Java-based Graphical User Interface using Swing framework

Steven Zaayter

March 2023

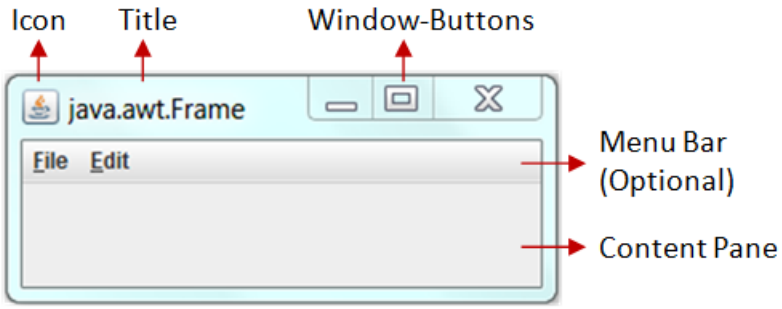
1 Introduction:

Swing is a GUI widget library in Java. It is part of Oracle's Java Foundation Classes, an API for providing a graphical user interface (GUI) for Java programs. Swing provide a more complex and versatile graphical components than its predecessor Abstract Window Toolkit (AWT).

2 Necessary Libraries:

```
1 import javax.swing.*;
2 import java.awt.*;
```

3 Graphical Components:



4 The 'JFrame':

A top level container that provide a window cover on the screen.

4.1 Constructors

```
3 JFrame(); // It constructs a new frame that is initially invisible.
4 JFrame(GraphicsConfiguration gc); // It creates a Frame in the
      specified GraphicsConfiguration of a screen device and a blank
      title.
5 JFrame(String title); // It creates a new, initially invisible Frame
       with the specified title.
6 JFrame(String title, GraphicsConfiguration gc); // It creates a
      JFrame with the specified title and the specified
      GraphicsConfiguration of a screen device.
```

```
7         //Creating a new JFrame
8         JFrame frame = new JFrame();
```

4.2 Common Methods

JFrame Methods		
Type	*Method*	*Description*
void	`.setTitle(String s);`	*It sets the title to be displayed as the Title for this window.*
void	`.setIconImage(Image image);`	*It sets the image to be displayed as the icon for this window.*
void	`.setDefaultCloseOperation(int);`	*Specify what happens when the user closes a JFrame or JDialog.*
void	`.setSize(int width, int height);`	*It sets the size of a JFrame, JPanel, or other Swing components.*
void	`.setLocation(int x, int y);`	*It sets the location of a window (e.g. JFrame, JDialog) on the screen.*
void	`.setLocationRelativeTo(null);`	*It center the frame on the screen.*
void	`.setVisible(boolean b);`	*It shows or hide a window, such as a JFrame or JDialog.*
void	`.setMenuBar(JMenuBar bar);`	*It sets the menu bar for a 'JFrame'.*
void	`.pack();`	*It adjusts the size of the JFrame so that it is just big enough to fit all its child components with their preferred sizes.*

4.3 'Frame' Class Exemple

```
public class Frame extends JFrame{
    JButton button = new JButton();
    public Frame(){
        initComponent();
    }
    private void initComponent(){
        setTitle("Test Title"); // this.setTitle();
        setSize(800,600);
        add(button, BorderLayout.North);
    }
    public static void main(String[] args){
        EventQueue.invokeLater(new Runnable(){
            @override
            public void run(){
                new Frame.setVisible(true);
            }
        });
    }
}
```

1. *The* `'EventQueue.invokeLater()'` *method schedules a* `'Runnable'` *object to be executed on the event-dispatching thread. This ensures that the code that creates and shows the* `Frame` *object is executed on the correct thread to avoid any thread-safety issues.*

2. *The* `new Runnable(){...}` *syntax creates a new anonymous inner class that implements the* `'Runnable'` *interface. The class overrides the* `'run(')` *method to define the code that will be executed on the event-dispatching thread.*

3. *The* `'run()'` *method contains the code that creates a new* `Frame` *object and sets its visibility to* `true` *using the* `'setVisible()'` *method.*

4. *When the* `'invokeLater()'` *method executes the* `'run()'` *method, a new* `Frame` *object is created and shown on the screen.*

Event Dispatching Thread: *The EDT in Swing is a special thread that is responsible for handling events generated by user interactions with a Swing GUI. The EDT is a dedicated thread that is created by the Java Virtual Machine (JVM) when a Swing application is started. It is responsible for dispatching events to the appropriate event handlers in the Swing application.*

5 The 'JButton':

5.1 Constructors

```
JButton(); // Creates a button with no set text or icon.
JButton(Action a); // Creates a button where properties are taken
    from the Action supplied.
JButton(String s); // Creates a button with the specified text.
JButton(Icon i); // Creates a button with an icon.
```

5.2 Common Methods

JButton Methods		
Type	*Method*	*Description*
void	`.setText(String s);`	*It sets the text to be displayed on the Button.*
String	`.getText();`	*It returns the text displayed on the Button.*
void	`.setEnabled(boolean b);`	*It enables or disables the button.*
void	`.setIcon(Icon i);`	*It sets the icon that is displayed on a Button component.*
void	`.addAcrionListener(ActionListener a);`	*It registers an 'ActionListener' object to the Button.*

5.3 Display a Button in a JFrame

To display a button in a JFrame, you need to create an instance of the JButton class, and then add the JFrame.

```
1        //Method 1:
2        JButton button = new JButton(); // Creating a JButton instance
3        frame.add(button, BorderLayout.NORTH); //add a button component to a
             JFrame using the BorderLayout layout manager and place it in
             the frame's northern region.
```

```
4        //Method 2:
5        frame.setLayout(new FlowLayout(FlowLayout.LEFT); // sets the layout
             manager of the frame object to FlowLayout with a left alignment.
6        frame.add(button); // //add a button component to a JFrame.
```

6 Layout Manager:

The layout manager automatically positions all the components within the container. Even if you do not use the layout manager, the components are still positioned by the default layout manager. It is possible to lay out the controls by hand, however, it becomes very difficult.

6.1 BorderLayout

'BorderLayout' is a popular layout manager used in Java Swing to arrange components in a container such as a JFrame. Here are some key points to know about BorderLayout:

1. *BorderLayout is the default layout manager for a JFrame.*
2. *BorderLayout divides a container into five regions: north, south, east, west, and center.*
3. *When adding a component to a container that uses BorderLayout, you must specify the region where you want to place the component. For example, to place a button in the north region, you would use:*

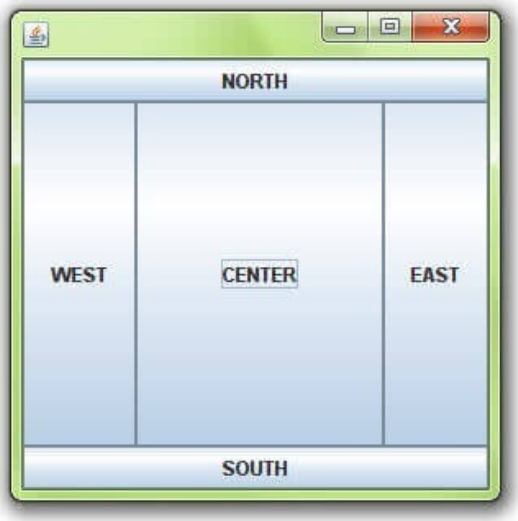


Figure 1: BorderLayout

```
frame.add(button, BorderLayout.NORTH);
```

4. *If a region is not specified when adding a component, it will be placed in the center region by default.*
5. *Components in the center region will take up all the available space in the container.*
6. *You can set the horizontal and vertical gaps between components using the 'setHgap(int hgap)' and 'setVgap(int vgap)' methods of the BorderLayout manager.*

6.2 FlowLayout

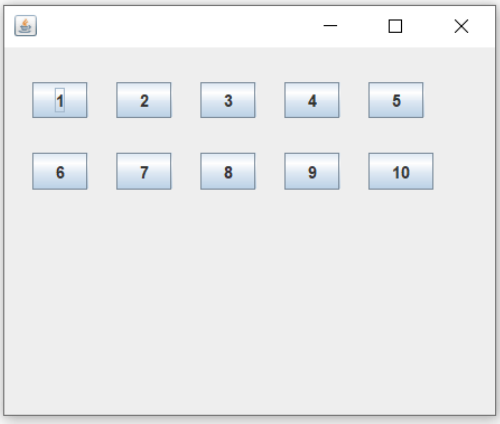


Figure 2: FlowLayout

'FlowLayout' is a layout manager in Java Swing that arranges components in a flow, either horizontally or vertically, based on their preferred size. It's one of the simplest and most straightforward layout managers, and it's often used in small Swing applications.

1. *Components are added to the container in the order they are added to the container.*

2. *When the container becomes full, the next component is added to the next row or column, depending on the orientation of the layout.*

3. *You can specify the alignment of components within each row or column using the 'FlowLayout.CENTER', 'FlowLayout.LEFT', or 'FlowLayout.RIGHT' constants. The default is 'FlowLayout.CENTER'.For example, in this code, 'FlowLayout' is defined with a horizontal alignment of CENTER:*

```
FlowLayout flowLayout = new FlowLayout(FlowLayout.CENTER);
frame.setLayout(flowLayout);
```

4. *The size of each component is determined by its preferred size, which is set by the 'setPreferredSize()' method or is calculated by the component itself. If a component's preferred size is greater than the size of the container, it will be truncated or clipped.*

6.3 GridLayout

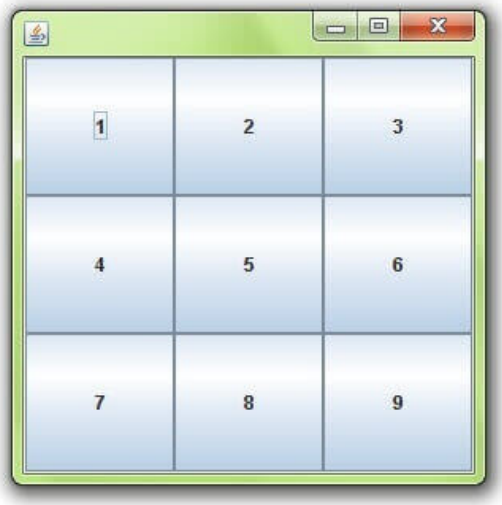


Figure 3: GridLayout

'GridLayout' is a layout manager that arranges components in a grid of cells, with each cell having the same size.

1. *The number of rows and columns in the grid is specified when the 'GridLayout' object is created:*

   ```
   frame.setLayout(new GridLayout(rows, columns));
   ```

2. *If you set the value of 'rows' or 'columns' to 0 in the constructor, the layout manager will calculate the number of 'rows' or 'columns' automatically based on the number of components you add to the container.*

3. *Components are added to the container in the order they are added to the container, filling each row from left to right and top to bottom.*

4. *All components in the grid have the same size, which is determined by dividing the container's size by the number of rows and columns in the grid.*

5. *You can set the horizontal and vertical gaps between components using the 'setHgap()' and 'setVgap()' methods of the 'GridLayout' object.*

6. *If a component's preferred size is greater than the size of its cell in the grid, it will be truncated or clipped.*

7 Menu Components:

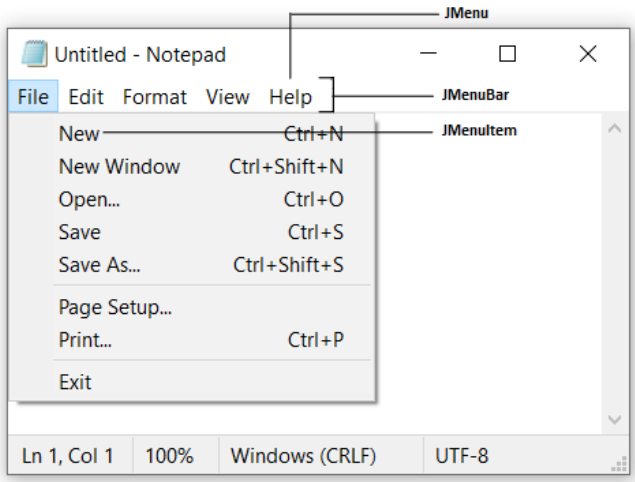


Figure 4: Menu Components

7.1 The "JMenuBar"

The JMenuBar class provides an implementation of a menu bar.

7.1.1 Constructors

```
1 JMenuBar(); // Creates a new menu bar.
```

7.1.2 Common Methods

JMenuBar Methods		
Type	*Method*	*Description*
JMenu	`.add(JMenu m);`	*It appends the specified menu to the end of the menu bar.*
void	`.addNotify();`	*It overrides 'JComponent.addNotify' to register this menu bar with the current keyboard manager.*
boolean	`.isSelectod();`	*It returns true if the menu bar currently has a component selected.*
String	`.paramString();`	*It returns a protected string representation of this JMenuBar.*

7.2 The "JMenu"

The JMenu Class represents the pull-down menu component which is deployed from a menu bar.

7.2.1 Constructors

```
JMenu(); // Creates a new JMenu with no text.
JMenu(Action a); // Creates a menu whose properties are taken
    from the Action supplied.
JMenu(String s); // Creates a new JMenu with the supplied string
     as its text.
JMenu(String s, boolean b); // Creates a new JMenu with the
    supplied string as its text and specified as a tear-off menu
     or not.
```

7.2.2 Common Methods

JMenu Methods		
Type	*Method*	*Description*
JMenuItem	`.add(JMenuItem menuItem);`	*It appends the specified menu item to the end of this menu.*
void	`.add(new JSeperator());`	*It adds a 'JSeparator' to a container, such as a 'JMenu'.*
boolean	`.addSeperator();`	*It adds a 'JSeparator' to a container, such as a 'JMenu'.*

***The reason to use 'menu.add(new JSeparator())' instead of 'menu.addSeparator()'** is that 'addSeparator()' is just a convenience method that creates and adds a new 'JSeparator' to the menu or popup menu, whereas 'add(new JSeparator())' gives you more control over the 'JSeparator' component. For example, you can set the orientation and size of the 'JSeparator' by calling its methods before adding it to the container.*

```
JSeparator separator = new JSeparator();
separator.setOrientation(SwingConstants.VERTICAL);
separator.setPreferredSize(new Dimension(10, 30));
```

7.3 The "JMenuItem"

The JMenuItem class represents the actual item in a menu.

7.3.1 Constructors

```
JMenuItem(); // Creates a JMenuItem with no set text or icon.
JMenuItem(String text, Icon icon); // Creates a JMenuItem with
    the specified text and icon.
JMenuItem(String text, int mnemonic); // Creates a JMenuItem
    with the specified text and keyboard mnemonic
```

7.3.2 Common Methods

JMenuItem Methods		
Type	*Method*	*Description*
void	`.setEnabled(boolean b);`	*It enables or disables the menu item.*
MenuElement[]	`.getSubElements();`	*It returns an array containing the sub-menu components for this menu component.*

8 Text Components:

8.1 The "JTextField"

The JTextField class is a text component that allows the editing of a single-line text.

8.1.1 Constructors

```
1 JTextField(); // Creates a new TextField.
2 JTextField(String text); // Creates a new TextField initialized
      with the specified text.
3 JTextField(int columns); // Creates a new empty TextField with
      the specified number of columns.
```

8.1.2 Common Methods

JTextField Methods		
Type	*Method*	*Description*
void	`.addActionListener(ActionListener l);`	*It adds the specified action listener to receive action events from this textfield*
void	`.setFont(Font f);`	*It sets the current font.*

8.2 The "JTextArea"

JTextArea class is a multi line region that displays text. It allows the editing of multiple line text.

8.2.1 Constructors

```
1 JTextArea(); // Creates a text area that displays no text
      initially.
2 JTextArea(String s); // Creates a text area that displays
      specified text initially.
3 JTextArea(String s, int row, int column); // Creates a text area
       with the specified number of rows and columns that displays
       specified text.
```

8.2.2 Common Methods

JTextArea Methods		
Type	*Method*	*Description*
void	`.setFont(Font f);`	*It sets the specified font.*
void	`.insert(String s, int position);`	*It inserts the specified text on the specified position.*
void	`.append(String s);`	*It appends the given text to the end of the document.*
void	`.setRows(int rows);` `.setColomns(int columns)`	*It sets the specified number of rows/-columns.*

***The Font class** is used to specify the appearance of text in graphical user interfaces and can be applied to various Swing components such as 'JLabel', 'JButton', 'JTextField'. Here's an example of how to create a Font object in Java:*

```
1         // create a Font object with default attributes
2             Font defaultFont = new Font("Arial", Font.PLAIN, 12);
3
4         // create a Font object with bold and italic style
5                 Font boldItalicFont = new Font("Times New Roman", Font.BOLD
                      | Font.ITALIC, 16);
```

8.3 The "JLabel"

The JLabel class display a single line of read-only text.

8.3.1 Constructors

```
1             JLabel(); // Creates a blank label with no text or image in it.
2             JLabel(String s); // Creates a new label with the string
                  specified.
3             JLabel(String s, Icon icon, int HorizontalAlign); // Creates a
                  new label with a string, an image, and a specified
                  horizontal alignment.
```

8.3.2 Common Methods

JLabel Methods		
Type	*Method*	*Description*
void	`.setFont(Font f);`	*It sets the specified font.*
void	`.getText();`	*It returns the text that the label will display.*
void	`.setText(String s);`	*It sets the text that the label will display.*
void	`.setHorizontalAlignement(int align);`	*It sets the specified horizontal alignment.*

9 The 'JPanel':

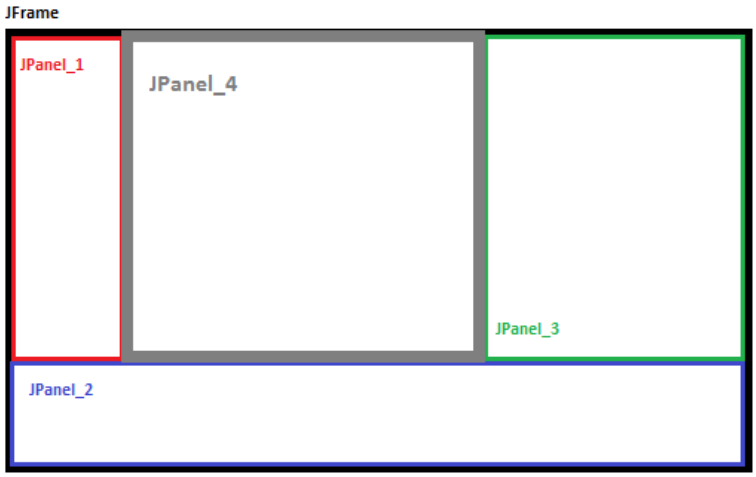


Figure 5: JPanel vs JFrame

The JPanel class is a container that can store a group of components.

9.1 Constructors

```
1  JPanel(); // Creates a new JPanel with a double buffer and a flow
       layout.
2  JPanel(LayoutManager l); // Creates a new buffered JPanel with the
       specified layout manager.
```

9.2 Common Methods

JPanel Methods		
Type	*Method*	*Description*
void	`.add(Component C);`	*It adds a Component (such as a button, label, or another panel) to the panel.*
void	`.setLayout(LayoutManager l);`	*It sets the layout manager for the panel.*
void	`.setBorder(Border b);`	*It sets the border of the panel to the specified Border object.(Use the 'BorderFactory' class)*

The "BorderFactory"

The `BorderFactory` *class is a utility class that provides methods for creating standard borders for Swing components. The class is part of the* `javax.swing` *package and can be imported using the following statement:*

```
import javax.swing.BorderFactory;
```

The following are some of the commonly used methods of the `BorderFactory` *class:*

- `createLineBorder(Color color)`*: Creates a line border with the specified color.*
- `createEtchedBorder()`*: Creates an etched border with a raised or lowered 3D effect.*
- `createTitledBorder(String title)`*: Creates a titled border with the specified title.*
- `createCompoundBorder(Border outsideBorder, Border insideBorder)`*: Creates a compound border with the specified outside and inside borders.*

Each of these methods returns a `Border` *object that can be passed to the* `setBorder()` *method of a Swing component to set the border.*

Here's an example of how to create a line border with a red color and set it as the border of a `JButton`*:*

```
JButton button = new JButton("Click me");
Border border = BorderFactory.createLineBorder(Color.RED);
button.setBorder(border);
```

This code creates a `JButton` *with the text "Click me", creates a line border with a red color using the* `createLineBorder()` *method, and sets the border of the button to the created border using the* `setBorder()` *method.*

10 The 'JScrollPane':

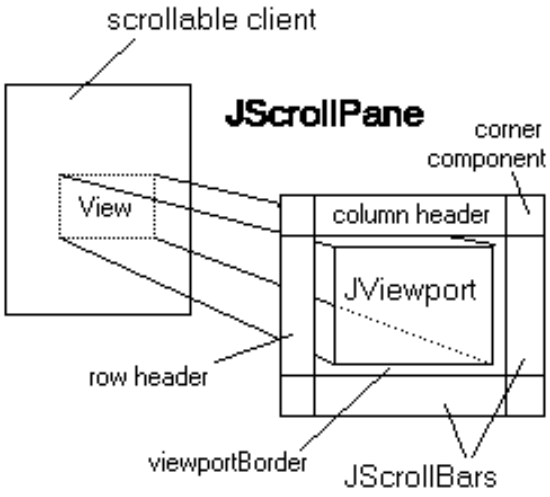


Figure 6: JPanel vs JFrame

The JScrollPane class is used to make scrollable view of a component.

10.1 Constructors

```
1        JScrollPane();
2        JScrollPane(Component c, int v, int h);
3    // It creates a scroll pane. The Component parameter, when present, sets
         the scroll pane's client. The two int parameters, when present, set
         the vertical and horizontal scroll bar policies.
```

10.2 Common Methods

JScrollPane Methods		
Type	*Method*	*Description*
void	`.setHorizontalScrollBarPolicy(JScrollBar horizontalScrollBar);` `.setVerticalScrollBarPolicy(JScrollBar verticalScrollBar);`	*It sets the horizontal/vertical scroll bar policy for the scroll pane.*
void	`.setViewportView(Component c);`	*Creates a viewport if necessary and then sets its view.*
void	`.setPreferredSize(Dimension preferredSize);`	*It sets the preferred size of the panel.*
JViewport	`.getViewport();`	*It returns the 'JViewport' object that displays the content of the scroll pane.*

*To **implement** a `JScrollPane` in your Swing application, you can follow these steps:*

1. *Create the component you want to display in the scroll pane, such as a `JTextArea`.*

```
JTextArea text = new JTextAre();
```

2. *Create a `JScrollPane` object and pass the component as a parameter to its constructor.*

```
JScrollPane scrollPane = new JScrollPane(text);
```

3. *Set the preferred size of the scroll pane and its view port.*

```
scrollPane.setPreferredSize(new Dimension(300, 200));
scrollPane.getViewport().setPreferredSize(new Dimension(280, 180));
```

   In this example, we set the preferred size of the scroll pane to 300x200 pixels, and the preferred size of its view port to 280x180 pixels. The view port is the visible part of the component that is displayed inside the scroll pane.

4. *Add the scroll pane to your user interface.*

```
frame.add(scrollPane);
```

11 The 'Look and Feel':

In Swing, the 'look and feel' determine the appearance of the graphical user interface components, such as buttons, menus, and dialogs. Swing provides several 'look and feel' implementations, including the default look and feel, which is called "Metal". You can change the look and feel of your Swing application to make it look like a native application on different operating systems, or to provide a custom theme that matches your application's style. Here's how you can change the 'look and feel' to match your OS 'look and feel':

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

---

12 The 'JComboBOx':

The JComboBox is used to show a popup menu of choices. The choice selected by the user is shown at the top of the menu.

12.1 Constructors

```
1 JComboBox(); // Creates a JComboBox with a default data model.
2 JComboBox(Object[] items); // Creates a JComboBox that contains
      the elements in the specified array.
```

12.2 Common Methods

JComboBox Methods		
Type	*Method*	*Description*
void	`.addItem(Object object);`	*It adds an item to the item list.*
void	`.removeItem(Object object);`	*It deletes an item to the item list.*
void	`.removeAllItem();`	*It deletes all the items from the list.*
void	`.setEditable(boolean b);`	*It determines whether the JComboBox is editable.*
void	`.addActionListener(ActionListener a);`	*It adds the ActionListener.*
void	`.addItemListener(ActionListener a);`	*It adds the ItemListener.*
int	`.getSelectedIndex();`	*It returns the index of the currently selected item in the combo box.*
string	`.geySelectedItem();`	*It returns the label of the selected item in the combo box.*

13 Event Handling:

Change in the state of an object is known as an 'Event'. An event describes the change in the state of the source object. Events are generated as a result of user interaction with the graphical user interface components. Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has a code which is known as an event handler, that is executed when an event occurs. The 2 major players in the Delegation Event Model are as follows.

- `Source`*: The source is an object on which the event occurs. Source is responsible for providing information about the occurred event to its handler. Java provides us with classes for the source object.*
- `Listener`*: The listener is responsible for generating a response to an event. From the point of view of Java implementation, a listener is also an object. The listener waits till it receives an event. Once the event is received, the listener processes the event and then returns.*

14 Event Listeners:

Event listeners represent the interfaces responsible to handle events. It is an object which is notified when an event occurs. The following are several standard Java interfaces along with the events that they can listen for and handle:

Event Listener Interfaces	
Interface	*Description*
`ActionListener`	*Listens for and handles button clicks.*
`KeyListener`	*Listens for and handles key events.*
`MouseListener`	*Listens for and handles mouse events.*
`MouseMotionListener`	*Listens for and handles mouse drag and move events.*
`TextListener`	*Listens for and handles text changing events.*
`ItemListener`	*Listens for and handles CheckBox and Listbox events.*
`AdjustmentListener`	*Listens for and handles scrolling events.*
`WindowListener`	*Listens for and handles window events.*
`FocusListener`	*Listens for and handles focus events.*

Here are the steps to implement an event listener in Swing:

1. *Create a class that implements the appropriate event listener interface, Here's an example of how to create an ActionListener implementation:*

```
class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // Action to perform when the button is clicked
    }
}
```

2. *Create an instance of the listener class and register it with the component. In this example, we will create an instance of the MyActionListener class and register it with a JButton component using the addActionListener() method:*

```
JButton myButton = new JButton("Click me!");
MyActionListener myListener = new MyActionListener();
myButton.addActionListener(new MyActionListener () ;);
```

3. *Implement the event handling code in the actionPerformed() method of your listener class. In this example, we will simply display a message dialog when the button is clicked:*

```
class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "Button clicked!
            ");
    }
}
```

Using an anonymous inner class that implements the ActionListener interface, you can define the event handling code inline, without the need to create a separate listener class. Here's an example:

```
JButton myButton = new JButton("Click me!");
myButton.addActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent e){
        // Action to perform
    }
});
```

14.1 Event Listeners Interfaces

1. ***ActionListener:*** *The class which processes the ActionEvent should implement this interface. The object of that class must be registered with a component. The object can be registered using the* `addActionListener()` *method.*

Interface Methods		
Type	*Method*	*Description*
void	`actionPerformed(ActionEvent e);`	*Invoked when an action occurs.*

2. ***ItemListener:*** *The class which processes the ItemEvent should implement this interface. The object of that class must be registered with a component. The object can be registered using the* `addItemListener()` *method.*

Interface Methods		
Type	*Method*	*Description*
void	`itemStateChanged(ItemEvent e);`	*Invoked when an item has been selected or deselected by the user.*

3. ***KeyListener:*** *The class which processes the KeyEvent should implement this interface. The object of that class must be registered with a component. The object can be registered using the* `addKeyListener()` *method.*

Interface Methods		
Type	*Method*	*Description*
void	`keyPressed(KeyEvent e);`	*Invoked when a key has been pressed.*
void	`keyReleased(KeyEvent e);`	*Invoked when a key has been released.*
void	`keyTyped(KeyEvent e);`	*Invoked when a key has been typed.*

4. ***MouseListener:*** *The class which processes the MouseEvent should implement this interface. The object of that class must be registered with a component. The object can be registered using the* `addMouseListener()` *method.*

Interface Methods		
Type	*Method*	*Description*
void	`mouseClicked(MouseEvent e);`	*Invoked when the mouse button has been clicked (pressed and released) on a component.*
void	`mouseEntered(MouseEvent e);`	*Invoked when the mouse enters a component.*
void	`mouseExited(MouseEvent e);`	*Invoked when the mouse exits a component.*
void	`mousePressed(MouseEvent e);`	*Invoked when a mouse button has been pressed on a component.*
void	`mouseReleased(MouseEvent e);`	*Invoked when a mouse button has been released on a component.*

5. ***MouseMotionListener:*** *The interface MouseMotionListener is used for receiving mouse motion events on a component. The class that processes mouse motion events needs to implement this interface.*

Interface Methods		
Type	*Method*	*Description*
void	`mouseDragged(MouseEvent e);`	*Invoked when a mouse button is pressed on a component and then dragged.*
void	`mouseMoved(MouseEvent e);`	*Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.*

6. ***FocusListener:*** *The interface FocusListener is used for receiving keyboard focus events. The class that processes focus events needs to implements this interface.*

Interface Methods		
Type	*Method*	*Description*
void	`focusGained(FocusEvent e);`	*Invoked when a component gains the keyboard focus.*
void	`focusLost(FocusEvent e);`	*Invoked when a component loses the keyboard focus.*

Event Adapters: *An event adapter is a class that provides empty implementations of all methods in an event listener interface. You would typically extend an event adapter if you only need to handle one or two methods in the interface, and you don't want to provide empty implementations for all the other methods.*

15 The 'Graphics2D' Class:

This Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-dimensional shapes, text and images on the Java(tm) platform.

15.1 Constructors

```
1 Graphics2D(); // Constructs a new Graphics2D object.
```

15.2 Common Methods

Graphics2D Methods		
Type	*Method*	*Description*
void	`.drawString(String str, float x, float y);`	*Renders the text specified by the specified String, using the current text attribute state in the Graphics2D context.*
void	`.setColor(Color c);`	*It sets the current drawing color to the give Color object.*
void	`.drawLine(int x, int y, Color c);`	*It draws a line between point (x1, y1) and (x2, y2).*
void	`.setStroke(BasicStrokes);`	*It sets the stroke (line thickness) used for subsequent drawing operations, such as lines, shapes, and text.*
void	`.repaint();`	*It requests that a component or a part of a component be repainted.*

Anti-aliasing: *is a technique used to smooth the jagged edges of geometric shapes and text that are displayed on a computer screen. It works by blending the color of the shape's edge with the colors of the pixels around it, giving the appearance of a smoother edge.In Swing, the method:*

```
1 setRenderingHint(RenderingHints.KEY_ANTIALIASING,RenderingHints.
      VALUE_ANTIALIASING_ON)
```

is used to turn on anti-aliasing for rendering graphics.

Class Cursor: *is a class to encapsulate the bitmap representation of the mouse cursor. Here's an example on how to change the cursor type in swing:*

```
1 Component.setCursor(Cursor.getPredefinedCursor(Cursor.
      CROSSHAIR_CURSOR)); // The crosshair cursor type.
2  CUSTOM_CURSOR // The default cursor type (gets set if no cursor
       is defined).
3  TEXT_CURSOR // The text cursor type.
4  WAIT_CURSOR // The wait cursor type.
```

References

[1] JavaTpoint. (n.d.). Java Swing Tutorial. Retrieved from https://www.javatpoint.com/java-swing

[2] TutorialsPoint. (n.d.). Swing Tutorial. Retrieved from https://www.tutorialspoint.com/swing/index.htm

[3] GeeksforGeeks. (n.d.). Introduction to Java Swing. Retrieved from https://www.geeksforgeeks.org/introduction-to-java-swing/

[4] TutorialsFields. (n.d.). JButton ActionListener. Retrieved from https://www.tutorialsfield.com/jbutton-click-event/

[5] Jiménez, O. (Fall 2008-2009).Draw, Paint, Repaint. Retrieved from https://web.stanford.edu/class/archive/cs/cs108/cs108.1092/handouts/27PaintRepaint.pdf

[6] Oracle. (n.d.).Java SE 7 and 8 Documentation. Retrieved from https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html