# GUI Development with Swing

This chapter introduces creating graphical user interfaces for applications (and also for applets) using the Java Foundation Classes (JFC) Swing packages. The previous library used to develop GUI in Java was Abstract Window Toolkit (AWT). Since it was awkward and its components heavyweight it was replaced by JFC and GUI portion of JFC was called Swing. However, some AWT components are also available in Java and in some cases they may be used.
Creating a basic swing application is very easy. First the swing package (`javax.swing.*`) could be imported and a subclass of `JFrame` could be created. In the following, a simple swing frame declaration is given. This application could be used as a template to create a window and display it within programs. The `initialize()` method in this program will be used to initialize the appearance and components within the window and last two lines of this method ensure that the window to be sized to fit the preferred size and layouts of its subcomponents and instructs that closing the window causes application to exit. The `main()` method ensures that there will be no thread safety problems that could break the program. This code could be copied and be used as it-is in the other programs.

```
import javax.swing.*;
public class SimpleJFrame extends JFrame {
        public SimpleJFrame(){
                initialize();
        }
        private void initialize(){
                // create and add components to the frame
                // using getContentPane().add(component)
                ...

        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
                pack();
        }
        public static void main(String[] args) {
                SwingUtilities.invokeLater(new Runnable() {
                        public void run(){
                                new
SimpleJFrame().setVisible(true);
                        }
                });
        }
}
```

# Swing Components

Swing contains some visual components like buttons, labels, text boxes which could be used to develop a GUI for an application. All components except top level containers are subclasses of JComponent class, and their names start with "J".
In order to place a component on a window (which is a top level container of type JFrame, JDialog, or JApplet), it should first be

created. Then, by using the interface provided by the component, its properties could be set. Finally, the component should be added to a container within the top level container. Components are not added to the top level container directly. Instead, they are added to a container within the top level container (use getContentPane() method of the top level container to learn the container to add components to). The most commonly used containers are JPanel, JScrollPane, and JTabbedPane.
JComponent provides common functionality to its users or descedants. Some of these are listed in the following table.

| Feature | Methods | Explanation |
|---|---|---|
| Properties | setEnabled(boolean)<br>setVisible(boolean)<br>setToolTipText(str) | Set whether the component is enabled or visible. The string is displayed when the mouse pointer pauses over the component. |
| Borders | setBorder(Border) | Adds a border around the component. |
| Painting | paintComponent() | Override this method for custom painting. |
| Colors | setForeground(Color)<br>setBackground(Color)<br>setOpaque(boolean) | Set background/foreground color of the component. Set whether the component is opaque. |
| Font | setFont(Font) | Sets the component's font for displayed text. |
| Size and Position | setLocation(x,y)<br>setSize(w,h)<br>setBounds(x,y,w,h) | Sets the component's absolute position and size. |

# Top Level Containers

A GUI for an application could be organized in windows. In swing, each window in the GUI should be a top level container. The other components are added to this top level container. The top level containers should be subclasses of `JFrame` or `JDialog` (`JApplet` for applets which can run inside a browser). Simple windows can be created by extending `JFrame`. A dialog created by subclassing `JDialog` is a limited version of frame, and it should be associated with a frame.
The components that will be displayed in the top level containers should be added to the content pane of the top level container. The content pane is a kind of general purpose container which can contain more than one component (or other containers). The general purpose containers will be explained in the following sections.
A component could be displayed by adding it to the content pane of one of the top level containers or one of the general purpose containers inside the top level container. A component can be contained in a single container at the same time. If the component is added to another container it is automatically removed from the first and added to the second. In order to add a component directly to the top level container's content pane `getContentPane().add()` method could be used. In order to add a

component to a general purpose container, that container's `add()` method could be used.

## Basic Components

Swing provides a set of components to build a GUI window. In order to be displayed, a control has to be added to a container. Within a container, the components are positioned and sized according to the layout policies specified for the contianer. Layout customization will be explained in later sections. For simplicity, the layout manager for the content pane could be set to `null` and absolute position and size of each component could be set using `setBounds(int, int, int, int)` method of each component, where parameters specify x coordinate, y coordinate, width, and height, respectively as follows:

```
// set layout manager to null
getContentPane().setLayout(null);
// create a component
JXXX component = new JXXX(...)
// set its position and size
component.setBounds(x,y,w,h);
// add to the content pane
getContentPane().add(component);
// create and add other components
...
```

In the following table, a set of basic controls that could be used is given.

| Component | Example | Purpose |
|---|---|---|
| `JLabel` | | It displays a short string or an image (or both). |
| `JTextField` | | It can be used for editing a single line of plain text. |
| `JTextArea` | | It can be used for editing a multi-line plain text. |
| `JButton` | | A standard push button that can display a text and an image. |
| `JCheckBox` | | It can be selected/deselected and displays its state. |
| `JRadioButton` | | It can be selected/deselected and displays its state. It can be used with a ButtonGroup instance which holds a set of buttons in which only one of them can be selected at a time. |
| `JComboBox` | | It displays a button or editable field and a drop-down list. A value can be chosen from the list. |
| `JList` | | It displays a list from which one or more items can be chosen. |
| `JSlider` | | It can be used to select a value by sliding a knob within the specified bounded interval. |
| `JProgressBar` | | It can be used to display an integer value within the specified bounded interval. |

# General Purpose Containers

The general purpose containers are used to contain other components (also other containers). Three of the most commonly used general purpose containers are `JPanel`, `JScrollPane`, and `JTabbedPane`.

`JPanel` is a generic lightweight container. The controls could be added to a `JPanel` instance using the `add()` method. Panels are useful while designing GUI to organize components and to set custom positioning and sizing options. It can also be used to wrap more than one component. By this way, more than one components could be added to a container which accepts a single component. A panel only draws its background and displays

components added to it. However, it can be customized to have borders and to paint itself.
`JScrollPane` provides a scrollable view for a component added to it. It manages a viewport, optional vertical and horizontal scroll bars, and optional row and column heading viewports. A component which may need scrolling (e.g., a `JTextArea` instance) could be simply added to a `JScrollPane` instance by specifying it in the constructor as a parameter as (the result is shown on the right):

```
JTextArea jTextArea = new
JTextArea();
JScrollPane jScrollPane = new
JScrollPane(jTextArea);
jScrollPane.setBounds(10, 10, 160,
110);
getContentPane().add(jScrollPane);
```

`JTabbedPane` allows switching between a group of components by clicking on a tab with a given title and/or icon. The components are added to a `JTabbedPane` instance by using the `addTab()` or `insertTab()` method with the component to be added, a name string, an index to place component, and/or an icon. A tab is represented by an index corresponding to the position it was added in.

# Handling Events

A GUI application usually interacts with its user (perhaps, with some other things). This interaction requires capturing the events occurring in the GUI and performing some actions consequently. In GUI applications, an event can be seen as something that has happened to a component such as mouse clicks, keyboard key presses, display of a window, change of focus, etc.
An event may cause an object to do something. For this purpose, a notification is sent to the object, and that notification may lead to execution of some code, which is known as an event handler. An event handler is a method in the code which determines the actions to be performed. When an event occurs, the event handler for the event is executed. Multiple handlers can be assigned to an event, and the handlers that handle particular events can be added/removed dynamically.
In Java, each event type has an associated event listener defined as an interface. In order to handle an event of a component, an object implementing the suitable listener `interface` should be registered to the event dispatcher. Every component has some methods to register handlers for categories of events that may occur. For example, `JButton`'s `addActionListener()` method could be used to register an object implementing `ActionListener` interface which handles button click events.

# Inner Classes and Anonymous Classes

Declaring a separate class for each component (each type of event) could be very cumbersome. Instead, a feature of Java, ability to declare classes (and also to create their instances) inside another class could be exploited. Such classes may be extending some other classes and/or implementing some interfaces. Each instance of an inner class has a link to the enclosing object that made it. Therefore, it can access the members of the enclosing object.

An inner class could be declared inside another class by putting the class declaration inside that class or its methods as:

```
public class MyClass {
        class Inner1 {
                public void doX(){
                        ...
                }
        }

        public void doY(){
                Inner1 i1 = new Inner1();
                ...
                class Inner2{
                        public void doZ(){
                                ...
                        }
                };
                Inner2 i2 = new Inner2();
                ...
        }
}
```

It is also possible to declare a class without name inside a method and create instances of it. These classes are called anonymous classes. For example, in the following, `doX()` method of `ClassX` creates an instance of an anonymous class extending `ClassY` (or implementing `ClassY` interface).

```
class ClassX{
        ...
        public void doX(){
                ...
                ClassY y = new ClassY(){
                        // define new methods/attributes
                        // or override/implement some methods
                        public void doZ(){
                                ...
                        }
                        ...
                };
                ...
        }
}
```

# Listener Interfaces

There is a different listener interface that should be implemented to handle different categories of events. In the following table some of these event listener interfaces and methods to be implemented are given. In order to handle an event of category `xxx` for a component, that component's `addXXXListener()` method should be called with an object implementing `XXXListener` interface. An added event handler could also be removed by calling `removeXXXListener()` method of the component.

| Listener Interface | Methods* | Explanation | Some of the Components Firing Event |
|---|---|---|---|
| `ComponentListener` | `componentHidden`<br>`componentMoved`<br>`componentResized`<br>`componentShown` | Change in the component's size, position, or visibility | `all` |
| `FocusListener` | `focusGained`<br>`focusLost` | Component gained or lost focus | `all` |
| `KeyListener` | `keyPressed`<br>`keyReleased`<br>`keyTyped` | A key press on the keyboard | `all` |
| `MouseListener` | `mouseClicked`<br>`mouseEntered`<br>`mouseExited`<br>`mousePressed`<br>`mouseReleased` | Mouse clicks and mouse pointer entrance/leave | `all` |
| `MouseMotionListener` | `mouseDragged`<br>`mouseMoved` | Mouse motion | `all` |
| `MouseWheelListener` | `mouseWheelMoved` | Mouse wheel movement | `all` |
| `ActionListener` | `actionPerformed` | An action is performed | `JButton`, `JCheckBox`, `JRadioButton`, `JComboBox` |
| `CaretListener` | `caretUpdate` | Caret position change | `JTextArea`, `JTextField` |
| `ChangeListener` | `stateChanged` | An object changed its state | `JButton`, `JCheckBox`, `JRadioButton`, `JSlider`, `JProgressBar`, `JTabbedPane` |
| `ListSelectionListener` | `valueChanged` | A list's selection changed | `Jlist` |
| `WindowListener` | `windowActivated`<br>`windowClosed`<br>`windowClosing`<br>`windowDeactivated`<br>`windowDeiconified`<br>`windowIconified`<br>`windowOpened` | Window events | `JFrame`, `JDialog` |

* Methods for an event listener XXXListener has an argument of type XXXEvent.

# Listener Adapters

In order to define event handlers, an object implementing a suitable listener interface should be registered to the component. However, some listener interfaces require more than one method to be implemented. This requirement causes too many unnecessary lines of code to be written. This problem could be avoided by using listener adapters. Each listener adapter is a class that defines empty methods for the corresponding listener interface. Therefore, if all events defined by the interface will not be handled, the class to handle events can be extended from the corresponding listener adapter class.
In the following table, the listener adapter classes defined for some of the listener interfaces are listed.

| Listener Interface | Listener Adapter |
|---|---|
| `ComponentListener` | `ComponentAdapter` |
| `FocusListener` | `FocusAdapter` |
| `KeyListener` | `KeyAdapter` |
| `MouseListener` | `MouseAdapter` |
| `MouseMotionListener` | `MouseMotionAdapter` |
| `WindowListener` | `WindowAdapter` |

# Customizing Layout

Layout managers simplify GUI design by automatically determining size and/or position of components inside a container. Components placed in a container are positioned and/or sized by the layout manager given to the container. Moreover, if the container is resized the layout manager adjusts the component's size and position accordingly to adapt to the change consistently. If a layout manager is used, the position and size specified for a component may not be used while displaying the component. Every container has a default layout manager. For example, content panes' default layout manager is `BorderLayout`, and every panel is initialized to use `FlowLayout` layout manager. In order to change a container's layout manager its `setLayout()` method could be used by passing a new layout manager as the argument. If `setLayout()` method is called with a `null` parameter, the components will be placed according to their position and size attributes set by the designer. There are many layout managers available. However, it is also possible to define a special layout manager.

# BorderLayout

`BorderLayout` is the default layout manager for the content panes. The figure on the right presents a simple frame having five buttons and whose content manager is `BorderLayout`.

Border layout has the following properties:

Layout manager for the container could be set as:

```
continer.setLayout(new BorderLayout())
```

While adding components, the area on which the component will be placed is specified as:

```
container.add(component, area);
```

- area should be one of `BorderLayout.PAGE_START`, `BorderLayout.PAGE_END`, `BorderLayout.LINE_START`, `BorderLayout.LINE_END`, `BorderLayout.CENTER`. As illustrated in the figure.

When a component is added or container is resized, the component in each area is resized as follows:

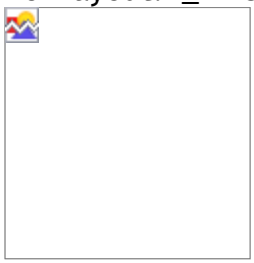
- `PAGE_START`, `PAGE_END`: width is adjusted,
- `LINE_START`, `LINE_END`: height is adjusted
- `CENTER`: width and height are adjusted

A component's position is determined according to the area it has been placed.

# BoxLayout

`BoxLayout` places the components in the container by forming either a column or row depending on the axis specified during its construction. The figure on the right illustrates these two options on a frame with four buttons. This layout manager only sets the positions of the components. The size of the each component is determined according to the component's properties.

BoxLayout.X_AXIS



BoxLayout.Y_AXIS

A `BoxLayout` instance could be created as: `new BoxLayout(container, axis);` where axis can be one of `BoxLayout.X_AXIS` or `BoxLayout.Y_AXIS`. Other axis values (`PAGE_AXIS` or `LINE_AXIS`) could also be used to lay components according to container's `ComponentOrientation`. After setting the layout manager, the components could be added to the container using container's `add()` method. In order to customize the layout, `Box` class's methods could also be used.

# FlowLayout

`FlowLayout` positions components on a row. If the container's width is not enough, a new row is added. That is, there can be multiple rows. While placing components their preferred sizes are used. Therefore, the height of a row is determined by the tallest component in that row. When the container is resized, the positions of the components may be changed. The figure on the right illustrates this property.

Before Resize

After Resize

A FlowLayout instance could be created as

```
new FlowLayout([alignment, [hgap, vgap]]);
```

where the value of the alignment argument must be one of `FlowLayout.LEFT`, `FlowLayout.RIGHT`, `FlowLayout.CENTER`, `FlowLayout.LEADING`, or `FlowLayout.TRAILING`, and `hgap` and `vgap` specifies the gap between components in the same row and the gap between consecutive rows. After the layout manager is set, the components could be added to the container using `add(component)` method of the container.

## GridLayout

`GridLayout` places components into the fixed size cells forming a rectangular grid. Each component fills the available space in the cell. Therefore, the size and location of each component is determined by the layout manager and when the container is resized, the components in the container are resized accordingly. The figure on the right illustrates this behavior of the `GridLayout`.

Before Resize

After Resize

A `GridLayout` instance could be created as:

```
new GridLayout([rows, columns, [hgap, vgap]]);
```

where `rows` and `columns` correspond to number of rows and columns to produce and `hgap` and `vgap` correspond to the spacing between rows and columns. After the layout manager is set, the components could be added to the container using `add()` method of the container. Then components are distributed to the grid according to their insertion order.

## GridBagLayout

The `GridBagLayout`, similar to `GridLayout`, aligns components into rectangular grid of cells vertically and horizontally. However, the cell width and cell height are determined according to the components' size, and when the container is resized how to resize components could also be specified. The example on the right illustrates these properties of the `GridBagLayout` layout manager.

Before Resize

After Resize

Some properties of GridBagLayout are as follows:

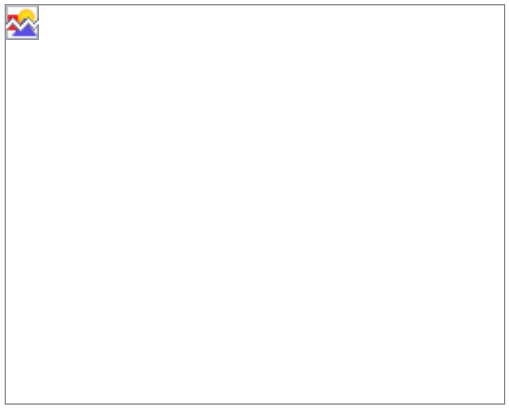
A GridBagLayout instance could be created as: new GridBagLayout();
Before adding a component to the container, a GridBagConstraints instance could be created and the constraints for how to lay the component in the container could be set by using GridBagLayout.setConstraints(component, constraints).
Then the component could be added to the container using add() method.
The following constraints could be set for a component (`GridBagConstraints` attributes):

- `gridx`, `gridy`: specifies the area to place the component
- `gridwidth`, `gridheight`: the component could span more than one cells horizontally and vertically.
- `fill`: specifies how the component fills its area. Could be one of `NONE`, `VERTICAL`, `HORIZONTAL`, `BOTH`. `NONE` uses components preferred width/height.
- `weightx`, `weighty`: Specifies how to resize the component when the container is resized. The component's width (or height) is changed according to the weights assigned to other components in the same row (or column)

- `anchor`: specifies where to place the component in the area if the component is smaller than the area. Could be one of `CENTER`, `NORTH`, `SOUTH`, `WEST`, `EAST`, `NORTHWEST`, `NORTHEAST`, `SOUTHWEST`, `SOUTHEAST`.

# Menus

Menus are the common tools the provide options to the user. In swing, a menu can be constructed by creating a `JMenuBar` instance and adding one or more `JMenu` instances. The `JMenu` instances added to the menubar are displayed on the menu bar. When the user clicks a menu, a list of menu items are displayed. Those menu items are provided by creating and adding JMenuItem instances to the menu object. It is possible to add a new `JMenu` instance to another `JMenu` instance to create a submenu within a menu. The `JMenu` class also has an `addSeparator()` method to create a separator within the menu. After the `JMenuBar` instance is prepared, it could be added to a frame by using `setJMenuBar(menuBar)` method.



The following program creates the menu (and the application) shown on the right.

```
import javax.swing.*;
import java.awt.*;
public class SimpleJFrame extends JFrame {
        public SimpleJFrame(){
                initialize();
        }
        private void initialize(){
                JMenuBar menuBar = new JMenuBar();
                JMenu menu = new JMenu("Menu");
                menuBar.add(menu);
                JMenuItem menuItem1 = new JMenuItem("Menu Item
1");
                menu.add(menuItem1);
                menu.addSeparator();
                JMenuItem menuItem2 = new JMenuItem("MenuItem
2");
                menu.add(menuItem2);
                JMenu subMenu = new JMenu("Submenu");
                JMenuItem submenuItem = new JMenuItem("Submenu
Item");
                subMenu.add(submenuItem);
                menu.add(subMenu);
                setJMenuBar(menuBar);

                setPreferredSize(new Dimension(250,200));
```

```
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
                pack();
        }
        public static void main(String[] args) {
                SwingUtilities.invokeLater(new Runnable() {
                        public void run(){
                                new
SimpleJFrame().setVisible(true);
                        }
                });
        }
}
```

# Popup Menus

A popup menu is a small window displaying a menu. It can be prepared by creating a `JPopupMenu` instance and adding one or more `JMenuItem` instances (and `JMenu` instances for submenus). A popup menu appears generally when the user activates it by right-clicking in a specified area. In order to show a popup menu, it must be associated with a component and a mouse listener should be registered to that component. The mouse listener monitors the events for that component to decide whether or not the popup menu should be displayed.

In the following a program creating a popup menu is given (the result is given on the right).

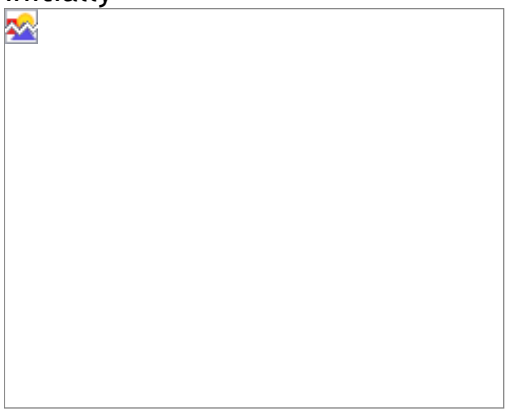
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SimpleJFrame extends JFrame {
        public SimpleJFrame(){
                initialize();
        }
        JPopupMenu popup;
        private void initialize(){
                popup = new JPopupMenu();
                JMenuItem menuItem1 = new JMenuItem("Menu Item
1");
                popup.add(menuItem1);
                popup.addSeparator();
                JMenuItem menuItem2 = new JMenuItem("MenuItem
2");
                popup.add(menuItem2);
                JMenu subMenu = new JMenu("Submenu");
                JMenuItem submenuItem = new JMenuItem("Submenu
Item");
                subMenu.add(submenuItem);
```

```
                popup.add(subMenu);

                JLabel label = new JLabel("Hello World");
                getContentPane().add(label,
BorderLayout.CENTER);
                label.addMouseListener(new MouseAdapter(){
                    public void mousePressed(MouseEvent e) {
                        maybeShowPopup(e);
                    }

                    public void mouseReleased(MouseEvent e) {
                        maybeShowPopup(e);
                    }

                    private void maybeShowPopup(MouseEvent e) {
                        if (e.isPopupTrigger()) {
                            popup.show(e.getComponent(),
                                       e.getX(), e.getY());
                        }
                    }

                });
                setPreferredSize(new Dimension(250,200));

        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
                pack();
        }
        public static void main(String[] args) {
                SwingUtilities.invokeLater(new Runnable() {
                        public void run(){
                                new
SimpleJFrame().setVisible(true);
                        }
                });
        }
}
```

# TOOLBARS

Toolbars could be used to display frequently used actions or controls on a frame. A simple toolbar can be prepared by creating a `JToolBar` instance and adding buttons to it. Another good feature of toolbars is that, it can be repositioned by the user by dragging the toolbar to a new area in the frame (or outside the frame as a separate window). The figure on the right illustrates this feature. The code to produce the example application is given below.

Initially



Toolbar has been moved to another area.

# Dialogs

A dialog is a restricted version of a frame, and it can be prepared by subclassing `JDialog` top level container. Every dialog should be associated with a frame. When that frame is destroyed the dialogs associated with it are also destroyed. When the frame is iconified, its dependent dialogs disappear from the screen. When the frame is deiconified, its dependent dialogs return to the screen. A dialog can be modal or non-modal. When a modal dialog is set to be visible, it blocks user input to all other windows in the same application.

There are some standard dialogs available to be used for specific purposes. These can be creted and shown by using the classes `JOptionPane`, `JColorChooser`, `JFileDialog`, etc.. The following table lists some of them.

| Class/method | use |
| --- | --- |
| JOptionPane.showConfirmDialog() | To ask the user a question and request confirmation |
| JOptionPane.showInputDialog() | To ask the user to enter some input |
| JOptionPane.showMessageDialog() | To inform the user about something |
| JOptionPane.showOptionDialog() | To have a combination of the features of above dialogs |
| JFileChooser.showOpenDialog() | To enable the user to choose a file in the file system. |
| JFileChooser.showSaveDialog() | To enable the user to give a file name and directory to create/save a file. |
| JColorChooser.showDialog() | To request the user to choose a color by displaying a color-chooser dialog. |