Java and C# in depth
Carlo A. Furia, Marco Piccioni, and Bertrand Meyer

Java: Graphical User Interfaces (GUI)

With material from Christoph Angerer
The essence of the Java Graphics API
Abstract Windowing Toolkit (AWT)

- The first available API for Java GUIs
- Platform independent (if there are JVMs)
- It does not look the same on all platforms
- Calls native libraries on the user system
- Handles events, cut and paste, drag and drop, keyboard focus, user input
- Still used directly for top-level containers (frames, applets, dialogs)
Java 2D

Introduced in 1.2
Basic and advanced drawing operations
Image manipulation
Text
Printing
Can be used directly, or indirectly via Swing
Handles Swing’s rendering operations (e.g. GUI component background and border)
The essence of the Java Graphics API
Java Swing

Introduced in 1.2

Main Java GUI library for desktop apps

Lightweight

- relies on 2D for graphics
- relies on AWT for top-level containers and application behavior via event management
Swing Concepts

UI structure

- Top-level containers (need native support): JFrame, JDialog, JApplet.
- N-level containers (implemented in Java): JPanel, JSplitPane, JTabbedPane, ...
- Leaf components: JButton, JTextField, JTable, JList, JProgressBar, JScrollPane, ...

UI design via layout managers
Top-level containers always contain a JRootPane

JRootPane contains JLayeredPane, a content pane, JMenuBar, and a glass pane

Can be used to intercept mouse clicks and paint over multiple components
Top-level containers structure: JLayeredPane

JRootPane contains a JLayeredPane

JLayeredPane contains and positions a content pane, an optional menu bar, and possibly dialogs and toolbars

Enables Z-ordering of the contained components
Top-level containers structure: content pane

In the content pane go your visible GUI components

JPanel is the typical content pane

First create a JPanel, then add components to it, and then:

myTopLevelContainer.setContentPane(myJPanel)
Top-level containers structure: glass pane

The glass pane sits on top of everything, fills the entire view, and it’s by default invisible.

Used to intercept mouse and keyboard clicks, drag & drop, and to draw over the entire UI, especially if there are already other components.
public class MyFrame extends JFrame {
    private JPanel contentPane;

    public MyFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(100, 100, 450, 300);
        contentPane = new JPanel();
        getContentPane().add(contentPane);
    }
}
public class MyFrame extends JFrame {
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    MyFrame frame = new MyFrame();
                    frame.setVisible(true);
                } catch (Exception e) { ... }
            }
        });
    }
}...
public class MyGlassPane extends JComponent {

@Override

protected void paintComponent(Graphics g) {
    Rectangle clip = g.getClipBounds();
    g.setColor(Color.BLUE);
    g.fillRect(clip.x, clip.y, clip.width, clip.height);
}
}
public class MyFrame extends JFrame {
  private JPanel contentPane;

  public MyFrame() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 450, 300);
    contentPane = new JPanel();
    getContentPane().add(contentPane);
    setGlassPane(new MyGlassPane());
    getGlassPane().setVisible(true);
  }
}...
Before and after applying the glass pane
Threading Concepts

Threads running in a Swing application

- Main thread: runs the application’s main method
- Toolkit thread: captures system events (mouse, keyboard, …)
- Event Dispatching Thread (EDT): responsible for executing any method that modifies a component’s state
  - dispatches the events captured by the toolkit thread to the appropriate components
  - the recommended way to interact with Swing
Caveats

Swing is **not** a thread-safe API. It is a single-threaded API, and that thread is the EDT.

Swing should be used only on the EDT (see previous example).

Hint: don’t perform long-lasting computations or I/O accesses in a method executed by the EDT.

Where do you perform these computations then?
Handling heavy computations and I/O

```java
SwingUtilities.invokeLater(new Runnable(){
    public void run(){
        // lengthy computation
    }
});
```

This posts a new task on the EDT by invoking `EventQueue.invokeLater(…)`

From Java 6 you can use `SwingWorker`, a utility class to run a task on a background thread, and post intermediate or final results on the EDT
SwingWorker usage

For running long-running tasks in a different thread so as to prevent the GUI from being unresponsive

For updating GUI with the results produced by the long-running task at the end of the task through method `done()`

For updating GUI from time to time with the intermediate results produced and published by the task through methods `publish()` and `process()`
Handling events

Observer design pattern

- Components maintain a list of objects implementing listener interfaces (listeners)
- You can register/unregister a listener XYZ on a component c:
  ```java
c.addXYZListener Or c.removeXYZListener
  ```
- When the component fires an event, all registered listeners are notified using a callback
- The reaction code is typically in the (anonymous inner) class implementing the listener interface
Adding a button and an associated action

```java
public class MyFrame extends JFrame {

private JButton myButton;

public MyFrame(){...

myButton = new JButton(“Push me!”);
contentPane.setLayout(new FlowLayout());
myButton.addActionListener(new ActionListener(){

    public void actionPerformed(ActionEvent e){
        myButton.setText("Works!");
    }
});

contentPane.add(myButton);
setContentPane(contentPane);
}...
```
Firing an ActionEvent

Push me!

Works!
# Some Listeners and related Components

<table>
<thead>
<tr>
<th>Event Listener</th>
<th>Listener methods</th>
<th>Register on</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionListener</td>
<td>actionPerformed()</td>
<td>JButton, JComboBox, JFileChooser, MenuItem, JTextField, ...</td>
</tr>
<tr>
<td>FocusListener</td>
<td>focusGained(), focusLost()</td>
<td>Component</td>
</tr>
<tr>
<td>MouseListener</td>
<td>mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()</td>
<td>Component</td>
</tr>
<tr>
<td>MouseMotionListener</td>
<td>mouseDragged(), mouseMoved()</td>
<td>Component</td>
</tr>
</tbody>
</table>
## Some more Listeners and Components

<table>
<thead>
<tr>
<th>Event Listener</th>
<th>Listener methods</th>
<th>Register on</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyListener</td>
<td>keyPressed(), keyReleased(), KeyTyped()</td>
<td>Component</td>
</tr>
<tr>
<td>TextListener</td>
<td>textValueChanged()</td>
<td>TextComponent</td>
</tr>
<tr>
<td>CaretListener</td>
<td>caretUpdate()</td>
<td>JTextComponent</td>
</tr>
<tr>
<td>MenuListener</td>
<td>menuCanceled(), menuDeselected(), menuSelected()</td>
<td>JMenu</td>
</tr>
</tbody>
</table>
Layout Managers

Used to harmonize component placement

They typically decide the component size

Can be composed with one another

React in a ‘predictable’ way when adding/removing components and resizing the application window

Absolute positioning (x, y, size) is still possible:

```java
contentPane.setLayout(null);
```
Some Layout Managers

Border Layout: 5 areas

BoxLayout: single row/column

GridLayout: cells are same size

FlowLayout

http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
Some more Layout Managers

CardLayout: different components at different times

GridBagLayout: cells of different size

SpringLayout: fine-grained control

http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
Do it yourself

You can experiment with the Java GUI APIs trying the RPN calculator assignment

A nice GUI designer tool that produces clean GUI code in the background and you may want to have a look at is Google’s Window Builder (Eclipse plug-in):

https://developers.google.com/java-dev-tools/wbpro/