CS193j, Stanford
Winter, 2002-03

Handout #32
Nick Parlante

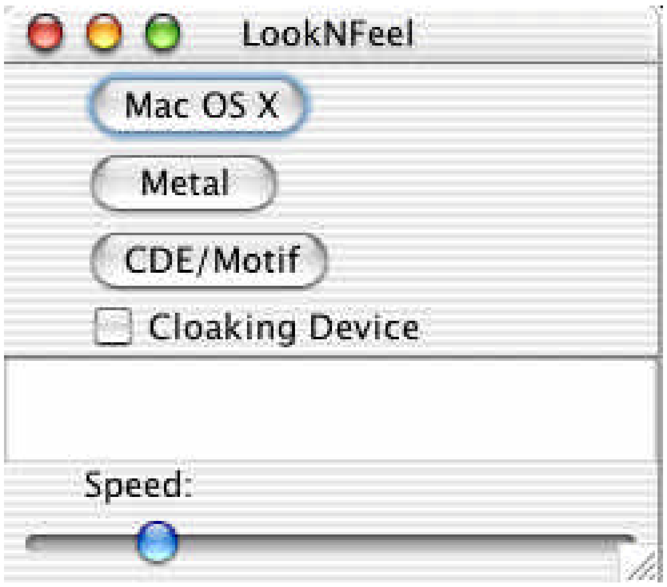
Advanced Java 2

Some advanced Java features...

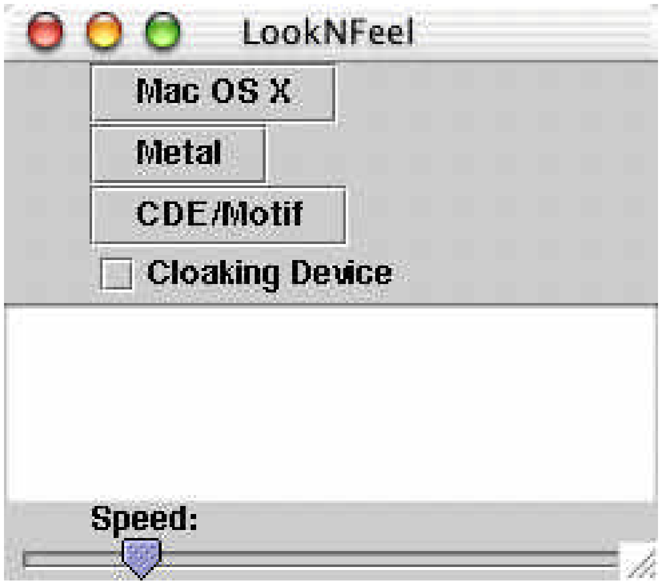
Look And Feel

Swing controls can take on different Look N Feel code, to resemble different operating systems.
The "metal" look and feel is neutral -- it looks the same on all platforms.
By default, a Swing app will use the LnF of the platform where it is running.

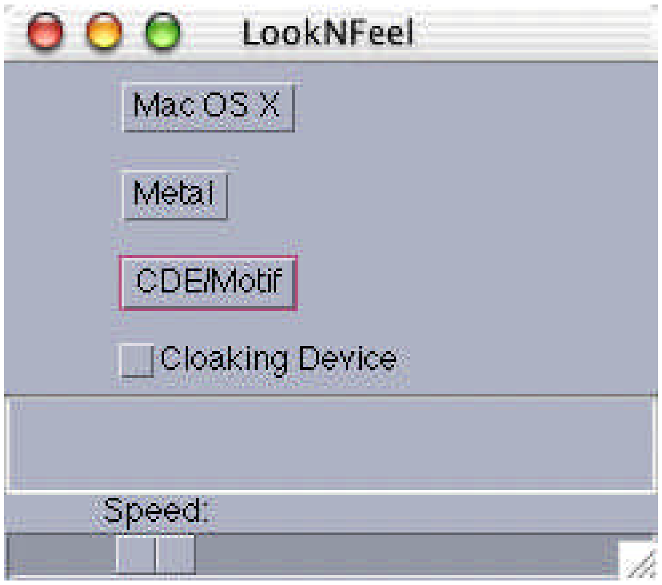
OS X



Metal



Motif / X-Windows



Look And Feel Code

```
// LookNFeel.java
/*
 Demonstrates changing the look and feel of a Swing app
*/
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;

public class LookNFeel extends JFrame {
    public LookNFeel() {
       super("LookNFeel");

       JComponent content = (JComponent) getContentPane();
       content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));

       // Get a list of the lnfs
       UIManager.LookAndFeelInfo[] looks = UIManager.getInstalledLookAndFeels();

       // Use a hash to map button pointers to lnf class names
       final HashMap map = new HashMap();

         final ActionListener lookListener = new ActionListener() {
          public void actionPerformed(ActionEvent e) {
             // Get the lnf name from the hash
             String look = (String) map.get(e.getSource());
             try {
                 // set the lnf
                   UIManager.setLookAndFeel(look);

                   // Need to do this to change an on-screen window
                SwingUtilities.updateComponentTreeUI(LookNFeel.this);
             }
             catch (Exception ignored) { }

          }
         };


       // For each look, create a button and put an entry
       // in the hashmap button->lnf-class
          for (int i=0; i<looks.length; i++) {
             JButton button = new JButton(looks[i].getName());
             button.addActionListener(lookListener);
             content.add(button);
             map.put(button, looks[i].getClassName());
          }

          // Put some junk in the window
          content.add(new JCheckBox("Cloaking Device"));
          content.add(new JTextField(10));
          content.add(new JLabel("Speed:"));
          content.add(new JSlider(0, 100, 20));

       pack();
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       setVisible(true);

       // Workaround for OSX bug where the content acts
       // like its  minimum size is its preferred size
       //content.setMinimumSize(new Dimension(100, 100));
    }

    public static void main(String[] args) {
       new LookNFeel();
    }
 }
```

New IO - NIO (1.4)

http://java.sun.com/j2se/1.4/docs/guide/nio/index.html
http://developer.java.sun.com/developer/technicalArticles/releases/nio/
Non-blocking IO for sockets
    vs. the old 1-thread-per-socket model
New buffering system
    Like a big array of binary data

Java Generics (probably in 1.5)

http://developer.java.sun.com/developer/technicalArticles/releases/generics/
Compile time types
The RT is the same -- really it's checking the type every time, but you don't have to put the cast in at CT
Cleans up the code and finds some erors at CT, which are now masked by all the casting

```
// Suppose Foo responds to the bar() message
ArrayList<Foo> list;
Foo f = ...
list.add(f);...
...
Iterator<Foo> it = list.iterator();
while(it.hasNext()) {
   it.next().bar();  // NOTE: no cast required, it.next() has correct CT type
   ...
}
```

foreach (1.5 or later)

There's a good chance that some sort of easy iterate-over-collections syntax will be added. It will work with anything that implements Iterator, and it will work with regular arrays.
This is not "elegant" in that a special syntax is added for a particular purpose.
However, I think it's a great idea -- iterating is so very common that having a special sytnax for it is great.

```
String[] strings ...;

for (String s : strings) {
   // use s
}
```