Java Course
Lecture 10 - Java UI (AWT/Swing)

IT Learning &
Outsourcing Center

www.pragmatic.bg

Summary

1. Java GUI Technologies:
   - AWT
   - Java Foundation Classes (JFC) / Swing
   - SWT
2. AWT Programming Model
   - AWT Components
   - Heavyweight and Lightweight Components
3. Swing Programming Model
   - Model-View-Controller (MVC)
   - Notifications Architecture

Java GUI Technologies

- Abstract Window Toolkit (AWT) – `java.awt.*`
  - The original Java GUI toolkit
  - Standard, old, stable
- Swing – `javax.swing.*`
  - Part of Java Foundation Classes (JFC)
  - Standard, more advanced, complex
- SWT – `org.ecipse.swt.*`
  - Non-standard, fast, well integrated in host
  - Low-level – JFace provides more power

Java GUI Technologies

- JavaFX – javafx.*
  - Shipped as part of Java 8.
  - Swing successor
  - Binding
  - Animations
  - Integration with Swing and SWT
  - Geometry 2D and 3D

AWT and Swing

- Java provides 3 standard sets of components for GUI programming:
  - AWT: classes in the `java.awt` package
  - Swing: classes in the `javax.swing` package
  - JavaFX: classes in `javafx`

  - `Note: JavaFX will not be part of this course.(since it is HUGE)`

Abstract Window Toolkit (AWT)

- The Abstract Window Toolkit is a portable GUI library
- AWT provides the connection between your application and the host native GUI
- AWT provides a high-level abstraction
  - AWT components depend on native code counterparts (called peers) to handle their functionality
  - These components are often called *heavyweight* components

Java Swing

- Swing implements GUI components that build on AWT technology
- Swing is implemented entirely in Java
- Swing components do not depend on peers to handle their functionality
  - These components are often called *lightweight* components

AWT - Pros and Cons

- AWT advantages
  - Speed: native components speed performance
  - Look and feel: AWT components more closely reflect the look and feel of the OS they run on
- AWT disadvantages
  - Portability: use of native peers creates platform specific limitations
  - Features: AWT supports only the lowest common denominator of features

Swing - Pros and Cons

- Swing advantages
  - Portability: Pure Java implementation
  - Features: Not limited by native components
  - Look and Feel: Pluggable look and feel
- Swing disadvantages
  - Performance: Swing components handle their own painting (instead of using APIs like DirectX on Windows)
  - Look and Feel: May look slightly different than native components

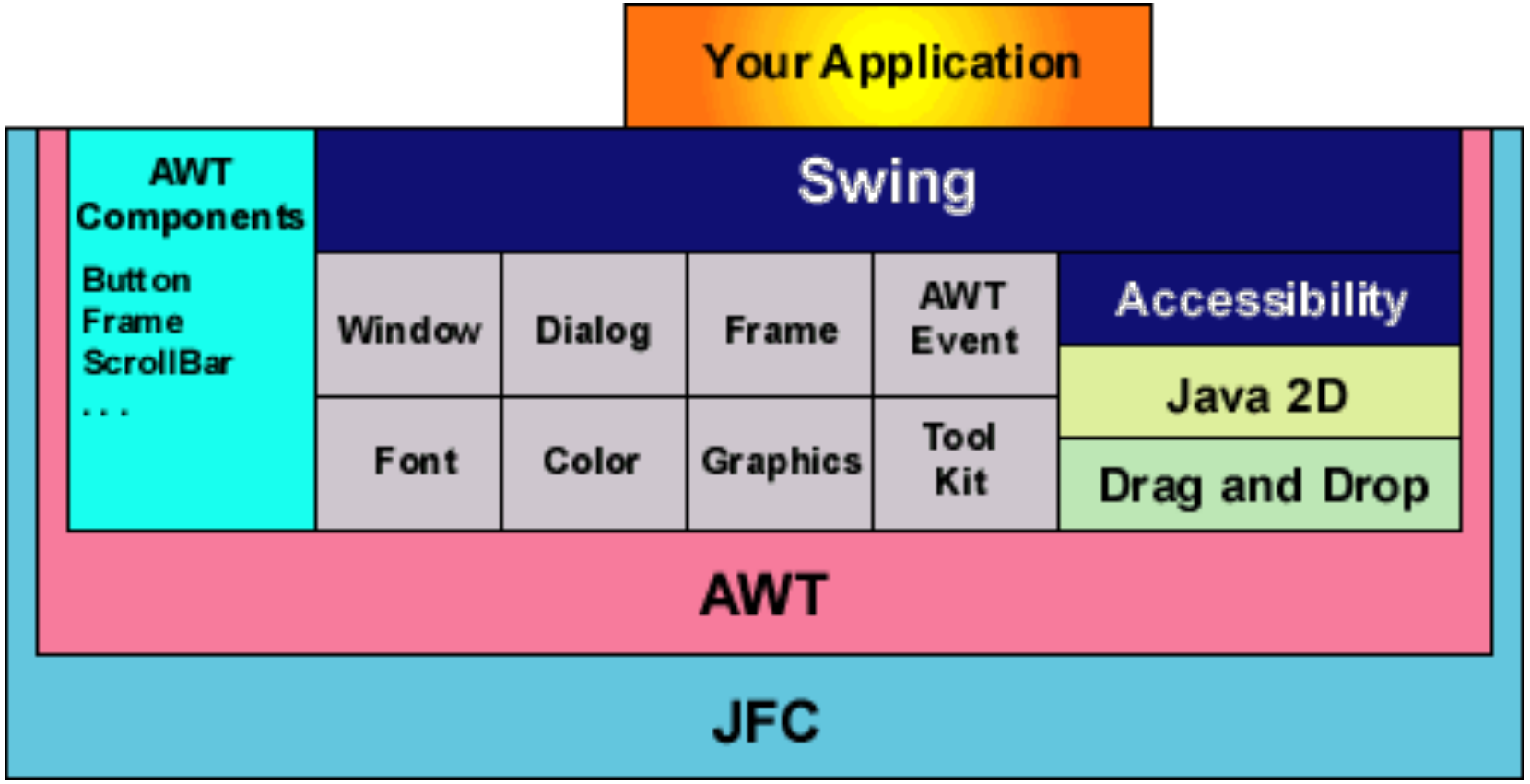
Java Foundation Classes (JFC) / Swing

Technology Overview

What is JFC?

- Java Foundation Classes (JFC) is a group of features for building GUIs
- First announced at the 1997 JavaOne conference
- JFC is a superset that contains AWT, Swing, Java2D and others
- JFC is a set of technologies designed to build GUI client applications that run on any client machine that supports J2SE

JFC/Swing Technologies Stack



JFC Features

- Abstract Window Toolkit (AWT) enables integration into the native desktop window system
- Java2D API provides high-quality 2D graphics, imaging, text and printing
- Swing GUI components extend the AWT to provide an extensible GUI component library with pluggable look-and-feel

JFC Features (2)

- Accessibility API enables assistive technologies as screen readers and Braille displays
- Internationalization allows applications to interact with users worldwide in their own languages and cultural conventions

AWT Programming Model

Abstract Window Toolkit (AWT)

- AWT was the original Java GUI framework
- AWT is composed of a set of basic components and containers
  - Components – buttons, labels, text boxes
  - Containers group components together according to a given layout policy
- All AWT components require associated "peer" classes, which are built using native code

AWT Components

- AWT components are objects that have some graphical representation on the UI
- The `java.awt.Component` class is the abstract superclass of the non-menu-related AWT components
- The `Component` class can be extended to create a lightweight component
- A *lightweight* component is a component that is not associated with a native window

AWT Heavyweight Components

- *Heavyweight* components are associated with their own native screen resources
- All AWT components are heavyweight
- Heavy components are always opaque

```
// Create heavyweight AWT button
Button heavy = new Button("So heavy");
```

Swing Programming Model

Swing Design Goals

- Pure Java implementation to promote cross-platform consistency
  - Extension to AWT
  - Provides compatibility with AWT
- Support multiple look-and-feels
- Harvest the benefits of model-driven programming
- Adhere to JavaBeans design principles to leverage IDEs and builder tools

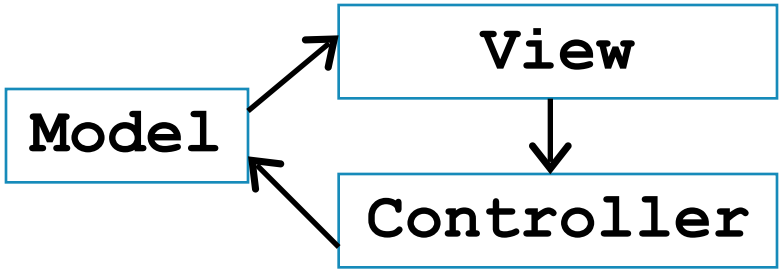
Swing Lightweight Components

- Lightweight components are pure Java
- Light components ”borrow” the screen resource on an ancestor
- All Swing components are light except for the top-level ones: `JWindow`, `JFrame`, `JDialog`, and `JApplet`
- Light components support transparency

```
JButton light = new JButton("Light as a breeze");
```

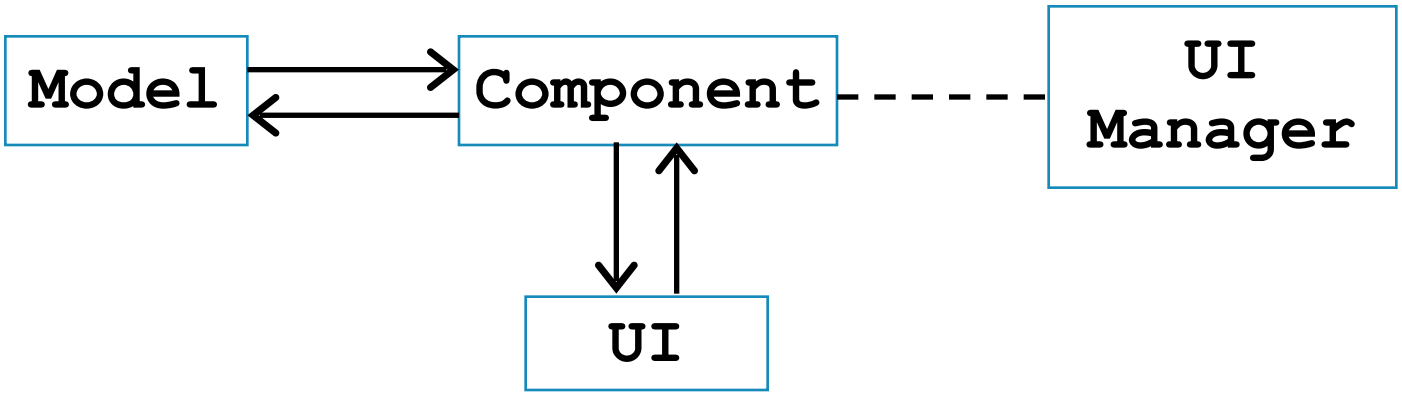
Model-View-Controller (MVC)

- Swing implements the MVC pattern
  - The *model* represents the data for the application
  - The *view* is responsible for the visual representation of the data
  - The *controller* accepts input from the user and translates it to changes in the model



To Use MVC or Not?

- It is very difficult to write a generic controller that is not specific of the view
- In Swing the view and the controller are tightly coupled into a single UI delegate



Swing MVC

- Swing treats the model as a separate element, just like traditional MVC
- Swing design is sometimes referred to as separable model architecture
- Swing defines abstract models shared among more than one component
  - Models in Swing are Java interfaces

`ButtonModel` `BoundedRangeModel` `TableModel`

`ComboBoxModel` `Document` `ListModel`

Separable Model API

- Swing components that use models support a JavaBean bound property for the model
  - `JButton` uses `ButtonModel` interface for its model and includes the following methods

```
public ButtonModel getModel()
public void setModel(ButtonModel model)
```

- If you don't supply your own model, one is created and installed internally

Ignoring Models Completely

- Most components provide model-defined API directly in the component class
- For example `JSlider`'s implementation of `getValue()` internally delegates the call to its model

```
public int getValue() {
  return getModel().getValue();
}
```

- **What's a model anyway?**

```
JSlider slider = new JSlider();
int value = slider.getValue();
```

Model Change Notification

- Models must notify views when their data changes
- There are two approaches for this in Swing:
  - Lightweight notification – a simple "changed" event is sent and model must be queried to find out what has changed
  - Stateful notification – a precise information of what has changed is sent in each event

Example of Lightweight Notification

- The `ChangeListener` interface

```
public void stateChanged(ChangeEvent e)
```

- `ChangeEvent` supplies event "source"
- Only one event instance per component is enough
- Models that use this interface define the following methods

```
public void addChangeListener(ChangeListener l)
public void removeChangeListener(ChangeListener l)
```

Model Notification

- We can subscribe to model changes

```
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
      // Need to query the model
      // to get updated value...
      BoundedRangeModel m =
         (BoundedRangeModel)e.getSource();
      System.out.println("model changed: " +
          m.getValue());
   }
});
```

- This way the source is the model

Component Notification

- Some components provide delegates for subscription

```
JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
      // The source will be
      // the slider this time..
      JSlider s = (JSlider)e.getSource();
      System.out.println("value changed: " +
         s.getValue());
   }
});
```

- This way the source is the component

Example of Stateful Notification

- The `ListSelectionListener`

```
void valueChanged(ListSelectionEvent e)
```

- `ListSelectionEvent` describes the change using a row interval
- New event instance is created for every change
- Models define the following methods

```
public void addListSelectionListener(ListSelectionListener l)
public void removeListSelectionListener(ListSelectionListener l)
```

Stateful Notification Example

- Listeners can query the event object directly to find out what has changed

```
String items[] = {"One", "Two", "Three");
JList list = new JList(items);
ListSelectionModel selModel =
   list.getSelectionModel();
selModel.addListSelectionListener(
      new ListSelectionListener() {
   public void valueChanged(ListSelectionEvent e) {
      // Get change information directly
      // from the event instance ...
      if (!e.getValueIsAdjusting()) {
         System.out.println("selection changed: " +
           e.getFirstIndex());
      }
   }
});
```

UI Plumbing

- UIs (look and feel) are implicitly installed/de-installed by Swing

```
protected void setUI(ComponentUI newUI) {
    if (ui != null) {
        ui.uninstallUI(this);
    }
    ComponentUI oldUI = ui;
    ui = newUI;
    if (ui != null) {
        ui.installUI(this);
    }
    invalidate();
    firePropertyChange("UI", oldUI, newUI);
}
```

Automatic View Updates

- UIs subscribe to model listeners upon installation
- UI gets updated when model change
- If you find that a component is not updating when the model changes, it is a bug!

JavaBeans

JavaBeans

- Any Java class that follows certain design conventions can be a JavaBeans component
- These conventions make it possible to have tools that can use, reuse, replace, and connect JavaBeans

JavaBeans

- The required conventions are:
  - It should have a default constructor
  - Its properties should be accessed using get, set and its methods following a standard naming convention
  - The class should be serializable

JavaBeans Example

```
import java.io.Serializable;

public class MyBean implements Serializable{
  protected int theValue;

  public MyBean() { // Default constructor
  }

  public int getMyValue() { // Property getter
    return theValue;
  }

  public void setMyValue(int newValue) { // Setter
    theValue = newValue;
  }
}
```

Problems

1. Describe the difference between AWT, Swing and SWT. Show the strong and weak points in these technologies.
2. Describe the AWT programming model. What is component? What is container?
3. Describe the Swing programming model. What is lightweight component?
4. Describe the Model-View-Controller (MVC) architecture.
5. Describe the notification mechanisms in Swing.

Problems

6. Create a bean `Car` which has three properties: type, model and price. Add a constructor and accessory methods.
7. Create a simple GUI application that enters two numbers and calculates their sum in:

- AWT
- Swing
- SWT
- Sample solution: