Object-Oriented Language and Theory

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn
Teaching Assistant: DO Minh Hieu, hieudominh@hotmail.com

Lab 9: GUI Programming

* Objectives:

In this lab, you will practice with:

- Create a simple GUI application with AWT
- Create a simple GUI application with Swing
- Work with JavaFX
- Convert the AimsProject from the console/command-line (CLI) application to the GUI one.

There are current three sets of Java APIs for graphics programming:

1. **AWT** (Abstract Windowing Toolkit) API was introduced in JDK 1.0. Most of the AWT components have become obsolete and should be replaced by newer Swing components.
2. **Swing** API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1. JFC consists of Swing, Java2D, Accessibility, Internationalization, and Pluggable Look-and-Feel Support APIs. JFC has been integrated into core Java since JDK 1.2.
3. The latest **JavaFX**, which was integrated into JDK 8, is meant to replace Swing. However, Oracle **removed JavaFX from JDK 11 onwards** but **continued commercial support** for JavaFX in the **Oracle JDK 8** through at least 2022.

Other than AWT/Swing/JavaFX graphics APIs provided in JDK, other organizations/vendors have also provided graphics APIs that work with Java, such as Eclipse's Standard Widget Toolkit (SWT) (used in Eclipse), Google Web Toolkit (GWT) (used in Android), 3D Graphics API such as Java bindings for OpenGL (JOGL) and Java3D.

You need to check the JDK API the AWT/Swing APIs (under module **`java.desktop`**) and JavaFX (under modules **`javafx.*`**) while reading this chapter. The best online reference for Graphics programming is the "Swing Tutorial" @ http://docs.oracle.com/javase/tutorial/uiswing/. For advanced 2D graphics programming, read "Java 2D Tutorial" @ http://docs.oracle.com/javase/tutorial/2d/index.html.

The below guideline helps you to practice with all three Java APIs for comparison. You may copy and paste the suggestion source code but you should analyse to understand and explain the source code.

0. Create GUIProject

- Create a Java Project named GUIProject
- Create the following packages in the GUIProject for this lab:

For Global ICT:

+ **`hust.soict.globalict.gui.awt`**
+ **`hust.soict.globalict.gui.swing`**
+ **`hust.soict.globalict.gui.javafx`**

For HEDSPI:

+ **`hust.soict.hedspi.gui.awt`**
+ **`hust.soict.hedspi.gui.swing`**
+ **`hust.soict.hedspi.gui.javafx`**

1. A simple GUI application with AWT

Note: All codes in this section should be put into the AWT package.

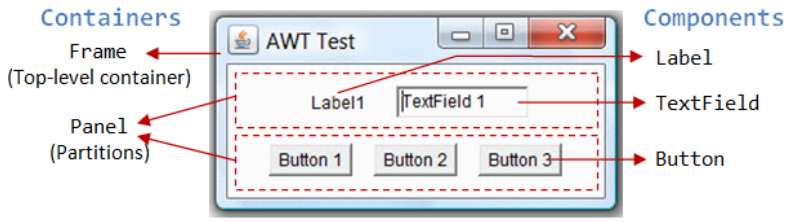
1.1. AWT Packages

AWT consists of 12 packages of 370 classes (Swing is even bigger, with 18 packages of 737 classes as of JDK 8, and then we have JavaFX with more than 30 packages). Fortunately, only 2 packages - **`java.awt`** and **`java.awt.event`** - are commonly-used.

1. The **`java.awt`** package contains the *core* AWT graphics classes:
   - GUI Component classes, such as **`Button`**, **`TextField`**, and **`Label`**.
   - GUI Container classes, such as **`Frame`** and **`Panel`**.
   - Layout managers, such as **`FlowLayout`**, **`BorderLayout`** and **`GridLayout`**.
   - Custom graphics classes, such as **`Graphics`**, **`Color`** and **`Font`**.
2. The **`java.awt.event`** package supports event handling:
   - Event classes, such as **`ActionEvent`**, **`MouseEvent`**, **`KeyEvent`** and **`WindowEvent`**,
   - Event Listener Interfaces, such as **`ActionListener`**, **`MouseListener`**, **`MouseMotionListener`**, **`KeyListener`**, and **`WindowListener`**
   - Event Listener Adapter classes, such as **`MouseAdapter`**, **`KeyAdapter`**, and **`WindowAdapter`**.

AWT provides a *platform-independent* and *device-independent* interface to develop graphic programs that runs on all platforms, including Windows, Mac OS X, and Unixes.

1.2. Containers and Components



There are two types of GUI elements:

1. *Component*: Components are elementary GUI entities, such as **`Button`**, **`Label`**, and **`TextField`**.
2. *Container*: Containers, such as **`Frame`** and **`Panel`**, are used to *hold components in a specific layout* (such as **`FlowLayout`** or **`GridLayout`**). A container can also hold sub-containers.

In the above figure, there are three containers: a **`Frame`** and two **`Panel`**s. A **`Frame`** is the *top-level container* of an AWT program. A **`Frame`** has a title bar (containing an icon, a title, and the minimize/maximize/close buttons), an optional menu bar and the content display area. A **`Panel`** is a *rectangular area* used to group related GUI components in a certain layout. In the above figure, the top-level **`Frame`** contains two **`Panel`**s. There are five components: a **`Label`** (providing description), a **`TextField`** (for users to enter text), and three **`Button`**s (for user to trigger certain programmed actions).

In a GUI program, a component must be kept in a container. You need to identify a container to hold the components. Every container has a method called **`add(Component c)`**. A container (say c) can invoke **`c.add(aComponent)`** to add **`aComponent`** into itself. For example,

```
Panel pnl = new Panel();           // Panel is a container
Button btn = new Button("Press"); // Button is a component
pnl.add(btn);                      // The Panel container adds a Button component
```

GUI components are also called *controls* (e.g., Microsoft ActiveX Control), *widgets* (e.g., Eclipse's Standard Widget Toolkit, Google Web Toolkit), which allow users to interact with (or control) the application.

1.3. AWTCounter application

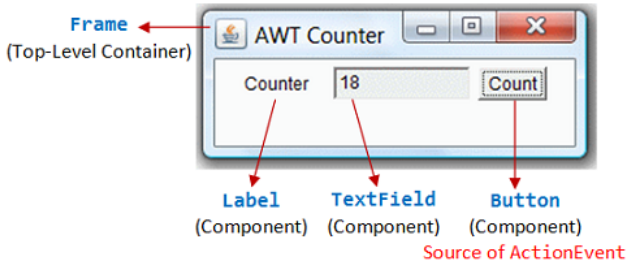
Let's assemble a few components together into a simple GUI counter program, as illustrated.

- Create a class **`AWTCounter`** in **`soict.hust.globalict.gui.awt`** package or **`soict.hust.hedspi.gui.awt`**. It has a top-level container **`Frame`**, which contains three components - a **`Label`** "Counter", a non-editable **`TextField`** to display the current count, and a "Count" **`Button`**. The **`TextField`** shall display count of 0 initially. Each time you click the button, the counter's value increases by 1.
- The sample code is presented below:

```
1import java.awt.*;        // Using AWT container and component classes
2import java.awt.event.*;  // Using AWT event classes and listener interfaces
3
4// An AWT program inherits from the top-level container java.awt.Frame
5public class AWTCounter extends Frame implements ActionListener {
6   private Label lblCount;    // Declare a Label component
7   private TextField tfCount; // Declare a TextField component
8   private Button btnCount;   // Declare a Button component
9   private int count = 0;     // Counter's value
10
11  // Constructor to setup GUI components and event handlers
12  public AWTCounter () {
13     setLayout(new FlowLayout());
14        // "super" Frame, which is a Container, sets its layout to FlowLayout to arrange
15        // the components from left-to-right, and flow to next row from top-to-bottom.
16
17     lblCount = new Label("Counter");  // construct the Label component
18     add(lblCount);                    // "super" Frame container adds Label component
19
20     tfCount = new TextField(count + "", 10); // construct the TextField component with initial text
21     tfCount.setEditable(false);       // set to read-only
22     add(tfCount);                     // "super" Frame container adds TextField component
23
24     btnCount = new Button("Count");   // construct the Button component
25     add(btnCount);                    // "super" Frame container adds Button component
26
27     btnCount.addActionListener(this);
28        // "btnCount" is the source object that fires an ActionEvent when clicked.
29        // The source add "this" instance as an ActionEvent listener, which provides
30        //   an ActionEvent handler called actionPerformed().
31        // Clicking "btnCount" invokes actionPerformed().
32
33     setTitle("AWT Counter");  // "super" Frame sets its title
34     setSize(250, 100);        // "super" Frame sets its initial window size
35
36     // For inspecting the Container/Components objects
37     // System.out.println(this);
38     // System.out.println(lblCount);
39     // System.out.println(tfCount);
40     // System.out.println(btnCount);
41
```

```
42      setVisible(true);         // "super" Frame shows
43
44      // System.out.println(this);
45      // System.out.println(lblCount);
46      // System.out.println(tfCount);
47      // System.out.println(btnCount);
48   }
49
50   // The entry main() method
51   public static void main(String[] args) {
52      // Invoke the constructor to setup the GUI, by allocating an instance
53      AWTCounter app = new AWTCounter();
54         // or simply "new AWTCounter();" for an anonymous instance
55   }
56
57   // ActionEvent handler - Called back upon button-click.
58   @Override
59   public void actionPerformed(ActionEvent evt) {
60      ++count; // Increase the counter value
61      // Display the counter value on the TextField tfCount
62      tfCount.setText(count + ""); // Convert int to String
63   }
64}
```

- Run and test the application by clicking the button **Count**



1.4. Do more practice at home

Visit https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html and practice example from 2 to 5.

2. A simple GUI application with Swing

Note: All codes in this section should be put into the Swing package.

Swing is part of the so-called "Java Foundation Classes (JFC)", which was introduced in 1997 after the release of JDK 1.1. JFC was subsequently included as an integral part of JDK since JDK 1.2. JFC consists of:

- Swing API: for advanced graphical programming.
- Accessibility API: provides assistive technology for the disabled.
- Java 2D API: for high quality 2D graphics and images.
- Pluggable look and feel supports.
- Drag-and-drop support between Java and native applications.

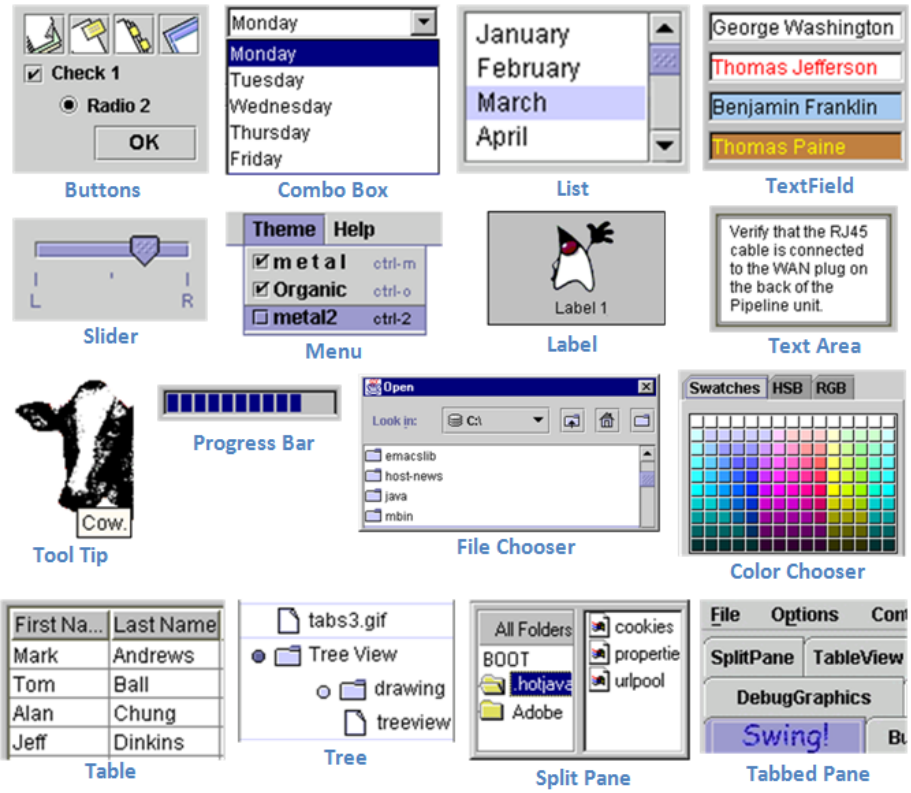
The goal of Java GUI programming is to allow the programmer to build GUI that looks good on ALL platforms. JDK 1.0's AWT was awkward and non-object-oriented (using many event.getSource()). JDK 1.1's AWT introduced

event-delegation (event-driven) model, much clearer and object-oriented. JDK 1.1 also introduced inner class and JavaBeans – a component programming model for visual programming environment (similar to Visual Basic).

Swing appeared after JDK 1.1. It was introduced into JDK 1.1 as part of an add-on JFC (Java Foundation Classes). Swing is a rich set of easy-to-use, easy-to-understand JavaBean GUI components that can be dragged and dropped as "GUI builders" in visual programming environment. Swing is now an integral part of Java since JDK 1.2.

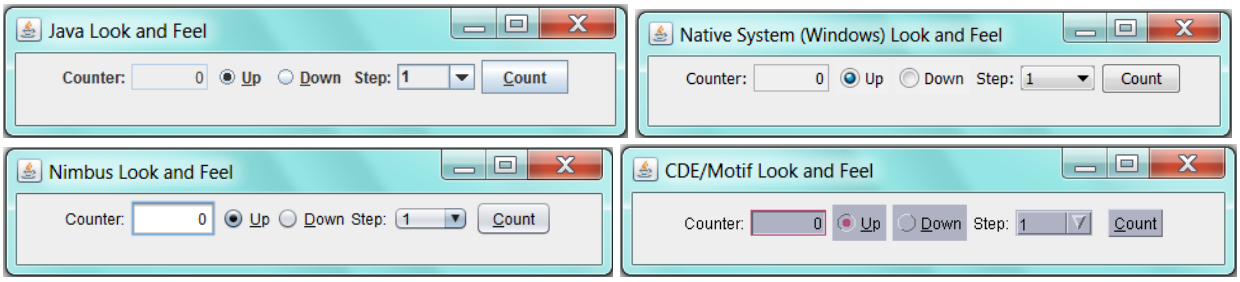
2.1. Swing's Features

Swing is huge (consists of 18 packages of 737 classes as in JDK 1.8) and has great depth. Compared with AWT, Swing provides a huge and comprehensive collection of reusable GUI components, as shown in the Figure below (extracted form Swing Tutorial).



The main features of Swing are (extracted from the Swing website):

1. Swing is written in pure Java (except a few classes) and therefore is 100% portable.
2. Swing components are *lightweight*. The AWT components are *heavyweight* (in terms of system resource utilization). Each AWT component has its own opaque native display, and always displays on top of the lightweight components. AWT components rely heavily on the underlying windowing subsystem of the native operating system. For example, an AWT button ties to an actual button in the underlying native windowing subsystem, and relies on the native windowing subsystem for their rendering and processing. Swing components (**`JComponents`**) are written in Java. They are generally not "weight-down" by complex GUI considerations imposed by the underlying windowing subsystem.
3. Swing components support *pluggable look-and-feel*. You can choose between *Java look-and-feel* and the *look-and-feel of the underlying OS* (e.g., Windows, UNIX or Mac). If the later is chosen, a Swing button runs on the Windows looks like a Windows' button and feels like a Window's button. Similarly, a Swing button runs on the UNIX looks like a UNIX's button and feels like a UNIX's button.



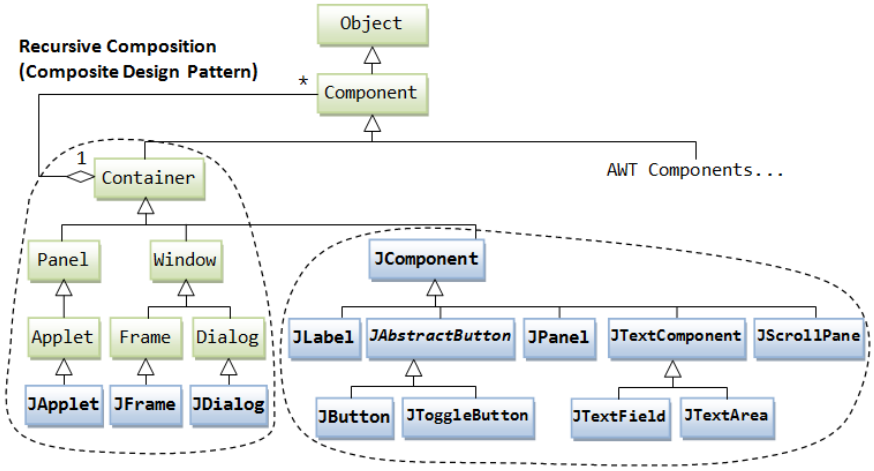
4. Swing supports *mouse-less operation*, i.e., it can operate entirely using keyboard.
5. Swing components support "tool-tips".
6. Swing components are *JavaBeans* – a Component-based Model used in Visual Programming (like Visual Basic). You can drag-and-drop a Swing component into a "design form" using a "GUI builder" and double-click to attach an event handler.
7. Swing application uses AWT event-handling classes (in package **`java.awt.event`**). Swing added some new classes in package **`javax.swing.event`**, but they are not frequently used.
8. Swing application uses AWT's layout manager (such as **`FlowLayout`** and **`BorderLayout`** in package **`java.awt`**). It added new layout managers, such as Springs, Struts, and **`BoxLayout`** (in package **`javax.swing`**).
9. Swing implements *double-buffering* and automatic repaint batching for smoother screen repaint.
10. Swing introduces **`JLayeredPane`** and **`JInternalFrame`** for creating Multiple Document Interface (MDI) applications.
11. Swing supports floating toolbars (in **`JToolBar`**), splitter control, "undo".
12. Others - check the Swing website.

2.2. Using Swing API

If you understood the AWT programming (in particular, container/component and event-handling), switching over to Swing (or any other Graphics packages) is straight-forward.

2.2.1. Swing's Components

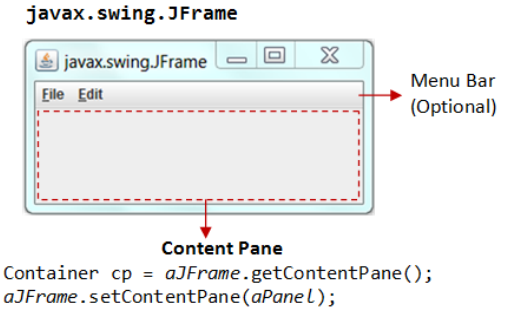
Compared with the AWT component classes (in package **`java.awt`**), Swing component classes (in package **`javax.swing`**) begin with a prefix "J",e.g., **`JButton`**, **`JTextField`**, **`JLabel`**, **`JPanel`**, **`JFrame`**, or **`JApplet`**.



The above figure shows the class hierarchy of the swing GUI classes. Similar to AWT, there are two groups of classes: *containers* and *components*. A container is used to hold components. A container can also hold containers because it is a (subclass of) component.

As a rule, do not mix heavyweight AWT components and lightweight Swing components in the same program, as the heavyweight components will always be painted *on top of* the lightweight components.

2.2.2. Swing's Top-Level and Secondary Containers



Just like AWT application, a Swing application requires a *top-level container*. There are three top-level containers in Swing:

1. **`JFrame`**: used for the application's main window (with an icon, a title, minimize/maximize/close buttons, an optional menu-bar, and a content-pane), as illustrated.
2. **`JDialog`**: used for secondary pop-up window (with a title, a close button, and a content-pane).
3. **`JApplet`**: used for the applet's display-area (content-pane) inside a browser's window.

Similarly to AWT, there are *secondary containers* (such as JPanel) which can be used to group and layout relevant components.

2.2.3. The Content-Pane of Swing's Top-Level Container

However, unlike AWT, the **`JComponents`** shall not be added onto the top-level container (e.g., **`JFrame`**, **`JApplet`**) directly because they are lightweight components. The **`JComponents`** must be added onto the so-called *content-pane* of the top-level container. Content-pane is in fact a **`java.awt.Container`** that can be used to group and layout components.

You could:

get the content-pane via **`getContentPane()`** from a top-level container, and add components onto it. For example,

```
1. public class SwingDemo extends JFrame {
2.    // Constructor
3.    public SwingDemo() {
4.       // Get the content-pane of this JFrame, which is a java.awt.Container
5.       // All operations, such as setLayout() and add() operate on the content-pane
6.       Container cp = getContentPane();
7.       cp.setLayout(new FlowLayout());
8.       cp.add(new JLabel("Hello, world!"));
9.       cp.add(new JButton("Button"));
10.      ......
11.   }
12.   .......
13.}
```

set the content-pane to a **`JPanel`** (the main panel created in your application which holds all your GUI components) via **`JFrame`**'s **`setContentPane()`**.

```
14.public class SwingDemo extends JFrame {
15.   // Constructor
16.   public SwingDemo() {
17.      // The "main" JPanel holds all the GUI components
```

```
      JPanel mainPanel = new JPanel(new FlowLayout());
      mainPanel.add(new JLabel("Hello, world!"));
      mainPanel.add(new JButton("Button"));

      // Set the content-pane of this JFrame to the main JPanel
      setContentPane(mainPanel);
      ......
   }
   .......
}
```

Notes: If a component is added directly into a **`JFrame`**, it is added into the content-pane of **`JFrame`** instead.

```
// Suppose that "this" is a JFrame
add(new JLabel("add to JFrame directly"));
// is executed as
getContentPane().add(new JLabel("add to JFrame directly"));
```

2.2.4. Event-Handling in Swing

Swing uses the AWT event-handling classes (in package **`java.awt.event`**). Swing introduces a few new event-handling classes (in package **`javax.swing.event`**) but they are not frequently used.

2.2.5. Writing Swing Applications

In summary, to write a Swing application, you have:

1. Use the Swing components with prefix "J" in package **`javax.swing`**
   e.g., **`JFrame`**, **`JButton`**, **`JTextField`**, **`JLabel`**, etc.
2. A top-level container (typically **`JFrame`**) is needed. The **`JComponents`** should not be added directly onto the top-level container. They shall be added onto the *content-pane* of the top-level container. You can retrieve a reference to the content-pane by invoking method **`getContentPane()`** from the top-level container.
3. Swing applications uses AWT event-handling classes
   e.g., ActionEvent/ ActionListener, MouseEvent/ MouseListener, etc.
4. Run the constructor in the Event Dispatcher Thread (instead of Main thread) for thread safety, as shown in the following program template.

2.2.6. Swing Program Template

```
1import java.awt.*;        // Using AWT layouts
2import java.awt.event.*;  // Using AWT event classes and listener interfaces
3import javax.swing.*;     // Using Swing components and containers
4
5// A Swing GUI application inherits from top-level container javax.swing.JFrame
6public class ..... extends JFrame {
7
8   // Private instance variables
9   // ......
10
11  // Constructor to setup the GUI components and event handlers
12  public ......() {
13     // Retrieve the top-level content-pane from JFrame
14     Container cp = getContentPane();
15
16     // Content-pane sets layout
```

```
      cp.setLayout(new ....Layout());

      // Allocate the GUI components
      // .....

      // Content-pane adds components
      cp.add(....);

      // Source object adds listener
      // .....

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         // Exit the program when the close-window button clicked
      setTitle("......");  // "super" JFrame sets title
      setSize(300, 150);   // "super" JFrame sets initial size
      setVisible(true);    // "super" JFrame shows
   }

   // The entry main() method
   public static void main(String[] args) {
      // Run GUI codes in Event-Dispatching thread for thread-safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            new ......();  // Let the constructor do the job
         }
      });
   }
}
```

2.3. SwingCounter application

- Let's convert the earlier AWT application example into Swing.
- Create a class named **`SwingCounter`** in the **`hust.soict.globalict.gui.swing`** or **`hust.soict.hedspi.gui.swing`**. Compare the two source files and note the changes (which are highlighted). The display is shown below. Note the differences in *look and feel* between the AWT GUI components and Swing's.

```
1import java.awt.*;       // Using AWT layouts
2import java.awt.event.*; // Using AWT event classes and listener interfaces
3import javax.swing.*;    // Using Swing components and containers
4
5// A Swing GUI application inherits from top-level container javax.swing.JFrame
6public class SwingCounter extends JFrame {   // JFrame instead of Frame
7   private JTextField tfCount;  // Use Swing's JTextField instead of AWT's TextField
8   private JButton btnCount;    // Using Swing's JButton instead of AWT's Button
9   private int count = 0;
10
11  // Constructor to setup the GUI components and event handlers
```

```
12   public SwingCounter() {
13      // Retrieve the content-pane of the top-level container JFrame
14      // All operations done on the content-pane
15      Container cp = getContentPane();
16      cp.setLayout(new FlowLayout());   // The content-pane sets its layout
17
18      cp.add(new JLabel("Counter"));
19      tfCount = new JTextField("0");
20      tfCount.setEditable(false);
21      cp.add(tfCount);
22
23      btnCount = new JButton("Count");
24      cp.add(btnCount);
25
26      // Allocate an anonymous instance of an anonymous inner class that
27      //  implements ActionListener as ActionEvent listener
28      btnCount.addActionListener(new ActionListener() {
29         @Override
30         public void actionPerformed(ActionEvent evt) {
31            ++count;
32            tfCount.setText(count + "");
33         }
34      });
35
36      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  // Exit program if close-window button clicked
37      setTitle("Swing Counter"); // "super" JFrame sets title
38      setSize(300, 100);         // "super" JFrame sets initial size
39      setVisible(true);          // "super" JFrame shows
40   }
41
42   // The entry main() method
43   public static void main(String[] args) {
44      // Run the GUI construction in the Event-Dispatching thread for thread-safety
45      SwingUtilities.invokeLater(new Runnable() {
46         @Override
47         public void run() {
48            new SwingCounter(); // Let the constructor do the job
49         }
50      });
51   }
52}
```

JFrame's Content-Pane

The **JFrams**'s method **getContentPane()** returns the content-pane (which is a **java.awt.Containter**) of the `JFrame`. You can then set its layout (the default layout is **BorderLayout**) and add components into it. For example,

```
Container cp = getContentPane(); // Get the content-pane of this JFrame
cp.setLayout(new FlowLayout());  // content-pane sets to FlowLayout
cp.add(new JLabel("Counter"));   // content-pane adds a JLabel component
......
cp.add(tfCount);    // content-pane adds a JTextField component
```

```
......
cp.add(btnCount);   // content-pane adds a JButton component
```

You can also use the **`JFrame`**'s **`setContentPane()`** method to directly set the content-pane to a **`JPanel`** (or a **`JComponent`**). For example,

```
JPanel displayPanel = new JPanel();

setContentPane(displayPanel);

      // "this" JFrame sets its content-pane to a JPanel directly
.....

// The above is different from:
getContentPane().add(displayPanel);
      // Add a JPanel into the content-pane. Appearance depends on the JFrame's layout.
```

JFrame's setDefaultCloseOperation()

Instead of writing a **`WindowEvent`** listener with a **`windowClosing()`** handler to process the "close-window" button, **`JFrame`** provides a method called **`setDefaultCloseOperation()`** to sets the default operation when the user initiates a "close" on this frame. Typically, we choose the option **`JFrame.EXIT_ON_CLOSE`**, which terminates the application via a **`System.exit()`.**

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Running the GUI Construction Codes on the Event-Dispatching Thread

In the previous examples, we invoke the constructor directly in the entry `main()` method to setup the GUI components. For example,

```
// The entry main method
public static void main(String[] args) {
   // Invoke the constructor (by allocating an instance) to setup the GUI
   new SwingCounter();
}
```

The constructor will be executed in the so-called "Main-Program" thread. This may cause multi-threading issues (such as unresponsive user-interface and deadlock).

It is recommended to execute the GUI setup codes in the so-called "Event-Dispatching" thread, instead of "Main-Program" thread, for thread-safe operations. Event-dispatching thread, which processes events, should be used when the codes updates the GUI.

To run the constructor on the event-dispatching thread, invoke `static` method `SwingUtilities.invokeLater()` to asynchronously queue the constructor on the event-dispatching thread. The codes will be run after all pending events have been processed. For example,

```
public static void main(String[] args) {

   // Run the GUI codes in the Event-dispatching thread for thread-safety
   SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
         new SwingCounter();  // Let the constructor do the job
      }
   });
}
```

Note: **`javax.swing.SwingUtilities.invokeLater()`** is a cover for `java.awt.EventQueue.invokeLater()` (which is used in the NetBeans' Visual GUI Builder).

At times, for example in game programming, the *constructor* or the **`main()`** may contains non-GUI codes. Hence, it is a common practice to create a dedicated method called **`initComponents()`** or **`createAndShowGUI()`** (used in Swing

tutorial) to handle all the GUI codes (and another method called `initGame()` to handle initialization of the game's objects). This GUI init method shall be run in the event-dispatching thread.

Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"

This warning message is triggered because **`java.awt.Frame`** (via its superclass **`java.awt.Component`**) implements the **`java.io.Serializable`** interface. This interface enables the object to be written out to an output stream *serially* (via method **`writeObject()`**); and read back into the program (via method **`readObject()`**). The serialization runtime uses a number (called **`serialVersionUID`**) to ensure that the object read into the program is compatible with the class definition, and not belonging to another version.

You have these options:

1. Simply ignore this warning message. If a **`serializable`** class does not explicitly declare a **`serialVersionUID`**, then the serialization runtime will calculate a default **`serialVersionUID`** value for that class based on various aspects of the class.
2. Add a **`serialVersionUID`** (Recommended), e.g.

```
private static final long serialVersionUID = 1L;  // version 1
```

3. Suppress this particular warning via annotation **`@SuppressWarmomgs`** (in package **`java.lang`**) (JDK 1.5):

```
@SuppressWarnings("serial")
public class MyFrame extends JFrame { ..... }
```

2.4. Do more practice at home

Visit https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html and practice example from 6 to 8.

3. Working with JavaFX

Note: All codes in this section should be put into the JavaFX package.
Put all examples/exercises of this section on the **`hust.soict.globalict.gui.javafx`** package or **`hust.soict.hedspi.gui.javafx`**.

3.1. JavaFX Packages

Overall, JavaFX is huge, yet we still can gradually master it. The followings are commonly used packages:

- **`javafx.application`**: JavaFX application
- **`javafx.stage`**: top-level container
- **`javafx.scene`**: scene and scene graph.
- **`javafx.scene.*`**: control, layout, shape, etc.
- **`javafx.event`**: event handling
- **`javafx.animation`**: animation

3.2. Using JavaFX

3.2.1. Preparation

Please go to this tutorial https://o7planning.org/en/11009/javafx, **read carefully** and do the following task **in order**:

- Install e(fx)clipse into Eclipse (JavaFX Tooling)
- Install JavaFX Scene Builder into Eclipse

3.2.2. First JavaFX Application

Please go to this tutorial JavaFX Tutorial for Beginners - Hello JavaFX, read carefully and do the task in the tutorial.

3.3. JavaFXHello application

```
1   import javafx.application.Application;
2   import javafx.event.ActionEvent;
3   import javafx.event.EventHandler;
4   import javafx.scene.Scene;
5   import javafx.scene.control.Button;
6   import javafx.scene.layout.StackPane;
7   import javafx.stage.Stage;
8
9   public class JavaFXHello extends Application {
10      private Button btnHello;  // Declare a "Button" control
11
12      @Override
13      public void start(Stage primaryStage) {
14          // Construct the "Button" and attach an "EventHandler"
15          btnHello = new Button();
16          btnHello.setText("Say Hello");
17          // Using JDK 8 Lambda Expression to construct an EventHandler<ActionEvent>
18          btnHello.setOnAction(evt -> System.out.println("Hello World!"));
19
20          // Construct a scene graph of nodes
21          StackPane root = new StackPane();  // The root of scene graph is a layout node
22          root.getChildren().add(btnHello);  // The root node adds Button as a child
23
24          Scene scene = new Scene(root, 300, 100);  // Construct a scene given the root of scen
25          primaryStage.setScene(scene);    // The stage sets scene
26          primaryStage.setTitle("Hello");  // Set window's title
27          primaryStage.show();             // Set visible (show it)
28      }
29
30      public static void main(String[] args) {
31          launch(args);
32      }
33  }
```

How It Works

1. A JavaFX GUI Program extends from **`javafx.application.Application`** (just like a Java Swing GUI program extends from **`javax.swing.JFrame`**).
2. JavaFX provides a huge set of controls (or components) in package **`javafx.scene.control`**, including **`Label`**, **`Button`** and **`TextField`**.
3. We declare and construct a **`Button`** control, and attach a **`javafx.event.EventHandler<ActionEvent>`** to the **`Button`**, via method **`setOnAction()`** (of **`ButtonBase`** superclass), which takes an **`EventHandler<ActionEvent>,`** as follows:

```
public final void setOnAction(EventHandler<ActionEvent> value)
```

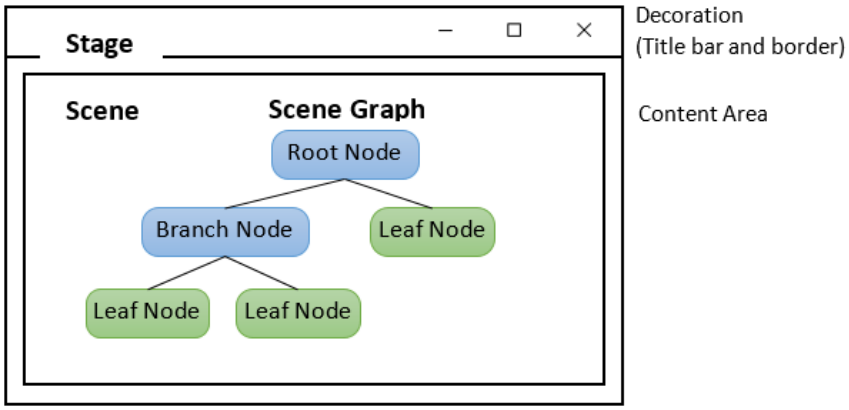
The **EventHandler** is a Functional Interface with an abstract method **handle(),** defined as follows:

```
package javafx.event;

@FunctionalInterface

public interface EventHandler<T extends Event> extends EventListener {
   void handle(T event);  // public abstract
}
```

We can trigger the **handle()** by firing the button, via clicking the button with the mouse or touch, key press, or invoke the **fire()** method programmatically.
In this example, we use a one-liner Lambda Expression (JDK 8) to construct an instance of Functional Interface **EventHandler**. You can also use an anonymous inner class (Pre JDK 8), as follows:

```
btnHello.setOnAction(new EventHandler<ActionEvent>() {
   @Override
   public void handle(ActionEvent evt) {
      System.out.println("Hello World!");
   }
});
```

4. JavaFX uses *the metaphor of a theater* to model the graphics application. A *stage* (defined by the **javafx.stage.Stage** class) represents the top-level container (window). The individual controls (or components) are contained in a *scene* (defined by the **javafx.scene.Scene** class). An application can have more than one scenes, but only one of the scenes can be displayed on the stage at any given time. The contents of a scene is represented in a hierarchical *scene graph of nodes* (defined by **javafx.scene.Node**).



5. To construct the UI:
   1. Prepare a scene graph.
   2. Construct a scene, with the root node of the scene graph.
   3. Setup the stage with the constructed scene.
6. In this example, the root node is a "layout" node (container) named **javafx.scene.layout.StackPane**, which layouts its children in a back-to-front stack. This layout node has one child node, which is the **Button**. To add child node(s) under a layout, use:

```
aLayout.getChildren().add(Node node)          // Add one node
aLayout.getChildren().addAll(Node... nodes)   // Add all nodes
```

Notes: A JavaFX's **Pane** is similar to Swing's **JPanel**. However, JavaFX has layout-specific **Pane**, such as **FlowPane**, **GridPane** and **BorderPane**, which is similar to a Swing's `JPanel` with **FlowLayout**, **GridLayout** and **BorderLayout**.

7. We allocate a **javafx.scene.Scene** by specifying the root of the scene graph, via constructor:

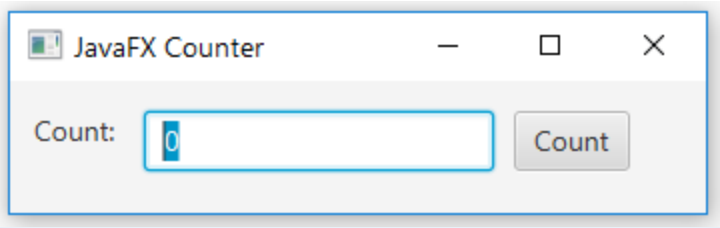
```
public Scene(Parent root, double width, double height)
```

where **javafx.scene.Parent** is a subclass of **javafx.scene.Node**, which serves as the base class for all nodes that have children in the scene graph.

8. We then set the stage's scene, title, and show it.

3.4. JavaFXCounter application

The following JavaFX GUI counter contains 3 controls (or components): a `Label`, a `TextField` and a `Button`. Clicking the button increases the count displayed in the textfield.




```
1    import javafx.application.Application;
2    import javafx.scene.Scene;
3    import javafx.scene.control.Label;
4    import javafx.scene.control.Button;
5    import javafx.scene.control.TextField;
6    import javafx.scene.layout.FlowPane;
7    import javafx.stage.Stage;
8    import javafx.geometry.Insets;
9    import javafx.geometry.Pos;
10
11   public class JavafxCounter extends Application {
12      private TextField tfCount;
13      private Button btnCount;
14      private int count = 0;
15
16      @Override
17      public void start(Stage primaryStage) {
18         // Allocate controls
19         tfCount = new TextField("0");
20         tfCount.setEditable(false);
21         btnCount = new Button("Count");
22         // Register event handler using Lambda Expression (JDK 8)
23         btnCount.setOnAction(evt -> tfCount.setText(++count + ""));
24
25         // Create a scene graph of node rooted at a FlowPane
26         FlowPane flow = new FlowPane();
27         flow.setPadding(new Insets(15, 12, 15, 12));  // top, right, bottom, left
28         flow.setVgap(10);  // Vertical gap between nodes in pixels
```

```
29          flow.setHgap(10);  // Horizontal gap between nodes in pixels
30          flow.setAlignment(Pos.CENTER);  // Alignment
31          flow.getChildren().addAll(new Label("Count: "), tfCount, btnCount);
32
33          // Setup scene and stage
34          primaryStage.setScene(new Scene(flow, 400, 80));
35          primaryStage.setTitle("JavaFX Counter");
36          primaryStage.show();
37       }
38
39       public static void main(String[] args) {
40          launch(args);
41       }
42    }
```

How It Works

1. We use 3 controls: **`Label`**, **`TextField`** and **`Button`** (in package **`javafx.scene.control`**).
2. We use a layout called **`FlowPane`**, which lays out its children in a flow that wraps at the flowpane's boundary (like a Swing's **`JPanel`** in **`FlowLayout`**). This layout node is the root of the scene graph, which has the 3 controls as its children.

See more at https://www3.ntu.edu.sg/home/ehchua/programming/java/Javafx1_intro.html.

3.5. Do more practice at home

- Working with some common Layout in JavaFX such as HBox, VBox, FlowPane, BorderPane, GridPane, … (from 4 to 10): Do **all** the tasks & examples.
- Working with some common controls in JavaFX such as ListView, ComboBox, TableView, TreeView, Menu, TextField, TextArea, … (from 11 to 41): Try to do **all** the tasks & examples.
- Working with dialogs and windows in JavaFX (from 42 to 45): Do **all** the tasks & examples.
- Working with charts, shapes, effects… in JavaFX (from 46 to 52): Do **all** the tasks & examples.

Although doing all the tasks and examples is time-consuming and exhausting, it helps us have initiative ideas about what JavaFX can do and how it works.

4. GUI Application for AimsProject

Please convert all features that you develop for the AimsProject from the CLI application to the GUI one:

- You should use Java Swing or JavaFX to do this exercise. AWT is strongly discouraged.
- Some suggestions for your GUI application:
    - Overuse of JOptionPane is unacceptable. Your evaluation result of this part will be 0 if you only use or abuse JOptionPane.
    - You should use Menu of Java Swing or JavaFX for the CLI menu

    - You should provide forms with GUI controls for listing medias/orders or entering a new item or any other features
- Please read the following links for some tips for UI/UX design:
    - https://www.cs.umd.edu/~ben/goldenrules.html
    - http://athena.ecs.csus.edu/~buckley/CSc238/Psychology%20of%20UX.pdf