Outline

- JNI
  - JNI
  - JNI sequence
  - JNI and Java Exceptions
- AWT
  - Swing/JFC
  - Swing Contains
  - Applet
  - Application
  - Event
  - Text Area
  - Controlling Layout
  - Bean
The Java Native Interface (JNI)

• allows users to call native methods from a Java application.
• added in Java 1.1
• the native method interface (NMI)
  – maintaining a certain degree of compatibility with its Java 1.0 equivalent:
  – NMI has design characteristics that make it unsuitable for adoption across all virtual machines
  – For this reason, future versions of the language might no longer support NMI, and it will not be covered here.
The Java Native Interface (JNI)

• JNI is designed to interface with native methods written only in C or C++

• Using JNI, your native methods can:
  – Create, inspect, and update Java objects (including arrays and Strings)
  – Call Java methods
  – Catch and throw exceptions
  – Load classes and obtain class information
  – Perform run-time type checking

• Thus, virtually everything you can do with classes and objects in ordinary Java you can also do in native methods.
Calling a native method

- Step 1: to write the Java code declaring a native method and its arguments

```java
//: appendixb:ShowMessage.java
public class ShowMessage {
    private native void ShowMessage(String msg);
    static {
        System.loadLibrary("MsgImpl");
        // Linux hack, if you can't get your library
        // path set in your environment:
        // System.load(
        //   "/home/bruce/tij2/appendixb/MsgImpl.so");
    }
    public static void main(String[] args) {
        ShowMessage app = new ShowMessage();
        app.ShowMessage("Generated with JNI");
    }
} ///:~
```
The header file generator: javah

• **Step2**: Now compile your Java source file and run **javah** on the resulting `.class` file, specifying the `—jni` switch

• `javah —jni ShowMessage`

```c
/* DO NOT EDIT THIS FILE
   - it is machine generated */
#include <jni.h>
/* Header for class ShowMessage */

#ifndef _Included_ShowMessage
#define _Included_ShowMessage

#ifdef __cplusplus
extern "C" {
#endif

/* Class:     ShowMessage
   * Method:    ShowMessage
   * Signature: (Ljava/lang/String;)V
   */
JNIEXPORT void JNICALL Java_ShowMessage_ShowMessage
    (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

All Rights Reserved.
Implementing your DLL

- **Step 3**: All you have to do is write a C or C++ source code file that includes the `javah`-generated header file and implements the native method, then compile it and generate a dynamic link library. This part is platform-dependent.

```c
extern "C" JNIEXPORT void JNICALL Java_ShowMessage_ShowMessage(jni.h>
#include <stdio.h>
#include "ShowMessage.h"

const char* msg=env->GetStringUTFChars(jMsg,0);
printf("Thinking in Java, JNI: %s\n", msg);
env->ReleaseStringUTFChars(jMsg, msg);
} ///:~
```

**JNIEnv** argument, the programmer has access to a large set of functions.
Passing and using Java objects

• **Step4**: You can also pass Java objects of your own creation to a native method. Inside your native method, you can access the fields and methods of the object that was received.

```
//: appendixb:UseObjects.java
class MyJavaClass {
  public int aValue;
  public void divByTwo() { aValue /= 2; }
}

public class UseObjects {
  private native void changeObject(MyJavaClass obj);
  static {
    System.loadLibrary("UseObjImpl");
    // Linux hack, if you can't get your library
    // path set in your environment:
    // System.load(
    // "/home/bruce/tij2/appendixb/UseObjImpl.so");
  }

  public static void main(String[] args) {
    UseObjects app = new UseObjects();
    MyJavaClass anObj = new MyJavaClass();
    anObj.aValue = 2;
    app.changeObject(anObj);
    System.out.println("Java: " + anObj.aValue);
  }
}
```
JNI and Java exceptions

- **Throw()**
  Throws an existing exception object. Used in native methods to rethrow an exception.

- **ThrowNew()**
  Generates a new exception object and throws it.

- **ExceptionOccurred()**
  Determines if an exception was thrown and not yet cleared.

- **ExceptionDescribe()**
  Prints an exception and the stack trace.

- **ExceptionClear()**
  Clears a pending exception.

- **FatalError()**
  Raises a fatal error. Does not return.
Swing/JFC

- Very easy to add keyboard accelerators, tooltips, graphics
- Pluggable look and feel
- Provides ways to change just about everything, but you must work to understand how
Swing Contains

- Borders
- Buttons
- Checkboxes
- ComboBoxes
- Image Icons
- Labels
- Layered Panes and Internal Frames (MDI)
- Lists and List Boxes
- Menus
- Popup Menus
- Radio Buttons
- Progress Bars
- Scrolling Support
- Scrollbars
- Splitter Control
- Tabbed Panes

- Tables
- Text Areas
- Text Components
- Text Fields
- Toolbars
- Trees (Outline Control)
- Keyboard Navigation
- Repaint Batching
- Support for Undo
- Custom Cursors
- Drag and Drop
- The LookAndFeel API
- All components are data-aware
The Basic Applet

- Application framework: inherit to build an application
- Override methods to add custom functionality
- Methods called automatically as part of basic framework
- Ideally: minimum amount of coding to create new application
Applet Methods to Override

- Often need to override only one method:
  - `init()` : Loading. Lay out all components, etc.,
- For more specialization you may also need:
  - `paint()` : For custom repainting of applet. Called automatically as part of an `update()`
  - `start()` : When page with applet appears on the screen
  - `stop()` : When page with applet moves off screen
  - `destroy()` : When applet is being unloaded
- Vestigial: `action()` (old AWT)
Simple First Applet

- Put a label on the applet’s “Content Pane”

```java
//: c13:Applet1.java
// Very simple applet.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
}
```
Applications

- Created in a JFrame
- `main()` creates an instance, sets the size, makes it visible
- You must also add a `WindowListener` to close the application
Combined Application/Applet

• Just add a `main()` to your applet class
• `main()` builds an instance of the applet inside a `JFrame`
• Then you can run the program from the command line or as an applet
• GUI programming is virtually identical between applets and applications, so the combined approach will be used
  – Allows easy display via HTML pages
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.);

public class Applet1c extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!");
    }
    // A main() for the application:
    public static void main(String[] args) {
        JApplet applet = new Applet1c();
        JFrame frame = new JFrame("Applet1c");
        // To close the application:
        Console.setupClosing(frame);
        frame.getContentPane().add(applet);
        frame.setSize(100,50);
        applet.init();
        applet.start();
        frame.setVisible(true);
    }
} ///:~
A Display Framework

• The code for running an application from the console command line is redundant
• Possibly being turned into several overloaded methods
• Label is created using reflection
package com.bruceeckel.swing;
import javax.swing.*;
import java.awt.event.*;

public class Console {
    // Create a title string from the class name:
    public static String title(Object o) {
        String t = o.getClass().toString();
        // Remove the word "class":
        if(t.indexOf("class") != -1)
            t = t.substring(6);
        return t;
    }

    public static void setupClosing(JFrame frame) {
        // The JDK 1.2 Solution as an
        // anonymous inner class:
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        // The improved solution in JDK 1.3:
        // frame.setDefaultCloseOperation(
        //     EXIT_ON_CLOSE);
        }

    public static void run(JFrame frame, int width, int height) {
        setupClosing(frame);
        frame.setSize(width, height);
        frame.setVisible(true);
    }

    public static void run(JApplet applet, int width, int height) {
        JFrame frame = new JFrame(title(applet));
        setupClosing(frame);
        frame.getContentPane().add(applet);
        frame.setSize(width, height);
        applet.init();
        applet.start();
        frame.setVisible(true);  
    }

    public static void run(JPanel panel, int width, int height) {
        JFrame frame = new JFrame(title(panel));
        setupClosing(frame);
        frame.getContentPane().add(panel);
        frame.setSize(width, height);
        frame.setVisible(true);  
    }
}

public static void run(JFrame frame, int width, int height) {
    setupClosing(frame);
    frame.setSize(width, height);
    frame.setVisible(true);
}

public static void run(JApplet applet, int width, int height) {
    JFrame frame = new JFrame(title(applet));
    setupClosing(frame);
    frame.getContentPane().add(applet);
    frame.setSize(width, height);
    applet.init();
    applet.start();
    frame.setVisible(true);  
}

public static void run(JPanel panel, int width, int height) {
    JFrame frame = new JFrame(title(panel));
    setupClosing(frame);
    frame.getContentPane().add(panel);
    frame.setSize(width, height);
    frame.setVisible(true);  
}
Buttons

- Simple constructor: takes button label

```java
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button1 extends JApplet {
    JButton b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
    }
    public static void main(String[] args) {
        Console.run(new Button1(), 200, 50);
    }
} ///:~
```
Capturing an Event

- Nothing happens in previous example
- Need a piece of code to be executed when the button is pressed: a “listener”
- Button press calls
  - the `actionPerformed()` method of the `ActionListener` interface
- Define `actionPerformed()`
  - to do whatever you want when the button is pressed
- A “Callback”
Callbacks for GUI events

- Register the implementation of an ActionListener with each button
- When the button is pressed, it “calls back” to the ActionListener
- Inner class used to implement the ActionListener
What kind of change?

- Change something on the screen
- Introduce a new Swing component
  - JTextField
- Text can be typed by user, or entered by program
Text Area

• Multi-line edit control
  – new JTextArea("Hello", 5, 40);
• Inherited from TextComponent
  – Has getText( ) & setText( ), appendText( ), etc.
• Generally place it inside a JScrollPane to allow scrolling of the visible area
public class TextArea extends JApplet {
    JButton b = new JButton("Add Data"),
    c = new JButton("Clear Data");
JTextArea t = new JTextArea(20, 40);
Map m = new HashMap();
public void init() {
    // Use up all the data:
    Collections2.fill(m,
        Collections2.geography,
        CountryCapitals.pairs.length);
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            for (Iterator it = m.entrySet().iterator();
                it.hasNext();)
                Map.Entry me = (Map.Entry)(it.next());
                t.append(me.getKey() + ": "
                    + me.getValue() + "\n");
        }
    });
    c.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText(""");
        }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(b);
    cp.add(c);
}
public static void main(String[] args) {
    Console.run(new TextArea(), 475, 425);
} ///:~
Controlling Layout

• Probably different than other GUls you’ve used
• All code, no resources
• Components are placed on panel using “layout manager” based on the order in which you `add()` the components
• Size, shape and placement are quite different depending on layout manager
• Window size and proportions modifies layout dynamically
FlowLayout

• Components “flow” onto form left-to-right and top-to-bottom
• Components take on “normal” size

```
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.);

public class FlowLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            cp.add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        Console.run(new FlowLayout1(), 300, 250);
    }
    } ///:~
```
BorderLayout

• Default for most things

```java
public class BorderLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.add(BorderLayout.NORTH,
                new JButton("North"));
        cp.add(BorderLayout.SOUTH,
                new JButton("South"));
        cp.add(BorderLayout.EAST,
                new JButton("East"));
        cp.add(BorderLayout.WEST,
                new JButton("West"));
        cp.add(BorderLayout.CENTER,
                new JButton("Center"));
    }
    public static void main(String[] args) {
        Console.run(new BorderLayout1(), 300, 250);
    }
} ///:~
```
GridLayout

- Organized in rows & columns

```java
public class GridLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
            cp.add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        Console.run(new GridLayout1(), 300, 250);
    }
} ///:~
```
Other Events

• You’re not limited to ActionListener
• Each type of event represented by a class
• Component responds to an event by making an event object and calling each “listener” registered for that event
• An event listener implements a particular listener interface using an inner class
• addXXXListener( ) adds a listener to your component, removeXXXListener( ) un-registers it
Other Events

- You’re not limited to `ActionListener`
- Each type of event represented by a class
- Component responds to an event by making an event object and calling each “listener” registered for that event
- An event listener `implements` a particular listener interface using an inner class
- `addXXXListener()` adds a listener to your component, `removeXXXListener()` un-registers it
## Other Events

<table>
<thead>
<tr>
<th>Event, listener interface and add- and remove-methods</th>
<th>Components supporting this event</th>
</tr>
</thead>
</table>
| ActionEvent  
ActionListener  
addActionListener( )  
removeActionListener( ) | JButton, JList, JTextField, JMenuItem and its derivatives including JCheckBoxMenuItem, JMenu, and JpopupMenu. |
| AdjustmentEvent  
AdjustmentListener  
addAdjustmentListener( )  
removeAdjustmentListener( ) | JScrollPane and anything you create that implements the Adjustable interface. |
| ComponentEvent  
ComponentListener  
addComponentListener( )  
removeComponentListener( ) | *Component and its derivatives, including JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileChooser, JFrame, JLabel, JList, JScrollPane, JTextArea, and JTextField. |
## Other Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContainerEvent</td>
<td>Container and its derivatives, including JPanel, JApplet, JScrollPane, Window, JDialog, JFileChooser, and JFrame.</td>
</tr>
<tr>
<td>FocusEvent</td>
<td>Component and derivatives*</td>
</tr>
<tr>
<td>TextEvent</td>
<td>Anything derived from JTextComponent, including JTextArea and JTextField.</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>Component and derivatives*</td>
</tr>
</tbody>
</table>

**Methods**

- `add*Listener()`: Adds an event listener.
- `remove*Listener()`: Removes an event listener.

All Rights Reserved.
Other Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Listeners/Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseEvent</td>
<td>Component and derivatives*</td>
</tr>
<tr>
<td>MouseListener</td>
<td>addMouseListener() removeMouseListener()</td>
</tr>
<tr>
<td>MouseMotionListener</td>
<td>addMouseMotionListener() removeMouseMotionListener()</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>Window and its derivatives, including JDialog, JFileDialog, and JFrame.</td>
</tr>
<tr>
<td>WindowListener</td>
<td>addWindowListener() removeWindowListener()</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>JCheckBox, JCheckBoxMenuItem, JComboBox, JList, and anything that implements the ItemSelectable interface.</td>
</tr>
<tr>
<td>ItemListener</td>
<td>addItemListener() removeItemListener()</td>
</tr>
</tbody>
</table>
# Other Events

<table>
<thead>
<tr>
<th>Listener interface w/ adapter</th>
<th>Methods in interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionListener</td>
<td>actionPerformed(ActionEvent)</td>
</tr>
<tr>
<td>AdjustmentListener</td>
<td>adjustmentValueChanged(AdjustmentEvent)</td>
</tr>
<tr>
<td>ComponentListener ComponentAdapter</td>
<td>componentHidden(ComponentEvent)</td>
</tr>
<tr>
<td></td>
<td>componentShown(ComponentEvent)</td>
</tr>
<tr>
<td></td>
<td>componentMoved(ComponentEvent)</td>
</tr>
<tr>
<td></td>
<td>componentResized(ComponentEvent)</td>
</tr>
<tr>
<td>ContainerListener ContainerAdapter</td>
<td>componentAdded(ContainerEvent)</td>
</tr>
<tr>
<td></td>
<td>componentRemoved(ContainerEvent)</td>
</tr>
<tr>
<td>FocusListener FocusAdapter</td>
<td>focusGained(FocusEvent)</td>
</tr>
<tr>
<td></td>
<td>focusLost(FocusEvent)</td>
</tr>
<tr>
<td>KeyListener KeyAdapter</td>
<td>keyPressed(KeyEvent)</td>
</tr>
<tr>
<td></td>
<td>keyReleased(KeyEvent)</td>
</tr>
<tr>
<td></td>
<td>keyTyped(KeyEvent)</td>
</tr>
</tbody>
</table>
### Other Events

<table>
<thead>
<tr>
<th>Event Listener</th>
<th>Event Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseListener MouseAdapter</td>
<td>mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)</td>
</tr>
<tr>
<td>MouseMotionListener MouseMotionAdapter</td>
<td>mouseDragged(MouseEvent) mouseMoved(MouseEvent)</td>
</tr>
<tr>
<td>WindowListener WindowAdapter</td>
<td>windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)</td>
</tr>
<tr>
<td>ItemListener</td>
<td>itemStateChanged(ItemEvent)</td>
</tr>
</tbody>
</table>
Other basic components

- JLabel
- JCheckBox
- JRadioButton (add to ButtonGroup)
- JComboBox
- JList (list box)
- JTabbedPane
- JMenu, JMenuItem, JMenuBar
- JDialog
- All quite intuitive to use
- See examples in Chapter 13 HTML
What is a Bean?

• Just a class (thus easy to learn & use)
• Supports three concepts:
  – Properties
  – Events
  – Methods
• Properties, Methods, Events
  – For a property named size, create two methods
    • getSize( ) and setSize( ). (First letter automatically decapitalized).
  – boolean property: may also use “is” instead of “get.”
  – Ordinary methods are public
  – Events use the same “Listeners,” with add- and remove-methods like before
  – You can create your own event types
What is a Bean?

package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
    private int jumps;
    private Color color;
    private Spots spots;
    private boolean jmpr;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Spots getSpots() { return spots; }
    public void setSpots(Spots newSpots) {
        spots = newSpots;
    }
    public boolean isJumper() { return jmpr; }
    public void setJumper(boolean j) { jmpr = j; }
    public void addActionListener(
        ActionListener l) {
        // ...
    }
    public void removeActionListener(
        ActionListener l) {
        // ...
    }
    public void addKeyListener(KeyListener l) {
        // ...
    }
    public void removeKeyListener(KeyListener l) {
        // ...
    }
    // An "ordinary" public method:
    public void croak() {
        System.out.println("Ribbet!");
    }
} ///:~
Summary

- “Listener” event model and Beans are a big step forward
- Swing is the best UI library I’ve seen
- All Swing components are JavaBeans
- Numerous application builders use Beans
- Beans enable RAD environments
- Beans support more sophistication than shown here