
  - Note that Applet is a subclass of Panel
- You can place one panel inside another
  - These are called "nested panels"
  - Used to easily create sophisticated layouts
- Let's look at a "login" type applet [NestedPanels]

A Nested Panel Example

- Here are the requirements for the applet
  - Title at top stretches, button on the bottom naturally sized
  - Labels and input fields in center arranged in rows/columns
  - Want center section naturally sized
- Solution
  - Overall layout is BorderLayout
  - Title label added to top of layout
  - Panel added to south with FlowLayout for login button
  - Panel in center uses GridLayout for input fields
  - Added to FlowLayout Panel for natural sizing
- Introducing Exercise 8B

Lesson C: Introducing Inheritance

- Inheritance and Composition: is-a versus has-a
  - Properties of the is-a relationship
    - Inherited methods and fields
    - Specialization, generalization, and extension
    - Substitutability and single inheritance
- Creating a custom widget class
  - The Canvas class versus the Component class
  - Creating WarningMessage and a test harness
  - Setting the size, color, and font
- Exercise 8C
Inheritance

- You've been using "pre-built" parts in your programs
  - Using Labels, Buttons, and other widgets
  - In this unit, you'll build new "parts" for your programs
- Two basic strategies for building new parts
  - Combine simpler parts into a more complex part
    - Combine a TextField and Label
    - This is called a has-a class relationship
  - Extend an existing class and add new features
    - This is called an is-a class relationship

The is-a Relationship

- Many components combine both of these techniques
  - A TextField has-a background color property
  - It also is-a TextComponent and a Component
  - TextField has an is-a relationship with TextComponent
    - TextField and TextArea are specialized versions
    - These specialized versions are called subclasses
    - TextComponent contains methods common to both
    - TextComponent is called a superclass
- Superclass and subclass are often confused

Why Superclass and Subclass?

- Divide Animals into Insect, Mammal, Amphibian subsets
  - Subset is called a subclass
  - Specialized kind of Animal
- Combine Rodent, Primate, Elephant into Mammal
  - Mammal is a superclass
  - Represents generalization
- Can say jumbo is-a Elephant
Creating a Subclass

- In Java, you create a subclass by using `extends`

  ```java
  public class ImageButton extends Button ...
  ```

- Every class can directly extend only one class
  - This is called single inheritance
  - Every class has only one immediate ancestor class

Inherited Methods and Fields

- Subclass normally represents specialization
  - Inherits all attributes and methods of superclass
  - Usually adds additional methods or fields

Substitutability

- Everything a Button can do, an ImageButton can do
- Can use ImageButton anywhere you can use a Button
  - Called "the principle of substitutability"

  ```java
  public void methodOne(Button b) ...
  public void methodTwo(ImageButton b) ...
  ```

- Pass an ImageButton to both, but a Button only to first
  - Every ImageButton is a Button
  - Every Button is not necessarily an ImageButton
Creating a new Widget

- Let’s create a new class called `WarningLabel`.
  - Like a `Label` only we’ll do our own painting.
  - Difficulty extending `Label` because of native peers.
- Java has two “blank” widgets, just made for extending.
- `Canvas` : uses a “blank” native peer.
  - Components are rectangular, heavyweight, and opaque.
- `Component` : does not use a native peer.
  - Can create non-rectangular and transparent regions.
  - A little more difficult to use correctly.

Step 1: The Test Harness

- Start by creating the applet that tests our new widget.
- Create a new empty project in JCreator, `TestWarning`.
- Add a new Java class named `TestWarning.java`.
- Write basic applet structure, add `init()` method.
- Create and add a new `WarningMessage` object.
- Save and add HTML file for `TestWarning`.
- Note: your program won’t compile at this point.
- If you get behind, step-by-step instructions in Lesson 8C.

Step 2: Add New Class

- Add another Java file: `WarningMessage.java`.
  - Only import `java.awt`, not `java.applet`.
  - Extend the `Canvas` class, not `Applet`.
  - Add a `paint()` method that prints a message.

```java
public void paint(Graphics g)
{
    g.drawString("This is a Warning", 25, 25);
}
```

- A placeholder method like this is called a stub.
Step 3: Compile and Run

- Compile both files using **Compile File** (not Project)
  - Both should compile OK
- Select **TestWarning.html** and Run File
  - Does your program work? Not yet!
- Why? **Canvas** objects don't have a default size
  - Add this line to **TestWarning.java**
    ```java
    warning.setSize(200, 30);
    ```
- Now, you can recompile and run (short-term fix)

Inherited Methods

- Let's see if we really inherited the color methods.
  - Add these to the **TestWarning.init()** method
    ```java
    warning.setBackground(Color.yellow);
    warning.setForeground(Color.red);
    ```
- How about fonts? Try this:
  ```java
  warning.setFont(new Font("Dialog", Font.BOLD, 18));
  ```

Four Lessons Learned

- We can create **WarningLabel** variables like **warning**
  - Just like we can create **Button** or **Label** variables
- The **WarningLabel** class has a constructor
  - We could use **new** to create **WarningLabel** objects
- The **WarningLabel** class is a **Component**
  - We could use Container's **add()** which expects a **Component**
- The **WarningLabel** class inherits **Component** methods
  - Got **setBackground()**, **setFont()** for "free"
- Introduction to Exercise 8C
Lesson D: Constructors

- Constructor basics: what is a constructor?
- How do you write a constructor?
- The default constructor and the invisible constructor
- The WarningMessage constructors
- Using this with constructors (arguments and fields)
- Exercise 8D

What Are Constructors?

- Constructors are like methods
  - They are defined inside a class like a method
  - But, they are not called directly like a method
  - Instead, they are called when you use the new operator
  - Java uses the constructor to initialize your object
- A class definition is the blueprint that defines an object
- A constructor is the factory that creates an object

Constructor Syntax

- A constructor looks just like a method, with two constraints:
  - The name is always the same as the class
  - The constructor cannot have a return type
The Default Constructor

- The constructor that takes no arguments
  - Sometimes called the no-arg constructor
  - Used to initialize each field to a predefined (default) value

```
Button btn = new Button();
Label lbl = new Label();
Panel pnl = new Panel();
```

- The “invisible” constructor
  - If you don’t write any constructors, Java supplies a default
  - Does not do any initialization beyond setting fields to 0

The WarningMessage Constructor

- The WarningMessage constructor is not very useful
  - We’d like to be able to write:

```
WarningMessage warning
    = new WarningMessage("All data will be erased. OK?");
```

- For this, we need to write a constructor with arguments
  - We’ll also need to define fields for the class

The WarningMessage Fields

- The purpose of a constructor is to initialize fields
- What fields should we use for the WarningMessage class?
  - Need to hold message supplied by the user
  - To hold text we use String variables
  - Add this in the class, but not inside a method

```
private String message;
```

- We don’t need variables to hold color or font size
  - These are inherited from Component
The Working Constructor

- Constructor that allows user to initialize all possible fields
  - Not all fields need be user initialized
- Open up WarningMessage.java and write the constructor
  - Start with skeleton: public, same name as class, no return
  - Add a formal argument: designed to initialize message
  - Finish the body: use the arg to initialize message
  - Adjust the paint() method to use the field message
- Adjust TestWarning.java to use new constructor
- Check your work: TestWarning.java, WarningMessage.java

Using this

- The keyword this means "the current object"
- Can use it when calling methods in the same class
  ```java
  this.add(myButton);
  ```
  - Most people leave it off, because it is implied
- In constructors, often use same name for args/fields
- Place this in front of field to differentiate between them
  ```java
  public WarningMessage(String message) {
      this.message = message;
  }
  ```

Constructor Review

- Constructors are "methods" that create objects
  - If class definition is blueprint, then constructor is a factory
- Constructors are called by using the new operator
- Constructors have the same name as the class
- Constructors never have a return value
- May have arguments so you can customize objects
- May provide a default value for every field instead
- Introduction to Exercise 8D
Lesson E: Sizing and Metrics

- Improving your custom component
- Sizing your component based upon font and text size
- Writing a value-producing method
  - How to define and override getPreferredSize()
- Measuring the message
  - Using the FontMetrics class
- Exercise 8E

Sizing Your Component

- Buttons are “auto-sized”, why not WarningMessage?
  - Textbook makes each object 60 x 40 (page 124)
  - Not big enough for many messages
  - Need to find a way to “dynamically” size component
  - Want to use the size of the message
  - Don’t want to size it if dropped into a GridLayout()
- Instead of sizing in constructor, use a special method
  - Container calls getPreferredSize() method
  - Method returns a Dimension with desired size

Value-Producing Methods

- You already know how to call methods like getSize()
  - Writing them is a little different than writing init(), etc.
- Value-producing method have two differences:
  - Must have a return type other than void
  - Must end with a return expression [see page 200]

```java
public returnType methodName( arguments )
{
    // statements
    return typeExpression;
}
```
Writing getPreferredSize() I

The `getPreferredSize()` method has this signature:

```java
Dimension getPreferredSize()
```

To override, add a method with this signature to your class.

The method should carry out these actions:
- Calculate the size you want the component to be
- Create a `Dimension` variable of that size
- Return the variable, using the `return` statement
- Let's walk through these steps with `WarningMessage`

Writing getPreferredSize() II

Step 1: Write the stub
- Take what you know from the signature and add a stub
- Make sure you add a JavaDoc comment

Step 2: Create the `Dimension` variable
- Use local variables `height` and `width` with 30 and 200
- Use test values to create a `Dimension` variable

Step 3: Return the variable using `return`

Step 4: Test the result
- Comment out the `setSize()` in `TestWarning.java`

Measuring the Message I

To really work, we need to know how wide the message is
- We have to set the size in pixels
- All we know is the size in characters
- Depends upon the font in use and the screen resolution
- Fonts are proportional and points depend upon screen

Class that measures this is called `FontMetrics`
- To create a `FontMetrics` object, don't use a constructor
- Instead, you use a `Graphics` method `getFontMetrics()`
- This is called a "factory" method
- Page 575 has explanation of the class

© Stephen D. Gilbert, 2004
Measuring the Message II

- Inside `paint()`, you get a `FontMetrics` like this:

```java
public void paint(Graphics g)
{
    // Statements that set the font, etc.
    FontMetrics fm = g.getFontMetrics();
    // Use the FontMetrics object here
}
```

- Don't change fonts after getting the `FontMetrics`
- If you do, the `FontMetrics` object is invalid

Measuring the Message III

- Outside `paint()`, you get a `FontMetrics` like this:

```java
public Dimension getPreferredSize()
{
    Graphics g = getGraphics();
    FontMetrics fm = g.getFontMetrics();
    g.dispose();
}
```

- You must dispose of `Graphics` objects outside `paint()`

Measuring the Message IV

- Once you have a `FontMetrics`, measure your message:
  - To find the `height` of a line of text use this:
    ```java
    int height = fm.getHeight();
    ```
  - To find the `width` of a line of text (in pixels), do this:
    ```java
    int width = fm.stringWidth(message);
    ```
  - Now, let's finish up `WarningMessage`
**Finishing WarningMessage**

- Fire up JCreator, and see if you can complete these steps:
  - In the constructor, set font to 18 point Dialog
  - In `getPreferredSize()`, get a `FontMetrics` object
    - You'll have retrieve and dispose of the `Graphics` object
  - Create a constant field named `PADDING` set to 20
    - Calculate `height`, allowing for padding on top/bottom
    - Calculate `width`, allowing for 2 padding on left, one on right
  - Modify `paint()` to draw red exclamations and message
- Check your work: `WarningMessage.java`, `TestWarning.java`
- Introduction to Exercise 8E

---

**Finish Up**

- Assignments due on Tuesday, April 13

  Remember this week's Midterm Exams

- Quiz 8, Week 8 Lab: Ovenator
  - Run the applet, examine and modify the code
- Homework 8 [ A Better Ovenator ] due April 13
  - Note that homework and assignments due together
- Homework 7 [ A Better Gigobite ] due April 6