

Introduction

With one exception, all of our programs used the console and text based input/output mechanisms to interact with the user. Although this is easy to understand and easy to program it is not the way programs usually interact with a user. All modern computers use windows and a mouse to interact with the person in front of the computer. This way of building a computer - user interaction is called a graphical user interface or short a GUI. If you want to create a GUI in Java you will usually use a *widget library* called [Swing](#).

Parts of a GUI

The typical parts of a GUI, no matter what OS or window system you are using, are

- **Windows**

Windows are (rectangular) portions of the screen. You can think of them as sheets of paper that you can drag around on your desktop. All data is displayed in windows.

- **Buttons**

Buttons are a way to let the user invoke a command. Buttons do not represent data but commands. A special form of a *Button* is a *Menu*. To start the command you click on the button.

- **Menus**

In a menu all available commands of the program are grouped together. Usually the menus are placed in a menu bar at the top of a window. Groups of logically related commands are represented as menus which are elements of the menubar.

- **Selection Lists**

Selection lists show a list of strings and the user can select one or more items in the list by clicking on an item.

- **Text fields**

Text fields provide a way to enter text which should be passed into a program.

Events

Whenever the user interacts with a GUI an event is generated by the widget library. So whenever you click a mouse button, press a key on the keyboard, expose a window, close a window or do something else an event is generated and delivered to the target of the event. When you click on a button the system creates a mouse down, then a mouseup event. When this happens within a predefined time interval a mouse clicked event is generated. This information is delivered to the button. If the button does not know what to do when an event occurs, it delivers the event to a parent widget and so on. If no component knows what to do with the event it is discarded.

Event Handlers

In Swing the events can be delegated to **event listener objects**. In this scenario the *button* delivers the raw GUI event with the help of an **event object** to the **event listener object**. One thing that is important to keep in mind is that the application cannot rely on a certain order when things happen. In our old text based interfaces we could somehow determine the flow of program input. In an event driven program the flow is determined by the order in which an event is delivered. An other thing that might be confusing is the fact that you write code that you never call yourself. For example the "clicked" handler of a button is usually called by the Swing library when the event is fired, not by your application.

Parts of a GUI program

A GUI program consists of three logical parts:

- **Graphical components**

This part is sometimes also called the *view* of the program and contains everything that makes up the visual part of the graphical user interface.

- **The event listeners**

This part is sometimes also called the *controller*. The controller listens for messages from the *view* and operates according to these messages on the *data model* of the program

- **The data model**

In a common application the logic should not be coded in the parts of the GUI. The *model* implements the logic of the program. This is the part where the data of your program is stored and manipulated by the *view*.

Usually, in a real world Java application you keep the first and the second part of the GUI in a single class where you set up the graphical components and wire their events to the appropriate event handlers. You should keep the third part always in a separate class.

Programming the interface

Open a window

A typical Swing application consists of one or more top level windows. This is different from the *Snowflake* we programmed a few weeks ago. The *Snowflake* class is an *applet* where the top level window is managed by the browser or the appletviewer.

To open a window you create an instance of class `JFrame`, set the size of the frame and make the window visible. This creates a basic window with basically no functionality. You can resize the window, but if you click on the *close window button* the program does not terminate.

Example code

```
import javax.swing.*;

public class TestFrame1
{
    public static void main ( String[] args )
    {
        JFrame frame = new JFrame("Test Frame 1");
        frame.setSize(200,100);
        frame.setVisible( true );
    }
}
```

To run the example type: `cd ; java TestFrame1`

It seems that `TestFrame1` does not realize that the close button has been clicked. To enable this, we must add a listener that is called when a window close event is fired. Swing provides a predefined class [WindowAdapter](#) that defines handlers for all window event types. Since we are only interested in *window closed events* it is sufficient to override the `void windowClosing(WindowEvent e)` method.

Example code

```
import javax.swing.*;
import java.awt.event.*;      // load window adapter

public class TestFrame2
{
    private JFrame frame;    // Our top level window

    /**
     * Constructor for TestFrame2
     * The window has a window listener that terminates the application
     * when the close window button is clicked.
     */
    TestFrame2() {
        frame = new JFrame("Test Frame 2");
        frame.setSize(200,100);

        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }

    /**
     * Open a TestFrame2
     */
    public static void main ( String[] args )
    {
        TestFrame2 aFrame = new TestFrame2();    // Open a TestFrame2
    }
}
```

To run the example type: `cd ; java TestFrame2`

Adding controls to a top level window

So far our window does not provide any real functionality. To make it useful we must add user interface parts to the top level JFrame. As a general rule, Java user interface elements are always grouped by instances of class [Container](#). Every container has a list of sub components and is responsible for the layout of its sub components. The layout of the components inside of a container is determined by a *LayoutManager*.

The basic recipe to add components to a top level frame is:

- Get the container of the top level window with `getContentPane()`
 1. If the default layout manager of a JFrame, the [Border Layout](#) manager is not what you need, add a different layout manager e.g. a [FlowLayout](#) manager, then add user interface parts directly to the content Pane.
or
 2. Create sub containers, add LayoutManagers to them and then add the parts to the sub containers.
Finally add the sub containers to the top level container. You may nest sub containers.

Layout Managers

FlowLayout

"A flow layout arranges components in a directional flow, much like lines of text in a paragraph. The flow direction is determined by the container's component Orientation property and may be one of two values:

`ComponentOrientation.LEFT_TO_RIGHT`

`ComponentOrientation.RIGHT_TO_LEFT`

Flow layouts are typically used to arrange buttons in a panel. It arranges buttons horizontally, until no more buttons fit on the same line. The line alignment is determined by the align property. The possible values are:

FlowLayout.LEFT
FlowLayout.RIGHT
FlowLayout.CENTER
FlowLayout.LEADING
FlowLayout.TRAILING
"

Example code

```
import javax.swing.*;
import java.awt.event.*;           // import window adapter
import java.awt.FlowLayout;       // import a FlowLayout layout manager
import java.awt.BorderLayout;     // import a BorderLayout layout manager

public class TestFrame3
{
    private JFrame frame;  // Our top level window

    /**
     * Constructor for TestFrame3
     * The window has a window listener that terminates the application
     * when the close window button is clicked. Two labels are displayed
     * with a FlowLayout manager
     */
    TestFrame3() {
        /**
         * Create a top level window with window title
         * "Test Frame 3"
         */
        frame = new JFrame("Test Frame 3");
        frame.setSize(300,200);           // set window size

        /**
         * Override JFrames default layout manager.
         * Components are left aligned and have a horizontal space of
         * 10 and a vertical space of 20 pixels
         */
        FlowLayout aFlowLayout = new FlowLayout(FlowLayout.LEFT,10,20);
        frame.getContentPane().setLayout(aFlowLayout);

        /**
         * Add two label to the content pane of frame
         */
        JLabel aLabel = new JLabel("Hello World");
        frame.getContentPane().add(aLabel);

        JLabel aLabel1 = new JLabel("Hello World2");
        frame.getContentPane().add(aLabel1);

        /**
         * Add a window listener
         */
        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );  // show window
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }

    /**
     * Open a TestFrame3
     */
    public static void main ( String[] args )
    {
        TestFrame3 aFrame = new TestFrame3();
    }
}
```

To run the example type: `cd ; java TestFrame3`

BorderLayout

The [BorderLayout](#) places its content into 5 different regions:


BorderLayout.NORTH

BorderLayout.WEST

BorderLayout.CENTER

BorderLayout.EAST

BorderLayout.SOUTH

 BorderLayout

To place a label in the northern sector of the manager:

```
JLabel aLabel = new JLabel("Hello World");  
frame.getContentPane().add(BorderLayout.NORTH,aLabel);
```

No Region may contain more than one component. If you add more components only the last one is visible. If you want to add more components to a region, you must create a wrapper component, usually a [JPanel](#), and add the components to the JPanel. The default layout manager of a JPanel is the FlowLayout manager.

Example code

```
import javax.swing.*;
import java.awt.event.*;           // import window adapter
import java.awt.FlowLayout;       // import a FlowLayout layout manager
import java.awt.BorderLayout;     // import a BorderLayout layout manager
import java.awt.Dimension;       // import Dimension
public class TestFrame4
{
    private JFrame frame;    // Our top level window

    /**
     * Constructor for TestFrame4
     * The window has a window listener that terminates the application
     * when the close window button is clicked. 5 Buttons are displayed
     * in the 5 areas of a BorderLayout
     */
    TestFrame4() {
        /**
         * Create a top level window with window title
         * "Test Frame 4"
         */
        frame = new JFrame("Test Frame 4");
        frame.setSize(300,200);

        /**
         * Add five buttons to the five different parts of the BorderLayout
         */

        JButton centerButton = new JButton("Center");
        frame.getContentPane().add(BorderLayout.CENTER,centerButton);

        JButton northButton = new JButton("North");
        frame.getContentPane().add(BorderLayout.NORTH,northButton);

        JButton southButton = new JButton("South");
        frame.getContentPane().add(BorderLayout.SOUTH,southButton);

        JButton eastButton = new JButton("East");
        frame.getContentPane().add(BorderLayout.EAST,eastButton);

        JButton westButton = new JButton("West");
        frame.getContentPane().add(BorderLayout.WEST,westButton);

        /**
         * Add a window listener
         */

        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );    // show window
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }

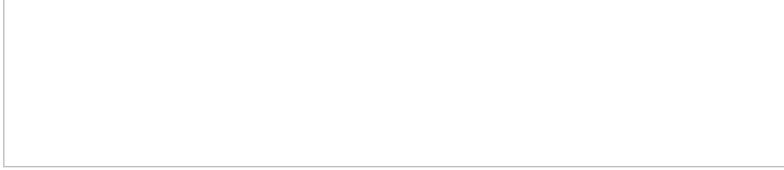
    /**
     * Open a TestFrame4
     */
    public static void main ( String[] args )
    {
        TestFrame4 aFrame = new TestFrame4();
    }
}
```

To run the example type: `cd ; java TestFrame4`

GridLayout

A [GridLayout](#) manager arranges components in a grid of rows and columns. All cells in the layout are of the same size.

To create a manager for 2 rows and 3 Columns: `getContentPane().setLayout(new GridLayout(2,3));`



No Region in a GridLayout manager may contain more than one component.

To add a component to a GridLayout simply `add()` the component to the layout. The cell of a component is determined by the order of the `add()` call. The first `add()` call adds the component to cell 1, the second call to cell 2, and so on. You are not allowed to skip a grid position. E.g. if you want to add a component to cell 1 and cell 3 of a GridLayout the program will throw an exception when the layout is displayed.

Example code

```
import javax.swing.*;
import java.awt.event.*;           // import window adapter
import java.awt.FlowLayout;       // import a FlowLayout layout manager
import java.awt.BorderLayout;     // import a BorderLayout layout manager
import java.awt.GridLayout;       // import a GridLayout layout manager
import java.awt.Dimension;       // import Dimension

public class TestFrame5
{
    private JFrame frame;  // Our top level window

    /**
     * Constructor for TestFrame5
     * The window has a window listener that terminates the application
     * when the close window button is clicked. 5 Buttons are displayed
     * in a GridLayout
     */
    TestFrame5() {
        frame = new JFrame("Test Frame 5");
        frame.setSize(300,200);

        /**
         * Override JFrames default layout manager.
         */
        GridLayout aGridLayout = new GridLayout(2,3);
        frame.getContentPane().setLayout(aGridLayout);

        /**
         * Add five buttons to the GridLayout
         */

        JButton firstButton = new JButton("first");
        frame.getContentPane().add(firstButton);

        JButton secondButton = new JButton("second");
        frame.getContentPane().add(secondButton);

        JButton thirdButton = new JButton("third");
        frame.getContentPane().add(thirdButton);

        JButton fourthButton = new JButton("fourth");
        frame.getContentPane().add(fourthButton);

        JButton fifthButton = new JButton("fifth");
        frame.getContentPane().add(fifthButton);

        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }

    /**
     * Open a TestFrame4
     */
    public static void main ( String[] args )
    {
        TestFrame5 aFrame = new TestFrame5();
    }
}
```

To run the example type: `cd ; java TestFrame5`

BoxLayout

A [BoxLayout](#) manager either stacks its components in a column or in a row. When you [predefine a size](#) for a component a BoxLayout manager, unlike a GridLayout or a BorderLayout manager, respects the preferred sizes of its components.

To create a BoxLayout manager you must pass the container and an orientation to the constructor of BoxLayout:

```
BoxLayout aBoxLayout = new BoxLayout(frame.getContentPane(),BoxLayout.Y_AXIS);
```

The orientation is determined by the following constants:

BoxLayout.Y_AXIS - layout components top to bottom

BoxLayout.X_AXIS - layout components left to right

The [Box](#) class provides invisible 'space' components. You can place these areas between components to get stretchable or rigid spaces. You can think of a stretchable space as a spring between two components. When you resize the bounding component of the two sub components, the spring stretches and pushes the components further apart. A rigid space stays always the same, even when you resize the window.

The following code adds a 5 pixel wide space between horizontally aligned components:

```
container.add(firstComponent);
container.add(Box.createRigidArea(new Dimension(5,0)));
container.add(secondComponent);
```

To add a stretchable vertical space:

```
container.add(firstComponent);
container.add(Box.createVerticalGlue());
container.add(secondComponent);
```

To add a stretchable horizontal space:

```
container.add(firstComponent);
container.add(Box.createHorizontalGlue());
container.add(secondComponent);
```

Example code

```
import javax.swing.*;
import java.awt.event.*;           // import window adapter
import java.awt.FlowLayout;       // import a FlowLayout layout manager
import java.awt.BorderLayout;     // import a BorderLayout layout manager
import java.awt.GridLayout;       // import a GridLayout layout manager
import javax.swing.BoxLayout;     // import a BoxLayout layout manager
import java.awt.Dimension;       // import Dimension

public class TestFrame7
{
    private JFrame frame;    // Our top level window

    /**
     * Constructor for TestFrame7
     * The window has a window listener that terminates the application
     * when the close window button is clicked. 5 Buttons are displayed
     * in a BoxLayout
     */
    TestFrame7() {
        frame = new JFrame("Test Frame 7");
        frame.setSize(300,200);

        /**
         * Override JFrames default layout manager.
         */
        BoxLayout aBoxLayout = new BoxLayout(frame.getContentPane(),BoxLayout.Y_AXIS);
        frame.getContentPane().setLayout(aBoxLayout);

        /**
         * This is the common size of the buttons
         */
        Dimension size = new Dimension(80,30);

        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        /**
         * Create a button, set its size and a rigid space, then add it to the
         * frame.
         */
        JButton firstButton = new JButton("first");
        firstButton.setMaximumSize(size);
        frame.getContentPane().add(firstButton);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        JButton secondButton = new JButton("second");
        secondButton.setMaximumSize(size);
        frame.getContentPane().add(secondButton);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        JButton thirdButton = new JButton("third");
        thirdButton.setMaximumSize(size);
        frame.getContentPane().add(thirdButton);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        JButton fourthButton = new JButton("fourth");
        fourthButton.setMaximumSize(size);
        frame.getContentPane().add(fourthButton);

        frame.getContentPane().add(Box.createVerticalGlue());

        JButton fifthButton = new JButton("fifth");
        fifthButton.setMaximumSize(size);
        frame.getContentPane().add(fifthButton);

        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
}
```

```

        private class MyWindowListener extends WindowAdapter {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }

        /**
         * Open a TestFrame7
         */
        public static void main ( String[] args )
        {
            TestFrame7 aFrame = new TestFrame7();
        }
    }
}

```

To run the example type: `cd ; java TestFrame7`

Controlling the size of component

The Swing layout managers sometimes have their own ideas about the size of their sub components. The `BoxLayout` at least accepts your choices. To change the size of a component you can call:

```

public void setMinimumSize(Dimension minimumSize);
public void setMaximumSize(Dimension maximumSize);
public void setPreferredSize(Dimension preferredSize);

```

Unfortunately only `setMaximumSize()` worked for me. But if you want to be sure call all three methods with the same `Dimension` instance.

Dimension

Class `Dimension` specifies the width and the height or a horizontal distance and a vertical distance. To create a `Dimension` with a width of 10 and a height of 100 pixels use:

```
Dimension size = new Dimension(10,100);
```

Buttons and ActionListeners

To add a `Button` you create a `JButton` object in the same way we did it with the `JLabel` already. The major difference is that a `JButton` object may invoke an action when you click on it. The class which performs the action must implement the `ActionListener` interface. The interface consist of a single method `void actionPerformed(ActionEvent e)`.

A lot of examples implement the `ActionListener` in the class that sets up the `JFrame`. This is probably not a very good idea. When you look at Savitch's example at page 884 you see a fairly complicated `if-else if - else` statement. When you have more than a single button on your interface, this becomes very unreadable and hard to maintain. It is a much better style to implement an `ActionListener` class for every button. To add the Listener to the button use `void addActionListener(ActionListener l)`.

Example code

```
import javax.swing.*;
import java.awt.event.*;           // import window adapter
import java.awt.FlowLayout;       // import a FlowLayout layout manager
import java.awt.BorderLayout;     // import a BorderLayout layout manager
import java.awt.GridLayout;       // import a GridLayout layout manager
import javax.swing.LayoutStyle;    // import a BoxLayout layout manager
import java.awt.Dimension;        // import Dimension
import javax.swing.JOptionPane;   // import Message dialogs

public class TestFrame8
{
    private JFrame frame;    // Our top level window

    /**
     * Constructor for TestFrame8
     * The window has a window listener that terminates the application
     * when the close window button is clicked. 5 Buttons are displayed
     * in a BoxLayout
     */
    TestFrame8() {
        frame = new JFrame("Test Frame 8");
        frame.setSize(300,200);

        /**
         * Override JFrames default layout manager.
         */
        BoxLayout aBoxLayout = new BoxLayout(frame.getContentPane(),BoxLayout.Y_AXIS);
        frame.getContentPane().setLayout(aBoxLayout);

        /**
         * This is the common size of the buttons
         */
        Dimension size = new Dimension(80,30);

        /** Add 5 Pixels rigid space */
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        /**
         * Create a button, set its size and a rigid space, add an ActionListener
         * then add it to the frame.
         */
        JButton firstButton = new JButton("first");
        firstButton.setMaximumSize(size);
        firstButton.addActionListener(new firstButtonHandler());
        frame.getContentPane().add(firstButton);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        /**
         * Ditto
         */
        JButton secondButton = new JButton("second");
        secondButton.setMaximumSize(size);
        secondButton.addActionListener(new secondButtonHandler());
        frame.getContentPane().add(secondButton);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        JButton thirdButton = new JButton("third");
        thirdButton.setMaximumSize(size);
        thirdButton.addActionListener(new thirdButtonHandler());
        frame.getContentPane().add(thirdButton);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));

        JButton fourthButton = new JButton("fourth");
        fourthButton.setMaximumSize(size);
        fourthButton.addActionListener(new fourthButtonHandler());
        frame.getContentPane().add(fourthButton);

        /**
         * Add a stretchable space
         */
        frame.getContentPane().add(Box.createVerticalGlue());

        JButton fifthButton = new JButton("fifth");
```

```

        fifthButton.setMaximumSize(size);
        fifthButton.addActionListener(new fifthButtonHandler());
        frame.getContentPane().add(fifthButton);

        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );
    }

    /**
     * ActionListeners for the JButtons
     */
    private class firstButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame,"First button was clicked");
        }
    }

    private class secondButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame,"Second button was clicked");
        }
    }

    private class thirdButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame,"Third button was clicked");
        }
    }

    private class fourthButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame,"Fourth button was clicked");
        }
    }

    private class fifthButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame,"Fifth button was clicked");
            new TestFrame8();
        }
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }

    /**
     * Open a TestFrame8
     */
    public static void main ( String[] args )
    {
        TestFrame8 aFrame = new TestFrame8();
    }
}

```

To run the example type: `cd ; java TestFrame8`

Lists and ListSelectionListener

With a [JList](#) component you can display a list of objects and the user can select one or more items from the list. By default the *JList* object does not provide a way to scroll through the list. To enable this, the *JList* must be wrapped in a [JScrollPane](#).

The connection between the selections in the list and your code is made by a class which implements the [ListSelectionListener](#) interface. This interface requires a single method `public void valueChanged(ListSelectionEvent event)`. To add the Listener to the List use `void addListSelectionListener(ListSelectionListener listener)` in *JList*. Whenever you click on an item in the list or you change the selection with the cursor keys, the selection handler is invoked.

Accessing the selected item

To get access to the selected item(s) in a list, the `JList` class provides the following methods:

`Object getSelectedValue()` - Returns the first selected value, or null if the selection is empty.

`int getSelectedIndex()` - Returns the first index of the first selected element; returns -1 if there is no selected item.

List datamodels

The data displayed by a list is contained in a model. When we create a `JList` with an array as a model, we have only read access to the elements in the list. We can retrieve elements from the list with the standard access methods. But as soon as we add elements the program will crash.

To change values in the list we must replace the read only model with an instance of class [DefaultListModel](#). This class provides methods to add or remove elements at a certain index in the list. If we want to add elements to the `JList` or remove elements from the `JList` we must change the underlying list model.

To edit the content of the `DefaultListModel` you can use:

`void addElement(Object obj)` - Adds the specified component to the end of this list.

`void add(int index, Object element)` - Inserts the specified element at the specified position in this list.

`Object remove(int index)` - Removes the element at the specified position in this list.

Example of a JList with a DefaultListModel:

```
/* Data to display in the list */
String[] entries = {"Fuchs","du","hast","die","Gans","gestohlen"};
/*
    Create a changable ListModel for our List. We need this because
    we later want to add or remove Strings from the list!!!
*/
DefaultListModel aListModel = new DefaultListModel();
for (int i=0;i < entries.length;i++)
    aListModel.addElement(entries[i]);
/* Create list with the new datamodel */
aList = new JList(aListModel);
```

Building your own ListModels

While it is easy and convenient to edit the data in the *JLists* data model, this is not desirable from a software engineering point of view. The model and not the view contains that data and programming logic of your application. The `DefaultListModel()` does not fit in that scenario, because it actually manages its data. When you change data on a component level, this is not necessarily reflected in your application model. Since every application and its model are different we must build a special data model that interfaces between the application model and the *JList*. Swing provides a class [AbstractListModel](#) that implements a basic list model. To create a specialized list model we usually have to provide the following methods:

- The constructor of your custom list model must connect to the desired aspect of the applications data model.
- `public Object getElementAt(int index)` - Retrieve element `index` from the model.
- `public int getSize()` - Calculate the number of elements in our list model.

For a simple data model the above might be sufficient. But for a more complicated data structure this might be inefficient to access the model for every click in the list box. It would be more efficient if the model tells everybody who is interested, that some aspect has changed and is ready to display. Java implements such a mechanism with the [Observer - Observable](#) pattern.

Example code

```
import javax.swing.*;
import java.awt.event.*;           // import window adapter
import java.awt.FlowLayout;       // import a FlowLayout layout manager
import java.awt.BorderLayout;     // import a BorderLayout layout manager
import java.awt.GridLayout;       // import a GridLayout layout manager
import javax.swing.BoxLayout;     // import a BoxLayout layout manager
import java.awt.Dimension;       // import Dimension
import javax.swing.JOptionPane;   // import Message dialogs
import javax.swing.event.*;

public class TestFrame9
{
    private JFrame frame; // Our toplevel window
    private JLabel aLabel; /* Label to display selection */

    /**
     * Constructor for TestFrame9
     * The window has a window listener that terminates the application
     * when the close window button is clicked. 5 Buttons are displayed
     * in a BoxLayout
     */
    TestFrame9() {
        frame = new JFrame("Test Frame 9");
        frame.setSize(200,100);

        /* Label to display selection */
        aLabel = new JLabel("Nothing selected");

        /* Data to display in the list */
        String[] entries = {"Fuchs","du","hast","die","Gans","gestohlen"};

        /* Create list and wrap it into a JScrollPane */
        JList aList = new JList(entries);
        /* connect List and SelectionListener */
        aList.addListSelectionListener(new ValueReporter());
        JScrollPane listPane = new JScrollPane(aList);

        /*
         Override JFrames default layout manager.
        */

        BoxLayout aBoxLayout = new BoxLayout(frame.getContentPane(),BoxLayout.Y_AXIS);
        frame.getContentPane().setLayout(aBoxLayout);

        /* Add 5 Pixels rigid space between components */
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));
        frame.getContentPane().add(aLabel);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));
        frame.getContentPane().add(listPane);

        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );
    }

    private class ValueReporter implements ListSelectionListener {
        public void valueChanged(ListSelectionEvent event) {
            if (!event.getValueIsAdjusting()) {
                JList l = (JList) event.getSource();
                aLabel.setText(l.getSelectedValue().toString());
            }
        }
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }
}
```

```
/**
 * Open a TestFrame9
 */
public static void main ( String[] args )
{
    TestFrame9 aFrame = new TestFrame9();
}
}
```

To run the example type: `cd ; java TestFrame9`

Text fields

So far, we only had JLabel objects to display text on our interfaces. For editing a single line of text Swing provides a [JTextField](#) component. Every JTextField component displays only a specific number of characters. You may enter more text into the text field, but the extra text that exceeds the text fields columns count is not displayed.

To change the text in the text field you must call the `public void setText(String t)` method, to get the contents of a text field call `public String getText()`.

To create a JTextFields with 30 columns:

```
JPanel aPanel = new JPanel();
JTextField entry = new JTextField(30);
aPanel.add(entry);
entry.setText("Foo Bar");
String notes = entry.getText();
```


Example code

```
import javax.swing.*;
import java.awt.event.*;           // import window adapter
import java.awt.FlowLayout;       // import a FlowLayout layout manager
import java.awt.BorderLayout;     // import a BorderLayout layout manager
import java.awt.GridLayout;       // import a GridLayout layout manager
import javax.swing.BoxLayout;     // import a BoxLayout layout manager
import java.awt.Dimension;       // import Dimension
import javax.swing.JOptionPane;   // import Message dialogs
import javax.swing.event.*;

public class TestFrame11
{
    private JFrame frame;           // Our toplevel window
    private JLabel aLabel;         /* Label to display selection */
    private JTextField aTextfield; // Textfield to get a String */
    /**
     * Constructor for TestFrame11
     * The window has a window listener that terminates the application
     * when the close window button is clicked. 5 Buttons are displayed
     * in a BoxLayout
     */
    TestFrame11() {
        frame = new JFrame("Test Frame 11");
        frame.setSize(200,120);

        /* Label to display selection */
        aLabel = new JLabel("Nothing selected");
        aTextfield = new JTextField(40);
        JButton aButton = new JButton("Change Label");
        aButton.addActionListener(new aButtonHandler());
        /*
         * Override JFrames default layout manager.
         */

        BoxLayout aBoxLayout = new BoxLayout(frame.getContentPane(),BoxLayout.Y_AXIS);
        frame.getContentPane().setLayout(aBoxLayout);

        /* Add 5 Pixels rigid space between components */
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));
        frame.getContentPane().add(aLabel);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));
        frame.getContentPane().add(aTextfield);
        frame.getContentPane().add(Box.createRigidArea(new Dimension(0,5)));
        frame.getContentPane().add(aButton);

        frame.addWindowListener(new MyWindowListener());

        frame.setVisible( true );
    }

    /**
     * ActionListeners for the JButtons
     */
    private class aButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String s = aTextfield.getText();
            if (s.length() > 0) {
                aLabel.setText(s);
                aTextfield.setText("");
            }
        }
    }

    /**
     * Our window listener terminates the program when the close window button
     * is clicked.
     */
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }
}
```

```
/**
 * Open a TestFrame11
 */
public static void main ( String[] args )
{
    TestFrame11 aFrame = new TestFrame11();
}
}
```

To run the example type: `cd ; java /afs/sfs/lehre/saile/TestFrame11`

Dialog boxes

Fairly often it is necessary to display messages or ask for confirmation of a certain action. Swing provides predefined dialog boxes in class [JOptionPane](#).

To get a message box use:

```
JOptionPane.showMessageDialog(frame,"Word is already in list");
```

`frame` is the parent window of the dialog box.

To get a confirmation dialog use:

```
JOptionPane.showConfirmDialog(frame,"Really delete element??")
```

The `showConfirmDialog()` method displays the message string and three buttons "Yes", "No" and "Cancel". The method returns `YES_OPTION` when "Yes" was selected, `NO_OPTION` when "No" was selected and `CANCEL_SELECTED` when the user selected "Cancel".

To enter a single line of text use:

```
public static String showInputDialog(Component parentComponent, Object message, Object
initialSelectionValue)
```

The `JOptionPane.showInputDialog()` method displays a message string `message` and a `JTextField` where you can enter text. If you want to provide a predefined entry for the text field, set `initialSelectionValue` to the desired value. The method returns the text in the `JTextField`.

Last modified: Tuesday, 12 May 2020, 11:05 AM