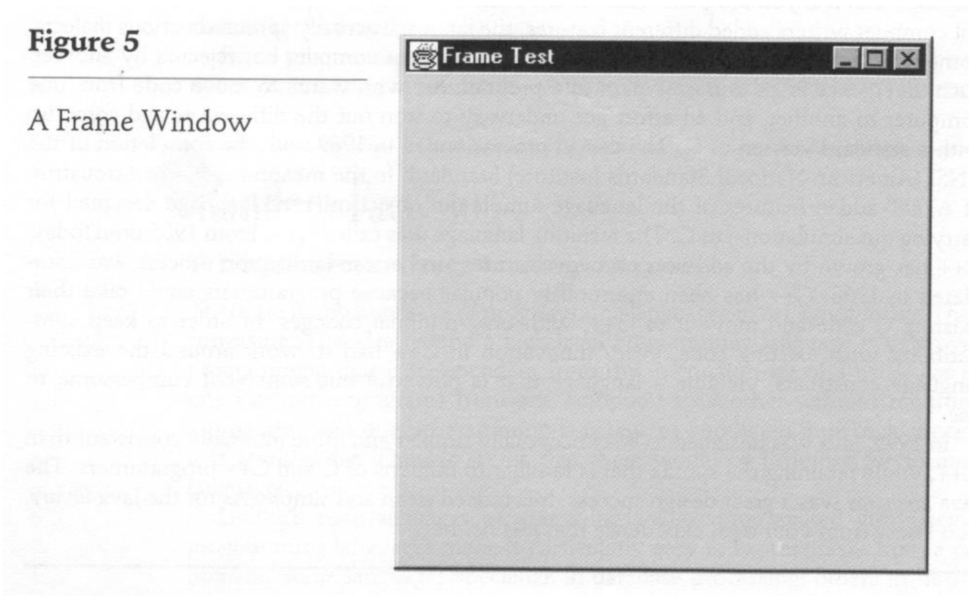
Graphical applications

- Graphical applications put up one or more frame windows, usually filled with user interface components such as buttons, text input fields, menus, etc.
- Graphical applications can display both text and graphical shapes and images.
- To show a frame, you can use the `JFrame` class in the `javax.swing` package.
- Before the creation of Swing, Java used components in the AWT (Abstract Window Toolkit) for graphical applications.
- Both AWT and Swing components are multiplatform – i.e. programs can run on Windows, the Macintosh, UNIX and other platforms without modification.
- The Java designers called this "*write once, run anywhere*".
- AWT uses the native user interface elements (buttons, textfields, menus, etc.) of the host platform. The programmers soon began complaining about "*write once, debug everywhere*".
- Swing takes a different approach – it paints the shapes for buttons, textfields, menus, and so on. That is slower but more consistent.
- The Swing classes are placed in the `javax.swing` package, where the `javax` package name denotes a standard extension of Java. For compatibility reasons, the package name was not changed from `javax` to `java`.
- Swing uses other parts of the AWT, such as the parts for drawing graphical shapes and handling events, so we shall continue to use some classes from the `java.awt` package.

- By default, frames have a rather useless size of 0x0 pixels. To define the size of the frame, use the `setSize()` method.
- To define the title of the frame, use the `setTitle()` method
- To display a frame, call the `setVisible()` method

Example: to show an empty frame



```
import javax.swing.JFrame;
public class FrameTest1
{
   public static void main(String[] args)
   {
      EmptyFrame frame = new EmptyFrame();
      frame.setTitle("Frame Test");
      frame.setVisible(true);
   }
}

class EmptyFrame extends JFrame
{
   public EmptyFrame()
   {
      final int DEFAULT_WIDTH = 300;
      final int DEFAULT_HEIGHT = 300;
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

Event Handling

- In the console applications we have seen so far, user input was under control of the program. The program asked the user for input in a specific order.
- In a program with a modern graphical user interface, the user can use both the mouse and the keyboard to enter information in any desired order. For example, the user can enter information into text fields, pull down menus, click buttons, drag scroll bars, and close windows in any order.
- The program must react to the user commands, in whatever order they arrive.

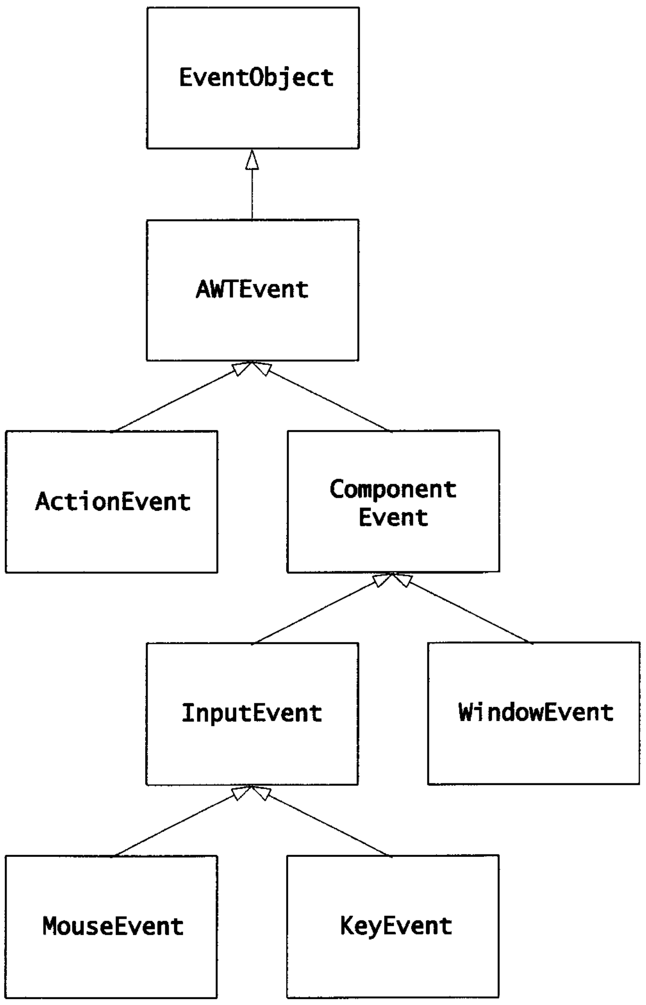
Events, Event Listeners, and Event Sources

- Whenever the user of a graphical program types in some characters, clicks on a button, or uses the mouse anywhere inside one of the windows of the program, the Java window manager sends a notification to the program that an *event* has occurred.
- A program can install *event listener* objects to capture the events that it likes to receive.
- To install a listener, you need to know the event source. The event source is the user interface component that generates a particular event. For example, a button is the event source for button click events.

- All event classes are subclasses of the `EventObject` class.
- The `EventObject` class has a method `getSource()` that returns the object that generates this event.

Figure 1

Event Classes



- In general, most GUI applications need to handle 3 types of events `ActionEvent`, `WindowEvent`, and `MouseEvent`.

7 kinds of window events can occur in a Java program.

1. A window has just opened for the first time.
2. A window has just closed as a result of the `dispose` method.
3. A window has just been *activated* (typically because the user clicked inside it).
4. A window has just been *deactivated* (typically because the user clicked inside another window).
5. A window is being *iconified* (typically because the user clicked on the "minimize" icon in the title bar).
6. A window is being *deiconified* (typically because the user clicked on the icon of the minimized window).
7. A window is being closed by the user (typically because the user clicked on the "close" icon in the title bar).

The `WindowListener` interface has 7 methods, and there is a convenience class called `WindowAdapter` defined in the `java.awt.event` package with all 7 methods implemented to do nothing.

```
public interface WindowListener
{
   void windowOpened(WindowEvent e);
   void windowClosed(WindowEvent e);
   void windowActivated(WindowEvent e);
   void windowDeactivated(WindowEvent e);
   void windowIconified(WindowEvent e);
   void windowDeinconified(WindowEvent e);
   void windowClosing(WindowEvent e);
}

public class WindowAdapter implements WindowListener
{
   public void windowOpened(WindowEvent e) { };
   public void windowClosed(WindowEvent e) { };
   public void windowActivated(WindowEvent e) { };
   public void windowDeactivated(WindowEvent e) { };
   public void windowIconified(WindowEvent e) { };
   public void windowDeinconified(WindowEvent e) { };
   public void windowClosing(WindowEvent e) { };
}
```

Consider our previous example that shows an empty frame.

- Graphical applications are multi-threaded programs.
- Once the `main` method shows a frame window, the program starts a new thread of execution that displays the graphical user interface.
- When the `main` method finishes (the main thread is completed), the Java program does not terminate because the user interface (event dispatcher) thread is still running.
- Even when you close the frame window (by clicking on the "close" icon in the title bar), the program keeps alive.
- To terminate the program, you must execute the statement `System.exit(0)`

```
public class FrameTest1
{
   public static void main(String[] args)
   {
      EmptyFrame frame = new EmptyFrame();
      frame.setTitle("Frame Test");
      frame.setVisible(true);
      System.exit(0); // Incorrect
   }
}
```

- If we put the `System.exit(0)` statement in the `main` method, the window will be shown for a brief moment, then the program exits immediately.
- We want to exit the program when the user clicks the "close" icon in the title bar. To do that, we must listen to window events and implement the `WindowListener` interface.

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

public class FrameTest2
{
   public static void main(String[] args)
   {
      EmptyFrame frame = new EmptyFrame();
      frame.setTitle("Close me!");
      frame.setVisible(true);
   }
}

class EmptyFrame extends JFrame
{
   public EmptyFrame()
   {
      final int DEFAULT_WIDTH = 300;
      final int DEFAULT_HEIGHT = 300;
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      WindowCloser listener = new WindowCloser();
      addWindowListener(listener);
   }

   //inner class
   private class WindowCloser extends WindowAdaptor
   {
      public void windowClosing(WindowEvent e)
      {
         System.exit(0);
      }
   }
}
```

- The window listener is defined as an inner class.
- Methods of the inner class can access the private variables of the outer class.

There is a simpler way to specify the handling of `windowClosing` in Java 5:

`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`

Structure of a Swing Frame

- The surface of a Swing frame is covered with 4 panes.
  - ➢ **The content pane holds the components that you want to display in the window**.
  - ➢ The glass pane is to capture mouse events.
  - ➢ The layered pane holds the menu bar and the content pane together.
  - ➢ The root pane holds the glass pane and the layered pane together.
  - ➢ The glass pane, layered pane and root pane are of no interest to most Java programmers.

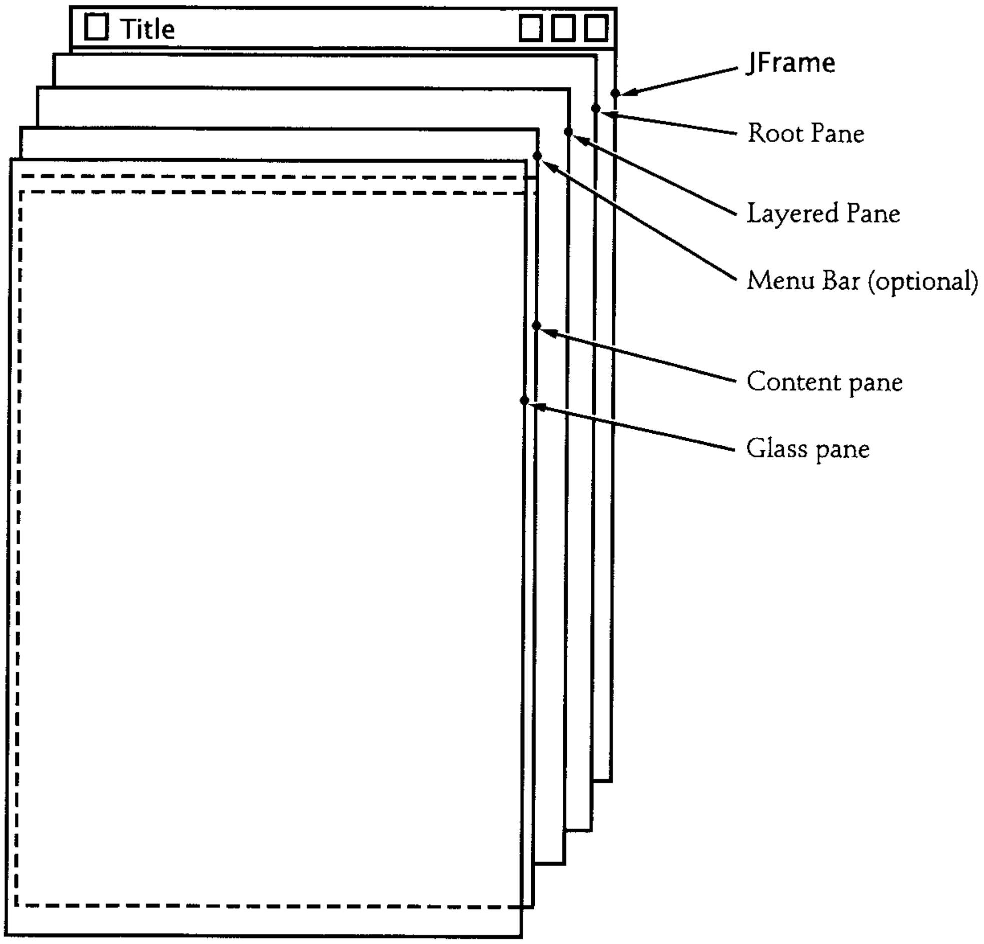


Figure 6

Anatomy of a Swing Frame

- You should not directly draw onto the surface of a frame. This will interferes with the display of the user interface components.
- The Swing user interface toolkit provides a special component, called `JPanel`, to hold the graphics components.
- You add a `JPanel` to the content pane and draws on the `JPanel`.
- To draw on a `JPanel`, you override the `paintComponent()` method (inherited from `JComponent`).
- When implementing your `paintComponent()` method, you must call the `paintComponent()` method of the superclass. This gives the superclass method a chance to erase the old contents of the panel.

```
public class MyPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D)g;

      // your drawing instructions go here
      ...
   }
}
```

- The `paintComponent()` method receives an object of type `Graphics`. The `Graphics` object stores the *graphics state*: the current color, font, and so on, that are used for the drawing operations.
- The `Graphics` class was included with the first version of Java. It is suitable for very basic drawings, but it does not use an object-oriented approach.
- Later, the designers of Java created the `Graphics2D` class. They did not want to inconvenience those programmers who had produced programs that used simple graphics, so they did not change the `paintComponent()` method. Instead, they made the `Graphics2D` class extends the `Graphics` class.
- The `Graphics` object is type-casted to `Graphics2D` in the `paintComponent()` method.

- You can use the draw method of the `Graphics2D` class to draw shapes such as rectangles, ellipses, line segments, polygons, and arcs.
- To refresh the display, call the `repaint()` method (inherited from `awt.Component`)
- In the older versions of Java, to add the panel to a `JFrame`, you must first get a reference to the content pane object by calling the `getContentPane()` method. This method returns a reference of type `Container`.
- A container is a window object that can contain other components. You use the add method of the `Container` to add your component.

```
class MyFrame extends JFrame
{
   public MyFrame()
   {
      MyPanel panel = new MyPanel();
      Container contentPane = getContentPane();
      contentPane.add(panel, "Center");
      ...
   }
   ...
}
```

- Starting from Java 5, you can simply add a GUI component to the JFrame directly.

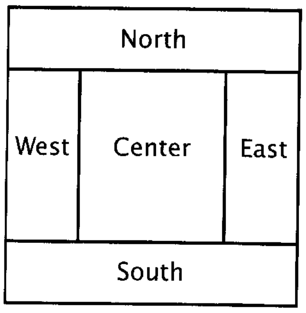
```
class MyFrame extends JFrame
{
   public MyFrame()
   {
      MyPanel panel = new MyPanel();

      add(panel, "Center");
      ...

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

Layout Management

- The content pane uses a *border layout* to arrange its components.

Figure 7

Component Areas of a Border Layout



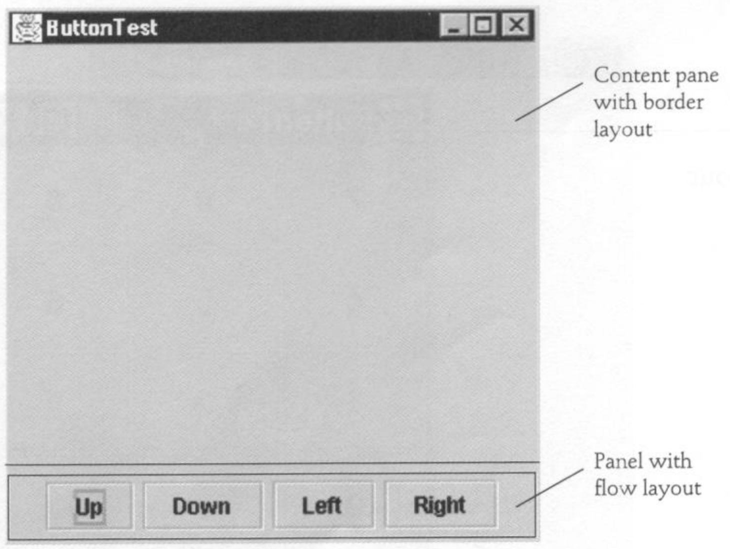
- If you want to add multiple buttons to the same area in a border layout, you need to put the buttons inside a panel, and then add the panel to the content pane.
- A panel supports three different layouts: flow layout (default), grid layout, and border layout
- To define the layout pattern of a panel, you call the `setLayout` method and supply the layout object.

```
JPanel numberPanel = new JPanel();
numberPanel.setLayout(new GridLayout(4, 3));
// define a grid with 4 rows and 3 columns
```

Example:

Figure 2

Containers with Separate Layout Managers



- When a button is clicked, an action event is generated and it is captured by the *action listener* that is associated with the button.
- The action listener only has one method, `actionPerformed()`
- You may install a separate action listener for each button, or you can share one listener among multiple buttons.
- If multiple buttons share an action listener, you can use the `getSource()` method to identify the button that generates the action event.

General program orgranization for Java GUI application.

```
public class MyFrame extends JFrame
{
   public static void main(String[] args)
   {
      MyFrame frame = new MyFrame();
      frame.setVisible(true);
   }

   // outer class instance variables

   public MyFrame()
   {
      // set up the GUI display
      . . .

      // create the action listener object
      MyListener listener = new MyListener();

      // register the listener with the event source
      anEventSource.addActionListener(listener);
      . . .

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }

   // outer class methods
   . . .

   // inner class
   private class MyListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         if (e.getSource() == eventSource1)
            // statements to handle the event
         else if (e.getSource() == eventSource2)
           ...
      }
   }
}
```

```
class MyFrame extends JFrame
{
   private JButton upButton, downButton,
                   leftButton, rightButton;

   public MyFrame()
   {  final int FRAME_WIDTH = 300;
      final int FRAME_HEIGHT = 300;
      setSize(FRAME_WIDTH, FRAME_HEIGHT);

      JPanel buttonPanel = new JPanel();
      upButton = new JButton("Up");
      downButton = new JButton("Down");
      leftButton = new JButton("Left");
      rightButton = new JButton("Right");

      buttonPanel.add(upButton);
      buttonPanel.add(downButton);
      buttonPanel.add(leftButton);
      buttonPanel.add(rightButton);

      add(buttonPanel, "South");

      //create the listener object
      ActionListener listener = new DirectionListener();

      //add the listener object to the buttons
      upButton.addActionListener(listener);
      downButton.addActionListener(listener);
      leftButton.addActionListener(listener);
      rightButton.addActionListener(listener);

      . . .

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);

   }  //end of the constructor


   //other methods
   . . .
   private class DirectionListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Object source = event.getSource();

         if (source == upButton)
            //handler for the upButton
         else if (source == downButton)
            //handler for the downButton
         else if (source == leftButton)
            //handler for the leftButton
         else if (source == rightButton)
            //handler for the rightButton
      }
   }  //end of inner class DirectionListener

}
```

Example codes in conventional style:

```
class MyFrame extends JFrame
{
   public MyFrame()
   {
      ...  // statements to set up GUI


      ActionListener listener = new MyListener();


      upButton.addActionListener(listener);
      ...

   }

   // private inner class
   private class MyListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         //implementation of the method
      }
   };

}
```

Anonymous inner class

- In the above example, the action listener is defined as an inner class with a name (i.e. `MyListener`).
- Since the class is not accessible outside the outer class, one may define it without giving it a name.

```
class MyFrame extends JFrame
{
   public MyFrame()
   {
      ...  // statements to set up GUI


      //Coding format using Anonymous class
      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               //implementation of the method
            }
         };

      /* Coding format using Lambda expression

      ActionListener listener = event -> {
              // implementation of the method
          };
      */

      upButton.addActionListener(listener);
      ...

   }
}
```

- The use of anonymous inner class (or lambda expression) is preferred if the `actionPerformed` method only has a few lines of codes.

- If the implementation of the `actionPerformed` method is long, then it is preferred to use the conventional coding format.

Text Components

- A text field holds a single line of text.
  - When constructing a text field, you can specify the desired number of characters in the constructor.

    ```
    JTextField interestRateField = new JTextField(5);
    ```

- To display multiple lines of text, you use the `JTextArea` class.
  - With a text area, you can specify the number of rows and columns.

    ```
    JTextArea resultArea = new JTextArea(10, 40);
    // 10 rows, 40 columns
    ```

- Both `JTextField` and `JTextArea` are subclasses of the `JTextComponent` class.

- When a user hits the "Enter" key in a `JTextField`, an `ActionEvent` is generated.
- However, if the user hits the "Enter" key in a `JTextArea`, no `ActionEvent` is generated – the "Enter" key just starts a new line.
- To find out when a user is done entering text into a text area, you may add a button at the bottom of the text area. When the user clicks the button, the handler for the button click calls the **`getText()`** method to retrieve the text.
- If you want to use the text component for display purposes only, then you can use the **`setEditable()`** method to disable the user from modifying the content of the text component. The program can still change the contents of the text component by calling the **`setText()`** method.

Example

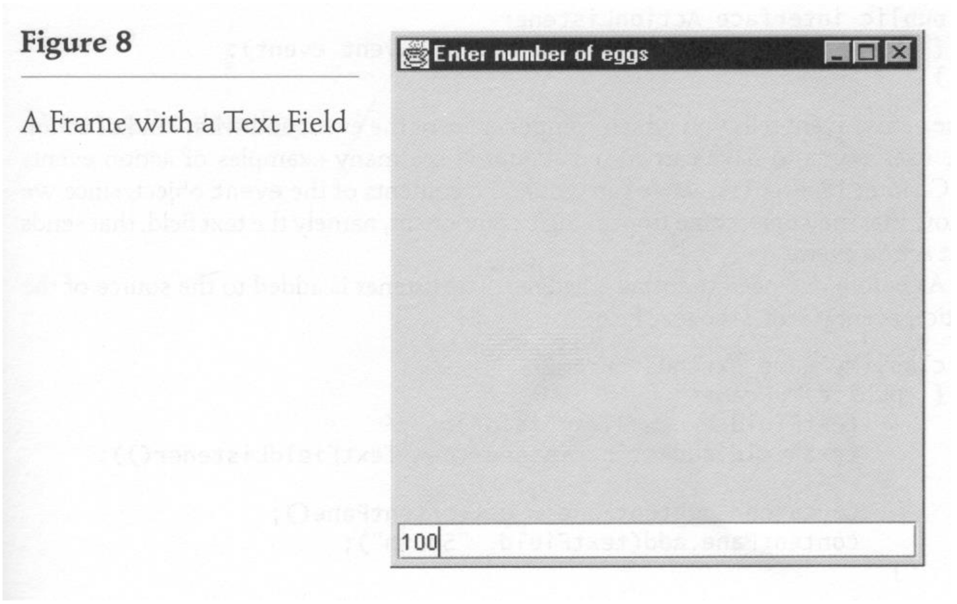


Figure 8

A Frame with a Text Field

```
class MyFrame extends JFrame
{  private JTextField textField;

   public MyFrame()
   {
      textField = new JTextField();
      add(textField, "South");

      textField.addActionListener(new TextFieldListener());

      . . .
   }

   private class TextFieldListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         // get user input
         String input = textField.getText();

         // process user input
         . . .

         // to clear the text field
         textField.setText("");
      }
   }
}
```

There are other types of popular user interface components

- Radio buttons
- Combo box
- Menu
- Slider

You are recommended to read related chapters of any Java book for details.

JavaFX is the new generation of GUI library intended to replace Swing.

Students may refer to related references, e.g.

K. Sharan, “Beginning Java 8 APIs, Extensions and Libraries, Swing, JavaFX, JavaScript, JDBC and Network Programming APIs”, Apress, 2014.