



# Internationalization, Localization and Java

Java is ready for creating internationalized applications:

- Unicode support
- Locale classes and locale sensitive classes like:
  - ▷ *Collator, BreakIterator, Calendar, DateFormat, NumberFormat, MessageFormat, Currency*
- Resource bundles

However:

- JFC/Swing only support *static* internationalization.
- With customized UI classes we can achieve *dynamic* internationalization.

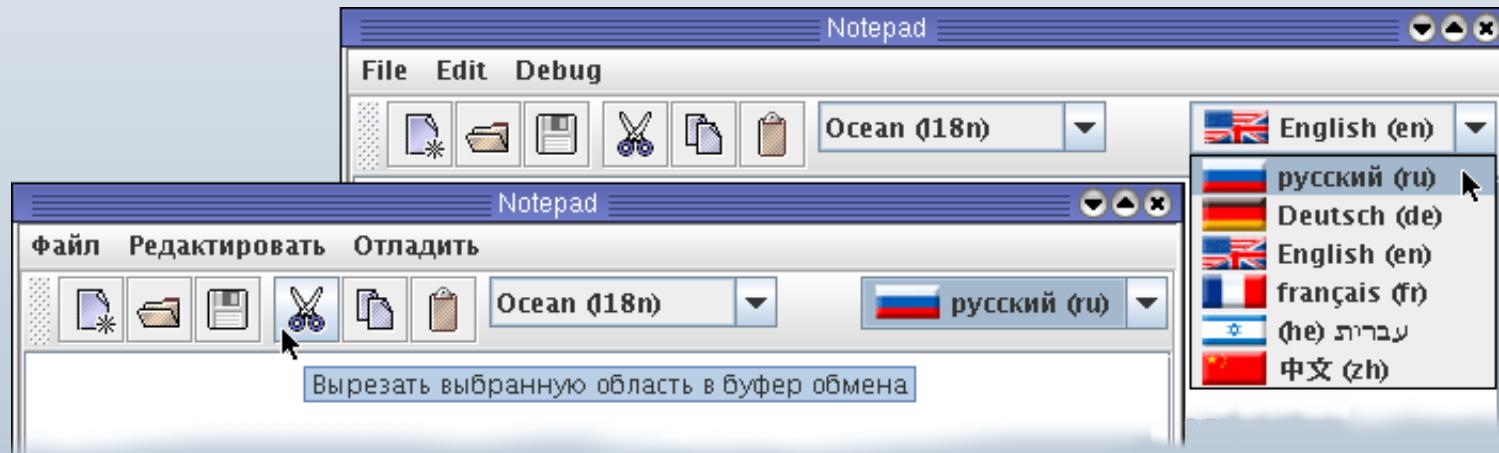
# Static vs. Dynamic Internationalization

Static Internationalization:

- *The locale of the application is determined once at startup.*

Dynamic Internationalization:

- *The locale of the application can be changed at runtime.*



Use cases for dynamic internationalization:

- *Multilingual applications, long running applications with complex GUI, enable “best choose” if none of the GUI languages is the user’s native language.*

## How to use the new I18n Look and Feels

- Typical coding in an application that is not internationalized:

```
JLabel label = new JLabel();
label.setText("Monday");
```

- Coding in an application that is *statically* internationalized:

```
ResourceBundle resource = ResourceBundle.getBundle("MyApplication", Locale.getDefault());
...
JLabel label = new JLabel();
label.setText(resource.getString("MyApplication.Monday"));
```

- Coding in a *dynamically* internationalized application (with an I18n L&F):

```
JLabel label = new JLabel();
label.setText("<i18n>MyApplication.Monday</i18n>");
```

..which can be abbreviated as:

```
JLabel label = new JLabel();
label.setText(I18nUtils.wrapI18nKey("MyApplication.Monday"));
```

## Resource bundles

Resource bundles contain key/value pairs of locale-specific objects:

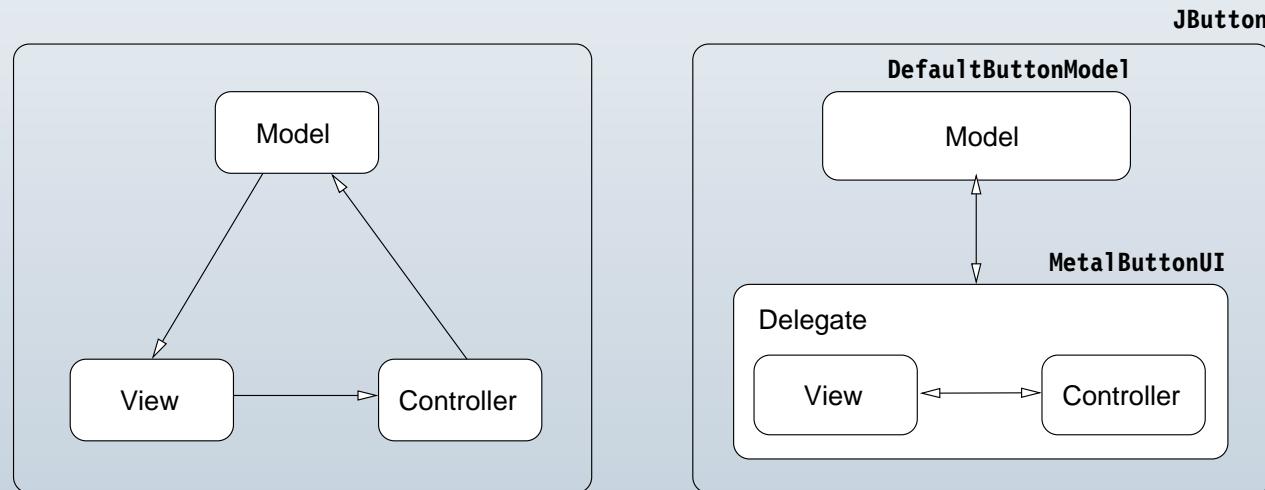
- ▷ A resource bundle can be a class or a property file.
- ▷ A resource bundle is identified by a base name and a locale.
- ▷ All resource files with the same base name share the same keys.
- ▷ Resource bundles are chained together with the most specialized bundle being at the top of the chain:

```
baseName + "_" + language1 + "_" + country1 + "_" + variant1  
baseName + "_" + language1 + "_" + country1  
baseName + "_" + language1  
baseName + "_" + language2 + "_" + country2 + "_" + variant2  
baseName + "_" + language2 + "_" + country2  
baseName + "_" + language2  
baseName
```

- ▷ Resources in classes take precedence over property files.

# The Swing MVC architecture

Swing uses a simplified MVC pattern called Model-Delegate:



- ▶ The UI-classes are the delegate part responsible for painting a component.  
(Every components `paintComponent` method calls the `paint` method of its UI class)

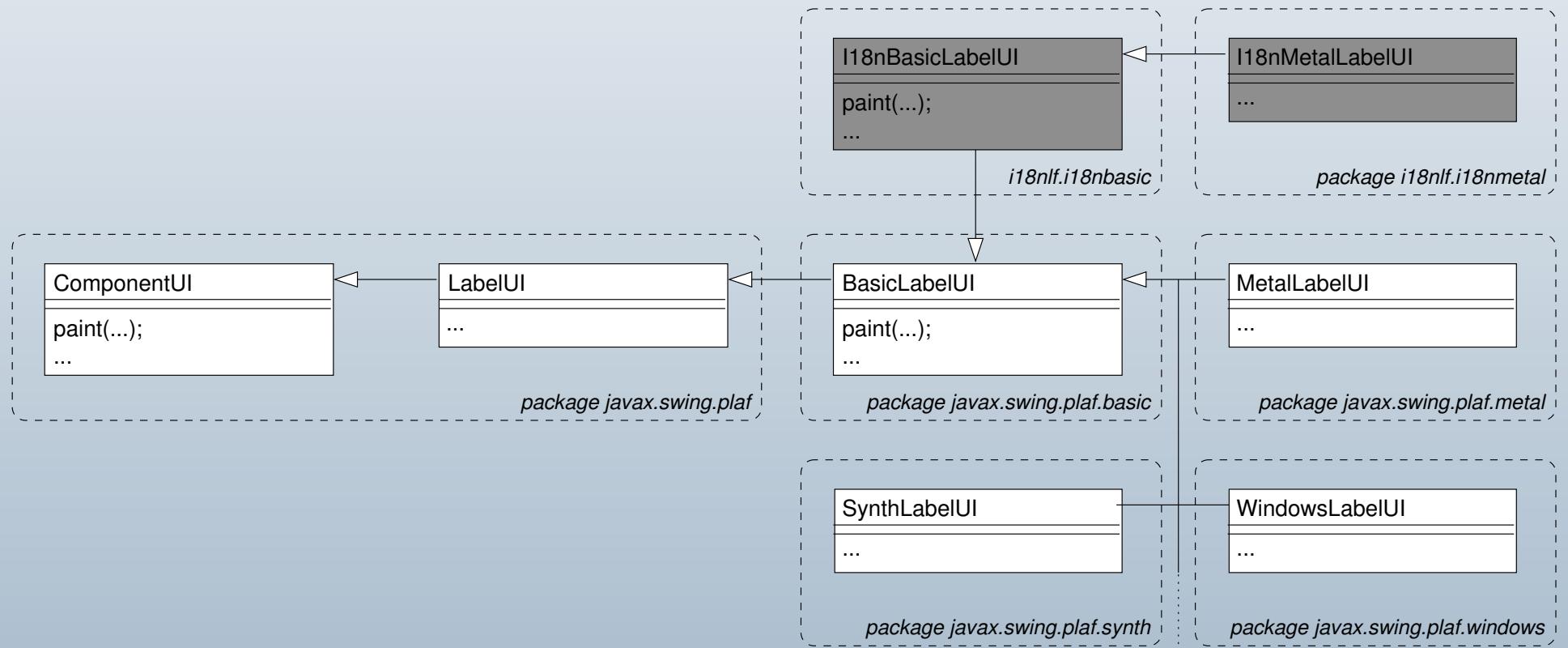
```
protected void paintComponent(Graphics g) {  
    UIManager.getUI(this).paint(g, this);  
}
```

- ▶ The UI-classes are Look and Feel dependent (e.g. `MetalButtonUI`, `WindowsButtonUI`)

# The Swing Pluggable Look and Feel architecture

Every Swing component has a Look and Feel dependent UI delegate:

- e.g. *MetalLabelUI* is the UI delegate of *JLabel*



# How to convert a classical L&F into an I18n L&F

The UI class queries the data of the model and uses it for layout and painting.

- ▷ *I18n L&F introduces a new level of indirection*
- ▷ *It uses the data and the current locale as key to query the real data*

```
// From BasicLabelUI.java
```

```
public void paint(Graphics g, JComponent c) {  
    JLabel label = (JLabel)c;  
    String text = label.getText();  
  
    ...  
    g.drawString(x, y, text);  
}  
  
public Dimension getPreferredSize(JComponent c) {  
    JLabel label = (JLabel)c;  
    String text = label.getText();  
  
    Font font = label.getFont();  
    ... // compute dimension based on font and text  
    return dimension;  
}
```

```
// From I18nBasicLabelUI.java
```

```
public void paint(Graphics g, JComponent c) {  
    JLabel label = (JLabel)c;  
    String text = label.getText();  
    text = I18nUtils.getResourceString(text); //(*)  
    ...  
    g.drawString(x, y, text);  
}  
  
public Dimension getPreferredSize(JComponent c) {  
    JLabel label = (JLabel)c;  
    String text = label.getText();  
    text = I18nUtils.getResourceString(text); //(*)  
    Font font = label.getFont();  
    ... // compute dimension based on font and text  
    return dimension;  
}
```

# The I18nUtils utility class

- Convenience functions for key handling:
  - ▷ *isI18nKey(), wrapI18nKey(), stripI18nKey()*
- Convenience functions for resource handling:
  - ▷ *getResourceMnemonic(), getResourceString()*
- Repainting the whole GUI according to the new locale:

```
public static void repaintI18nComponents() {
    Frame[] frames = Frame.getFrames();
    boolean ltr = ComponentOrientation.getOrientation(Locale.getDefault()).isLeftToRight();
    for (int i = 0; i < frames.length; i++) {
        if (ltr) {
            frames[i].applyComponentOrientation(ComponentOrientation.LEFT_TO_RIGHT);
        } else {
            frames[i].applyComponentOrientation(ComponentOrientation.RIGHT_TO_LEFT);
        }
        javax.swing.SwingUtilities.updateComponentTreeUI(frames[i]);
    }
}
```

I18nUtils
-resourceBundles:Hashtable
+isI18nKey(key:String):String
+wrapI18nKey(key:String):String
+stripI18nKey(key:String):String
+getResourceMnemonic(key:String):int
+getResourceString(key:String):String
+repaintI18nComponents():void

## The LocaleChooser

A nice component for locale switches:

- *Takes a directory which contains subdirectories for supported locales.*
- *Subdirectories have the name of the locale they represent.*
- *Subdirectories may contain a flag picture of the locale:*

i18n/  
i18n/de  
i18n/de/flag.gif  
i18n/de\_CH  
i18n/de\_CH/flag.gif



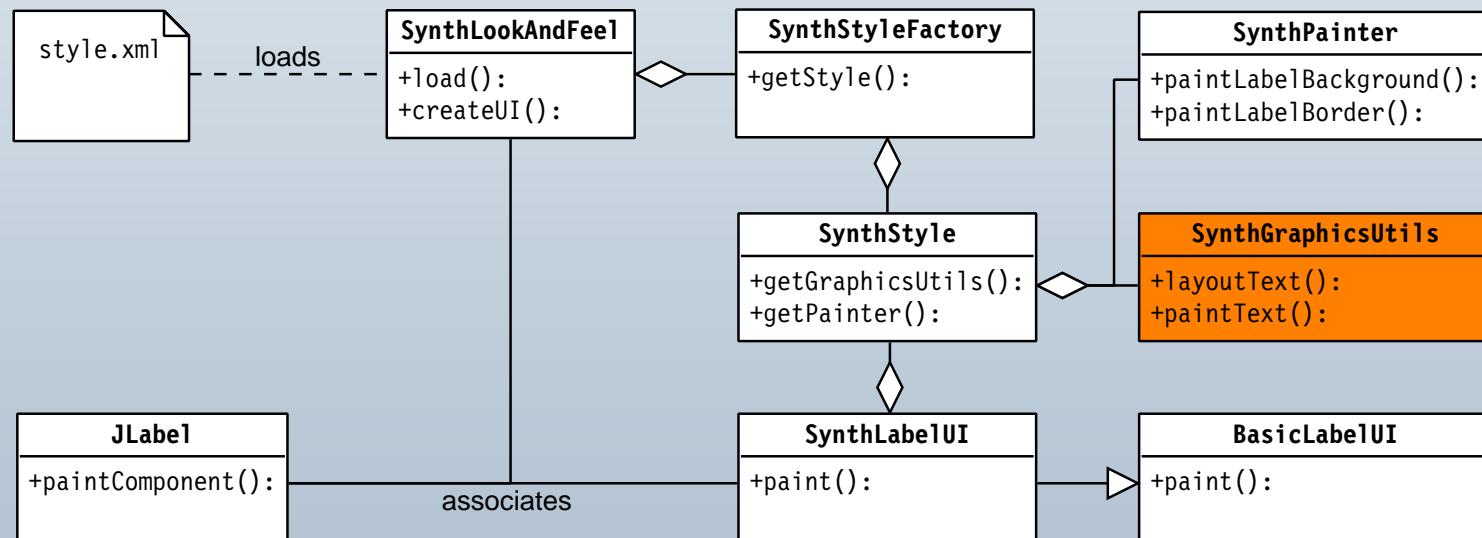
- *It changes the default locale and repaints the whole application.*

```
public void actionPerformed(ActionEvent e) {  
    ...  
    Locale locale = getSelectedLocale();  
    if (!Locale.getDefault().equals(locale)) {  
        Locale.setDefault(locale);  
        System.setProperty("user.locale", localeStr);  
        I18nUtils.repaintI18nComponents();  
    }  
}
```

# The architecture of the Synth Look and Feel

In Synth all text-related painting and layout is done by SynthGraphicsUtils.

- ▷ *The changes for dynamic internationalization have to be done only in one central place.*
- ▷ *We have to workaround the fact that the rendering of HTML labels is still handled by the BasicUI classes!*



# A customized SynthGraphicsUtils class

```
public class I18nSynthUtils extends SynthGraphicsUtils {  
  
    public void paintText(SynthContext ss, Graphics g, String text,  
                          Rectangle bounds, int mnemonicIndex) {  
        // Fetch the localized version of text according to the current locale.  
        text = I18nUtils.getResourceString(text);  
        JComponent comp = ss.getComponent();  
        if (BasicHTML.isHTMLString(text)) {  
            BasicHTML.updateRenderer(comp, text);  
        }  
        super.paintText(ss, g, text, bounds, mnemonicIndex);  
    }  
  
    public Dimension getPreferredSize(...) {  
        ...  
    }  
  
    public String layoutText(...) {  
        ...  
    }  
    ...  
}
```