Lecture 6:
Building GUIs in Java with Swing

Wholeness of the Lesson

Swing is a windowing toolkit that allows developers to create GUIs that are rich in content and functionality. The ultimate provider of tools for the creation of beautiful and functional content is pure intelligence itself; all creativity arises from this field's self-interacting dynamics.

Introduction to Swing

- **Java Swing** is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.
- ***Sun's AWT.*** The original version of Java (jdk1.0) came with a primitive windowing toolkit (the AWT) for making simple GUIs. GUI components were built by using the native GUI toolkit of the target platform (Windows, MacIntosh, Solaris, etc). It is platform dependent.
- Unlike AWT, Java Swing provides platform-independent and lightweight components.

- ***AWT Still Used.*** Swing components still make use of aspects of the AWT – Swing is built "on top of" the old AWT. In particular, handling of events relies on the old event-handling model.

- ***JavaFX.*** In 2014, Oracle declared that Swing libraries would be developed no further, and that the windowing toolkit of choice had become JavaFX. JavaFX has more modern-looking components and has a more flexible API. Since Swing is still (as of 2016) far more widely used than JavaFX, Swing is presented here.

- ***Industry Standard.*** For standalone GUI development in Java, Swing is the toolkit most often used.

- ***Visual Designers.***
  - Most widely used (as of 2016) is Netbeans, which provides excellent visual support for Swing.
  - Usually, to use a visual designer effectively, you need to have a good understanding of how to write code to produce the effects you want.

The Main Idea in Swing

- *Components and containers.* Swing provides components (like text boxes, buttons, checkboxes) and containers (frames, windows, panels, applets) in which such components can be placed.

- *Containers placed in other containers.* In Swing, a container is also considered to be another kind of component, so containers can be placed in other containers.

- *LayoutManagers for containers.* Every container supports the use of a layout strategy. To achieve the visual objectives in building Swing screens requires skillful use of layouts on multiple containers.

- *Listeners = Event Handlers.* A Swing GUI becomes responsive to user actions (like button presses, item selections, etc) by means of an event handling model. In this model, there are "listeners" for user actions (like button presses and mouse clicks). When a relevant user action occurs, the listener is informed and the code that you have written to handle the event will then be executed.

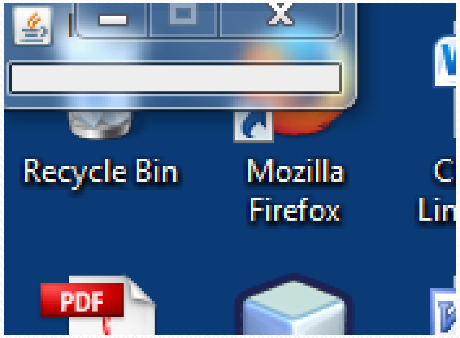
JFrames

- The top-level container class in Swing is `JFrame`. ("Top-level" means "not contained in any other containers.") JFrame is equipped with a title bar whose value can be modified.
- demo: `lesson6.MyFrame0`

```
class MyFrame extends JFrame {
    MyFrame() {
          setTitle("Hello World");
    }
}
```

To see the result so far, create an instance of `MyFrame` and call the `setVisible` method on it. Because of the non-threadsafe nature of Swing components, all component-building (to be safe) must be done through the EventQueue, so we have to create our JFrame and make it visible with the following mysterious code, which places our GUI-building thread in Swing's event queue, where it will be executed in the proper order.

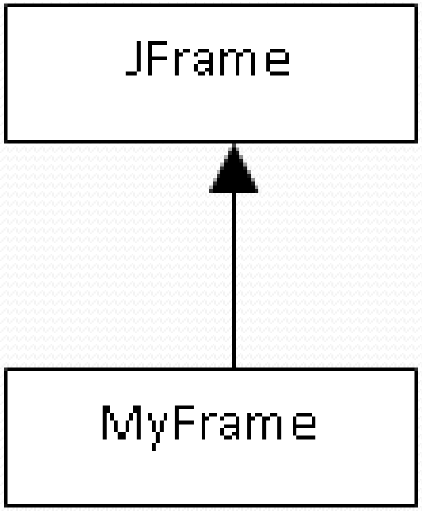
```
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable()
    {
        public void run() {
            MyFrame1 mf = new MyFrame1();
            mf.setVisible(true);
        }
    });
}
```



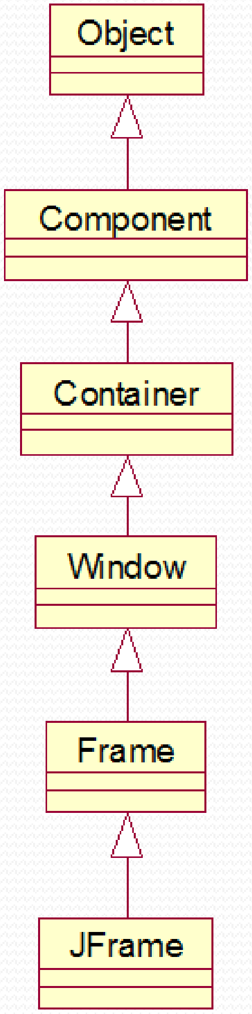
The JFrame that is created is placed by default in the upper left corner of the screen, squeezed into the smallest possible area

Inheritance in Swing

The code makes it clear that, when you design a Swing application, you start by creating a *subclass* of Jframe. The class diagram in UML is the following:



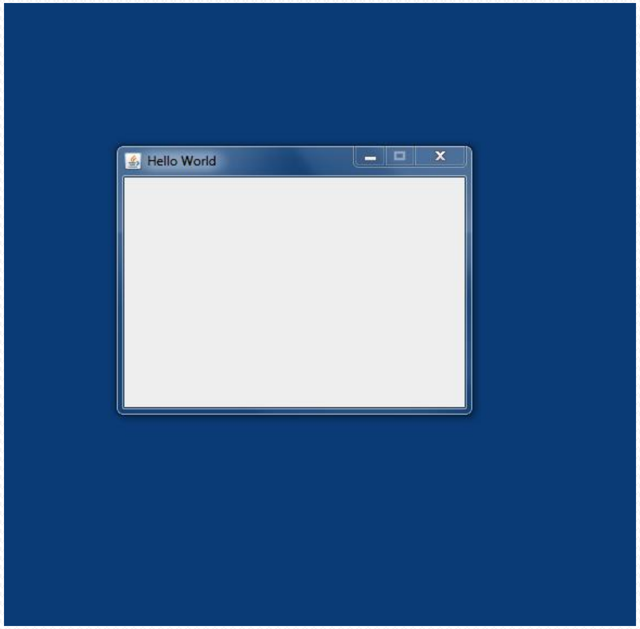
Inheritance Hierarchy for JFrame



- In the Example, next step is to adjust size and position.
- demo: `lesson6.MyFrame1`

```
public class MyFrame extends JFrame {
    public MyFrame()
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Hello World");
        setSize(320,240);
        centerFrameOnDesktop(this);
        setResizable(false);
    }
    public static void centerFrameOnDesktop(Component f) {
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        int height = toolkit.getScreenSize().height;
        int width  = toolkit.getScreenSize().width;
        int frameHeight = f.getSize().height;
        int frameWidth  = f.getSize().width;
        f.setLocation(((width-frameWidth)/2),
                   (height-frameHeight)/3);
    }
}

//To invoke, use the same main method as given above
```



Jframe is now centered in the desktop window and has the specified width and height

Tips:

- *Use* `pack()` *instead of hard-coding size*: Will make the window just large enough to fit in all the components.
- Call `pack()` after all components have been added to the container.
- Centering of window should be done *after* size has been set or `pack()` has been called

Adding Components

- `JFrame` is the top-level container class. It is the area of the screen occupied by a Swing application. It is responsible for closing the application, setting the size of the window, etc. You cannot add a component to a `JFrame` - you use the `JFrame's` content pane for that.
- The content pane is the only "child" of a `JFrame` and you may add components to it. However the default layout manager for the content pane is border layout, which does not preserve dimensions of components.
- So we create a “top-level” panel (`JPanel`) and add it to the content pane. `JPanel's` default layout manager is flow layout, which does preserve component dimensions.

Adding Components

- Organize components into containers (called "panels") and assemble panels into the main frame.

- Design Tip: Create a "top-level" panel that will contain all the other panels that you define.

```
//make the text field and label instance variables in MyFrame
JTextField text;
JLabel label;
public MyFrame(){
      //put initializations like setSize, setTitle, centerFrame here
         initializeWindow();

         JPanel mainPanel = new JPanel();
         text = new JTextField(10);
         label = new JLabel("My Text");
         JButton button = new JButton("My Button");
         mainPanel.add(text);
         mainPanel.add(label);
         mainPanel.add(button);

         getContentPane().add(mainPanel);
}
```

demo: `lesson6.MyFrame2`

Main Point

Swing classes are of two kinds: *components* and *containers*. A screen is created by creating components (like buttons, textfields, labels) and arranging them in one or more containers. Components and containers are analgous to the *manifest* and *unmanifest* fields of life; manifest existence, in the form of individual expressions, lives and moves within the unbounded container of pure existence.

Layout Managers: FlowLayout and BorderLayout

A Layout Manager is a Java class that decides how components will be arranged in a container and to what extent the *preferred size* of these components will be honored.

- The preferred size of a component, is, roughly, the minimum size it can have and still be visually meaningful (for example, a button's preferred size is "just big enough" so that you can see the button's label)

- The general rule is that the components in a container will be given their preferred size unless the policy of the container's layout manager conflicts with this

FlowLayout Policy

- All components are given their preferred size
- When components are added to the container, they are added from left to right in horizontal rows; when a row is filled up, components are placed in a new line below the first
- The default distance between successive components (both horizontally and vertically) is 5 pixels – this quantity can be modified using setHgap, setVgap.
- The entire cluster of components in a row can be justified left, justified right, or centered using these arguments, respectively, in the FlowLayout constructor: `FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.CENTER` (CENTER is the default)

  Example:

  ```
  myPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
  ```

BorderLayout Policy

1. When components are added, they are placed in one of 5 regions in the container, specified by

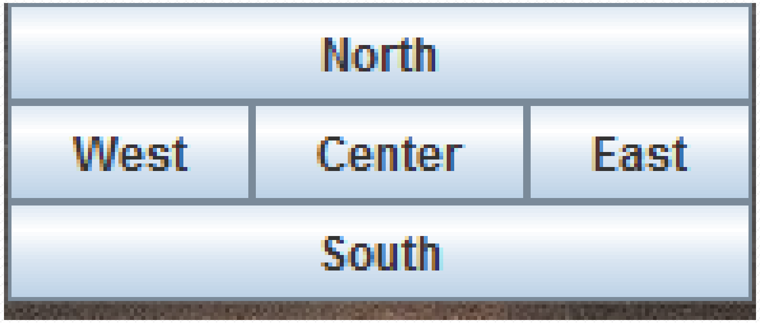
   `BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, BorderLayout.CENTER`

   If no region is specified, CENTER is the default. It is not necessary to populate every region with a component.

2. The gaps between these regions is, as with `FlowLayout`, 5 pixels both vertically and horizontally.

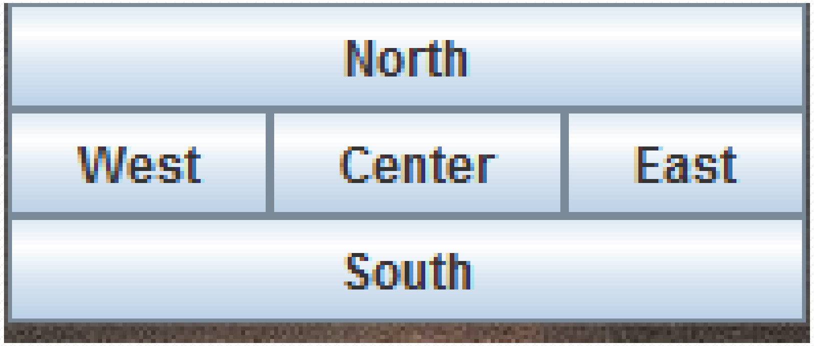
BorderLayout Policy

3. The preferred *height* of components placed North or South is honored, but the *width* of such components is made to be as wide as the container itself.
4. The preferred *width* of components placed East or West is honored. The *height* of such a component is forced to extend to the top and bottom of the container *unless* a component occupies North or South position. If North is occupied, then the height of West (and East) extends up to the North component. If South is occupied, the height of West (and East) extends down to the South component.
5. A component that occupies the Center position is stretched to fill out the region up to the components in the other positions.



BorderLayout Example

1. Demo: `lesson6.borderlayout`
2. Demo places five buttons to the five directions.



Main Point

Components are arranged in a container through the use of *layout managers* that organize components in different ways. FlowLayout preserves the size of components and lays components out horizontally, from left to right. BorderLayout lays out components in five positions – north, south, east, west and center; to preserve the size of components, BorderLayout is used in conjunction with FlowLayout. Likewise, all of manifest life is conducted by a vast network of natural laws.

Applying Layout Managers

JTextField and JLabel

Create the following panels in order to left justify the text field and label, placing the label below the text field

- a `textPanel` that will have a `BorderLayout`, so we can arrange the text field and label vertically
- place the text field in the `NORTH` position and label in the `CENTER` position of the `textPanel`.

```
private void defineTextPanel() {
    text = new JTextField(10);
    label = new JLabel("My Text");
    label.setFont(makeSmallFont(label.getFont()));

    textPanel = new JPanel();
    textPanel.setLayout(new BorderLayout());
    textPanel.add(text,BorderLayout.NORTH);
    textPanel.add(label,BorderLayout.CENTER);
}
```

Layout of the textfield/label combination and the Button.

To place the textfield/label component in the upper left of the screen and to place the button in the middle of the screen, create the following panels:

- a `topPanel` to hold the textfield/label – use a `FlowLayout` to left-justify
- a `middlePanel` to hold the button – use a `FlowLayout` to center the button
- layout the `mainPanel` with `BorderLayout`, and place `topPanel` in the `NORTH` and `middlePanel` in the `CENTER`.

```
public MyFrame(){
        initializeWindow();
        JPanel mainPanel = new JPanel();
        defineTopPanel();
        defineMiddlePanel();
        mainPanel.setLayout(new BorderLayout());
        mainPanel.add(topPanel, BorderLayout.NORTH);
        mainPanel.add(middlePanel, BorderLayout.CENTER);
        getContentPane().add(mainPanel);
}
private void defineTopPanel() {
        topPanel = new JPanel();
        defineTextPanel();
        topPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
        topPanel.add(textPanel);

}
private void defineMiddlePanel(){
        middlePanel=new JPanel();
        middlePanel.setLayout(new FlowLayout(FlowLayout.CENTER));
        button = new JButton("My Button");
        middlePanel.add(button);

}
```

Main Point

Because containers are themselves a certain type of component, containers can be organized inside of other containers. Attractive visual design of GUIs is accomplished in Swing through the creative use of multiple layouts of container classes. The natural order of existence is created and maintained by the hidden dynamics of pure intelligence.

Handling Events

To get a response from a button click, we associate a "listener" to the button; the listener will be informed (by way of an `ActionEvent`) whenever the button is clicked at runtime.

Here is a detailed overview of event handling in the AWT:

- A listener object is an instance of a class that implements a *listener interface* – typical example: `ActionListener` -- used for the most common GUI components in Java.
- Here is `ActionListener` from the source code for the Java libraries:

```
public interface ActionListener {
   public void actionPerformed(ActionEvent e);
}
```

When a class *implements* an interface, it means that the class provides an implementation for each of the method declared in the interface.

- An event source is an object that can register listener objects and send them event objects – examples: buttons, menu items, checkboxes, combo boxes
- The event source sends out event objects to all registered listeners when that event occurs – for instance, when a button is clicked, all listeners for this button receive an `ActionEvent` instance
- Listener objects may use the information in the event object received to determine their reaction to the event

Example of a Listener

For our GUI example, we register an `ActionListener` – named `MyButtonListener` – when we define our button. We specify the response to a button click in the body of the `actionPerformed` method.

ButtonListener Code

demo: `lesson6.MyFrame3`

```
//define the listener class
public class MyButtonListener implements ActionListener {
    //the text field we are listening to
    private JTextField text;
    public MyButtonListener(JTextField text) {
        this.text = text;
    }
    public void actionPerformed(ActionEvent evt){
        String textVal = text.getText();
        final String prompt = "Type a string";
        final String youWrote = "You wrote: ";
        if(textVal.equals("") ||
                textVal.equals(prompt) ||
                textVal.startsWith(youWrote)){

            text.setText(prompt);
        }
        else {

            text.setText(youWrote+"\""+textVal+"\".");
        }
    }
}
```

Attaching the ButtonListener

```
//Inside MyFrame, register your new
//listener class when the button is defined
button = new JButton("My Button");

//because our text field is stored as an instance variable
//we can pass it in to the listener like this:
button.addActionListener(new MyButtonListener(text));
```

Running the Code

When the user clicks the button….

User types “Hello”

User clicks My Button

User clicks My Button a second time

Brief Introduction to Inner Classes: Listeners As Inner Classes

The class `MyButtonListener` is closely associated with `MyFrame` – it relies on the text field of `MyFrame` and has behavior that is customized to the requirements of this particular application.

It is therefore natural to think of `MyButtonListener` as an auxiliary class for the private use of `MyFrame`.

Java supports this need with *nested classes* – a nested class is a class that is defined within another class. When a nested class has access to all the instance variables of its surrounding class it is called an *inner class.*

If we make `MyButtonListener` an inner class of `MyFrame`, then there is no longer a need to pass a text field into the listener class since it will automatically have access to it.

Implementing Listener As an Inner Class

demo: `lesson6.MyFrame4`

```
public class MyFrame extends JFrame {
    private JTextField text;
    private JLabel label;
    private JButton button;
    public MyFrame() {
        // . . .
    }
    private void defineMiddlePanel(){
        middlePanel=new JPanel();
        // . . .
        button = new JButton("My Button");
        button.addActionListener(new
                                MyButtonListener());
        // . . .
    }
```

```
// . . .
//now defined as an innner class
class MyButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent evt){
        //automatic access to MyFrame's instance variables
        String textVal = text.getText();
        final String prompt = "Type a string";
        final String youWrote = "You wrote: ";
        if(textVal.equals("") ||
                textVal.equals(prompt) ||
                textVal.startsWith(youWrote)){

            text.setText(prompt);
        }
        else if(textVal.equalsIgnoreCase("error")){
            showMessage("An error has occurred!");
            text.setText(prompt);
        }
        else {

            text.setText(youWrote+"\""+textVal+"\".");
        }
    }
}
```

Main Point

A GUI becomes responsive to user interaction (for example, button clicks and mouse clicks) through Swing's event-handling model in which event sources are associated with listener classes, whose actionPerformed method is called (and is passed an event object) whenever a relevant action occurs. To make use of this event-handling model, the developer defines a listener class, implements actionPerformed, and, when defining an event source (like a button), registers the listener class with this event source component. The "observer" pattern that is used in Swing mirrors the fact that in creation, the influence of every action is felt everywhere; existence is a field of infinite correlation; every behavior is "listented to" throughout creation.

Summary

Development in Swing requires knowledge of three areas:

1. ***Containers and Components***. The elements that a user makes use of to interact with a UI – like buttons, textfields, etc – are *components,* which are arranged in Swing *containers.*

2. ***Layout Managers***. Design of a UI first requires the developer to visualize, and sketch out, the desired appearance of windows. This design is translated into Swing components and containers by skillful use of LayoutManagers, which provide rules that determine dimensions and positions of components on the window

3. ***Event-Handling.*** The functionality of a UI – by which a user can initiate an action to obtain a response – is achieved in Swing with *listeners.* Typically on a UI, *ActionListeners,* which are implemented with event-handling code, are attached to components. The event-handling mechanism of Java translates user actions into events that causes the ActionListener code to execute.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

The self-referral dynamics
arising from the reflexive association of container classes

1. In Swing, components are placed and arranged in container classes for attractive display.
2. In Swing, containers are also considered to be components; this makes it possible to place and arrange container classes inside other container classes. These self-referral dynamics support a much broader range of possibilities in the design of GUIs.

---

3. **Transcendental Consciousness:** TC is the self-referral field of all possibilities.
4. **Wholeness moving within itself**: In Unity Consciousness, all activity is appreciated as the self-referral dynamics of one's own Self.