UIs in Java

AWT
The first user-interface package for Java was the Abstract Windowing Toolkit (AWT).

A major goal of the AWT was to insulate the programmer from platform-specific windowing issues.

As a result, the AWT provided a set classes that did not depend on artifacts of operating-system windowing widgets.

Thus,

- a Java program calls a Java AWT component.
- Then the abstract component calls Java code that manipulates the corresponding component in the native windowing system.

This native GUI component is called a peer.

For example, when you used the AWT to create an instance of the Button class, the Java runtime system created an instance of the Button peer.

This Button peer did the real work of displaying the button and managing its behavior.

- A Solaris JDK would create a Motif button peer.
- A Windows 95 JDK would create a Windows 95 button peer.
- An OS/2 JDK would create a Presentation Manager button peer.

What were the advantages of using peers?
As the user interface grew more complicated, it was realized that peers were not an ideal way to implement a UI.

What were the disadvantages of using peers?

• Bugs

The Swing API
So, Sun created a new UI implemented with no native code. This came to be known as Swing.

Swing is one of the APIs that make up the Java Foundation Classes (JFC). Others include AWT, Java 2D, Accessibility, and Drag and Drop.

Swing extends the AWT by adding—

• a set of components, the JComponents, and
• a group of related support classes.

Swing components, like AWT components, are all JavaBeans and use the JavaBeans event model. JavaBeans are are reusable software components that can be combined into an application.
Some Swing components resemble AWT widgets, but don’t use peers to interface to native operating-system widgets.

Examples of Swing components can be found at http://java.sun.com/docs/books/tutorial/uiswing/components/components.html.

Containers vs. components

In AWT, subclasses of Container can contain Components. Some of these Components may themselves be containers.

Thus, containers may be nested arbitrarily deep.

In Swing, all components are subclasses of java.awt.Container. Thus, any Swing widget

The hierarchy is divided into classes, some of which provide basic UI functionality, e.g., JLabel and JList.

Others provide additional functionality, e.g., JFileChooser, JProgressBar, and JSplitPane.

Some of the classes provide features that can be used with other widgets, e.g., Icons, which can be embedded in other components, like JButtons and JLabels. It automatically caches icons, so it usually does not need to fetch them over the network when they are redisplayed.

MVC and “look and feel”: The AWT forces you to use a "look and feel" for widgets that depends on what platform the application is running on.
In Swing, the components do not have peer classes. Rather, the look and feel is “pluggable.”

In general, a **pluggable** component or attribute is one that can be changed independently of the surrounding framework.

In Swing, you can make these changes by using an object called `AbstractLookAndFeel`.

You can change the look and feel of a GUI at run time by using the `setLookAndFeel` method:

```java
try {
    UIManager.setLookAndFeel(
        "com.sun.java.swing.jlf.JLFLookAndFeel");
} catch (java.lang.ClassNotFoundException e) {
    // Can't change looks.
}
```

There are some limitations on the changes you can make. For example, you can’t change to Windows look and feel on non-Windows platforms because of possible copyright infringement.

**Simple Swing examples**

**Example 1**

Let’s take a look at a very simple example program, from the online Java Tutorial ([http://java.sun.com/docs/books/tutorial/uiswing/mini/firstexample.html](http://java.sun.com/docs/books/tutorial/uiswing/mini/firstexample.html)):

```java
import javax.swing.*;

public class HelloWorldSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("HelloWorldSwing");
        final JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```
The first line of this program tells the compiler that this program is going to be using the `javax.swing` package. The “javax” stands for Java extension (Swing was originally a Java extension).

Most Swing programs also need to

```java
import java.awt.*;
import java.awt.event.*;
```

Java programs build a GUI by placing `components` in `containers`.

Thus, every Swing program needs to have at least one “top-level” Swing container. The top-level Swing components are—

- frames (class `JFrame`)—independent main windows,
- dialogs (class `JDialog`)—secondary windows that are dependent on other windows, and
- applets (class `JApplet`)—displaying an applet’s widgets within a browser window.

Note that a frame has a title bar, a border, and buttons for iconifying and closing the window.

The code for manipulating the `JFrame` in this sample application is—

```java
JFrame frame = new JFrame("HelloWorldSwing");
...  
frame.pack();
frame.setVisible(true);
```

The first line creates the frame and specifies its title.

The `pack()` call sets the size of the frame to be the preferred size (and layout) of its subcomponents. We always need to specify how large a window is.

`setVisible(true)` makes the frame visible on the display. When windows are created, they are not yet visible.
If there were more than one frame in the program, we should probably call `show()` instead of `setVisible(true)`, because `show()` not only makes a window visible, but brings it to the front.

This frame is a container that contains only one component—

These lines of code manipulate it:

```java
    final JLabel label = new JLabel("Hello World");
    frame.getContentPane().add(label);
```

The call to `getContentPane()` returns the container which is the “content pane” for this window. Components have to be added to content panes.

The only other line in the program is

```java
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

`JFrame`'s `setDefaultCloseOperation` method specifies the default action for when the user clicks the close button.

In Java 1.3 and above, you can specify this by using the call above. In earlier versions of Java, you have to implement an event listener, as we will describe in Lecture 13.

**Example 2**

Let's take a look at another very simple program [Horstman & Cornell 7.1, mod.] that illustrates a slightly different style.

```java
import javax.swing.*;

class FirstFrame extends JFrame {
    public FirstFrame() {
        setTitle("FirstFrame");
        setSize(200, 100);
    }
}

class FirstTest {
    public static void main(String[] args) {
        JFrame frame = new FirstFrame();
```
```java
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.show();
```

What's different about this program?

Component inheritance hierarchy

[Horstman & Cornell, p. 270] The JFrame class contains only a few methods for changing how frames look.

Most of the methods it uses are inherited from its ancestors.

Here is the inheritance tree for JFrame and the other top-level containers.

Many of JFrame's important methods are inherited from Frame. These include setTitle(...) and—

- `setIconImage`, which takes an Image object to use as the icon when the window is minimized.
- `setResizable(...)`, which takes a boolean to determine whether a window will be resizable by the user.

Inherited from Window are—
• **void dispose()**, which closes the window and returns to the system the resources this window and all of its subcomponents were using.

• **void toFront()**, which puts the window on top of any other windows on the desktop.

• **void toBack()**.

Inherited from **Component** are—

• **boolean isVisible()**, which checks whether a component is set to be visible. All components are initially visible, except for top-level components.

• **void setVisible(boolean b)**.

• **boolean isEnabled()**, which checks whether a component is enabled, that is, whether it can receive keyboard input. Components are initially enabled.

• **void setVisible(boolean b)**.

• **void setEnabled(boolean b)**.

• **point getLocation()**, which returns the location of the top left corner of this component, relative to the top left corner of the surrounding container.

• **point getLocationOnScreen()**, which returns the location of the top left corner of this component, relative to the top left corner of the screen.

• **void setLocation(int x, int y)**, which (re)positions a component in the coordinate space of its parent. Its top left corner is located x pixels across and y pixels down.

• **void setLocation(Point p)**

• **void setBounds(int x, int y, int width, int height)**, which moves and resizes this component.

• **Dimension getSize()**, which gets the current size of this component.

• **void setSize(int width, int height)**

• **void setLocation(Dimension d)**
Java’s coordinate system has its origin (0, 0) in the upper left-hand corner of the screen. Coordinates are measured in pixels relative to the origin.

`Dimension` and `Point` have _______ instance variables.

To find out the resolution of the screen, you need to use the `Toolkit` class, which allows a program to see system-dependent information.

```java
Toolkit mySystem = Toolkit.getDefaultToolkit();
Dimension d = mySystem.getScreenSize();
```

*Exercise:* Redo our first `HelloWorldSwing` example to put the frame in the center of the display.

Let’s take a look at some of the components provided by Swing. This index is available at

http://java.sun.com/docs/books/tutorial/uiswing/components/components.html

These components are classified into several categories:

- **Top-level containers**. A top-level container must be at the top of any Swing containment hierarchy.
- General-purpose containers, like `JPanel` and `JScrollPane`, which can be used for a variety of purposes.
- Special-purpose containers, like `JRootPane` and `JLayeredPane`, which are used to provide flexibility and functionality in the Swing UI.
• Basic controls. Atomic components like JButton, JList, and JMenu, which exist primarily to get input from the user; they generally also show simple state.

• Uneditable information displays, like JLabel and JProgressBar, which exist solely to give the user information.

• Editable displays of formatted information, like JFileChooser and JColorChooser. Atomic components that display specially formatted information that can be edited by the user.

Collections in Java

As we saw in Lecture 5, Smalltalk has a rich hierarchy of collection classes. These classes provide the functionality that programs need, and avoid the need to build data structures out of low-level primitives like arrays.

In Smalltalk, many of the collection classes are abstract. Can you give an example?

How would you expect abstract collections to be implemented in Java?

Let’s take a look at the hierarchy of abstract collections:
• A Java Collection is very similar to a Smalltalk collection. The main difference is that it is not a superinterface of Maps, which are like Smalltalk Dictionaries.

• A Java Set is very similar to a Smalltalk Set; it doesn’t allow duplicates. The main difference is that it is not a concrete class—because there are different implementations of it.

  ◆ HashSet implements a Set using a hashtable. This is an efficient implementation if the elements are considered to be unordered.

  ◆ TreeSet implements a Set using a tree. This is an efficient implementation if the elements need to be traversed in a particular order. (It uses a TreeMap in its implementation.)

• A SortedSet is a set whose elements are kept sorted. Note that TreeSet also implements the SortedSet interface.

• A List allows precise control of where a new element is inserted in the collection. This makes it very similar to a Smalltalk

• A Map maps keys to values, like a Smalltalk Dictionary.

Let’s take a more detailed look at the Collection interface:

```java
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element); // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
```


boolean containsAll(Collection c);
boolean addAll(Collection c); // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear(); // Optional

// Array Operations
Object[] toArray();
Object[] toArray(Object a[]);

Note that Java collections store Objects. What are the implications of this?

The “optional” operations allow for immutable collections. If a particular collection does not implement these modifying operations, if they are not implemented, invoking them will raise an UnsupportedOperation

How do you suppose this effect is achieved?

Now let’s look at some code that uses one of these classes:

import java.util.*;
public class SetTest {
    public static void main(String argv[]) {
        Set s = new HashSet();
        s.add(new Integer(3));
        s.add(new Integer(4));
        s.add(new Integer(4));
        System.out.println(s.size());
    }
}

© 2003 Edward F. Gehringer  CSC 517/ECE 517 Lecture Notes, Fall 2003
What does this method print?

What is the advantage of declaring \( s \) to be a Set rather than a HashSet?

Here is a table showing the various implementations of the core Collection interfaces. Note that each interface is implemented in exactly two ways.

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Hash Table</th>
<th>Resizable Array</th>
<th>Balanced Tree</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>HashSet</td>
<td></td>
<td>TreeSet</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
<td>ArrayList</td>
<td></td>
<td>LinkedList</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap</td>
<td></td>
<td>TreeMap</td>
<td></td>
</tr>
</tbody>
</table>