Advanced Programming
Graphical User Interface (GUI)
Human-Machine Interfaces

The ways in which a software system interacts with its users.

- Command Line
- Graphical User Interface - GUI
- Touch User Interface - TUI
- Multimedia (voice, animation, etc.)
- Intelligent (gesture recognition, conversational, etc.)
Graphical User Interfaces
Visual communication between software and users.

- **AWT** (Abstract Windowing Toolkit)
- **Swing** – part of JFC (Java Foundation Classes)
- **SWT** (IBM)
- **Java FX**
- **XUL**
- ...  
  - Java 2D
  - Java 3D
The Stages of Creating a GUI Application

- **Design**
  - Create the **containers**
  - Create and arrange the **components**

- **Functionality**
  - Define the user-components **interaction**
  - Attach **actions** to components
  - Create the action **handlers**

- **Considerations**
  - Programmatic – Declarative – Visual
  - Separation between the GUI and application logic
import java.awt.*;
public class AWTExample {
    public static void main (String args []) {
        // Create the window (frame)
        Frame f = new Frame("O fereastra");

        // Set the layout of the frame
        f.setLayout (new FlowLayout());

        // Create the components
        Button b1 = new Button("OK");
        Button b2 = new Button("Cancel");

        // Add the components to the frame
        f.add(b1);
        f.add(b2);
        f.pack();

        // Show the frame
        f.setVisible(true);
    }
}
AWT Components

- Button
- Canvas
- Checkbox
- CheckBoxGroup
- Choice
- Container
- Label
- List
- Scrollbar
- TextComponent
- TextField
- TextArea

AWT Components are platform-dependent, each of them having an underlying native peer.
Infrastructure

- **Components**: *Button, CheckBox, etc.*
  - A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Properties common to all components are:
    - location, x, y, size, height, width, bounds, foreground, background, font, visible, enabled,...

- **Containers**: *Window, Frame, Dialog, Panel, etc.*
  - A generic component containing other components.

- **LayoutManager**: *FlowLayout, GridLayout, etc.*
  - The interface for classes that know how to lay out Containers.

- **EventObject**: *ActionEvent, TextEvent, etc.*
  - An event indicates that a component-defined action occurred.
A layout manager is an object that controls the size and arrangement (position) of components inside a container.

Each Container object has a layout manager.

All classes that instantiate objects for managing positioning implements LayoutManager interface.

Upon instantiation of a container it is created an implicit layout manager associated with it:

- frames: BorderLayout
- panels: FlowLayout
import java.awt.*;
public class TestLayout {
    public static void main(String args[])
    {
        Frame f = new Frame("Grid Layout");
        f.setLayout(new GridLayout(3, 2));
        Button b1 = new Button(" Button 1");
        Button b2 = new Button("2");
        Button b3 = new Button(" Button 3");
        Button b4 = new Button("Long - Named Button 4");
        Button b5 = new Button(" Button 5");
        f.add(b1); f.add(b2); f.add(b3); f.add(b4); f.add(b5);
        f.pack();
        f.setVisible(true);
    }
}

Frame f = new Frame("Flow Layout");
f.setLayout(new FlowLayout());
import java.awt.*;
public class TestBorderLayout {
    public static void main ( String args []) {

        Frame f = new Frame (" Border Layout "); // This is the default for frames
        f.setLayout (new BorderLayout());

        f.add(new Button(" North "), BorderLayout.NORTH );
        f.add(new Button(" South"), BorderLayout.SOUTH );
        f.add(new Button(" East"), BorderLayout.EAST );
        f.add(new Button(" West "), BorderLayout.WEST );
        f.add(new Button(" Center "), BorderLayout.CENTER );
        f.pack ();
        f.setVisible(true);
    }
}
User Interactions

**Event-Driven Programming**

**Event**: clicking a button, altering the text, checking an option, closing a frame, etc.

**Source**: the component that generates an event.

**Listener**: the responsible for receiving and handling (consuming) events.

*many-to-many*

Observing the state of an entity within a system (*Publish-Subscribe*)
Using Anonymous Classes

class MyFrame extends Frame {
    public MyFrame ( String title ) {
        ...
        button.addActionListener( new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                MyFrame.this.setTitle(
                    "You pressed the button " + e.getActionCommand());
            }
        });
        ...
    }
    ...
}

Using Lambda Expressions

    ...
    button.addActionListener( (ActionEvent e) -> {
        MyFrame.this.setTitle(
            "You pressed the button " + e.getActionCommand());
    });
    ...
}
Using Method References

class MyFrame extends Frame {

    public MyFrame ( String title ) {
        ... 
        button.addActionListener( this::onButtonPressed );

        checkbox.addItemListener( this::onItemChanged );
        ...
    }

    //Your own, suggestively called, methods

    private void onButtonPressed(ActionEvent e) {
        this.setTitle("You pressed the button");
    }

    private void onItemChanged(ItemEvent e) {
        this.setTitle("Checkbox state: "+ check.getState());
    }
}

Swing

- **Extends** the core concepts and mechanisms of AWT; *we still have components, containers, layout managers, events and event listeners.*
- **Replaces completely** the AWT component set, providing a new set of components, capable of sorting, printing, drag and drop and other “cool” features.
- Brings **portability** to the GUI level; no more *native peers*, all components are “pure”.
- Based on **Separable Model-and-View** design pattern.
- "**Component Oriented Programming"**
Swing Components

- **Atomic Components**
  - JLabel, JButton, JCheckBox, JRadioButton, JToggleButton, JScrollPane, JSlider, JProgressBar, JSeparator

- **Complex Components**
  - JTable, JTree, JComboBox, JSpinner, JList, JFileChooser, JColorChooser, JOptionPane

- **Text Editing Components**
  - JTextField, JFormattedTextField, JPasswordField, JTextArea, JEditorPane, JTextPane

- **Menus**
  - JMenuBar, JMenu, JPopupMenu, JMenuItem, JCheckboxMenuItem, JRadioButtonMenuItem

- **Intermediate Containers**
  - JPanel, JScrollPane, JSplitPane, JTabbedPane, JDesktopPane, JToolBar

- **High-Level Containers**
  - JFrame, JDialog, JWindow, JInternalFrame, JApplet
Similarities and Differences with AWT

"J" Convention

java.awt.Frame - javax.swing.JFrame
java.awt.Button - javax.swing.JButton
java.awt.Label - javax.swing.JLabel

New Layout Managers

BoxLayout, SpringLayout, GroupLayout, OverlayLayout, etc.

HTML Aware Components

JButton simple = new JButton("Dull text");

JButton html = new JButton("<html><u>Cool</u> <i>text</i></html>");
**JComponent**

*JComponent* is the base class for all Swing components, except top-level containers: `JFrame`, `JDialog`, `JApplet`.

*JComponent* extends *Container*

- Support for **tool tips** - `setToolTip`
- Support for **borders** - `setBorder`
- Enhanced support for **sizing and positioning**
  - `setPreferredSize`, ...
- **Opacity** control - `setOpaque`
- **Keyboard bindings**
- “Pluggable” **look and feel**
- Double-Buffering, Support for accessibility, etc.
Swing Architecture

Swing architecture is “rooted” in the MVC design:

- *Model* – the data for the application
- *View* – the visual representation of the data
- *Controller* – takes user input on the view and translates that to changes in the model.

**Separable Model Architecture**

Model + (Presentation, Control)
Example: JTable

class MyTableModel extends AbstractTableModel {
    private String[] columns = {"Nume", "Varsta", "Student"};
    private Object[][] elements = {
        {"Ionescu", new Integer(20), Boolean.TRUE},
        {"Popescu", new Integer(80), Boolean.FALSE}};

    public int getColumnCount() {
        return columns.length;
    }

    public int getRowCount() {
        return elements.length;
    }

    public Object getValueAt(int row, int col) {
        return elements[row][col];
    }

    public String getColumnName(int col) {
        return columns[col];
    }

    public boolean isCellEditable(int row, int col) {
        // Doar numele este editabil
        return (col == 0);
    }
}
Customizing the View
CellRenderes and CellEditors

![SwingSet2 GUI with a table demo](image)

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Favorite Color</th>
<th>Favorite Movie</th>
<th>Favorite Number</th>
<th>Favorite Food</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike</td>
<td>Albers</td>
<td>Green</td>
<td>Brazil</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Mark</td>
<td>Andrews</td>
<td>Blue</td>
<td>Curse of the Demon</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Brian</td>
<td>Beck</td>
<td>Black</td>
<td>The Blues Brothers</td>
<td>2,718</td>
<td></td>
</tr>
<tr>
<td>Lara</td>
<td>Bunni</td>
<td>Red</td>
<td>Airplane (the whol...</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Roger</td>
<td>Brinkley</td>
<td>Blue</td>
<td>The Man Who Kne...</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Brent</td>
<td>Christian</td>
<td>Black</td>
<td>Blade Runner (Dir...</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Mark</td>
<td>Davidson</td>
<td>Dark Green</td>
<td>Brazil</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>Jeff</td>
<td>Dinkins</td>
<td>Blue</td>
<td>The Lady Vanishes</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Ewan</td>
<td>Dinkins</td>
<td>Yellow</td>
<td>A Bug's Life</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Amy</td>
<td>Fowler</td>
<td>Violet</td>
<td>Reservoir Dogs</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Hanna</td>
<td>Gajewska</td>
<td>Purple</td>
<td>Jules et Jim</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Press Shift-F10 to activate popup menu
Intermission...
The “Drawing” Concept

• Graphical interfaces are built using components. The “system” draws the components automatically:
  – when they are displayed for the first time,
  – at minimize, maximize operations,
  – when resizing the display area;

• The support methods for defining the graphical representation of a Component are:
  – void paint(Graphics g)
  – void update(Graphics g)
  – void repaint()
Java 2D

- Two-dimensional graphics, text, and imaging
- A uniform rendering model for display devices and printers
- Geometric primitives: any geometric shape
- Hit detection on shapes, text, and images
- Control over how overlapping objects are rendered
- Enhanced color support that facilitates color management
- Support for printing complex documents
- Control of the quality of the rendering (hints)
The *paint* method

This method is called when the contents of the component should be painted; such as when the component is first being shown or is damaged and in need of repair. The *clip rectangle* in the *Graphics* parameter is set to the area which needs to be painted.

```java
public class MyFrame extends Frame {
    public MyFrame(String title) {
        super(title);
        setSize(200, 100);
    }

    public void paint(Graphics g) { 
        super.paint(g);
        // Apelam metoda paint a clasei Frame
        g.setFont(new Font("Arial", Font.BOLD, 11));
        g.setColor(Color.red);
        g.drawString("DEMO Version", 5, 35);
    }
}
```
The `paintComponent` method

- `JComponent.paint` delegates the work of painting to three protected methods: `paintComponent`, `paintBorder`, and `paintChildren`. They're called in the order listed to ensure that children appear on top of component itself.

- Swing components should just override `paintComponent`.

```java
/* Creating a custom component */
class MyCustomComponent extends JPanel {

    // Define the representation of the component
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g;
        Graphics2D g2d = (Graphics2D) g;
        ...  
    }

    // Methods used by the layout managers
    public Dimension getPreferredSize() { return ... };
    public Dimension getMinimumSize() { return ... };
    public Dimension getMaximumSize() { return ... };
}
```
public class MyComponent extends JPanel {
    private int x, y, radius;
    public MyComponent() {
        init();
    }
    private void init() {
        setPreferredSize(new Dimension(400, 400));
        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                x = e.getX(); y = e.getY();
                radius = 50 + (int) (100 * Math.random());
                repaint();
            }
        });
    }
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawOval(x - radius / 2, y - radius / 2, radius, radius);
    }
}

JFrame frame = new JFrame("demo");
frame.add(new MyComponent());
frame.pack();
frame.setVisible(true);
Graphics, Graphics2D

- **Graphics** is the base class for all **graphics contexts** that allow an application to draw onto components realized on various devices, as well as onto off-screen images.

- **Graphics2D** class extends the *Graphics* class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout.

- A graphic context offers:
  - Methods for configuring the **drawing properties**:
    - *color, paintMode, font, stroke, clip, renderingHints*, ...
  - **Geometric primitives**
  - Support for working with **texts** and **images**
  - Support for **printing**
Geometric Primitives

- **Coordinates**
  - **User space** – in which graphics primitives are specified
  - **Device space** – screen, window, or a printer
  - The origin of user space is the upper-left corner

- **Primitives**:
  - `drawLine`, `drawPolyline`, `drawOval`, `fillOval`, `drawPolygon`, `fillPolygon`, `drawRect`, `fillRect`, ...
  - `draw(Shape)`, `fill(Shape)`
  - The *Shape interface* provides definitions for objects that represent some form of geometric shape. The Shape is described by a PathIterator object, which can express the outline of the Shape as well as a rule for determining how the outline divides the 2D plane into interior and exterior points.
Working with Texts

- **Font** - A collection of *glyphs* (unique marks that collectively add up to the spelling of a word) → *name, style, size*

  ```java
  Label label = new Label("Some text");
  label.setFont(new Font("Dialog", Font.PLAIN, 12));

  void paint(Graphics g) {
    g.setFont(new Font("Courier", Font.BOLD, 10));
    g.drawString("Another text", 10, 20);
  }
  ```

- **FontMetrics** - encapsulates information about the rendering of a particular font on a particular screen.

  ```java
  Font f = new Font("Arial", Font.BOLD, 11);
  FontMetrics fm = g.getFontMetrics();
  int height = fm.getHeight();
  int width = fm.stringWidth("frog");
  int xWidth = fm.charWidth('g');
  ```

- **TextLayout** - highlighting, strings with mixed fonts, mixed languages, bidirectional text.
Using Colors

- **Paint interface** defines how color patterns can be generated for Graphics2D operations.

- **Color** encapsulates colors in the sRGB space

  ```java
  Color standardRed = Color.RED;
  Color plainWhite = new Color(1.0, 1.0, 1.0);
  Color translucentRed = new Color(255, 0, 0, 128);
  ```

- **SystemColor** encapsulate symbolic colors representing the color of native GUI objects on a system.

  ```java
  SystemColor.desktop
  ```

- **GradientColor** provides a way to fill a Shape with a linear color gradient pattern.

  ```java
  Hello world!
  ```

- **TexturePaint** provides a way to fill a Shape with a texture that is specified as a BufferedImage.

  ```java
  Hello again...
  ```
Using Images

- **Image** is the superclass of all classes that represent graphical images.

- **BufferedImage**
  - Loading from a file
    ```java
    BufferedImage image = ImageIO.read(new File("hello.jpg"));
    ```
  - Creating in memory (off-screen)
    ```java
    BufferedImage image = new BufferedImage(w, h, type);
    Graphics g = image.getGraphics();
    ```
  - Drawing using a graphic context
    ```java
    graphics.drawImage(image);
    ```
  - Saving in a file (GIF, PNG, JPEG, etc.)
    ```java
    ImageIO.write(image, "png", new File("drawing.png"));
    ```
Working with Large Images

- **Displaying a large image**

  ```java
  BufferedImage img = ImageIO.read(new URL("http://www.remoteServer.com/hugeImage.jpg"));
  ...
  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
  ```

- **ImageObserver** - an asynchronous update interface for receiving notifications about information as the `Image` is constructed.

  ```java
  public boolean imageUpdate(Image image, int flags, int x, int y, int width, int height) {
    // If the image has finished loading, repaint the window.
    if ((flags & ALLBITS) != 0) {
      repaint();
      return false; // finished, no further notification.
    }
    return true; // not finished loading, need further notification.
  }
  ```
Intermission...
JavaFX

• A set of **graphics** and **media** packages that enables developers to design, create, test, debug, and deploy **rich client applications**.

• **High-performance**, **modern** user interface that features audio, video, graphics, and animation.

• Deployed across **multiple platforms**: desktop, browsers, mobile, etc.

• Coexists with Swing – however, it may replace Swing as the standard GUI library;
JavaFX Key Features

- **FXML** → MVC Pattern Support
- **WebView** (embed web pages within a JavaFX application)
- Built-in UI controls, **CSS** and **Themes** (Modena, Caspian, etc.)
- **3D Graphics** Features (*Shape3D*)
- Multi-touch Support, Hi-DPI support, Rich Text Support
- **Hardware-accelerated** graphics (uses optimally the GPU)
- High-performance media engine (playback of web multimedia content)
- Self-contained application deployment model
- IDEs offer tools for **rapid application development**
  → JavaFX Scene Builder
Hello World

// The main class extends Application
public class HelloWorld extends Application {
    @Override
    public void start(Stage primaryStage) {
        // The main entry point
        Button helloBtn = new Button();
        helloBtn.setText("Hello World!");

        FlowPane root = new FlowPane();
        root.getChildren().add(helloBtn);

        Scene scene = new Scene(root, 300, 250);

        // The UI is defined by a stage and a scene.
        // Stage class is the top-level JavaFX container.
        // The Scene class is the container for all content.
        primaryStage.setTitle("Hello World Application");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args); // not required for JavaFX applications...
    }
}
The Scene Graph

The JavaFX scene graph is a \textit{retained mode} API

```java
Group group = new Group();
Rectangle blueSquare = new Rectangle(50, 50);
blueSquare.setFill(Color.BLUE);
group.getChildren().add(blueSquare);

Circle redCircle = new Circle(50, new Color(1,0,0,0.5f));
group.getChildren().add(redCircle);
```
Each item in the scene graph is called a *Node*. Each node in the scene graph can be given a **unique id**. Each node has a **bounding rectangle** and a **style**. Any Node can have **transformations** applied to it: translation, rotation, scaling, or shearing.
Layout Management

Setting the position and size for UI element.

- A “combo” of a Swing JPanel + LayoutManager
- javafx.scene.layout.Pane - Base class for layout panes; used directly in cases where absolute positioning of children is required.
- Uses preferred, minimum and maximum properties
- FlowPane, BorderPane, AnchorPane, StackPane, TilePane, GridPane, TextFlow, HBox, VBox, etc.

```java
borderPane.setCenter(new ListView());
borderPane.setBottom(new Label("Hello"));
```
public class HelloWorld extends Application {

@Override
        public void start(Stage primaryStage) {
      Button helloBtn = new Button();
      helloBtn.setText("Hello World!");

      helloBtn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello Button was clicked!");
            }
        });

      //The anonymous inner class
      //can be turned into a lambda expression

      Button ciaoBtn = new Button("Ciao Mondo!");
      ciaoBtn.setOnAction((ActionEvent event) -> {
            System.out.println("Ciao Mondo e stato cliccato!");
        });
    }
JavaFX Events

An event represents an occurrence of something of interest to the application.

`javafx.event.Event` - Base class for FX events.
- **source** → *origin* of the event
- **target** → *the path* through which the event will travel when posted.
- **type** → hierarchy

```
InputEvent.ANY

KeyEvent.ANY
  - KeyEvent.KEY_PRESSED
  - KeyEvent.KEY_RELEASED
  - KeyEvent.KEY_TYPED

MouseEvent.ANY
  - MouseEvent.MOUSE_PRESSED
  - MouseEvent.MOUSE_RELEASED
  - ...

WindowEvent.ANY
  - WindowEvent.WINDOW_SHOWING
  - WindowEvent.WINDOW_SHOWN
  - ...
```

...
Event Delivery Process

• **Target Selection**
  - the node that has focus,
  - the node location of the cursor, etc.

• **Route Construction**
  - *the event dispatch chain* →

• **Event Capturing**
  - passed **down** to the target
  - **filters** are invoked

• **Event Bubbling**
  - the event returns **up** from the target to the root
  - **handlers** are invoked
Event Handling

- **EventHandler** functional interface

- **Filters** (going down...)

```java
redCircle.addEventFilter(
    MouseEvent.MOUSE_CLICKED, (MouseEvent e) -> {
        System.out.println("Click: going down");
        //e.consume();
    });
```

- **Handlers** (going up...)

```java
redCircle.addEventHandler(
    MouseEvent.MOUSE_CLICKED, (MouseEvent e) -> {
        System.out.println("Click: going up");
    });
```

- **Convenience methods**

```java
setOnEvent-type(EventHandler<? super event-class> value)
helloBtn.setOnAction(new EventHandler<ActionEvent>() {...});
redCircle.setOnMouseEntered(new EventHandler<MouseEvent>() {...});
```
Transitions and Animations

TranslateTransition translate =
   new TranslateTransition(Duration.millis(750));
translate.setToX(300); translate.setToY(250);

FillTransition fill = new FillTransition(Duration.millis(750));
fill.setToValue(Color.RED);

RotateTransition rotate = new
   RotateTransition(Duration.millis(750));
rotate.setToAngle(360);

ScaleTransition scale =
   new ScaleTransition(Duration.millis(750));
scale.setToX(0.1); scale.setToY(0.1);

ParallelTransition transition =
   new ParallelTransition(blueSquare, translate, fill, rotate, scale);
transition.setCycleCount(Timeline.INDEFINITE);
transition.setAutoReverse(true);
transition.play();
Pulse

• A pulse is an event that indicates to the JavaFX scene graph that it is time to synchronize the state of the elements on the scene graph with Prism.

• A pulse is throttled at 60 frames per second (fps) maximum and is fired whenever animations are running or when something in the scene graph is changed. For example, if a position of a button is changed, a pulse is scheduled.

• When a pulse is fired, the state of the elements on the scene graph is synchronized down to the rendering layer.

• A pulse enables application developers a way to handle events asynchronously. This important feature allows the system to batch and execute events on the pulse.

• The Glass Windowing Toolkit is responsible for executing the pulse events. It uses the high-resolution native timers to make the execution.
Styling with CSS

- Define Style Sheets Files

```
.root {
    -fx-background-image: url("background.jpg");
}
.label {
    -fx-font-size: 12px;
    -fx-font-weight: bold;
    -fx-text-fill: #333333;
}
```

- Specify the CSS

```java
scene.getStylesheets().add("path/stylesheet.css");
```

- Inline

```java
helloBtn.setStyle(
    "-fx-background-color: slateblue; " +
    "-fx-text-fill: white;");
```
FXML

- XML-based language that provides the structure for building a user interface separate from the application logic of your code.

- Java (Programatic)

```java
BorderPane border = new BorderPane();
Label helloLabel = new Label("Hello");
border.setTop(helloLabel);
Label worldLabel = new Label("World");
border.setCenter(worldLabel);
```

- FXML (Declarative)

```xml
<BorderPane>
  <top>
    <Label text="Hello"/>
  </top>
  <center>
    <Label text="World"/>
  </center>
</BorderPane>
```

JavaFX Scene Builder
Using FXML to Create UI

• FXML Loader

```java
Parent root = FXMLLoader.load(
    getClass().getResource("example.fxml"));
Scene scene = new Scene(root, 300, 275);
```

• Create the link between view and control

```xml
<GridPane fx:controller="FXMLExampleController">
    <Button text="Sign In"
        onAction="#handleSubmitButtonAction"/>
    <Text fx:id="actiontarget" />
</GridPane>
```

• Define the code to handle events

```java
public class FXMLExampleController {
    @FXML
    private Text actiontarget;

    @FXML
    protected void handleSubmitButtonAction(ActionEvent event) {
        actiontarget.setText("Sign in button pressed");
    }
}
```
Swing or JavaFX?

- **Swing**
  - Maturity, Stability
  - Component Libraries and Frameworks
  - Large amount of resources

- **JavaFX**
  - Modern, MVC friendly, CSS, FXML
  - Spectacular (3D, Animations, etc.)
  - May not be “rock-solid” in production, yet
  - Not so many resources