Lesson 13
GUI Components: Part 2

Assoc. Prof. Marenglen Biba

today

In this Chapter you'll learn:

- To create and manipulate sliders, menus, pop-up menus and windows.
- To programatically change the look-and-feel of a GUI, using Swing's pluggable look-and-feel.
- To create a multiple-document interface with `JDesktopPane` and `JInternalFrame`.
- To use additional layout managers.

14.18 Introduction to Layout Managers

- Layout managers arrange GUI components in a container for presentation purposes
- Can use for basic layout capabilities
- Enable you to concentrate on the basic look-and-feel — the layout manager handles the layout details.
- Layout managers implement interface `LayoutManager` (in package `java.awt`).
- `Container`'s `setLayout` method takes an object that implements the `LayoutManager` interface as an argument.

14.18 Introduction to Layout Managers (cont.)

- There are three ways for you to arrange components in a GUI:
  - Absolute positioning
    - Greatest level of control.
    - Set `Container`'s layout to `null`.
    - Specify the absolute position of each GUI component with respect to the upper-left corner of the `Container` by using `Component` methods `setSize` and `setLocation` or `setBounds`.
    - Must specify each GUI component's size.

14.18 Introduction to Layout Managers (cont.)

- Layout managers
  - Simpler and faster than absolute positioning.
  - Lose some control over the size and the precise positioning of GUI components.
- Visual programming in an IDE
  - Use tools that make it easy to create GUIs.
  - Allows you to drag and drop GUI components from a tool box onto a design area.
  - You can then position, size and align GUI components as you like.

Layout manager	Description
`FlowLayout`	Default for `javax.swing.JPanel`. Places components sequentially (left to right) in the order they were added. It's also possible to specify the order of the components by using the `Container` method `add`, which takes a `Component` and an integer index position as arguments.
`BorderLayout`	Default for `JFrames` (and other windows). Arranges the components into five areas: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
`GridLayout`	Arranges the components into rows and columns.

Fig. 14.38 | Layout managers.

Look-and-Feel Observation 14.17

Each individual container can have only one layout manager, but multiple containers in the same application can each use different layout managers.

```
1  // Fig. 14.39: FlowLayoutFrame.java
2  // Demonstrating FlowLayout alignments.
3  import java.awt.FlowLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class FlowLayoutFrame extends JFrame
11 {
12    private JButton leftJButton; // button to set alignment left
13    private JButton centerJButton; // button to set alignment center
14    private JButton rightJButton; // button to set alignment right
15    private FlowLayout layout; // layout object
16    private Container container; // container to set layout
17
18    // set up GUI and register button listeners
19    public FlowLayoutFrame()
20    {
21       super( "FlowLayout Demo" );
22
```

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 1 of 4.)

```
23          layout = new FlowLayout(); // create FlowLayout
24          container = getContentPane(); // get container to layout
25          setLayout( layout ); // set frame layout
26
27          // set up leftJButton and register listener
28          leftJButton = new JButton( "Left" ); // create Left button
29          add( leftJButton ); // add Left button to frame
30          leftJButton.addActionListener(
31
32             new ActionListener() // anonymous inner class
33             {
34                // process leftJButton event
35                public void actionPerformed( ActionEvent event )
36                {
37                   layout.setAlignment( FlowLayout.LEFT );
38
39                   // realign attached components
40                   layout.layoutContainer( container );
41                } // end method actionPerformed
42             } // end anonymous inner class
43          ); // end call to addActionListener
44
```

Left aligns the components.

Lays out the container's components again based on the layout changes.

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 2 of 4.)

```
45          // set up centerJButton and register listener
46          centerJButton = new JButton( "Center" ); // create Center button
47          add( centerJButton ); // add Center button to frame
48          centerJButton.addActionListener(
49
50             new ActionListener() // anonymous inner class
51             {
52                // process centerJButton event
53                public void actionPerformed( ActionEvent event )
54                {
55                   layout.setAlignment( FlowLayout.CENTER );
56
57                   // realign attached components
58                   layout.layoutContainer( container );
59                } // end method actionPerformed
60             } // end anonymous inner class
61          ); // end call to addActionListener
62
63          // set up rightJButton and register listener
64          rightJButton = new JButton( "Right" ); // create Right button
65          add( rightJButton ); // add Right button to frame
```

Center aligns the components.

Lays out the container's components again based on the layout changes.

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 3 of 4.)

```
66          rightJButton.addActionListener(
67
68             new ActionListener() // anonymous inner class
69             {
70                // process rightJButton event
71                public void actionPerformed( ActionEvent event )
72                {
73                   layout.setAlignment( FlowLayout.RIGHT );
74
75                   // realign attached components
76                   layout.layoutContainer( container );
77                } // end method actionPerformed
78             } // end anonymous inner class
79          ); // end call to addActionListener
80       } // end FlowLayoutFrame constructor
81    } // end class FlowLayoutFrame
```

Right aligns the components.

Lays out the container's components again based on the layout changes.

Fig. 14.39 | `FlowLayout` allows components to flow over multiple lines. (Part 4 of 4.)

```
1  // Fig. 14.40: FlowLayoutDemo.java
2  // Testing FlowLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class FlowLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
10       flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       flowLayoutFrame.setSize( 300, 75 ); // set frame size
12       flowLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class FlowLayoutDemo
```

Fig. 14.40 | Test class for `FlowLayoutFrame`. (Part 1 of 2.)

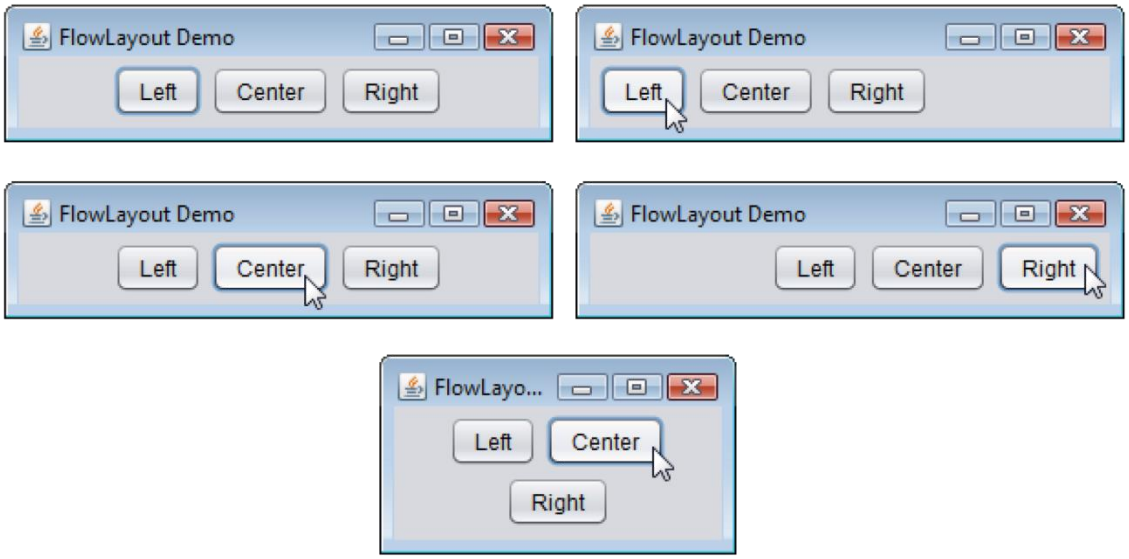


Fig. 14.40 | Test class for `FlowLayoutFrame`. (Part 2 of 2.)

14.18.1 FlowLayout (cont.)

- FlowLayout method setAlignment changes the alignment for the FlowLayout.
  - FlowLayout.LEFT
  - FlowLayout.CENTER
  - FlowLayout.RIGHT
- LayoutManager interface method layoutContainer (which is inherited by all layout managers) specifies that a container should be rearranged based on the adjusted layout.

14.18.2 BorderLayout

- BorderLayout
  - the default layout manager for a Jframe
  - arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER.
  - NORTH corresponds to the top of the container.
- BorderLayout implements interface LayoutManager2 (a subinterface of LayoutManager that adds several methods for enhanced layout processing).
- BorderLayout limits a Container to at most five components — one in each region.
  - The component placed in each region can be a container to which other components are attached.

```
1  // Fig. 14.41: BorderLayoutFrame.java
2  // Demonstrating BorderLayout.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8
9  public class BorderLayoutFrame extends JFrame implements ActionListener
10 {
11    private JButton[] buttons; // array of buttons to hide portions
12    private static final String[] names = { "Hide North", "Hide South",
13       "Hide East", "Hide West", "Hide Center" };
14    private BorderLayout layout; // borderlayout object
15
16    // set up GUI and event handling
17    public BorderLayoutFrame()
18    {
19       super( "BorderLayout Demo" );
20
21       layout = new BorderLayout( 5, 5 ); // 5 pixel gaps
22       setLayout( layout ); // set frame layout
23       buttons = new JButton[ names.length ]; // set size of array
```

Custom `BorderLayout` with horizontal and vertical gap space.

Fig. 14.41 | `BorderLayout` containing five buttons. (Part 1 of 3.)

```
24
25          // create JButtons and register listeners for them
26          for ( int count = 0; count < names.length; count++ )
27          {
28             buttons[ count ] = new JButton( names[ count ] );
29             buttons[ count ].addActionListener( this );
30          } // end for
31
32          add( buttons[ 0 ], BorderLayout.NORTH ); // add button to north
33          add( buttons[ 1 ], BorderLayout.SOUTH ); // add button to south
34          add( buttons[ 2 ], BorderLayout.EAST ); // add button to east
35          add( buttons[ 3 ], BorderLayout.WEST ); // add button to west
36          add( buttons[ 4 ], BorderLayout.CENTER ); // add button to center
37       } // end BorderLayoutFrame constructor
38
```

This `BorderLayoutFrame` handles each `JButton`'s `ActionEvent`.

Fig. 14.41 | `BorderLayout` containing five buttons. (Part 2 of 3.)

```
39      // handle button events
40      public void actionPerformed( ActionEvent event )
41      {
42         // check event source and lay out content pane correspondingly
43         for ( JButton button : buttons )
44         {
45            if ( event.getSource() == button )
46               button.setVisible( false ); // hide button clicked
47            else
48               button.setVisible( true ); // show other buttons
49         } // end for
50
51         layout.layoutContainer( getContentPane() ); // lay out content pane
52      } // end method actionPerformed
53   } // end class BorderLayoutFrame
```

Hides the button.

Shows the button.

Re-lays out the container.

Fig. 14.41 | `BorderLayout` containing five buttons. (Part 3 of 3.)

```
1  // Fig. 14.42: BorderLayoutDemo.java
2  // Testing BorderLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class BorderLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
10       borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       borderLayoutFrame.setSize( 300, 200 ); // set frame size
12       borderLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class BorderLayoutDemo
```

Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 1 of 3.)

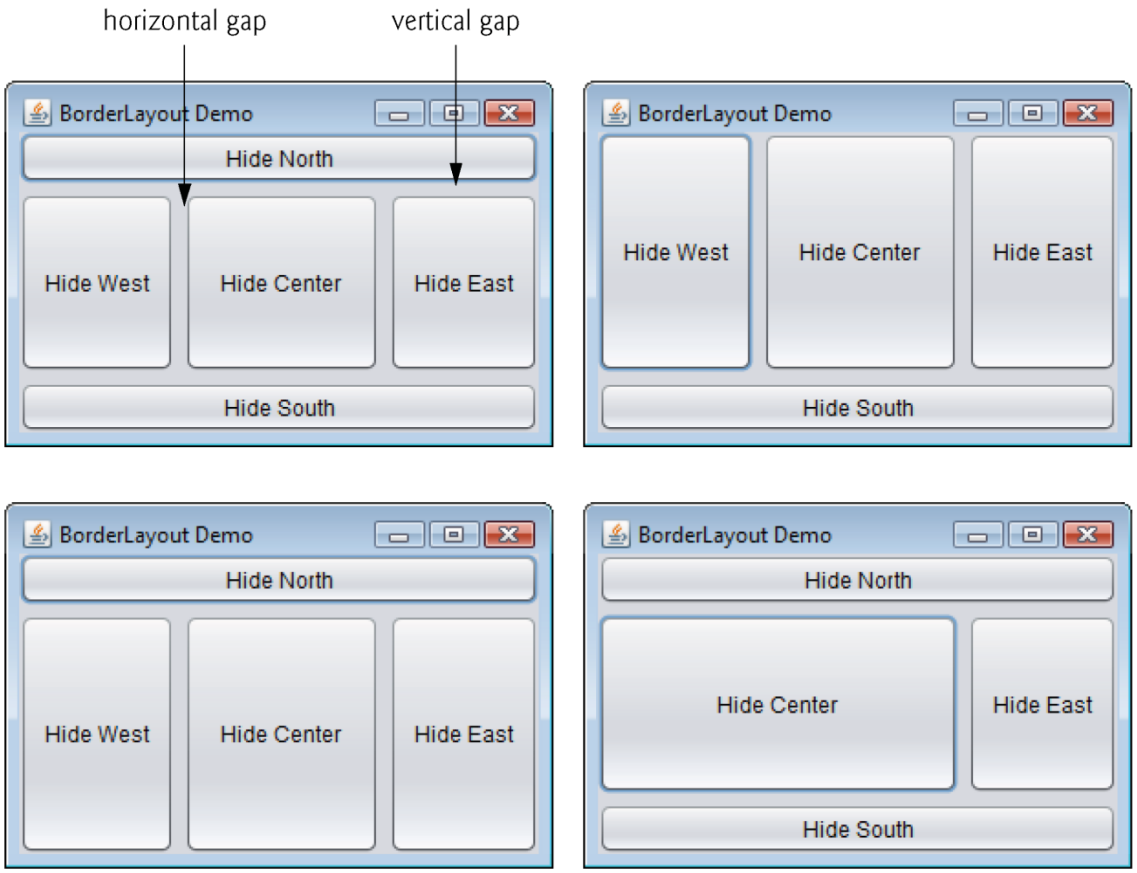


Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 2 of 3.)

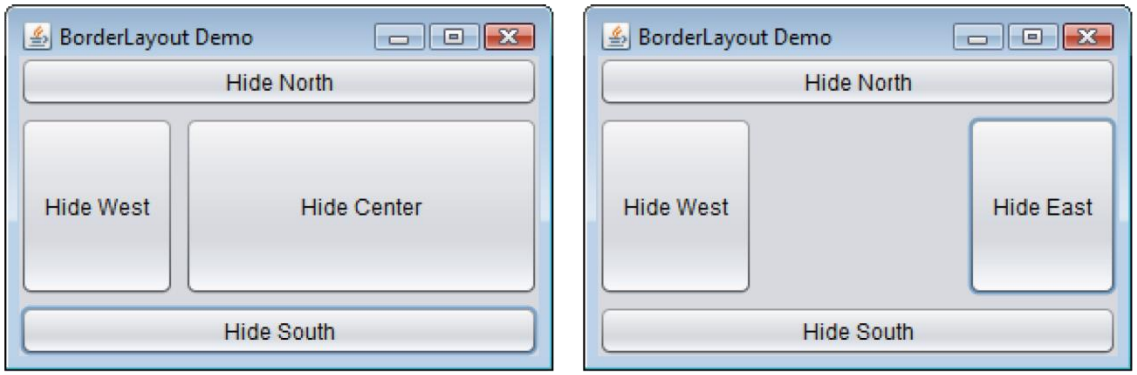


Fig. 14.42 | Test class for `BorderLayoutFrame`. (Part 3 of 3.)

Look-and-Feel Observation 14.18

If no region is specified when adding a `Component` *to a* `BorderLayout`*, the layout manager assumes that the* `Component` *should be added to region* `BorderLayout.CENTER`.

Common Programming Error 14.6

When more than one component is added to a region in a `BorderLayout`*, only the last component added to that region will be displayed. There is no error that indicates this problem.*

14.18.3 GridLayout

- `GridLayout` divides the container into a grid of rows and columns.
  - Implements interface `LayoutManager`.
  - Every `Component` has the same width and height.
  - Components are added starting at the top-left cell of the grid and proceeding left to right until the row is full.
  - Then the process continues left to right on the next row of the grid, and so on.
- `Container` method `validate` recomputes the container's layout based on the current layout manager and the current set of displayed GUI components.

```
1  // Fig. 14.43: GridLayoutFrame.java
2  // Demonstrating GridLayout.
3  import java.awt.GridLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class GridLayoutFrame extends JFrame implements ActionListener
11 {
12    private JButton[] buttons; // array of buttons
13    private static final String[] names =
14       { "one", "two", "three", "four", "five", "six" };
15    private boolean toggle = true; // toggle between two layouts
16    private Container container; // frame container
17    private GridLayout gridLayout1; // first gridlayout
18    private GridLayout gridLayout2; // second gridlayout
19
```

Fig. 14.43 | `GridLayout` containing six buttons. (Part 1 of 3.)

```
20      // no-argument constructor
21      public GridLayoutFrame()
22      {
23         super( "GridLayout Demo" );
24         gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 by 3; gaps of 5
25         gridLayout2 = new GridLayout( 3, 2 ); // 3 by 2; no gaps
26         container = getContentPane(); // get content pane
27         setLayout( gridLayout1 ); // set JFrame layout
28         buttons = new JButton[ names.length ]; // create array of JButtons
29
30         for ( int count = 0; count < names.length; count++ )
31         {
32            buttons[ count ] = new JButton( names[ count ] );
33            buttons[ count ].addActionListener( this ); // register listener
34            add( buttons[ count ] ); // add button to JFrame
35         } // end for
36      } // end GridLayoutFrame constructor
37
```

Custom `GridLayout`s: One with 2 rows, 3 columns and 5 pixels of gap space between components and the other with 3 rows, two columns and the default gap space.

Fig. 14.43 | `GridLayout` containing six buttons. (Part 2 of 3.)

```
38      // handle button events by toggling between layouts
39      public void actionPerformed( ActionEvent event )
40      {
41         if ( toggle )
42            container.setLayout( gridLayout2 ); // set layout to second
43         else
44            container.setLayout( gridLayout1 ); // set layout to first
45
46         toggle = !toggle; // set toggle to opposite value
47         container.validate(); // re-lay out container
48      } // end method actionPerformed
49   } // end class GridLayoutFrame
```

Changes the layout

Re-lays out the container.

Fig. 14.43 | `GridLayout` containing six buttons. (Part 3 of 3.)

```
1  // Fig. 14.44: GridLayoutDemo.java
2  // Testing GridLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class GridLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
10       gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       gridLayoutFrame.setSize( 300, 200 ); // set frame size
12       gridLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class GridLayoutDemo
```

Fig. 14.44 | Test class for `GridLayoutFrame`.

14.19 Using Panels to Manage More Complex Layouts

- Complex GUIs require that each component be placed in an exact location.
  - Often consist of multiple panels, with each panel's components arranged in a specific layout.
- Class `JPanel` extends `JComponent` and `JComponent` extends class `Container`, so every `JPanel` is a `Container`.
- Every `JPanel` may have components, including other panels, attached to it with `Container` method `add`.
- `JPanel` can be used to create a more complex layout in which several components are in a specific area of another container.

```
1  // Fig. 14.45: PanelFrame.java
2  // Using a JPanel to help lay out components.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import javax.swing.JButton;
8
9  public class PanelFrame extends JFrame
10 {
11    private JPanel buttonJPanel; // panel to hold buttons
12    private JButton[] buttons; // array of buttons
13
14    // no-argument constructor
15    public PanelFrame()
16    {
17       super( "Panel Demo" );
18       buttons = new JButton[ 5 ]; // create buttons array
19       buttonJPanel = new JPanel(); // set up panel
20       buttonJPanel.setLayout( new GridLayout( 1, buttons.length ) );
21
```

Fig. 14.45 | `JPanel` with five `JButtons` in a `GridLayout` attached to the `SOUTH` region of a `BorderLayout`. (Part 1 of 2.)

```
22          // create and add buttons
23          for ( int count = 0; count < buttons.length; count++ )
24          {
25             buttons[ count ] = new JButton( "Button " + ( count + 1 ) );
26             buttonJPanel.add( buttons[ count ] ); // add button to panel
27          } // end for
28
29          add( buttonJPanel, BorderLayout.SOUTH ); // add panel to JFrame
30       } // end PanelFrame constructor
31    } // end class PanelFrame
```

Places the `JPanel` with 5 buttons in the South region of the window's `BorderLayout`.

Fig. 14.45 | `JPanel` with five `JButtons` in a `GridLayout` attached to the `SOUTH` region of a `BorderLayout`. (Part 2 of 2.)

```
1  // Fig. 14.46: PanelDemo.java
2  // Testing PanelFrame.
3  import javax.swing.JFrame;
4
5  public class PanelDemo extends JFrame
6  {
7     public static void main( String[] args )
8     {
9        PanelFrame panelFrame = new PanelFrame();
10       panelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       panelFrame.setSize( 450, 200 ); // set frame size
12       panelFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class PanelDemo
```

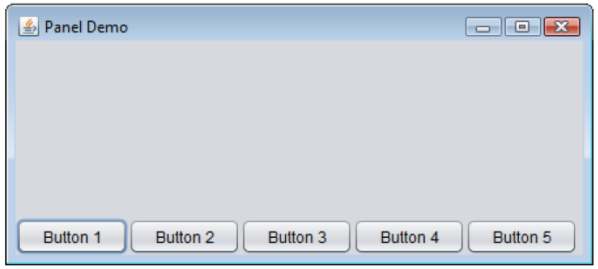


Fig. 14.46 | Test class for `PanelFrame`.

14.20 JTextArea

- A `JTextArea` provides an area for manipulating multiple lines of text.
- `JTextArea` is a subclass of `JTextComponent`, which declares common methods for `JTextField`s, `JTextArea`s and several other text-based GUI components.

```
1  // Fig. 14.47: TextAreaFrame.java
2  // Copying selected text from one textarea to another.
3  import java.awt.event.ActionListener;
4  import java.awt.event.ActionEvent;
5  import javax.swing.Box;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10
11 public class TextAreaFrame extends JFrame
12 {
13    private JTextArea textArea1; // displays demo string
14    private JTextArea textArea2; // highlighted text is copied here
15    private JButton copyJButton; // initiates copying of text
16
17    // no-argument constructor
18    public TextAreaFrame()
19    {
20       super( "TextArea Demo" );
21       Box box = Box.createHorizontalBox(); // create box
22       String demo = "This is a demo string to\n" +
23          "illustrate copying text\nfrom one textarea to \n" +
24          "another textarea using an\nexternal event\n";
```

Container that arranges components horizontally.

Fig. 14.47 | Copying selected text from one `JTextArea` to another. (Part 1 of 2.)

```
25
26          textArea1 = new JTextArea( demo, 10, 15 ); // create textarea1
27          box.add( new JScrollPane( textArea1 ) ); // add scrollpane
28
29          copyJButton = new JButton( "Copy >>>" ); // create copy button
30          box.add( copyJButton ); // add copy button to box
31          copyJButton.addActionListener(
32
33             new ActionListener() // anonymous inner class
34             {
35                // set text in textArea2 to selected text from textArea1
36                public void actionPerformed( ActionEvent event )
37                {
38                   textArea2.setText( textArea1.getSelectedText() );
39                } // end method actionPerformed
40             } // end anonymous inner class
41          ); // end call to addActionListener
42
43          textArea2 = new JTextArea( 10, 15 ); // create second textarea
44          textArea2.setEditable( false ); // disable editing
45          box.add( new JScrollPane( textArea2 ) ); // add scrollpane
46
47          add( box ); // add box to frame
48       } // end TextAreaFrame constructor
49    } // end class TextAreaFrame
```

Copies selected text into `textArea2`.

Fig. 14.47 | Copying selected text from one `JTextArea` to another. (Part 2 of 2.)

```
1  // Fig. 14.48: TextAreaDemo.java
2  // Copying selected text from one textarea to another.
3  import javax.swing.JFrame;
4
5  public class TextAreaDemo
6  {
7     public static void main( String[] args )
8     {
9        TextAreaFrame textAreaFrame = new TextAreaFrame();
10       textAreaFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textAreaFrame.setSize( 425, 200 ); // set frame size
12       textAreaFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class TextAreaDemo
```

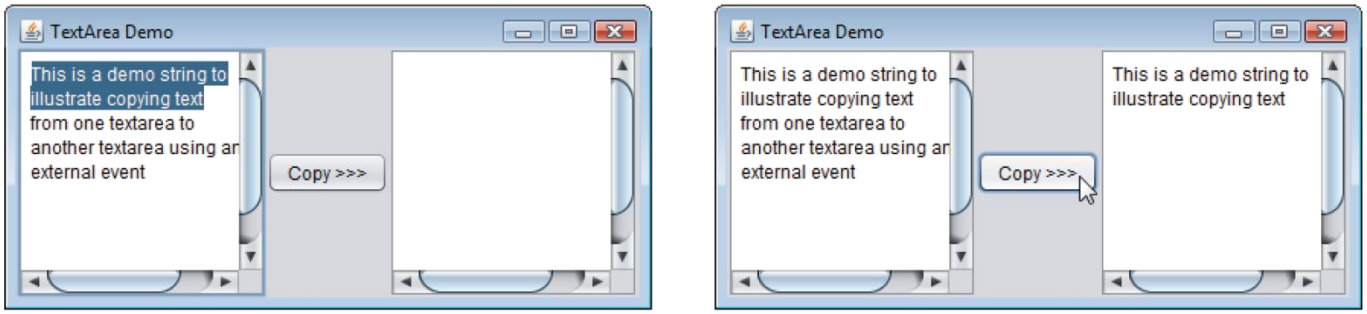


Fig. 14.48 | Test class for `TextAreaFrame`.

Look-and-Feel Observation 14.19

To provide line wrapping functionality for a `JTextArea`*, invoke* `JTextArea` *method* `setLineWrap` *with a* `true` *argument.*

14.20 JTextArea (cont.)

- Box is a subclass of Container that uses a BoxLayout to arrange the GUI components horizontally or vertically.
- Box static method createHorizontalBox creates a Box that arranges components left to right in the order that they are attached.

- JTextArea method getSelectedText (inherited from JTextComponent) returns the selected text from a JTextArea.
- JTextArea method setText changes the text in a JTextArea.
- When text reaches the right edge of a JTextArea the text can wrap to the next line.
  - Referred to as line wrapping.
  - By default, JTextArea does not wrap lines.

14.20 JTextArea (cont.)

- You can set the horizontal and vertical scrollbar policies of a `JScrollPane` when it's constructed.
- You can also use `JScrollPane` methods `setHorizontalScrollBarPolicy` and `setVerticalScrollBarPolicy` to change the scrollbar policies.

14.20 JTextArea (cont.)

- Class `JScrollPane` declares the constants
  - `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`
    `JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS`
  - to indicate that a scrollbar should always appear, constants
  - `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED`
    `JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED`
  - to indicate that a scrollbar should appear only if necessary (the defaults) and constants
  - `JScrollPane.VERTICAL_SCROLLBAR_NEVER`
    `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`
  - to indicate that a scrollbar should never appear.
- If policy is set to `HORIZONTAL_SCROLLBAR_NEVER`, a `JTextArea` attached to the `JScrollPane` will automatically wrap lines.

End of Part 2-1

- Readings
  - Chapter 14.

Part 2-2

25.1 Introduction

- In this chapter, we cover
  - Additional components and layout managers and lay the groundwork for building more complex GUIs.
  - Menus that enable the user to effectively perform tasks in the program.
  - Swing's pluggable look-and-feel (PLAF).
  - Multiple-document interface (MDI) — a main window (often called the parent window) containing other windows (often called child windows) to manage several open documents in parallel.

25.3 Windows: Additional Notes

- A `JFrame` *is a* window with a title bar and a border.
- `JFrame` is a subclass of `Frame`, which is a subclass of `Window`.
  - These are heavyweight Swing GUI components.
- A window is provided by the local platform's windowing toolkit.
- By default, when the user closes a `JFrame` window, it is hidden, but you can control this with `JFrame` method `setDefaultCloseOperation`.
  - Interface `WindowConstants` (package `javax.swing`), which class `JFrame` implements, declares three constants—`DISPOSE_ON_CLOSE`, `DO_NOTHING_ON_CLOSE` and `HIDE_ON_CLOSE` (the default) — for use with this method.

25.3 Windows: Additional Notes (cont.)

- Class `Window` (an indirect superclass of `JFrame`) declares method `dispose` to return a window's resources to the system.
  - When a `Window` is no longer needed in an application, you should explicitly dispose of it.
  - Can be done by calling the `Window`'s `dispose` method or by calling method `setDefaultCloseOperation` with the argument `WindowConstants.DISPOSE_ON_CLOSE`.
- A window is not displayed until the program invokes the window's `setVisible` method with a `true` argument.
- A window's size should be set with a call to method `setSize`.
- The position of a window when it appears on the screen is specified with method `setLocation`.

25.3 Windows: Additional Notes (cont.)

- When the user manipulates the window, window events occur.
- Event listeners are registered for window events with `window` method `addWindowListener`.
- Interface `WindowListener` provides seven window-event-handling methods
  - `windowActivated` (called when user makes a window the active window)
  - `windowClosed` (called after the window is closed)
  - `windowClosing` (called when the user initiates closing of the window)
  - `windowDeactivated` (called when the user makes another window the active window)
  - `windowDeiconified` (called when window is restored from minimized state)
  - `windowIconified` (called when window minimized)
  - `windowOpened` (called when window first displayed)

25.4 Using Menus with Frames

- Menus are an integral part of GUIs.
- Allow the user to perform actions without unnecessarily cluttering a GUI with extra components.
- In Swing GUIs, menus can be attached only to objects of the classes that provide method `setJMenuBar`.
  - Two such classes are `JFrame` and `JApplet`.
- The classes used to declare menus are `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem` and class `JRadioButtonMenuItem`.

25.4 Using Menus with Frames (cont.)

- Class `JMenuBar` (a subclass of `JComponent`) manages a menu bar, which is a container for menus.
- Class `JMenu` (a subclass of `javax.swing.JMenuItem`) — menus.
  - Menus contain menu items and are added to menu bars or to other menus as submenus.
- Class `JMenuItem` (a subclass of `javax.swing.AbstractButton`) — menu items.
  - A menu item causes an action event when clicked.
  - Can also be a submenu that provides more menu items from which the user can select.

25.4 Using Menus with Frames (cont.)

- Class `JCheckBoxMenuItem` (a subclass of `javax.swing.JMenuItem`) — menu items that can be toggled on or off.
- Class `JRadioButtonMenuItem` (a subclass of `javax.swing.JMenuItem`) — menu items that can be toggled on or off like `JCheckBoxMenuItem`s.
  - When multiple `JRadioButtonMenuItem`s are maintained as part of a `ButtonGroup`, only one item in the group can be selected at a given time.
- Mnemonics can provide quick access to a menu or menu item from the keyboard.
  - Can be used with all subclasses of `javax.swing.AbstractButton`.
- `JMenu` method `setMnemonic` (inherited from class `AbstractButton`) indicates the mnemonic for a menu.

```
1  // Fig. 25.5: MenuFrame.java
2  // Demonstrating menus.
3  import java.awt.Color;
4  import java.awt.Font;
5  import java.awt.BorderLayout;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.event.ItemListener;
9  import java.awt.event.ItemEvent;
10 import javax.swing.JFrame;
11 import javax.swing.JRadioButtonMenuItem;
12 import javax.swing.JCheckBoxMenuItem;
13 import javax.swing.JOptionPane;
14 import javax.swing.JLabel;
15 import javax.swing.SwingConstants;
16 import javax.swing.ButtonGroup;
17 import javax.swing.JMenu;
18 import javax.swing.JMenuItem;
19 import javax.swing.JMenuBar;
20
21 public class MenuFrame extends JFrame
22 {
23    private final Color[] colorValues =
24       { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };
```

Fig. 25.5 | JMenus and mnemonics. (Part 1 of 9.)

```
25      private JRadioButtonMenuItem[] colorItems; // color menu items
26      private JRadioButtonMenuItem[] fonts; // font menu items
27      private JCheckBoxMenuItem[] styleItems; // font style menu items
28      private JLabel displayJLabel; // displays sample text
29      private ButtonGroup fontButtonGroup; // manages font menu items
30      private ButtonGroup colorButtonGroup; // manages color menu items
31      private int style; // used to create style for font
32
33      // no-argument constructor set up GUI
34      public MenuFrame()
35      {
36         super( "Using JMenus" );
37
38         JMenu fileMenu = new JMenu( "File" ); // create file menu
39         fileMenu.setMnemonic( 'F' ); // set mnemonic to F
40
41         // create About... menu item
42         JMenuItem aboutItem = new JMenuItem( "About..." );
43         aboutItem.setMnemonic( 'A' ); // set mnemonic to A
44         fileMenu.add( aboutItem ); // add about item to file menu
45         aboutItem.addActionListener(
46
47            new ActionListener() // anonymous inner class
48            {
```

Fig. 25.5 | JMenus and mnemonics. (Part 2 of 9.)

```
49             // display message dialog when user selects About...
50             public void actionPerformed( ActionEvent event )
51             {
52                JOptionPane.showMessageDialog( MenuFrame.this,
53                   "This is an example\nof using menus",
54                   "About", JOptionPane.PLAIN_MESSAGE );
55             } // end method actionPerformed
56          } // end anonymous inner class
57       ); // end call to addActionListener
58
59       JMenuItem exitItem = new JMenuItem( "Exit" ); // create exit item
60       exitItem.setMnemonic( 'x' ); // set mnemonic to x
61       fileMenu.add( exitItem ); // add exit item to file menu
62       exitItem.addActionListener(
63
64          new ActionListener() // anonymous inner class
65          {
66             // terminate application when user clicks exitItem
67             public void actionPerformed( ActionEvent event )
68             {
69                System.exit( 0 ); // exit application
70             } // end method actionPerformed
71          } // end anonymous inner class
72       ); // end call to addActionListener
```

Fig. 25.5 | JMenus and mnemonics. (Part 3 of 9.)

```
73
74          JMenuBar bar = new JMenuBar(); // create menu bar
75          setJMenuBar( bar ); // add menu bar to application
76          bar.add( fileMenu ); // add file menu to menu bar
77
78          JMenu formatMenu = new JMenu( "Format" ); // create format menu
79          formatMenu.setMnemonic( 'r' ); // set mnemonic to r
80
81          // array listing string colors
82          String[] colors = { "Black", "Blue", "Red", "Green" };
83
84          JMenu colorMenu = new JMenu( "Color" ); // create color menu
85          colorMenu.setMnemonic( 'C' ); // set mnemonic to C
86
87          // create radio button menu items for colors
88          colorItems = new JRadioButtonMenuItem[ colors.length ];
89          colorButtonGroup = new ButtonGroup(); // manages colors
90          ItemHandler itemHandler = new ItemHandler(); // handler for colors
91
92          // create color radio button menu items
93          for ( int count = 0; count < colors.length; count++ )
94          {
95             colorItems[ count ] =
96                new JRadioButtonMenuItem( colors[ count ] ); // create item
```

Fig. 25.5 | JMenus and mnemonics. (Part 4 of 9.)

```
97             colorMenu.add( colorItems[ count ] ); // add item to color menu
98             colorButtonGroup.add( colorItems[ count ] ); // add to group
99             colorItems[ count ].addActionListener( itemHandler );
100         } // end for
101
102         colorItems[ 0 ].setSelected( true ); // select first Color item
103
104         formatMenu.add( colorMenu ); // add color menu to format menu
105         formatMenu.addSeparator(); // add separator in menu
106
107         // array listing font names
108         String[] fontNames = { "Serif", "Monospaced", "SansSerif" };
109         JMenu fontMenu = new JMenu( "Font" ); // create font menu
110         fontMenu.setMnemonic( 'n' ); // set mnemonic to n
111
112         // create radio button menu items for font names
113         fonts = new JRadioButtonMenuItem[ fontNames.length ];
114         fontButtonGroup = new ButtonGroup(); // manages font names
115
116         // create Font radio button menu items
117         for ( int count = 0; count < fonts.length; count++ )
118         {
119             fonts[ count ] = new JRadioButtonMenuItem( fontNames[ count ] );
120             fontMenu.add( fonts[ count ] ); // add font to font menu
```

Fig. 25.5 | JMenus and mnemonics. (Part 5 of 9.)

```
121            fontButtonGroup.add( fonts[ count ] ); // add to button group
122            fonts[ count ].addActionListener( itemHandler ); // add handler
123         } // end for
124
125         fonts[ 0 ].setSelected( true ); // select first Font menu item
126         fontMenu.addSeparator(); // add separator bar to font menu
127
128         String[] styleNames = { "Bold", "Italic" }; // names of styles
129         styleItems = new JCheckBoxMenuItem[ styleNames.length ];
130         StyleHandler styleHandler = new StyleHandler(); // style handler
131
132         // create style checkbox menu items
133         for ( int count = 0; count < styleNames.length; count++ )
134         {
135            styleItems[ count ] =
136               new JCheckBoxMenuItem( styleNames[ count ] ); // for style
137            fontMenu.add( styleItems[ count ] ); // add to font menu
138            styleItems[ count ].addItemListener( styleHandler ); // handler
139         } // end for
140
141         formatMenu.add( fontMenu ); // add Font menu to Format menu
142         bar.add( formatMenu ); // add Format menu to menu bar
143
```

Fig. 25.5 | JMenus and mnemonics. (Part 6 of 9.)

```
144         // set up label to display text
145         displayJLabel = new JLabel( "Sample Text", SwingConstants.CENTER );
146         displayJLabel.setForeground( colorValues[ 0 ] );
147         displayJLabel.setFont( new Font( "Serif", Font.PLAIN, 72 ) );
148
149         getContentPane().setBackground( Color.CYAN ); // set background
150         add( displayJLabel, BorderLayout.CENTER ); // add displayJLabel
151      } // end MenuFrame constructor
152
153      // inner class to handle action events from menu items
154      private class ItemHandler implements ActionListener
155      {
156         // process color and font selections
157         public void actionPerformed( ActionEvent event )
158         {
159            // process color selection
160            for ( int count = 0; count < colorItems.length; count++ )
161            {
162               if ( colorItems[ count ].isSelected() )
163               {
164                  displayJLabel.setForeground( colorValues[ count ] );
165                  break;
166               } // end if
167            } // end for
```

Fig. 25.5 | `JMenus` and mnemonics. (Part 7 of 9.)

```
168
169          // process font selection
170          for ( int count = 0; count < fonts.length; count++ )
171          {
172             if ( event.getSource() == fonts[ count ] )
173             {
174                displayJLabel.setFont(
175                   new Font( fonts[ count ].getText(), style, 72 ) );
176             } // end if
177          } // end for
178
179          repaint(); // redraw application
180       } // end method actionPerformed
181    } // end class ItemHandler
182
183    // inner class to handle item events from checkbox menu items
184    private class StyleHandler implements ItemListener
185    {
186       // process font style selections
187       public void itemStateChanged( ItemEvent e )
188       {
189          String name = displayJLabel.getFont().getName(); // current Font
190          Font font; // new font based on user selections
191
```

Fig. 25.5 | `JMenus` and mnemonics. (Part 8 of 9.)

```
192            // determine which items are checked and create Font
193            if ( styleItems[ 0 ].isSelected() &&
194                 styleItems[ 1 ].isSelected() )
195               font = new Font( name, Font.BOLD + Font.ITALIC, 72 );
196            else if ( styleItems[ 0 ].isSelected() )
197               font = new Font( name, Font.BOLD, 72 );
198            else if ( styleItems[ 1 ].isSelected() )
199               font = new Font( name, Font.ITALIC, 72 );
200            else
201               font = new Font( name, Font.PLAIN, 72 );
202
203            displayJLabel.setFont( font );
204            repaint(); // redraw application
205         } // end method itemStateChanged
206      } // end class StyleHandler
207   } // end class MenuFrame
```

Fig. 25.5 | JMenus and mnemonics. (Part 9 of 9.)

```
1  // Fig. 25.6: MenuTest.java
2  // Testing MenuFrame.
3  import javax.swing.JFrame;
4
5  public class MenuTest
6  {
7     public static void main( String[] args )
8     {
9        MenuFrame menuFrame = new MenuFrame(); // create MenuFrame
10       menuFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       menuFrame.setSize( 500, 200 ); // set frame size
12       menuFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class MenuTest
```

Fig. 25.6 | Test class for `MenuFrame`. (Part 1 of 2.)

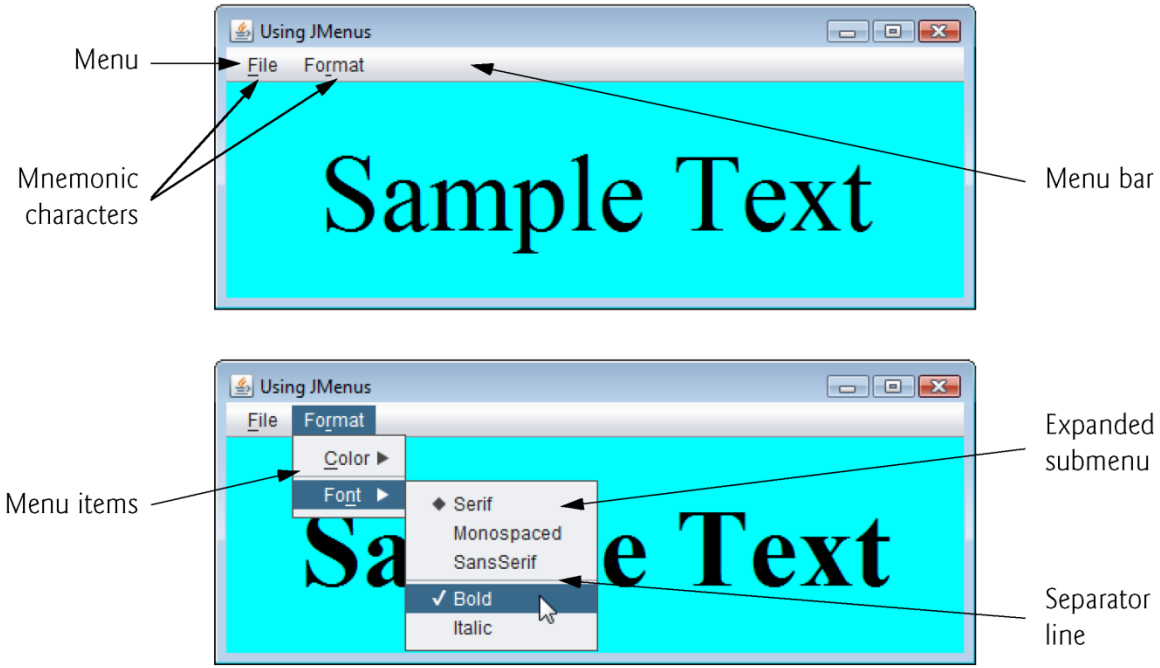


Fig. 25.6 | Test class for `MenuFrame`. (Part 2 of 2.)

25.4 Using Menus with Frames (cont.)

- In most prior uses of `showMessageDialog`, the first argument was `null`.
  - The first argument specifies the parent window that helps determine where the dialog box will be displayed.
  - If `null`, the dialog box appears in the center of the screen.
  - Otherwise, it appears centered over the specified parent window.
- When using the `this` reference in an inner class, specifying `this` by itself refers to the inner-class object.
  - To reference the outer-class object's `this` reference, qualify `this` with the outer-class name and a dot (`.`).

25.4 Using Menus with Frames (cont.)

- Dialog boxes are typically modal — does not allow any other window in the application to be accessed until the dialog box is dismissed.
- Class `JDialog` can be used to create your own modal or nonmodal dialogs.
- `JMenuBar` method `add` attaches a menu to a `JMenuBar`.
- `AbstractButton` method `setSelected` selects the specified button.
- `JMenu` method `addSeparator` adds a horizontal separator line to a menu.
- `AbstractButton` method `isSelected` determines if a button is selected.

Common Programming Error 25.3

Forgetting to set the menu bar with `JFrame` *method* `setJMenuBar` *prevents the menu bar from displaying in the* `JFrame`.

Look-and-Feel Observation 25.4

Menus appear left to right in the order that they are added to a `JMenuBar`.

Look-and-Feel Observation 25.5

A submenu is created by adding a menu as a menu item in another menu. When the mouse is positioned over a submenu (or the submenu's mnemonic is pressed), the submenu expands to show its menu items.

Look-and-Feel Observation 25.7

Any lightweight GUI component (i.e., a component that is a subclass of `JComponent`*) can be added to a* `JMenu` *or to a* `JMenuBar`.

25.5 JPopupMenu

- Context-sensitive pop-up menus are created with class `JPopupMenu` (a subclass of `JComponent`).
  - Provide options that are specific to the component for which the popup trigger event was generated — typically occurs when the user presses and releases the right mouse button.

- `MouseEvent` method `isPopupTrigger` returns `true` if the popup trigger event occurred

- `JPopupMenu` method `show` displays a `JPopupMenu`.
  - The first argument specifies the origin component — helps determine where the `JPopupMenu` will appear on the screen.
  - The last two arguments are the *x*-*y* coordinates (measured from the origin component's upper-left corner) at which the `JPopupMenu` is to appear.

Look-and-Feel Observation 25.8

The pop-up trigger event is platform specific. On most platforms that use a mouse with multiple buttons, the pop-up trigger event occurs when the user clicks the right mouse button on a component that supports a pop-up menu.

```
1  // Fig. 25.7: PopupFrame.java
2  // Demonstrating JPopupMenus.
3  import java.awt.Color;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JRadioButtonMenuItem;
10 import javax.swing.JPopupMenu;
11 import javax.swing.ButtonGroup;
12
13 public class PopupFrame extends JFrame
14 {
15    private JRadioButtonMenuItem[] items; // holds items for colors
16    private final Color[] colorValues =
17       { Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used
18    private JPopupMenu popupMenu; // allows user to select color
19
20    // no-argument constructor sets up GUI
21    public PopupFrame()
22    {
23       super( "Using JPopupMenus" );
24
```

Fig. 25.7 | JPopupMenu for selecting colors. (Part 1 of 4.)

```
25          ItemHandler handler = new ItemHandler(); // handler for menu items
26          String[] colors = { "Blue", "Yellow", "Red" }; // array of colors
27
28          ButtonGroup colorGroup = new ButtonGroup(); // manages color items
29          popupMenu = new JPopupMenu(); // create pop-up menu
30          items = new JRadioButtonMenuItem[ colors.length ]; // color items
31
32          // construct menu item, add to pop-up menu, enable event handling
33          for ( int count = 0; count < items.length; count++ )
34          {
35             items[ count ] = new JRadioButtonMenuItem( colors[ count ] );
36             popupMenu.add( items[ count ] ); // add item to pop-up menu
37             colorGroup.add( items[ count ] ); // add item to button group
38             items[ count ].addActionListener( handler ); // add handler
39          } // end for
40
41          setBackground( Color.WHITE ); // set background to white
42
43          // declare a MouseListener for the window to display pop-up menu
44          addMouseListener(
45
46             new MouseAdapter() // anonymous inner class
47             {
```

Fig. 25.7 | `JPopupMenu` for selecting colors. (Part 2 of 4.)

```
48             // handle mouse press event
49             public void mousePressed( MouseEvent event )
50             {
51                checkForTriggerEvent( event ); // check for trigger
52             } // end method mousePressed
53
54             // handle mouse release event
55             public void mouseReleased( MouseEvent event )
56             {
57                checkForTriggerEvent( event ); // check for trigger
58             } // end method mouseReleased
59
60             // determine whether event should trigger pop-up menu
61             private void checkForTriggerEvent( MouseEvent event )
62             {
63                if ( event.isPopupTrigger() )
64                   popupMenu.show(
65                      event.getComponent(), event.getX(), event.getY() );
66             } // end method checkForTriggerEvent
67          } // end anonymous inner class
68       ); // end call to addMouseListener
69    } // end PopupFrame constructor
70
```

Fig. 25.7 | `JPopupMenu` for selecting colors. (Part 3 of 4.)

```
71      // private inner class to handle menu item events
72      private class ItemHandler implements ActionListener
73      {
74         // process menu item selections
75         public void actionPerformed( ActionEvent event )
76         {
77            // determine which menu item was selected
78            for ( int i = 0; i < items.length; i++ )
79            {
80               if ( event.getSource() == items[ i ] )
81               {
82                  getContentPane().setBackground( colorValues[ i ] );
83                  return;
84               } // end if
85            } // end for
86         } // end method actionPerformed
87      } // end private inner class ItemHandler
88   } // end class PopupFrame
```

Fig. 25.7 | `JPopupMenu` for selecting colors. (Part 4 of 4.)

```
1  // Fig. 25.8: PopupTest.java
2  // Testing PopupFrame.
3  import javax.swing.JFrame;
4
5  public class PopupTest
6  {
7     public static void main( String[] args )
8     {
9        PopupFrame popupFrame = new PopupFrame(); // create PopupFrame
10       popupFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       popupFrame.setSize( 300, 200 ); // set frame size
12       popupFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class PopupTest
```

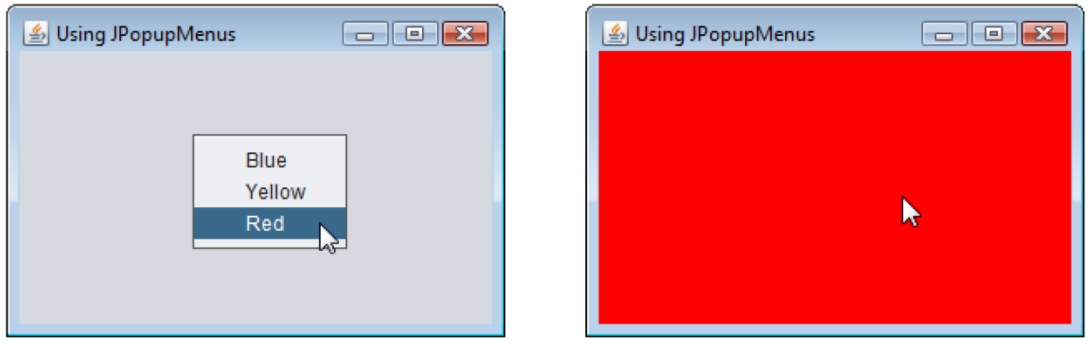


Fig. 25.8 | Test class for PopupFrame.

25.7 JDesktopPane and JInternalFrame

- Multiple-document interface (MDI)
  - a main window (called the parent window) containing other windows (called child windows), to manage several open documents that are being processed in parallel.
- Swing's `JDesktopPane` and `JInternalFrame` classes implement multiple-document interfaces.

```
1  // Fig. 25.11: DesktopFrame.java
2  // Demonstrating JDesktopPane.
3  import java.awt.BorderLayout;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.util.Random;
9  import javax.swing.JFrame;
10 import javax.swing.JDesktopPane;
11 import javax.swing.JMenuBar;
12 import javax.swing.JMenu;
13 import javax.swing.JMenuItem;
14 import javax.swing.JInternalFrame;
15 import javax.swing.JPanel;
16 import javax.swing.ImageIcon;
17
18 public class DesktopFrame extends JFrame
19 {
20    private JDesktopPane theDesktop;
21
22    // set up GUI
23    public DesktopFrame()
24    {
```

Fig. 25.11 | Multiple-document interface. (Part 1 of 4.)

```
25          super( "Using a JDesktopPane" );
26
27          JMenuBar bar = new JMenuBar(); // create menu bar
28          JMenu addMenu = new JMenu( "Add" ); // create Add menu
29          JMenuItem newFrame = new JMenuItem( "Internal Frame" );
30
31          addMenu.add( newFrame ); // add new frame item to Add menu
32          bar.add( addMenu ); // add Add menu to menu bar
33          setJMenuBar( bar ); // set menu bar for this application
34
35          theDesktop = new JDesktopPane(); // create desktop pane
36          add( theDesktop ); // add desktop pane to frame
37
38          // set up listener for newFrame menu item
39          newFrame.addActionListener(
40
41             new ActionListener() // anonymous inner class
42             {
43                // display new internal window
44                public void actionPerformed( ActionEvent event )
45                {
46                   // create internal frame
47                   JInternalFrame frame = new JInternalFrame(
48                      "Internal Frame", true, true, true, true );
```

Fig. 25.11 | Multiple-document interface. (Part 2 of 4.)

```
49
50                MyJPanel panel = new MyJPanel(); // create new panel
51                frame.add( panel, BorderLayout.CENTER ); // add panel
52                frame.pack(); // set internal frame to size of contents
53
54                theDesktop.add( frame ); // attach internal frame
55                frame.setVisible( true ); // show internal frame
56             } // end method actionPerformed
57          } // end anonymous inner class
58       ); // end call to addActionListener
59    } // end DesktopFrame constructor
60 } // end class DesktopFrame
61
62 // class to display an ImageIcon on a panel
63 class MyJPanel extends JPanel
64 {
65    private static Random generator = new Random();
66    private ImageIcon picture; // image to be displayed
67    private final static String[] images = { "yellowflowers.png",
68       "purpleflowers.png", "redflowers.png", "redflowers2.png",
69       "lavenderflowers.png" };
70
```

Fig. 25.11 | Multiple-document interface. (Part 3 of 4.)

```
71       // load image
72       public MyJPanel()
73       {
74          int randomNumber = generator.nextInt( images.length );
75          picture = new ImageIcon( images[ randomNumber ] ); // set icon
76       } // end MyJPanel constructor
77
78       // display imageIcon on panel
79       public void paintComponent( Graphics g )
80       {
81          super.paintComponent( g );
82          picture.paintIcon( this, g, 0, 0 ); // display icon
83       } // end method paintComponent
84
85       // return image dimensions
86       public Dimension getPreferredSize()
87       {
88          return new Dimension( picture.getIconWidth(),
89             picture.getIconHeight() );
90       } // end method getPreferredSize
91    } // end class MyJPanel
```

Fig. 25.11 | Multiple-document interface. (Part 4 of 4.)

```
1  // Fig. 25.12: DesktopTest.java
2  // Demonstrating JDesktopPane.
3  import javax.swing.JFrame;
4
5  public class DesktopTest
6  {
7     public static void main( String[] args )
8     {
9        DesktopFrame desktopFrame = new DesktopFrame();
10       desktopFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       desktopFrame.setSize( 600, 480 ); // set frame size
12       desktopFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class DesktopTest
```

Fig. 25.12 | Test class for `DeskTopFrame`. (Part 1 of 3.)

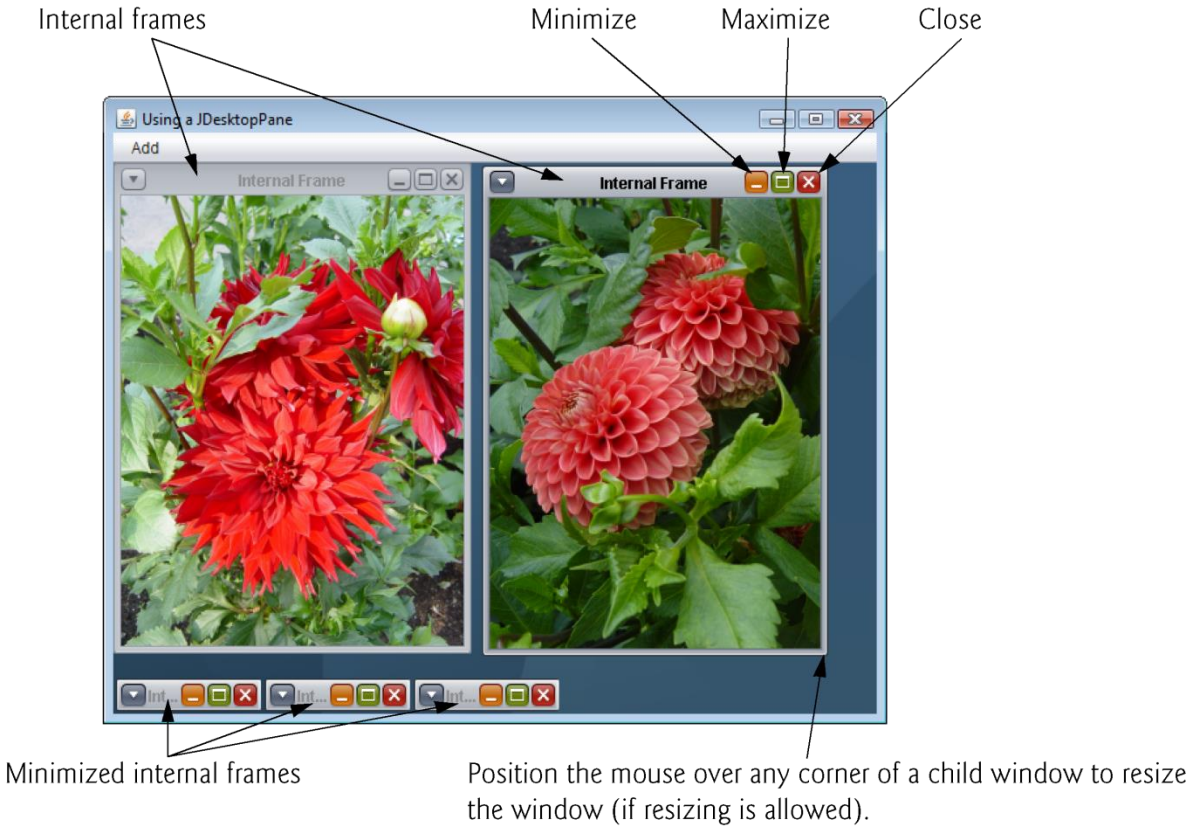


Fig. 25.12 | Test class for `DeskTopFrame`. (Part 2 of 3.)

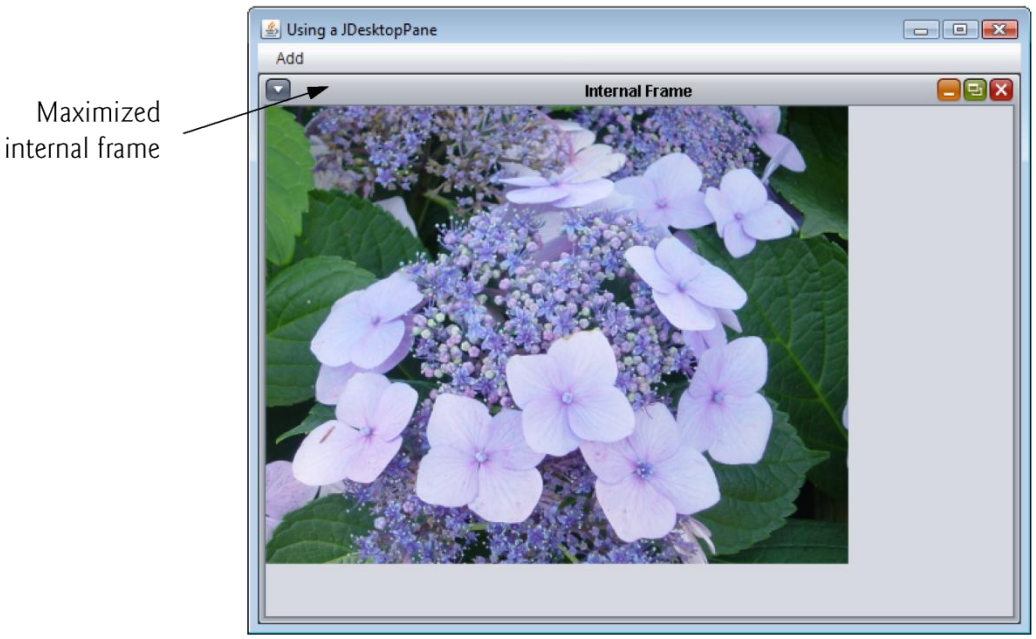


Fig. 25.12 | Test class for `DeskTopFrame`. (Part 3 of 3.)

25.7 JDesktopPane and JInternalFrame (cont.)

- The `JInternalFrame` constructor used here takes five arguments
  - a `String` for the title bar of the internal window
  - a `boolean` indicating whether the internal frame can be resized by the user
  - a `boolean` indicating whether the internal frame can be closed by the user
  - a `boolean` indicating whether the internal frame can be maximized by the user
  - a `boolean` indicating whether the internal frame can be minimized by the user.
- For each of the `boolean` arguments, a `true` value indicates that the operation should be allowed (as is the case here).

25.7 JDesktopPane and JInternalFrame (cont.)

- A `JInternalFrame` has a content pane to which GUI components can be attached.
- `JInternalFrame` method `pack` sets the size of the child window.
  - Uses the preferred sizes of the components to determine the window's size.
- Classes `JInternalFrame` and `JDesktopPane` provide many methods for managing child windows.

25.8 JTabbedPane

- A `JTabbedPane` arranges GUI components into layers, of which only one is visible at a time.
- Users access each layer by clicking a tab.
- The tabs appear at the top by default but also can be positioned at the left, right or bottom of the `JTabbedPane`.
- Any component can be placed on a tab.
  - If the component is a container, such as a panel, it can use any layout manager to lay out several components on the tab.
- Class `JTabbedPane` is a subclass of `JComponent`.

```
1  // Fig. 25.13: JTabbedPaneFrame.java
2  // Demonstrating JTabbedPane.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JTabbedPane;
7  import javax.swing.JLabel;
8  import javax.swing.JPanel;
9  import javax.swing.JButton;
10 import javax.swing.SwingConstants;
11
12 public class JTabbedPaneFrame extends JFrame
13 {
14    // set up GUI
15    public JTabbedPaneFrame()
16    {
17       super( "JTabbedPane Demo " );
18
19       JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane
20
21       // set up pane11 and add it to JTabbedPane
22       JLabel label1 = new JLabel( "panel one", SwingConstants.CENTER );
23       JPanel panel1 = new JPanel(); // create first panel
```

Fig. 25.13 | JTabbedPane used to organize GUI components. (Part 1 of 2.)

```
24          panel1.add( label1 ); // add label to panel
25          tabbedPane.addTab( "Tab One", null, panel1, "First Panel" );
26
27          // set up panel2 and add it to JTabbedPane
28          JLabel label2 = new JLabel( "panel two", SwingConstants.CENTER );
29          JPanel panel2 = new JPanel(); // create second panel
30          panel2.setBackground( Color.YELLOW ); // set background to yellow
31          panel2.add( label2 ); // add label to panel
32          tabbedPane.addTab( "Tab Two", null, panel2, "Second Panel" );
33
34          // set up panel3 and add it to JTabbedPane
35          JLabel label3 = new JLabel( "panel three" );
36          JPanel panel3 = new JPanel(); // create third panel
37          panel3.setLayout( new BorderLayout() ); // use borderlayout
38          panel3.add( new JButton( "North" ), BorderLayout.NORTH );
39          panel3.add( new JButton( "West" ), BorderLayout.WEST );
40          panel3.add( new JButton( "East" ), BorderLayout.EAST );
41          panel3.add( new JButton( "South" ), BorderLayout.SOUTH );
42          panel3.add( label3, BorderLayout.CENTER );
43          tabbedPane.addTab( "Tab Three", null, panel3, "Third Panel" );
44
45          add( tabbedPane ); // add JTabbedPane to frame
46       } // end JTabbedPaneFrame constructor
47    } // end class JTabbedPaneFrame
```

Fig. 25.13 | `JTabbedPane` used to organize GUI components. (Part 2 of 2.)

```
1  // Fig. 25.14: JTabbedPaneDemo.java
2  // Demonstrating JTabbedPane.
3  import javax.swing.JFrame;
4
5  public class JTabbedPaneDemo
6  {
7     public static void main( String[] args )
8     {
9        JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
10       tabbedPaneFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       tabbedPaneFrame.setSize( 250, 200 ); // set frame size
12       tabbedPaneFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class JTabbedPaneDemo
```

Fig. 25.14 | Test class for `JTabbedPaneFrame`. (Part 1 of 2.)

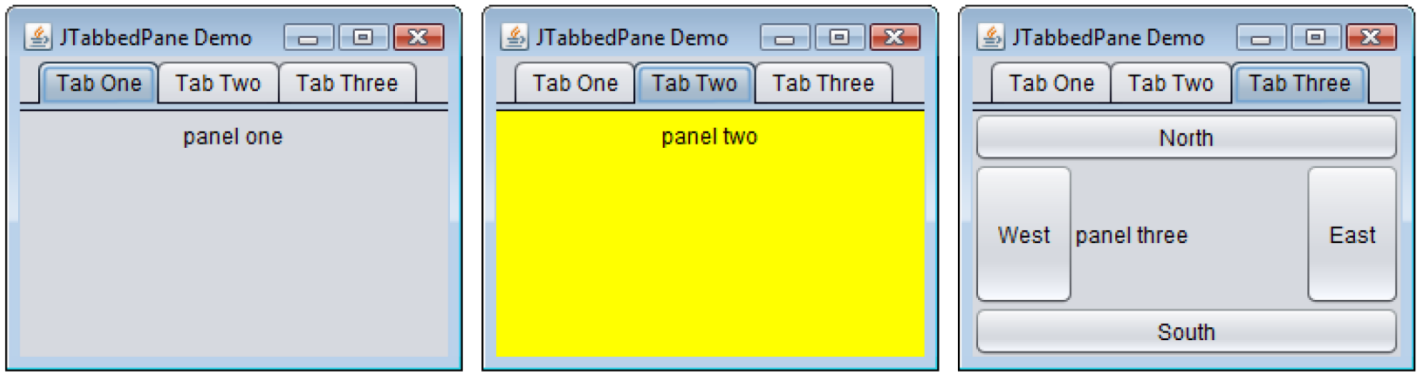


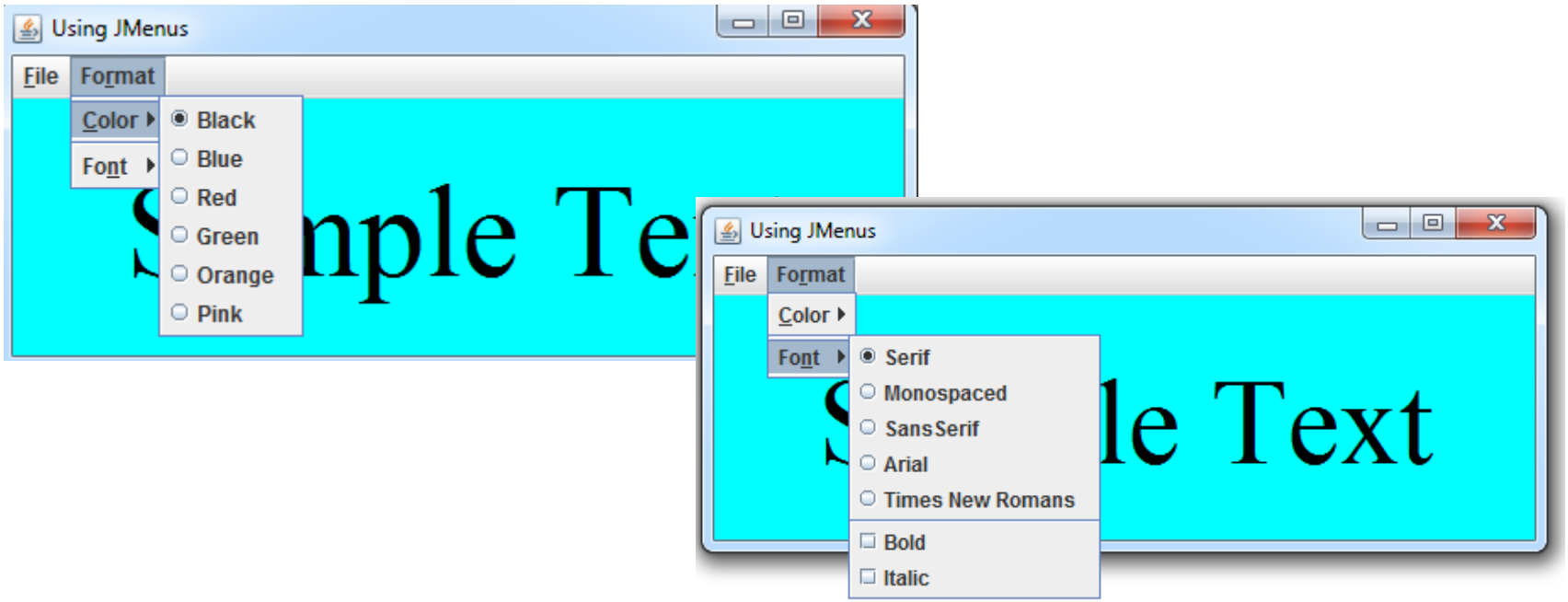
Fig. 25.14 | Test class for `JTabbedPaneFrame`. (Part 2 of 2.)

25.9 JTabbedPane (cont.)

- `JTabbedPane` method `addTab` adds a new tab. In the version with four arguments:
  - The first is a `String` that specifies the title of the tab.
  - The second is an `Icon` reference that specifies an icon to display on the tab — can be `null`
  - The third is a `Component` to display when the user clicks the tab.
  - The last is a `String` that specifies the tab's tool tip.

Exercise 1

- Implement the code for the following:



- Add two more colors
- Add two more fonts: Arial and Times New Roman

End of Class