Why Java UI Programming is Special

- Platform independence ("write once runs anywhere"): Windows, Linux, Mac OS
- But user interfaces are highly platform dependent
  - Look; e.g., sizes, colors, layout of UI elements
  - Feel; e.g., their click- and selection behavior
- Question: how to write one UI that works on all platforms equally well?
  (not necessarily: “is on all platforms equal”!)
Outline

- **Graphical User Interfaces with Java: Basics**
  - Abstract Window Toolkit (AWT)
  - Java SWING
  - (Standard Widget Toolkit (SWT)): not today...

- **Advanced SWING Topics**
  - Pluggable Look And Feel (PLAF)
  - Event Dispatching Thread
  - SWING Animation
AWT

... heavyweight components
AWT was the first Java GUI API

- Released in 1995 by Sun Microsystems
- Now part of the Java Foundation Classes (JFC)

AWT was “superseded” by SWING, but it is still important:

- AWT provides the core services for SWING: interface to the OS, event dispatching, drag&drop support...

- AWT is the GUI toolkit for several mobile java profiles
Features of AWT

- Limited set of basic widgets:
  - Frame, Button, TextField, ...
- Event handling model
  - Keyboard, Mouse, ...
- Graphics and imaging APIs
  - Classes for fonts, colors, shapes
- LayoutManager mechanism
- Data Transfer from/to native platform clipboard
//imports
public class BeepButton extends Applet {
    public void init() {
        // Compose the GUI
        Button beepButton = new Button("Beep");
        this.add(beepButton);

        // Setup Event handlers
        beepButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                Component c = (Component)event.getSource();
                c.getToolkit().beep();
            }
        });
    }
    // STEP 3: Display the GUI (automatic -- this is an applet)
}
}
Native Peers

- AWT Widgets are not implemented in Java but by native peers

```java
public class JavaSideCanvas extends Canvas {
    static { //load peer’s library
        System.loadLibrary("NativeSideCanvas");
    }
    ...
    public native void paint(Graphics g); //call peer
    ...
}
```

An AWT widget with a native peer for the paint() method
(file JavaSideCanvas.java)

```c++
#include "jawt_md.h"
#include "JavaSideCanvas.h"

JNICALL Java_JavaSideCanvas_paint(
    JNIEnv* env,
    jobject canvas,
    jobject graphics)
{
    ... //do painting here
}
```

C++ implementation of the native peer’s paint() method
(file NativeSideCanvas.cpp)
SWING

... lightweight components
Problems with AWT

- AWT and its heavyweight native-peer approach was the first Java UI
  - Problem: UI design became hard, because every platform UI looks/behaves different
- Successor: Java SWING
  - only the top level containers (frames, applets, dialogs) are proxies for native widgets. The rest is written entirely in Java ("lightweight")
AWT / SWING Relationship
Essential SWING Concepts
(or: an intermediate outline)

- **UI structure**
  - Top-level containers
  - n-level containers
  - Leaf components

- **Application behavior**
  - Event Management: ActionListener, FocusListener, ...

- **UI design**
  - LayoutManagers
Top-level Containers

- Top-level containers (generally) represent a concrete window/dialog box on the OS desktop
  - JWindow, JFrame, JDialog, JApplet
- Therefore, they need native support from the platform OS
  - through a native peer and AWT
- SWING components that are placed inside a top-level container are then implemented purely in Java
Top-level Architecture

- Top-level containers consist of several layers:
  - Layered pane: contains a content pane and menu bar. May include more layers: dialogs, toolbars, tooltips, ...
  - Content pane: contains the visible UI components
  - Glass pane: usually invisible. Can be used to intercept mouse clicks, drag&drop actions etc.
n-level Containers

- SWING provides several containers that can hold other SWING UI components and implement different layouts and behavior:
  - JPanel: just a container with background
  - JSplitPane: divides two components with a bar
  - JTabbedPane: each component on a single tab
  - ...

- Containers can (in theory) also contain AWT components, but this is discouraged because of z-ordering issues
Leaf Components

- Very similar to AWT: JButton, JTextField, ...
  - SWING components are prefixed with "J"
- But SWING also provides more complex elements: JTable, JList, JProgressBar, JScrollBar, ...
- See Sun’s java-doc:
  
  http://java.sun.com/j2se/1.5.0/docs/api/javax/SWING
Event handling uses listener pattern (MVC):
- register listener at a component
- Component creates some event object and notifies all listeners by calling a callback
- Listener reacts by inspecting the event object’s source component and the payload ("type of event")
- Events and listeners provide the bridge between the UI and the model classes
Listeners

- Listeners are specified as interfaces
- Component provides addXYListener() and removeXYListener() methods
- Most important: ActionListener
- Other listeners:
  - ListSelectionListener, FocusListener, AncestorListener, DropTargetListener, and much more...
import javax.swing.*;
import java.awt.*;
import java.awt.event.*; // definiert Listener-Interfaces

class MyProgram implements ActionListener {
    private JButton button;

    void drawStuff()
    {
        ... 
        button = new JButton("Klick");
        button.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        if ( e.getSource() == button ) {
            System.err.println("Button geklickt!");
        }
    }
}
Anonymous Classes

• Java does not support function pointers

• Anonymous classes are the replacement for this; very common to provide listeners as anonymous classes:

```java
class MyProgram {
    private JButton button;

    void drawStuff() {
        ...
        button = new JButton("Klick");
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.err.println("Button geklickt");
                }
            }); //end of button.addActionListener();
    }
```
Usually, components are not placed on absolute x and y coordinates

Instead, layout managers do the actual layouting

Advantages:

- Portability

- Reacting on events such as resizing the window, adding/removing components

- If necessary to place components absolutely, set the layout manager to null
Some Layout Managers

- BorderLayout
- BoxLayout
- GridLayout
Layout Parametrization

- The sizes of components are generally determined by the layout manager.
- You can give “hints” like: setPreferredSize(), setMinimumSize(), ...
- Position in the logical layout: second argument to add(); e.g.:
  
  ```java
  add(new JLabel(“foo”), BorderLayout.SOUTH)
  ```
Layouting: When?

- Layouting a container is first done when calling `pack()` in the beginning.
- If you add/remove new components and the layout has to be updated, call `revalidate()` and then `paint()`.
//import several things

public class ExampleController {
    private SomeModelClass myModel = new SomeModelClass(); //the model
    private JButton starterButton; //the view

    public ExampleController() { //constructor
        starterButton = new JButton("Start Computation"); //create a button
        starterButton.addActionListener( //define behavior
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    starterButton.setActive(false); //update view
                    myModel.startComputation(); //call model
                } //end actionPerformed()
            } //end anonymous listener
        ); //end addActionListener call

        JFrame frame = new JFrame("Example"); //create a frame
        Container content = frame.getContentPane(); //get its content pane
        content.setLayout(new FlowLayout()); //define a layout manager
        content.add(starterButton); //add the button

        frame.pack(); //initiate layout computation
        frame.setVisible(true); //show the frame
    } //end constructor
}

Putting it all together: an example
Break
Outline

- **Graphical User Interfaces with Java: Basics**
  - Abstract Window Toolkit (AWT)
  - Java SWING
  - (Standard Widget Toolkit (SWT)): not today...

- **Advanced SWING Topics**
  - Pluggable Look And Feel (PLAF)
  - Event Dispatching Thread
  - SWING Animation
Pluggable Look&Feel

= PLAF
What is PLAF

- PLAF provides a way to change the concrete look of the UI (color schemes, icons, etc) and some behavior (double clicks etc.)
- Important PLAFs: the “native” L&F of the concrete platform and the cross-platform “metal” L&F
How it works

- Idea: components don’t contain the code for drawing. Instead they have a UI delegate (e.g., ButtonUI) that can be exchanged

- The delegate inherits from ComponentUI

- A LookAndFeel class collects information about the mapping between a JComponent and its corresponding ComponentUI

- Most of these mappings are stored as key-value pairs in a UIDefaults object
The Basic PLAF Mechanism

This is kind of hacky: the class just has to provide a static createUI method!
public static void setNativeLookAndFeel() {
    try {
        UIManager.setLookAndFeel(
            UIManager.getSystemLookAndFeelClassName());
    } catch (Exception e) { ... }
}

public static void setJavaLookAndFeel() {
    try {
        UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { ... }
}

public static void setMotifLookAndFeel() {
    try {
        UIManager.setLookAndFeel(
            "com.sun.java.SWING.plaf.motif.MotifLookAndFeel");
    } catch (Exception e) { ... }
}
Event Dispatching Thread

... updating the UI from background threads
SWING uses an Event Dispatching Thread (EDT) to process user input and drawing events.

Event-callbacks should return fast or else UI stalls.
- Querying a database from within the `actionPerformed()` method will make the UI unusable until the query is processed.

**But:** SWING is mostly not thread safe.
- For performance and lock ordering reasons.
- Updating the UI from within a separate database-query thread might crash the application! (or the UI just gets scattered...)
Executing Code in the EDT

- Important: Listener-callbacks (e.g., actionPerformed()) are called within the EDT

- `SWINGUtilities.invokeLater()` and `SWINGUtilities.invokeAndWait()` can be used to execute a method within the EDT thread

- `SWINGWorker` implements the Worker-pattern for more sophisticated inter-process communications

- Some thread-safe methods: repaint(), revalidate(), addXYListener(), removeXYListener()
private ResultSet result; //will contain the result
...
JButton button = new JButton("Query DB");

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        SWINGWorker worker = new SWINGWorker<ResultSet, Void>() {
            public ResultSet doInBackground() { //executed in worker thread
                ResultSet res = ...; //expensive DB query
                return res;
            }
            public void done() { //executed in event dispatch thread
                result = get(); //get the object returned by doInBackground()
            }
        }; //end anonymous SWINGWorker implementation
        worker.execute(); //execute worker thread
    } //end action performed
}); //end anonymous ActionListenerImplementation
SWING Animation

... embedding animated objects
Custom “static” Components

- Implementing a “normal” custom component in SWING
  - Extend JComponent and implement programmers’ API and does low-level event dispatching
  - If needed: interfaces for model and/or listeners
  - One or more ComponentUI classes that implement the actual painting routines
  - Register everything in the Look and Feel framework
- This does not work for animated components
  - SWING does not call paint(), if not needed
  - Animations need an animation thread
Custom Animated Components

- SWING does not provide a standard framework for animated components
- One simple approach: `javax.SWING.Timer`

```java
timer = new Timer(1000 / 30, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //do some slight changes, e.g., increase the color
    }
});
timer.start();
```
An Advanced Animation Framework

### Classes

- **JPanel**
- **Runnable**
- **AnimationThread**
  - isRunning: boolean
  - isPaused: boolean
  - buffer: Graphics
  - + start()
  - + stop()
  - + pause()
  - + resume()
  - + run()
  - animateObjects()
  - renderObjects()
  - paintScreen()

- **AnimatedObject**
  + animate()
  + render(Graphics g)

---

The diagram illustrates the relationships and methods of the classes involved in the animation framework.
```java
private Vector<AnimatedObject> objects;
private Image doubleBuffer;
private AnimatedPanel panel;
...
public void run() {
    while(running) {
        animateObjects();
        renderObjects();
        paintScreen();
        //compute time to sleep and sleep
    }
}
private void animateObjects() {
    for(AnimatedObject ao : objects) ao.animate();
}
private void renderObjects() {
    for (AnimatedObject ao : objects) ao.render(doubleBuffer.getGraphics());
}
private void paintScreen() {
    //draw doubleBuffer to panel
}
```
Computing Sleep Time

- Getting accurate time
  - System.currentTimeMillis(); //inaccurate
  - System.nanoTime(); //since J2SE 1.4.2
  - J2DTimer.getValue(); //good, but non J2SE

- Simplest case: just sleep for x ms (or ns)

- Better: measure execution time of animate/render/paint cycle and try to adjust to desired frame rate

- Best: Try to compensate execution time variations to have a defined frames-per-second ratio
Questions?