High-level Structure

Do the work outside of main, in a separate class that implements the GUI by extending the JFrame class:

```
import javax.swing.*;
public class SqrtCalculator {
    public static void main( String [] args ) {
        MyWindow myWindow = new MyWindow();   // Create the window
        myWindow.setVisible( true );          // Make it visible
    }
}

class MyWindow extends JFrame {               // "extends JFrame" says "I'm a window."

    public MyWindow() {                               // constructor creates Window
        final int WIDTH=300, HEIGHT=200;
        JLabel myLabel = new JLabel( "Hello World!" ); // Make label widget
        this.add( myLabel );                          // Add label widget to window
        this.setPreferredSize( WIDTH, HEIGHT );       // Resize window.
        this.setLocationRelativeTo( null );           // Center the window
        // Tell jvm to kill program when window closes.
        this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

    }
}
```

Separate the program's layout, interaction (listeners), and business logic so that each can be tweaked independently of the others.

Now we'll apply some stepwise refinement to the constructor. This is critically important, because it makes your program much more robust. And it's easy to do. This section contains the source code for a simple square root calculator. You'll see that now the window's constructor comprises just two method calls:

```
layoutComponents();
addListeners();
```

Note: the `addListeners()` method introduces two new concepts:

- Anonymous inner classes
- Interfaces can behave like classes

These concepts have been thoroughly discussed in my recorded Swing sections. The rationale is discussed in the code's comments.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SqrtCalculator {
    public static void main( String [] args ) {
        MyWindow myWindow = new MyWindow();
        myWindow.setVisible( true );
    }
}

class MyWindow extends JFrame {

    // Widgets
    private JTextField        txtNumber       = new JTextField( 10 );
    private JButton           btnSquareRoot   = new JButton( ""+'\u221A' );

    // Constructor
    public MyWindow () {
        layoutComponents();
        addListeners();
    }

    // SET UP THE PROGRAM'S LAYOUT. There is NO listener logic or business logic here. So
    // we can change the layout independently of the listener logic and business logic.
    private void layoutComponents() {

        this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        this.setTitle( "Square Root Calculator" );
        this.setLocationRelativeTo( null ); // Centers the window

        this.txtNumber.setHorizontalAlignment( JTextField.RIGHT );
        this.txtNumber.setBackground( Color.LIGHT_GRAY );
        this.txtNumber.setBorder( BorderFactory.createLineBorder( Color.BLACK ) );

        JPanel panel = new JPanel();
        panel.add( btnSquareRoot ); // JPanel's FlowLayout centers button

        Box box = Box.createVerticalBox();
        box.add( this.txtNumber );
        box.add( panel );

        this.add( box, BorderLayout.NORTH );

        // Optimize JFrame's size
        this.setPreferredSize( new Dimension( 600, 200 ) );
        this.pack();
    }
```

```
    // SET UP THE PROGRAM'S LISTENERS. There is NO layout or business logic here.
    // So we can change the listeners independently of the layout and business logic.
    private void addListeners() {
        // Buttons fire action events, which are handled by action listeners.
        // Register an action listener for the button's action events.

        // The argument to addActionListener must be an instance of a class that
        // implements the ActionListener interface. We satisfy that requirement with
        // an anonymous inner class derived from the ActionListener interface.

        // This is a good strategy because 1. it avoids scope issues (inner classes'
        // code can see things that live in the outer class), 2. it avoids namespace
        // pollution (no need to invent names for ActionListener classes), and 3. you
        // can easily have a separate actionPerformed method for each widget instead
        // of a single actionPerformed method having to serve all widgets.
        this.btnSquareRoot.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent ae ) {
                btnSquareRootClicked( ae );
            }
        } );
    }

    // THIS IS THE PROGRAM'S BUSINESS LOGIC. There is NO layout or listener logic here.
    // So we can change the business logic independently of the layout and listener logic.

    // Click handler for the "sqrt" button. This click handler doesn't need access to
    // the ActionEvent object. I included it only to show you how it's done in case
    // your event handlers do need it.
    public void btnSquareRootClicked( ActionEvent ae ) {
        String number = this.txtNumber.getText().trim();
        String error = "Enter a number >= 0.0";
        if ( number.length() > 0 ) {
            double result;
            try {
                result = Double.parseDouble( number );
                if ( result >= 0 ) {
                    error = "";
                    result = Math.sqrt( result );
                    this.txtNumber.setText( String.format( "%.2f",result ) );
                }
            }
            catch( NumberFormatException e ) { } // error is already set
        }
        if ( !error.equals( "" ) ) { show_error_message( error ); }
    }

    private static void show_error_message( String error_message ) {
        JOptionPane.showMessageDialog( null, error_message, "Input Error",
                                       JOptionPane.ERROR_MESSAGE
                                     );
    }
}
```

Summary of some useful widgets

Widget	Description
`JFrame`	Analogous to a window.
`JPanel`	Container used to group other widgets
`JLabel`	Read-only text
`JTextField`	Get a single line of text input
`JButton`	Enable user interaction
`JOptionPane`	Provides a standard dialog box that prompts users for a value or informs them of something.

Summary of some useful methods

Layout Methods	Description

`add( Component )`
 	
`add( Component, constraints )`
	Adds component to container. A container might be a `JFrame` or a `JPanel`. A constraint tells the layout manager where to put the component. E.g. to add a component to the North region of a border layout, the constraint is specified as `BorderLayout.NORTH`.

`pack`	Sizes the `JFrame` to fit its widgets' preferred sizes.
`setTitle( title )`	Sets a `JFrame`'s title.
`setBackground( Color )`	Sets a component's background color. Accepts a `Color` object, such as `Color.BLUE`.
`setBorder( Border )`	Sets a component's border. Accepts an object from any class that implements the `Border` interface. That object describes the desired border.
`setFont( Font )`	Set a `JComponent`'s font.
`setHorizontalAlignment( alignment )` `setHorizontalAlignment( alignment )`	Sets a `JLabel`'s or `JTextField`'s horizontal alignment
`setVerticalAlignment( alignment )`	Sets a `JLabel`'s vertical alignment Note: `JTextField` does not have a `setVerticalAlignment()` instance method.
`setLayout( LayoutManager )`	Associates a `LayoutManager` such as a `FlowLayout`, `BorderLayout`, or `GridLayout` with a container such as a `JFrame` or a `JPanel`.
`setLocationRelativeTo( Component )`	Sets the location of the window relative to the specified component. If the component is `null` or not currently showing, the window is placed at the center of the screen.

`setSize( Dimension )`
 	
`setSize( width, height )`
	Sets a `JFrame`'s or `JComponent`'s width and height.

`setDefaultCloseOperation( operation )`	Defines the action that occurs when the `JFrame` is closed. Typically operation is `JFrame.EXIT_ON_CLOSE`.

Event Handling Methods	Description
`addActionListener( ActionListener )`	Associates an `ActionListener` instance with a widget.
`actionPerformed( ActionEvent )`	Resides in a widget's `ActionListener`. Runs when the widget fires an `ActionEvent`. Receives an `ActionEvent` object containing information about the event.
`setActionCommand( actionCommand )`	An action command is just a string that this method associates with a widget that fires `ActionEvent`s. By default a `JButton`'s action command is the `JButton`'s text.
`getActionCommand()`	The widget's `ActionListener`'s `actionPerformed()` method receives an `ActionEvent` object, upon which this method can be invoked to retrieve the widget's action command.
`getSource()`	The widget's `ActionListener`'s `actionPerformed()` method receives an `ActionEvent` object, upon which this method can be invoked to retrieve the widget instance that fired the action.

Business Logic Methods	Description
`setEditable( boolean )`	Defines whether a `TextComponent` (e.g. a `JTextField`) is editable.
`setText( String )`	Set the text of a TextComponent like a `JTextField.`
`getText()`	Get the text of a TextComponent like a `JTextField.`
`setVisible( boolean )` `setVisible( boolean )`	Shows or hides a `Jframe` or a `JComponent`.
`showMessageDialog( parent, message, title, messageType )`	Static `JOptionPane` method that pops up a dialog box.