The Java Swing tutorial

Introduction to the Java Swing Toolkit

About this tutorial

This is an introductory Swing tutorial. The purpose of this tutorial is to get you started with the Java Swing toolkit. The tutorial has been created and tested on Linux.

About Swing

Swing library is an official Java GUI toolkit released by Sun Microsystems.

The main characteristics of the Swing toolkit

- platform independent
- customizable
- extensible
- configurable
- lightweight

Swing consists of the following packages

- javax.swing
- javax.swing.border
- javax.swing.colorchooser
- javax.swing.event
- javax.swing.filechooser
- javax.swing.plaf
- javax.swing.plaf.basic
- javax.swing.plaf.metal
- javax.swing.plaf.multi
- javax.swing.plaf.synth
- javax.swing.table
- javax.swing.text
- javax.swing.text.html

- javax.swing.text.html.parser
- javax.swing.text.rtf
- javax.swing.tree
- javax.swing.undo

Swing is probably the most advanced toolkit on this planet. It has a rich set of widgets. From basic widgets like Buttons, Labels, Scrollbars to advanced widgets like Trees and Tables.

Swing is written in 100% java.

Swing is a part of JFC, Java Foundation Classes. It is a collection of packages for creating full featured desktop applications. JFC consists of AWT, Swing, Accessibility, Java 2D, and Drag and Drop. Swing was released in 1997 with JDK 1.2. It is a mature toolkit.

The Java platform has Java2D library, which enables developers to create advanced 2D graphics and imaging.

There are basically two types of widget toolkits.

- Lightweight
- Heavyweight

A heavyweight toolkit uses OS's API to draw the widgets. For example Borland's VCL is a heavyweight toolkit. It depends on WIN32 API, the built in Windows application programming interface. On Unix systems, we have GTK+ toolkit, which is built on top of X11 library. Swing is a lightweight toolkit. It paints it's own widgets. It is in fact the only lightweight toolkit I know about.

SWT library

There is also another GUI library for the Java programming language. It is called SWT. The Standard widget toolkit. The SWT library was initially developed by the IBM corporation. Now it is an open source project, supported by IBM. The SWT is an example of a heavyweight toolkit. It lets the underlying OS to create GUI. SWT uses the java native interface to do the job. The main advantages of the SWT are speed and native look and feel. The SWT is on the other hand more error prone. It is less powerful then Swing. It is also quite Windows centric library.

Java Swing first programs

In this chapter, we will program our first programs in Swing toolkit. The examples are going to be very simple. We will cover some basic functionality.

Our first example

In our first example, we will show a basic window.

```
import javax.swing.JFrame;



public class Simple extends JFrame {


    public Simple() {


        setSize(300, 200);

        setTitle("Simple");

        setDefaultCloseOperation(EXIT_ON_CLOSE);

   }


    public static void main(String[] args) {


        Simple simple = new Simple();
```

```
        simple.setVisible(true);



    }

}
```

While this code is very small, the application window can do quite a lot. It can be resized, maximized, minimized. All the complexity that comes with it has been hidden from the application programmer.

```
import javax.swing.JFrame;
```

Here we import the JFrame widget. It is a toplevel container, which is used for placing other widgets.

```
setSize(300, 200);


setTitle("Simple");
```

This code will resize the window to be 300px wide and 200px tall. It will set the title of the window to Simple.

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

This method will close the window, if we click on the close button. By default nothing happens.

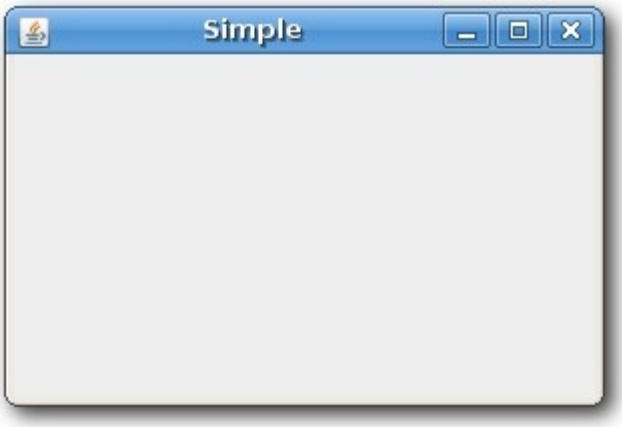


Figure: Simple

Centering window on the screen

By default, when the window is shown on the screen, it is not centered. It will pop up most likely in the upper left corner. The following code example will show the window centered on the screen.

```
import java.awt.Dimension;

import java.awt.Toolkit;



import javax.swing.JFrame;



public class CenterOnScreen extends JFrame {



    public CenterOnScreen() {
```

```
        setSize(300, 200);

        setTitle("CenterOnScreen");

        setDefaultCloseOperation(EXIT_ON_CLOSE);



        Toolkit toolkit = getToolkit();

        Dimension size = toolkit.getScreenSize();

        setLocation(size.width/2 - getWidth()/2,

             size.height/2 - getHeight()/2);

    }



    public static void main(String[] args) {



        CenterOnScreen cos = new CenterOnScreen();

        cos.setVisible(true);



    }

}
```

To center the window on the screen, we must know the resolution of the monitor. For this, we use the *Toolkit* class.

```
Toolkit toolkit = getToolkit();
```

```
Dimension size = toolkit.getScreenSize();
```

We get the toolkit and figure out the screen size.

```
setLocation(size.width/2 - getWidth()/2, size.height/2 - getHeight()/2);
```

To place the window on the screen, we call the *setLocation()* method.

Buttons

In our next example, we will show two buttons. The first button will beep a sound and the second button will close the window.

```
import java.awt.Dimension;

import java.awt.Toolkit;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;




public class Buttons extends JFrame {


    private Toolkit toolkit;
```

```
    public Buttons() {



        setTitle("Buttons");

        setSize(300, 200);



        toolkit = getToolkit();

        Dimension size = toolkit.getScreenSize();

        setLocation((size.width - getWidth())/2, (size.height -
getHeight())/2);

        setDefaultCloseOperation(EXIT_ON_CLOSE);



        JPanel panel = new JPanel();

        getContentPane().add(panel);



      panel.setLayout(null);



        JButton beep = new JButton("Beep");

        beep.setBounds(150, 60, 80, 30);

        beep.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                toolkit.beep();
```

```
            }
        });

        JButton close = new JButton("Close");
        close.setBounds(50, 60, 80, 30);
        close.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.exit(0);
            }
        });

        panel.add(beep);
        panel.add(close);

    }

    public static void main(String[] args) {

        Buttons buttons = new Buttons();
        buttons.setVisible(true);
```

```
    }

}
```

In this example, we will see two new topics. Layout management and event handling. They will be touched only briefly. Both of the topics will have their own chapter.

```
JPanel panel = new JPanel();

getContentPane().add(panel);
```

We create a *JPanel* component. It is a generic lightweight container. We add the JPanel to the JFrame.

```
panel.setLayout(null);
```

By default, the JPanel has a *FlowLayout* manager. The layout manager is used to place widgets onto the containers. If we call *setLayout(null)* we can position our components absolutely. For this, we use the *setBounds()* method.

```
JButton beep = new JButton("Beep");

beep.setBounds(150, 60, 80, 30);

beep.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent event) {

        toolkit.beep();

    }

});
```

Here we create a button. We position it by calling the *setBounds()* method. Then we add an action listener. The action listener will be called, when we perform an action on the button. In our case, if we click on the button. The beep button will play a simple beep sound.

```
System.exit(0);
```

The close button will exit the application. For this, we call the *System.exit()* method.

```
panel.add(beep);


panel.add(close);
```

In order to show the buttons, we must add them to the panel.

Figure: Buttons

A tooltip

Tooltips are part of the internal application's help system. The Swing shows a small rectangular window, if we hover a mouse pointer over an object.

```
import java.awt.Dimension;

import java.awt.Toolkit;
```

```
import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;




public class Tooltip extends JFrame {



    private Toolkit toolkit;



    public Tooltip() {



        setTitle("Tooltip");

        setSize(300, 200);



        toolkit = getToolkit();

        Dimension size = toolkit.getScreenSize();

        setLocation((size.width - getWidth())/2, (size.height -
getHeight())/2);

        setDefaultCloseOperation(EXIT_ON_CLOSE);



        JPanel panel = new JPanel();
```

```
        getContentPane().add(panel);



        panel.setLayout(null);

        panel.setToolTipText("A Panel container");



        JButton button = new JButton("Button");

        button.setBounds(100, 60, 80, 30);

        button.setToolTipText("A button component");



        panel.add(button);



    }



    public static void main(String[] args) {



        Tooltip tooltip = new Tooltip();

        tooltip.setVisible(true);



    }

}
```

In the example, we set the tooltip for the frame and the button.

```
panel.setToolTipText("A Panel container");
```

To enable a tooltip, we call the *setTooltipText()* method.

Figure: Tooltip

Menus and toolbars in Java Swing

Creating a menubar

A menubar is one of the most visible parts of the GUI application. It is a group of commands located in various menus. While in console applications you had to remember all those arcane commands, here we have most of the commands grouped into logical parts. There are accepted standards that further reduce the amount of time spending to learn a new application.

In Java Swing, to implement a menubar, we use three objects. A **JMenuBar**, a **JMenu** and a **JMenuItem**.

Simple menubar

We begin with a simple menubar example.

```
package com.zetcode;


import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.KeyEvent;


import javax.swing.ImageIcon;

import javax.swing.JFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;




public class Menu extends JFrame {


    public Menu() {


        setTitle("JMenuBar");


        JMenuBar menubar = new JMenuBar();

        ImageIcon icon = new ImageIcon("exit.png");
```

```
        JMenu file = new JMenu("File");
        file.setMnemonic(KeyEvent.VK_F);

        JMenuItem fileClose = new JMenuItem("Close", icon);
        fileClose.setMnemonic(KeyEvent.VK_C);
        fileClose.setToolTipText("Exit application");
        fileClose.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.exit(0);
            }

        });

        file.add(fileClose);

        menubar.add(file);

        setJMenuBar(menubar);

        setSize(250, 200);
```

```
        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }


    public static void main(String[] args) {


        new Menu();


    }

}
```

Our example will show a menu with one item. Selecting the close menu item we close the application.

```
JMenuBar menubar = new JMenuBar();
```

Here we create a menubar.

```
ImageIcon icon = new ImageIcon("exit.png");
```

We will display an icon in the menu.

```
JMenu file = new JMenu("File");


file.setMnemonic(KeyEvent.VK_F);
```

We create a menu object. The menus can be accessed via the keybord as well. To bind a menu to a particular key, we use the **setMnemonic** method. In our case, the menu can be opened with the **ALT + F** shortcut.

```
fileClose.setToolTipText("Exit application");
```

This code line creates a tooltip for a menu item.

Figure: JMenuBar

Submenu

Each menu can also have a submenu. This way we can group similar commnads into groups. For example we can place commands that hide/show various toolbars like personal bar, address bar, status bar or navigation bar into a submenu called toolbars. Within a menu, we can seperate commands with a separator. It is a simple line. It is common practice to separate commands like new, open, save from commands like print, print preview with a single separator. Menus commands can be launched via keyboard shortcuts. For this, we define menu item accelerators.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.KeyEvent;



import javax.swing.ImageIcon;
```

```
import javax.swing.JFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

import javax.swing.KeyStroke;




public class Submenu extends JFrame {


    public Submenu() {


        setTitle("Submenu");


        JMenuBar menubar = new JMenuBar();

        ImageIcon iconNew = new ImageIcon("new.png");

        ImageIcon iconOpen = new ImageIcon("open.png");

        ImageIcon iconSave = new ImageIcon("save.png");

        ImageIcon iconClose = new ImageIcon("exit.png");


        JMenu file = new JMenu("File");

        file.setMnemonic(KeyEvent.VK_F);
```

```
        JMenu imp = new JMenu("Import");
        imp.setMnemonic(KeyEvent.VK_M);

        JMenuItem newsf = new JMenuItem("Import newsfeed list...");
        JMenuItem bookm = new JMenuItem("Import bookmarks...");
        JMenuItem mail = new JMenuItem("Import mail...");

        imp.add(newsf);
        imp.add(bookm);
        imp.add(mail);

        JMenuItem fileNew = new JMenuItem("New", iconNew);
        fileNew.setMnemonic(KeyEvent.VK_N);

        JMenuItem fileOpen = new JMenuItem("Open", iconOpen);
        fileNew.setMnemonic(KeyEvent.VK_O);

        JMenuItem fileSave = new JMenuItem("Save", iconSave);
        fileSave.setMnemonic(KeyEvent.VK_S);
```

```
        JMenuItem fileClose = new JMenuItem("Close", iconClose);

        fileClose.setMnemonic(KeyEvent.VK_C);

        fileClose.setToolTipText("Exit application");

        fileClose.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W,

            ActionEvent.CTRL_MASK));



        fileClose.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                System.exit(0);

            }



        });



        file.add(fileNew);

        file.add(fileOpen);

        file.add(fileSave);

        file.addSeparator();

        file.add(imp);

        file.addSeparator();

        file.add(fileClose);
```

```
        menubar.add(file);



        setJMenuBar(menubar);



        setSize(360, 250);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }



    public static void main(String[] args) {

        new Submenu();

    }

}
```

In this example, we create a submenu, a menu separator and an accelerator key.

```
JMenu imp = new JMenu("Import");


...


file.add(imp);
```

A submenu is just like any other normal menu. It is created the same way. We simply add a menu to existing menu.

```
fileClose.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W,
```

```
    ActionEvent.CTRL_MASK));
```

An accelerator is a key shortcut that launches a menu item. In our case, by pressing **Ctrl + W** we close the application.

```
file.addSeparator();
```

A separator is a vertical line that visually separates the menu items. This way we can group items into some logical places.

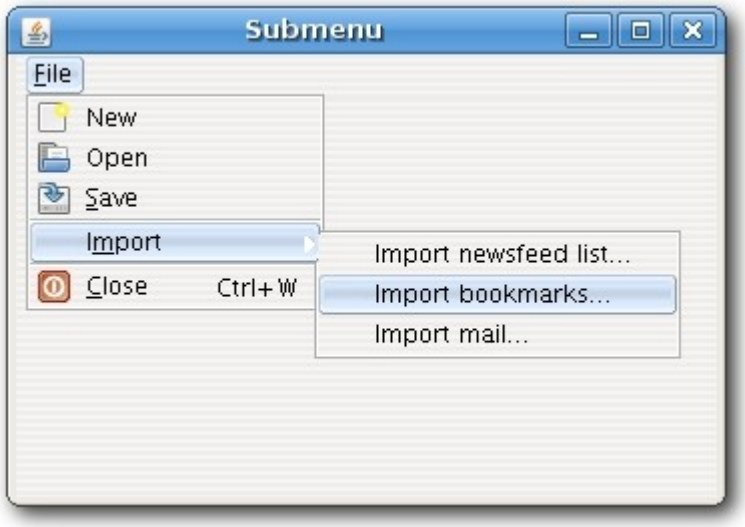


Figure: Submenu

JCheckBoxMenuItem

A menu item that can be selected or deselected. If selected, the menu item typically appears with a checkmark next to it. If unselected or deselected, the menu item appears without a checkmark. Like a regular menu item, a check box menu item can have either text or a graphic icon associated with it, or both.

```
import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;
```

```
import java.awt.event.KeyEvent;



import javax.swing.BorderFactory;

import javax.swing.JCheckBoxMenuItem;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.UIManager;

import javax.swing.border.EtchedBorder;




public class CheckMenuItem extends JFrame {



    private JLabel statusbar;


    public CheckMenuItem() {


        setTitle("CheckBoxMenuItem");
```

```
        JMenuBar menubar = new JMenuBar();

        JMenu file = new JMenu("File");

        file.setMnemonic(KeyEvent.VK_F);



        JMenu view = new JMenu("View");

        view.setMnemonic(KeyEvent.VK_V);



        JCheckBoxMenuItem sbar = new JCheckBoxMenuItem("Show StatuBar");

        sbar.setState(true);



        sbar.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                if (statusbar.isVisible()) {

                    statusbar.setVisible(false);

                } else {

                    statusbar.setVisible(true);

                }

            }



        });
```

```
        view.add(sbar);


        menubar.add(file);
        menubar.add(view);


        setJMenuBar(menubar);


        statusbar = new JLabel(" Statusbar");
        statusbar.setBorder(BorderFactory.createEtchedBorder(
             EtchedBorder.RAISED));
        add(statusbar, BorderLayout.SOUTH);




        setSize(360, 250);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    public static void main(String[] args) {
        new CheckMenuItem();
```

```
    }

 }
```

The example shows a **JCheckBoxMenuItem.**. By selecting the menu item, we toggle the visibility of the statusbar.

```
JCheckBoxMenuItem sbar = new JCheckBoxMenuItem("Show StatuBar");

sbar.setState(true);
```

We create the **JCheckBoxMenuItem** and check it by default. The statusbar is initially visible.

```
if (statusbar.isVisible()) {

        statusbar.setVisible(false);

} else {

        statusbar.setVisible(true);

}
```

Here we toggle the visibility of the statusbar.

```
statusbar = new JLabel(" Statusbar");

statusbar.setBorder(BorderFactory.createEtchedBorder(EtchedBorder.RAISED));
```

The statusbar is a simple **JLabel** component. We put a raised **EtchedBorder** around the label, so that it is discernible.

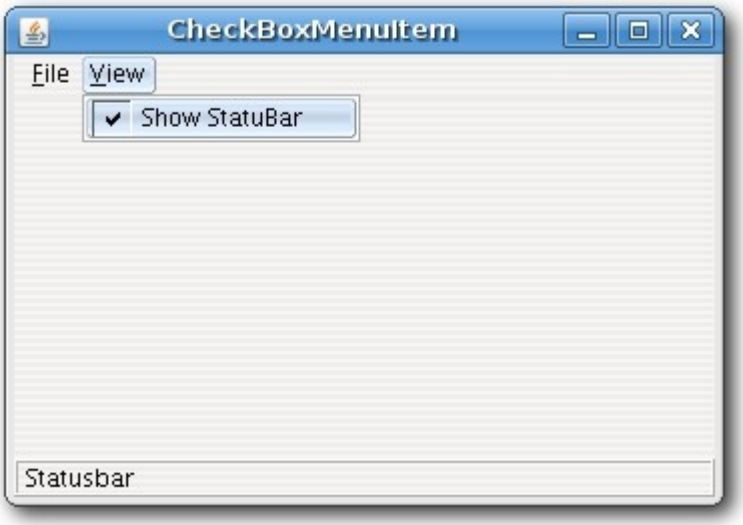


Figure: JCheckBoxMenuItem

A popup menu

Another type of a menu is a popup menu. It is sometimes called a context menu. It is usually shown, when we right click on a component. The idea is to provide only the commands, that are relevant to the current context. Say we have an image. By right clicking on the image, we get a window with commands to save, rescale, move etc the image.

```
import java.awt.Toolkit;


import javax.swing.*;

import java.awt.event.*;


public class PopupMenu {
```

```
    private JPopupMenu menu;

    private Toolkit toolkit;


  public PopupMenu(){



    JFrame frame = new JFrame("JPopupMenu");

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


    toolkit = frame.getToolkit();


    menu = new JPopupMenu();

    JMenuItem menuItemBeep = new JMenuItem("Beep");


    menuItemBeep.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {

            toolkit.beep();

        }

    });


    menu.add(menuItemBeep);
```

```
JMenuItem menuItemClose = new JMenuItem("Close");
menuItemClose.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }

});

menu.add(menuItemClose);

frame.addMouseListener(new MouseAdapter() {
    public void mouseReleased(MouseEvent e) {
        if (e.getButton() == e.BUTTON3) {
            menu.show(e.getComponent(), e.getX(), e.getY());
        }
    }
});

frame.setSize(250, 200);
frame.setLocationRelativeTo(null);
```

```
    frame.setVisible(true);

  }

  public static void main(String[] args) {

        new PopupMenu();

  }

}
```

Our example shows a demonstrational popup menu with two commands. The first option of the popup menu will beep a sound, the second one will close the window.

In our example, we create a submenu, menu separators and create an accelerator key.

```
menu = new JPopupMenu();
```

To create a popup menu, we have a class called **JPopupMenu**.

```
JMenuItem menuItemBeep = new JMenuItem("Beep");
```

The menu item is the same, as with the standard **JMenu**

```
frame.addMouseListener(new MouseAdapter() {

    public void mouseReleased(MouseEvent e) {

        if (e.getButton() == e.BUTTON3) {

            menu.show(e.getComponent(), e.getX(), e.getY());

        }
```

```
    }


});
```

The popup menu is shown, where we clicked with the mouse button.

The **BUTTON3** constant is here to enable the popup menu only for the mouse right click.

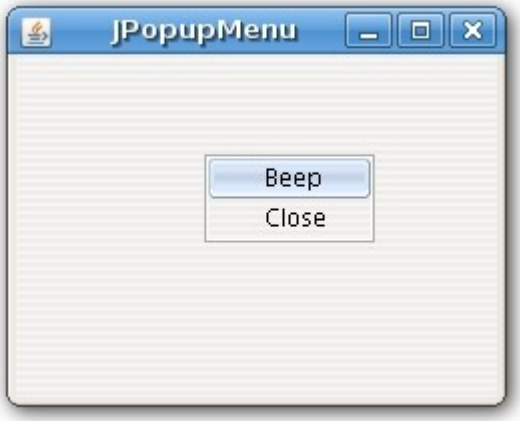


Figure: Popup menu

JToolbar

Menus group commands that we can use in an application. Toolbars provide a quick access to the most frequently used commands. In Java Swing, the **JToolBar** class creates a toolbar in an application.

```
import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.ImageIcon;

import javax.swing.JButton;
```

```
import javax.swing.JFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JToolBar;



public class SimpleToolbar extends JFrame {



    public SimpleToolbar() {


        setTitle("SimpleToolbar");


        JMenuBar menubar = new JMenuBar();

        JMenu file = new JMenu("File");

        menubar.add(file);

        setJMenuBar(menubar);


        JToolBar toolbar = new JToolBar();
```

```
        ImageIcon icon = new
ImageIcon(getClass().getResource("exit.png"));



        JButton exit = new JButton(icon);

        toolbar.add(exit);

        exit.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                System.exit(0);

            }



        });




        add(toolbar, BorderLayout.NORTH);



        setSize(300, 200);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }
```

```
    public static void main(String[] args) {

        new SimpleToolbar();

    }

}
```

The example creates a toolbar with one exit button.

```
JToolBar toolbar = new JToolBar();
```

This is the **JToolBar** constructor.

```
JButton exit = new JButton(icon);


toolbar.add(exit);
```

We create a button and add it to the toolbar.

Figure: Simple JToolBar

Toolbars

Say, we wanted to create two toolbars. The next example shows, how we could do it.

```
import java.awt.BorderLayout;
```

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.BoxLayout;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JToolBar;




public class Toolbars extends JFrame {


    public Toolbars() {


        setTitle("Toolbars");


        JToolBar toolbar1 = new JToolBar();

        JToolBar toolbar2 = new JToolBar();


        JPanel panel = new JPanel();
```

```
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));


        ImageIcon newi = new ImageIcon(
             getClass().getResource("new.png"));
        ImageIcon open = new ImageIcon(
             getClass().getResource("open.png"));
        ImageIcon save = new ImageIcon(
             getClass().getResource("save.png"));
        ImageIcon exit = new ImageIcon(
             getClass().getResource("exit.png"));


        JButton newb = new JButton(newi);
        JButton openb = new JButton(open);
        JButton saveb = new JButton(save);


        toolbar1.add(newb);
        toolbar1.add(openb);
        toolbar1.add(saveb);
        toolbar1.setAlignmentX(0);


        JButton exitb = new JButton(exit);
```

```
        toolbar2.add(exitb);

        toolbar2.setAlignmentX(0);



        exitb.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                System.exit(0);

            }



        });



        panel.add(toolbar1);

        panel.add(toolbar2);



        add(panel, BorderLayout.NORTH);



        setSize(300, 200);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }
```

```
    public static void main(String[] args) {

        new Toolbars();

    }

}
```

We show only one way, how we could create toolbars. Of course, there are several possibilities. We put a **JPanel** to the north of the **BorderLayout** manager. The panel has a vertical **BoxLayout**. We add the two toolbars into this panel.

```
JToolBar toolbar1 = new JToolBar();


JToolBar toolbar2 = new JToolBar();
```

Creation of two toolbars.

```
JPanel panel = new JPanel();


panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
```

The panel has a vertical **BoxLayout**.

```
toolbar1.setAlignmentX(0);
```

The toolbar is left aligned.

```
panel.add(toolbar1);


panel.add(toolbar2);




add(panel, BorderLayout.NORTH);
```

We add the toolbars to the panel. Finally, the panel is located into the north part of the frame.

Figure: Toolbars

A vertical toobar

The following example shows a vertical toobar.

```
import java.awt.BorderLayout;



import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JToolBar;

import javax.swing.UIManager;




public class VerticalToolbar extends JFrame {
```

```
public VerticalToolbar() {

    setTitle("Vertical toolbar");

    JToolBar toolbar = new JToolBar(JToolBar.VERTICAL);

    ImageIcon select = new ImageIcon(
         getClass().getResource("select.gif"));
    ImageIcon freehand = new ImageIcon(
         getClass().getResource("freehand.gif"));
    ImageIcon shapeed = new ImageIcon(
         getClass().getResource("shapeed.gif"));
    ImageIcon pen = new ImageIcon(
         getClass().getResource("pen.gif"));
    ImageIcon rectangle = new ImageIcon(
         getClass().getResource("rectangle.gif"));
    ImageIcon ellipse = new ImageIcon(
         getClass().getResource("ellipse.gif"));
    ImageIcon qs = new ImageIcon(
         getClass().getResource("qs.gif"));
```

```
        ImageIcon text = new ImageIcon(

             getClass().getResource("text.gif"));



        JButton selectb = new JButton(select);

        JButton freehandb = new JButton(freehand);

        JButton shapeedb = new JButton(shapeed);

        JButton penb = new JButton(pen);

        JButton rectangleb = new JButton(rectangle);

        JButton ellipseb = new JButton(ellipse);

        JButton qsb = new JButton(qs);

        JButton textb = new JButton(text);



        toolbar.add(selectb);

        toolbar.add(freehandb);

        toolbar.add(shapeedb);

        toolbar.add(penb);

        toolbar.add(rectangleb);

        toolbar.add(ellipseb);

        toolbar.add(qsb);

        toolbar.add(textb);
```

```
        add(toolbar, BorderLayout.WEST);



        setSize(250, 350);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }



    public static void main(String[] args) {

        try {

            UIManager.setLookAndFeel(

                UIManager.getSystemLookAndFeelClassName());

        }

        catch (Exception e) {

            System.out.println("Error:" + e.getStackTrace());

        }

        new VerticalToolbar();

    }

}
```

In the example, we put a vertical toolbar to the left side of the window. This is typical for a graphics applications like**Xara Extreme** or **Inkscape**. In our example, we use icons from Xara Extreme application.

```
JToolBar toolbar = new JToolBar(JToolBar.VERTICAL);
```

We create a vertical toolbar.

```
add(toolbar, BorderLayout.WEST);
```

The toolbar is placed into the left part of the window.

```
UIManager.setLookAndFeel(


    UIManager.getSystemLookAndFeelClassName());
```

I used the system look and feel. In my case, it was the gtk theme. The icons are not transparent, and they would not look ok on a different theme.

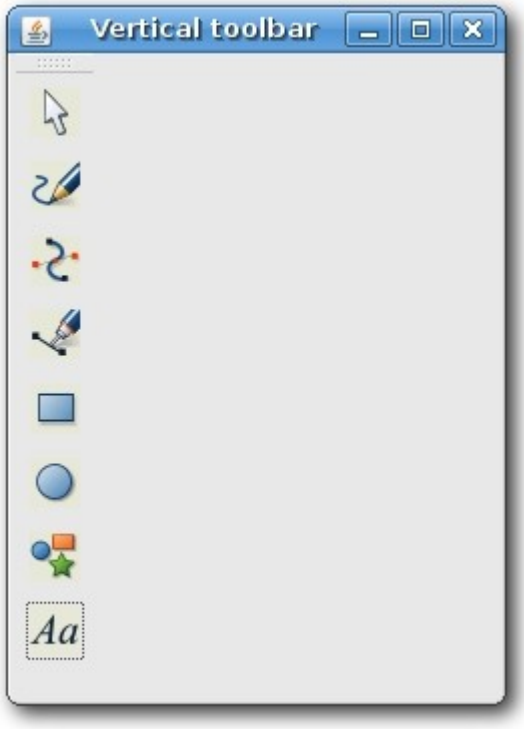


Figure: Vertical toolbar

The Swing layout management

The Java Swing toolkit has two kind of components. Containers and children. The containers group children into suitable layouts. To create layouts, we use **layout managers**. Layout managers are one of the most difficult parts of modern GUI programming. Many beginning programmers have too much respect for layout managers. Mainly because they are usually poorly documented. I believe, that GUI builder's like Matisse cannot replace the proper understanding of layout managers.

ZetCode offers a dedicated 196 pages e-book for the Swing layout management process: Java Swing layout management tutorial

No manager

We can use no layout manager, if we want. There might be situations, where we might not need a layout manager. For example, in my code examples, I often go without a manager. It is because I did not want to make the examples too complex. But to create truly portable, complex applications, we need layout managers.

Without layout manager, we position components using absolute values.

```
import javax.swing.JButton;

import javax.swing.JFrame;




public class Absolute extends JFrame {




    public Absolute() {


        setTitle("Absolute positioning");
```

```
        setLayout(null);


        JButton ok = new JButton("OK");
        ok.setBounds(50, 150, 80, 25);


        JButton close = new JButton("Close");
        close.setBounds(150, 150, 80, 25);


        add(ok);
        add(close);


        setSize(300, 250);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }


    public static void main(String[] args) {
        new Absolute();
    }
```

```
}
```

This simple example shows two buttons.

```
setLayout(null);
```

We use absolute positioning by providing null to the **setLayout()** method.

```
ok.setBounds(50, 150, 80, 25);
```

The **setBounds()** method positions the ok button. The parameters are the x, y location values and the width and height of the component.

FlowLayout manager

This is the simplest layout manager in the Java Swing toolkit. It is mainly used in combination with other layout managers. When calculating it's children size, a flow layout lets each component assume its natural (preferred) size.

The manager puts components into a row. In the order, they were added. If they do not fit into one row, they go into the next one. The components can be added from the right to the left or vice versa. The manager allows to align the components. Implicitly, the components are centered and there is 5px space among components and components and the edges of the container.

```
FlowLayout()

FlowLayout(int align)

FlowLayout(int align, int hgap, int vgap)
```

There are three constructors available for the FlowLayout manager. The first one creates a manager with implicit values. Centered and 5px spaces. The others allow to specify those parametes.

```
import java.awt.Dimension;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTree;



public class FlowLayoutExample extends JFrame {



    public FlowLayoutExample() {

        setTitle("FlowLayout Example");

        JPanel panel = new JPanel();

        JTextArea area = new JTextArea("text area");
        area.setPreferredSize(new Dimension(100, 100));

        JButton button = new JButton("button");
```

```
        panel.add(button);



        JTree tree = new JTree();

        panel.add(tree);



        panel.add(area);



        add(panel);



        pack();



        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLocationRelativeTo(null);

        setVisible(true);

    }



    public static void main(String[] args) {

        new FlowLayoutExample();

    }

}
```

The example shows a button, a tree and a text area component in the window. Interesingly, if we create an empty tree component, there are some default values inside the component.

```
JPanel panel = new JPanel();
```

The implicit layout manager of the **JPanel** component is a flow layout manager. We do not have to set it manually.

```
JTextArea area = new JTextArea("text area");


area.setPreferredSize(new Dimension(100, 100));
```

The flow layout manager sets a **preferred** size for it's components. This means, that in our case, the area component will have 100x100 px. If we didn't set the preferred size, the component would have a size of it's text. No more, no less. Without the text, the component would not be visible at all. Try to write or delete some text in the area component. The component will grow and shrink accordingly.

```
panel.add(area);
```

To put a component inside a container, we simply call the **add()** method.

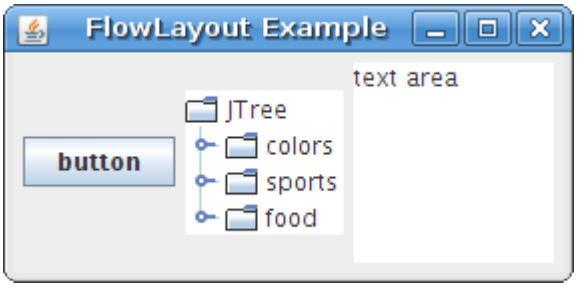


Figure: FlowLaout manager

GridLayout

The **GridLayout** layout manager lays out components in a rectangular grid. The container is divided into equally sized rectangles. One component is placed in each rectangle.

```
import java.awt.GridLayout;
```

```
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;



public class GridLayoutExample extends JFrame {

    public GridLayoutExample() {

        setTitle("GridLayout");

        JPanel panel = new JPanel();

        panel.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
        panel.setLayout(new GridLayout(5, 4, 5, 5));

        String[] buttons = {
            "Cls", "Bck", "", "Close", "7", "8", "9", "/", "4",
```

```
            "5", "6", "*", "1", "2", "3", "-", "0", ".", "=", "+"
        };



        for (int i = 0; i < buttons.length; i++) {


            if (i == 2)
                panel.add(new JLabel(buttons[i]));
            else
                panel.add(new JButton(buttons[i]));
        }


        add(panel);


        setSize(350, 300);


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


    }
```

```
    public static void main(String[] args) {

        new GridLayoutExample();

    }

}
```

The example shows a skeleton of a simple calculator tool. It is an ideal example for this layout manager. We put 19 buttons and one label into the manager. Notice, that each button is of the same size.

```
panel.setLayout(new GridLayout(5, 4, 5, 5));
```

Here we set the grid layout manager for the panel component. The layout manager takes four parameters. The number of rows, the number of columns and the horizontal and vertical gaps between components.

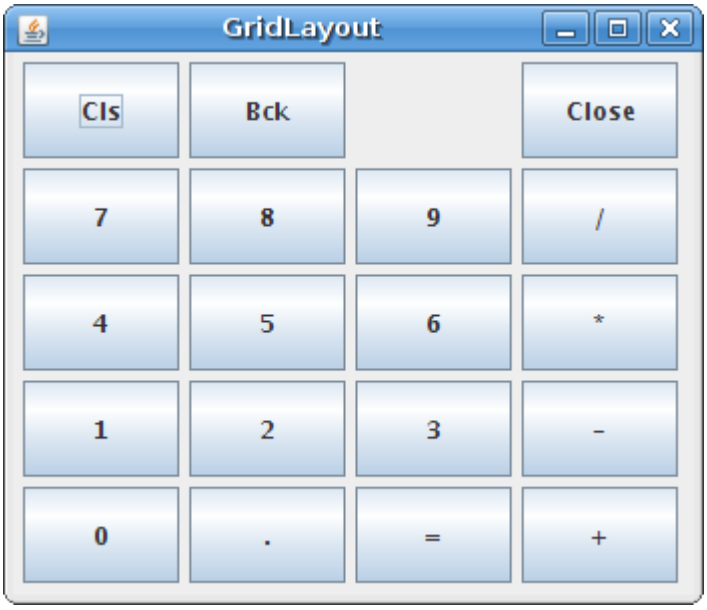


Figure: GridLayout manager

BorderLayout

A **BorderLayout** manager is a very handy layout manager. I haven't seen it elsewhere. It divides the space into five regions. North, West, South, East and Centre. Each region can have only one component. If we need to put more components into a region, we can simply put a panel there with a manager of our choice. The components in N, W, S, E regions get their **preferred** size. The component in the centre takes up the whole space left.

It does not look good, if child components are too close to each other. We must put some space among them. Each component in Swing toolkit can have borders around it's edges. To create a border, we either create a new instance of an **EmptyBorder** class or we use a **BorderFactory**. Except for EmptyBorder, there are other types of borders as well. But for layout management we will use only this one.

```
import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Insets;



import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.border.EmptyBorder;




public class BorderExample extends JFrame {




    public BorderExample() {
```

```
        setTitle("Border Example");


        JPanel panel = new JPanel(new BorderLayout());
        JPanel top = new JPanel();


        top.setBackground(Color.gray);
        top.setPreferredSize(new Dimension(250, 150));
        panel.add(top);


        panel.setBorder(new EmptyBorder(new Insets(20, 20, 20, 20)));


        add(panel);


        pack();


        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }
```

```
    public static void main(String[] args) {

        new BorderExample();

    }

}
```

The example will display a gray panel and border around it.

```
JPanel panel = new JPanel(new BorderLayout());


JPanel top = new JPanel();
```

We place a panel into a panel. We used a **BorderLayout** manager for the first panel, because this manager will resize it's children.

```
panel.add(top);
```

Here we placed a top panel into the panel component. More precisely, we placed into the center area of the**BorderLayout** manager.

```
panel.setBorder(new EmptyBorder(new Insets(20, 20, 20, 20)));
```

Here we created a 20px border around the panel. The border values are as follows: top, left, bottom and right. They go counterclockwise.

Figure: Border example

The next example will show a typical usage of a border layout manager.

```
import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.Insets;



import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JTextArea;

import javax.swing.JToolBar;

import javax.swing.border.EmptyBorder;

import javax.swing.border.LineBorder;




public class BorderLayoutExample extends JFrame {




    public BorderLayoutExample() {
```

```
        setTitle("BorderLayout");


        JMenuBar menubar = new JMenuBar();

        JMenu file = new JMenu("File");


        menubar.add(file);

        setJMenuBar(menubar);


        JToolBar toolbar = new JToolBar();

        toolbar.setFloatable(false);


        ImageIcon exit = new ImageIcon("exit.png");

        JButton bexit = new JButton(exit);

        bexit.setBorder(new EmptyBorder(0 ,0, 0, 0));

        toolbar.add(bexit);


        add(toolbar, BorderLayout.NORTH);


        JToolBar vertical = new JToolBar(JToolBar.VERTICAL);

        vertical.setFloatable(false);
```

```
        vertical.setMargin(new Insets(10, 5, 5, 5));


        ImageIcon select = new ImageIcon("drive.png");

        ImageIcon freehand = new ImageIcon("computer.png");

        ImageIcon shapeed = new ImageIcon("printer.png");


        JButton selectb = new JButton(select);

        selectb.setBorder(new EmptyBorder(3, 0, 3, 0));


        JButton freehandb = new JButton(freehand);

        freehandb.setBorder(new EmptyBorder(3, 0, 3, 0));

        JButton shapeedb = new JButton(shapeed);

        shapeedb.setBorder(new EmptyBorder(3, 0, 3, 0));


        vertical.add(selectb);

        vertical.add(freehandb);

        vertical.add(shapeedb);


        add(vertical, BorderLayout.WEST);


        add(new JTextArea(), BorderLayout.CENTER);
```

```
        JLabel statusbar = new JLabel(" Statusbar");
        statusbar.setPreferredSize(new Dimension(-1, 22));
        statusbar.setBorder(LineBorder.createGrayLineBorder());
        add(statusbar, BorderLayout.SOUTH);


        setSize(350, 300);


        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }


    public static void main(String[] args) {
        new BorderLayoutExample();
    }
}
```

The example shows a typical application skeleton. We show a vertical and horizontal toolbars, a statusbar and a central component. (a text area)

A default layout manager for a **JFrame** component is **BorderLayout** manager. So we don't have to set it.

```
add(toolbar, BorderLayout.NORTH);
```

Simply put the toolbar to the north of the layout.

```
add(vertical, BorderLayout.WEST);
```

Put the vertical toobar to the west.

```
add(new JTextArea(), BorderLayout.CENTER);
```

Put the text area to the center.

```
add(statusbar, BorderLayout.SOUTH);
```

Put the statusbar to the south.

That's it.

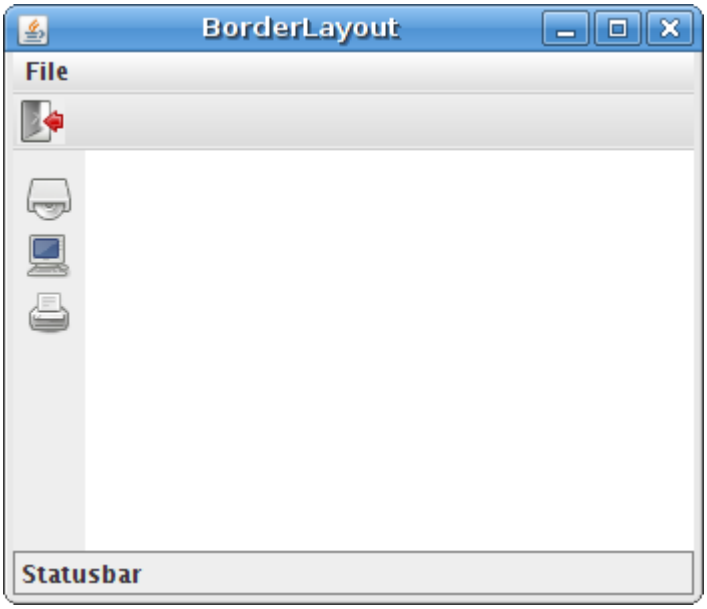


Figure: BorderLayout manager

BoxLayout

BoxLayout is a powerful manager, that can be used to create sofisticated layouts. This layout manager puts components into a row or into a column. It enables **nesting**, a powerful feature, that makes this manager very flexible. It means, that we can put a box layout into another box layout.

The box layout manager is often used with the **Box** class. This class creates several invisible components, which affect the final layout.

- glue
- strut
- rigid area

Let's say, we want to put two buttons into the right bottom corner of the window. We will use the boxlayut managers to accomplish this.

```
import java.awt.Dimension;



import javax.swing.Box;

import javax.swing.BoxLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;




public class TwoButtons extends JFrame {


    public TwoButtons() {


        setTitle("Two Buttons");
```

```
        JPanel basic = new JPanel();

        basic.setLayout(new BoxLayout(basic, BoxLayout.Y_AXIS));

        add(basic);



        basic.add(Box.createVerticalGlue());



        JPanel bottom = new JPanel();

        bottom.setAlignmentX(1f);

        bottom.setLayout(new BoxLayout(bottom, BoxLayout.X_AXIS));



        JButton ok = new JButton("OK");

        JButton close = new JButton("Close");



        bottom.add(ok);

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

        bottom.add(close);

        bottom.add(Box.createRigidArea(new Dimension(15, 0)));



        basic.add(bottom);

        basic.add(Box.createRigidArea(new Dimension(0, 15)));
```

```
        setSize(300, 250);



        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLocationRelativeTo(null);

        setVisible(true);



    }



    public static void main(String[] args) {

        new TwoButtons();

    }

}
```

The following drawing illustrates the example.

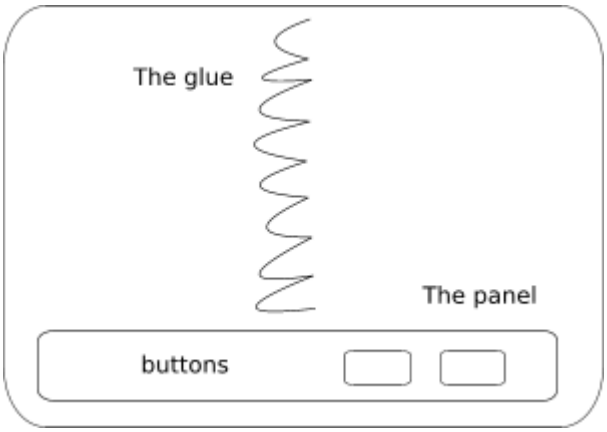


Figure: Two buttons

We will create two panels. The basic panel has a vertical box layout. The bottom panel has a horizontal one. We will put a bottom panel into the basic panel. We will right align the

bottom panel. The space between the top of the window and the bottom panel is expandable. It is done by the vertical glue.

```
basic.setLayout(new BoxLayout(basic, BoxLayout.Y_AXIS));
```

Here we create a basic panel with the vertical **BoxLayout**.

```
JPanel bottom = new JPanel();

bottom.setAlignmentX(1f);

bottom.setLayout(new BoxLayout(bottom, BoxLayout.X_AXIS));
```

The bottom panel is right aligned. This is done by the **setAlignmentX()** method. The panel has a horizontal layout.

```
bottom.add(Box.createRigidArea(new Dimension(5, 0)));
```

We put some rigid space between the buttons.

```
basic.add(bottom);
```

Here we put the bottom panel with a horizontal box layout to the vertical basic panel.

```
basic.add(Box.createRigidArea(new Dimension(0, 15)));
```

We also put some space between the bottom panel and the border of the window.

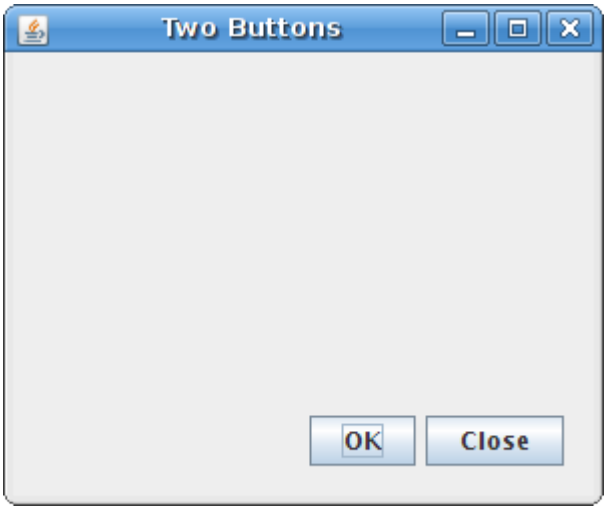


Figure: Two buttons

When we use a **BoxLayout** manager, we can set a rigid area among our components.

```
import java.awt.Dimension;



import java.awt.Insets;



import javax.swing.Box;

import javax.swing.BoxLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.border.EmptyBorder;




public class RigidArea extends JFrame {


    public RigidArea() {


        setTitle("RigidArea");


        JPanel panel = new JPanel();
```

```
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));


        panel.setBorder(new EmptyBorder(new Insets(40, 60, 40, 60)));


        panel.add(new JButton("Button"));
        panel.add(Box.createRigidArea(new Dimension(0, 5)));
        panel.add(new JButton("Button"));
        panel.add(Box.createRigidArea(new Dimension(0, 5)));
        panel.add(new JButton("Button"));
        panel.add(Box.createRigidArea(new Dimension(0, 5)));
        panel.add(new JButton("Button"));


        add(panel);


        pack();


        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }
```

```
    public static void main(String[] args) {

        new RigidArea();

    }

}
```

In this example, we display four buttons. By default, there is no space among the buttons. To put some space among them, we add some rigid area.

```
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
```

We use a vertical **BoxLayout** manager for our panel.

```
panel.add(new JButton("Button"));

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

panel.add(new JButton("Button"));
```

We add buttons and create a rigid area in between them.

Figure: Rigid area

Tip of the Day

JDeveloper has a dialog window called Tip of the Day.

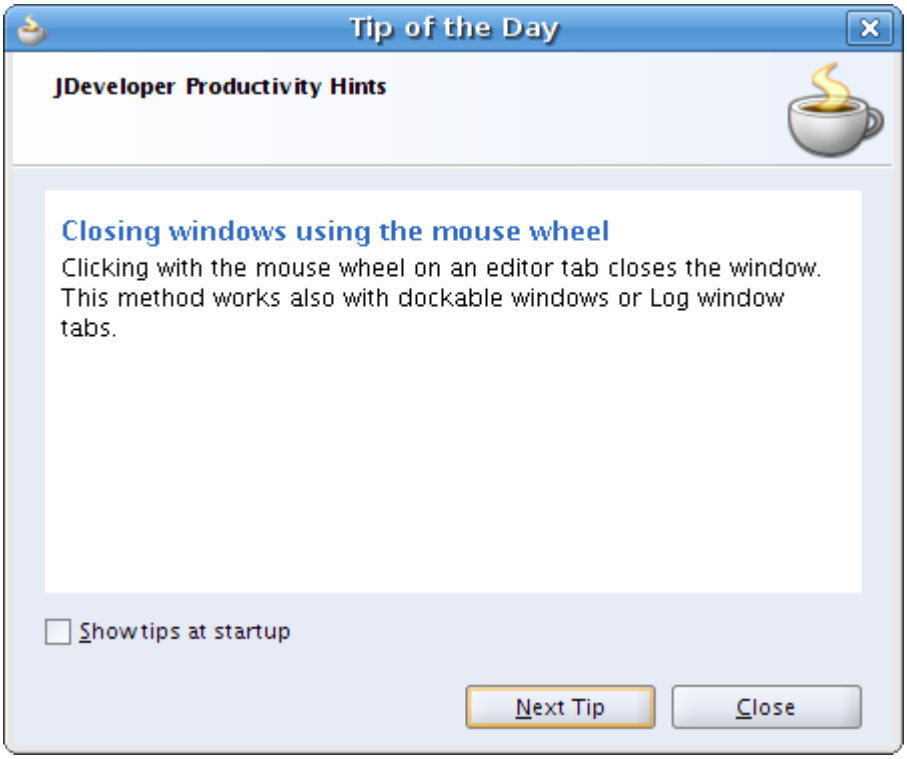


Figure: Tip of the Day

We will create a similar dialog. We will use a combination of various layout managers. Namely a border layout, flow layout and box layout manager.

```
import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.FlowLayout;

import java.awt.event.KeyEvent;



import javax.swing.BorderFactory;
```

```
import javax.swing.BoxLayout;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JCheckBox;

import javax.swing.JDialog;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JSeparator;

import javax.swing.JTextPane;




public class TipOfDay extends JDialog {



    public TipOfDay() {


        setTitle("Tip of the Day");


        JPanel basic = new JPanel();

        basic.setLayout(new BoxLayout(basic, BoxLayout.Y_AXIS));

        add(basic);
```

```
JPanel topPanel = new JPanel(new BorderLayout(0, 0));
topPanel.setMaximumSize(new Dimension(450, 0));
JLabel hint = new JLabel("JDeveloper Productivity Hints");
hint.setBorder(BorderFactory.createEmptyBorder(0, 25, 0, 0));
topPanel.add(hint);

ImageIcon icon = new ImageIcon("jdev.png");
JLabel label = new JLabel(icon);
label.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
topPanel.add(label, BorderLayout.EAST);

JSeparator separator = new JSeparator();
separator.setForeground(Color.gray);

topPanel.add(separator, BorderLayout.SOUTH);

basic.add(topPanel);

JPanel textPanel = new JPanel(new BorderLayout());
```

```
        textPanel.setBorder(BorderFactory.createEmptyBorder(15, 25, 15,
25));

        JTextPane pane = new JTextPane();



        pane.setContentType("text/html");

        String text = "<p><b>Closing windows using the mouse
wheel</b></p>" +

            "<p>Clicking with the mouse wheel on an editor tab closes the
window. " +

            "This method works also with dockable windows or Log window
tabs.</p>";

        pane.setText(text);

        pane.setEditable(false);

        textPanel.add(pane);



        basic.add(textPanel);



        JPanel boxPanel = new JPanel(new FlowLayout(FlowLayout.LEFT, 20,
0));



        JCheckBox box = new JCheckBox("Show Tips at startup");

        box.setMnemonic(KeyEvent.VK_S);



        boxPanel.add(box);
```

```
        basic.add(boxPanel);


        JPanel bottom = new JPanel(new FlowLayout(FlowLayout.RIGHT));


        JButton ntip = new JButton("Next Tip");
        ntip.setMnemonic(KeyEvent.VK_N);
        JButton close = new JButton("Close");
        close.setMnemonic(KeyEvent.VK_C);


        bottom.add(ntip);
        bottom.add(close);
        basic.add(bottom);


        bottom.setMaximumSize(new Dimension(450, 0));


        setSize(new Dimension(450, 350));
        setResizable(false);
        setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }
```

```
    public static void main(String[] args) {

        new TipOfDay();

    }

}
```

The example uses a mix of layout managers. Simply we put four panels into the vertically organized basic panel.

```
JPanel basic = new JPanel();

basic.setLayout(new BoxLayout(basic, BoxLayout.Y_AXIS));

add(basic);
```

This is the very bottom panel. It has a vertical box layout manager. The basic panel is added to the default **JDialog**component. This component has a border layout manager by default.

```
JPanel topPanel = new JPanel(new BorderLayout(0, 0));
```

The topPanel panel has a border layout manager. We will put three components into it. Two labels and a separator.

```
topPanel.setMaximumSize(new Dimension(450, 0));
```

If we want to have a panel, that is not greater than it's components, we must set it's maximum size. The zero value is ignored. The manager calculates the necessary heights.

```
JPanel textPanel = new JPanel(new BorderLayout());

...

textPanel.add(pane);
```

The text pane component is added to the center area of the border layout manager. It takes all space left. Exactly, as we wanted.

```
JPanel boxPanel = new JPanel(new FlowLayout(FlowLayout.LEFT, 20, 0));
```

The check box is shown in the boxPanel panel. It is left aligned. The flow layout manager has a 20px horizontal gap. Other components have 25px. Why is that? It is because the flow layout manager puts some space to between the component and the edge as well.

```
JPanel bottom = new JPanel(new FlowLayout(FlowLayout.RIGHT));


...


bottom.setMaximumSize(new Dimension(450, 0));
```

The bottom panel displays two buttons. It has a right aligned flow layout manager. In order to show the buttons on the right edge of the dialog, the panel must stretch horizontally from the beginning to the end.

Java Swing Events

Events are an important part in any GUI program. All GUI applications are event-driven. An application reacts to different event types which are generated during it's life. Events are generated mainly by the user of an application. But they can be generated by other means as well. e.g. internet connection, window manager, timer. In the event model, there are three participants:

- event source
- event object
- event listener

The **Event source** is the object whose state changes. It generates Events. The **Event object** (Event) encapsulates the state changes in the event source. The **Event listener** is the object that wants to be notified. Event source object delegates the task of handling an event to the event listener.

Event handling in Java Swing toolkit is very powerful and flexible. Java uses Event Delegation Model. We specify the objects that are to be notified when a specific event occurs.

An event object

When something happens in the application, an event object is created. For example, when we click on the button or select an item from list. There are several types of events. An **ActionEvent**, **TextEvent**, **FocusEvent**,**ComponentEvent** etc. Each of them is created under specific conditions.

Event object has information about an event, that has happened. In the next example, we will analyze an **ActionEvent**in more detail.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import java.text.DateFormat;



import java.util.Calendar;

import java.util.Date;

import java.util.Locale;



import javax.swing.DefaultListModel;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JList;
```

```
import javax.swing.JPanel;



public class EventObject extends JFrame {



    private JList list;

    private DefaultListModel model;


    public EventObject() {


        setTitle("Event Object");


        JPanel panel = new JPanel();

        panel.setLayout(null);


        model = new DefaultListModel();

        list = new JList(model);

        list.setBounds(150, 30, 220, 150);


        JButton ok = new JButton("Ok");
```

```
        ok.setBounds(30, 35, 80, 25);


        ok.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {


                Locale locale = Locale.getDefault();

                Date date = new Date();

                String s = DateFormat.getTimeInstance(DateFormat.SHORT,

                    locale).format(date);


                if ( !model.isEmpty() )

                    model.clear();


                if (event.getID() == ActionEvent.ACTION_PERFORMED)

                    model.addElement(" Event Id: ACTION_PERFORMED");


                model.addElement(" Time: " + s);


                String source = event.getSource().getClass().getName();

                model.addElement(" Source: " + source);
```

```
            int mod = event.getModifiers();


            StringBuffer buffer = new StringBuffer(" Modifiers: ");


            if ((mod & ActionEvent.ALT_MASK) > 0)
                buffer.append("Alt ");


            if ((mod & ActionEvent.SHIFT_MASK) > 0)
                buffer.append("Shift ");


            if ((mod & ActionEvent.META_MASK) > 0)
                buffer.append("Meta ");


            if ((mod & ActionEvent.CTRL_MASK) > 0)
                buffer.append("Ctrl ");


            model.addElement(buffer);
        }
    });


    panel.add(ok);
```

```
        panel.add(list);

        add(panel);



        setSize(420, 250);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }



    public static void main(String[] args) {

        new EventObject();

    }

}
```

The code example shows a button and a list. If we click on the button, information about the event is displayed in the list. In our case, we are talking about an **ActionEvent** class. The data will be the time, when the event occured, the id of the event, the event source and the modifier keys.

```
public void actionPerformed(ActionEvent event) {
```

Inside the action listener, we have an event parameter. It is the instance of the event, that has occured. In our case it is an **ActionEvent**.

```
cal.setTimeInMillis(event.getWhen());
```

Here we get the time, when the event occured. The method returns time value in milliseconds. So we must format it appropriately.

```
String source = event.getSource().getClass().getName();


model.addElement(" Source: " + source);
```

Here we add the name of the source of the event to the list. In our case the source is a **JButton**.

```
int mod = event.getModifiers();
```

We get the modifier keys. It is a bitwise-or of the modifier constants.

```
if ((mod & ActionEvent.SHIFT_MASK) > 0)


    buffer.append("Shift ");
```

Here we determine, whether we have pressed a Shift key.

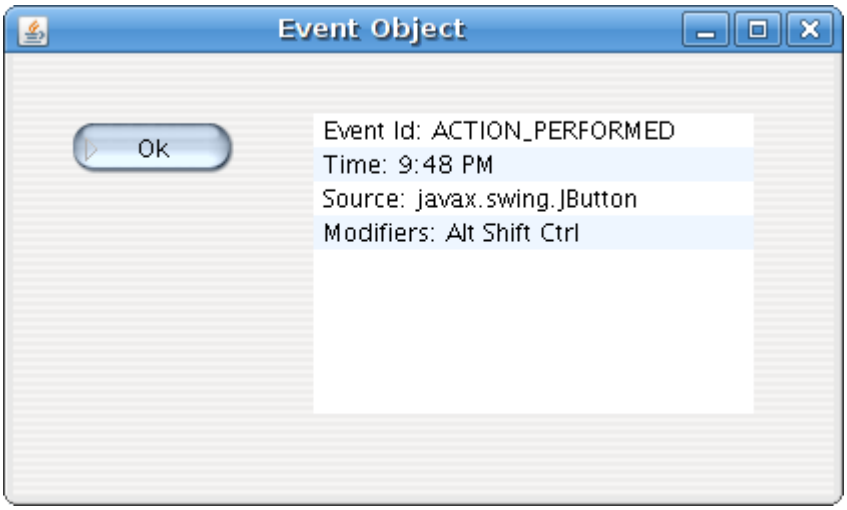


Figure: Event Object

Implementation

There are several ways, how we can implement event handling in Java Swing toolkit.

- Anonymous inner class
- Inner class
- Derived class

Anonymous inner class

We will illustrate these concepts on a simple event example.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;




public class SimpleEvent extends JFrame {



    public SimpleEvent() {


        setTitle("Simle Event");


        JPanel panel = new JPanel();

        panel.setLayout(null);


        JButton close = new JButton("Close");
```

```
        close.setBounds(40, 50, 80, 25);


        close.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.exit(0);
            }


        });


        panel.add(close);
        add(panel);


        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    public static void main(String[] args) {
        new SimpleEvent();
    }
```

```
}
```

In this example, we have a button that closes the window upon clicking.

```
JButton close = new JButton("Close");
```

The button is the **event source**. It will generate events.

```
close.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent event) {

        System.exit(0);

    }

});
```

Here we **register** an action listener with the button. This way, the events are sent to the **event target**. The event target in our case is **ActionListener** class. In this code, we use an **anonymous inner class**.

Inner class

Here we implement the example using an inner **ActionListener** class.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;
```

```
public class InnerClass extends JFrame {


    public InnerClass() {


        setTitle("Using inner class");


        JPanel panel = new JPanel();
        panel.setLayout(null);


        JButton close = new JButton("Close");
        close.setBounds(40, 50, 80, 25);


        ButtonListener listener = new ButtonListener();
        close.addActionListener(listener);


        panel.add(close);
        add(panel);


        setSize(300, 200);
        setLocationRelativeTo(null);
```

```
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e)
        {
            System.exit(0);
        }
      }

    public static void main(String[] args) {
        new InnerClass();
    }
 }
```

```
ButtonListener listener = new ButtonListener();

close.addActionListener(listener);
```

Here we have a non anonymous inner class.

```
class ButtonListener implements ActionListener {

   public void actionPerformed(ActionEvent e)
```

```
    {

        System.exit(0);

    }

}
```

The button listener is defined here.

A derived class implementing the listener

The following example will derive a class from a component and implement an action listener inside the class.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;




public class UsingInterface extends JFrame {




    public UsingInterface() {
```

```
        setTitle("Using inner class");


        JPanel panel = new JPanel();
        panel.setLayout(null);


        MyButton close = new MyButton("Close");
        close.setBounds(40, 50, 80, 25);


        panel.add(close);
        add(panel);


        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    class MyButton extends JButton implements ActionListener {


        public MyButton(String text) {
            super.setText(text);
```

```
            addActionListener(this);

        }


        public void actionPerformed(ActionEvent e)

        {

            System.exit(0);

        }


    }


    public static void main(String[] args) {

        new UsingInterface();

    }

}
```

In this example, we create a MyButton class, which will implement the action listener.

```
MyButton close = new MyButton("Close");
```

Here we create the MyButton custom class.

```
class MyButton extends JButton implements ActionListener {
```

The MyButton class is extended from the **JButton** class. It implements the **ActionListener** interface. This way, the event handling is managed within the MyButton class.

```
addActionListener(this);
```

Here we add the action listener to the MyButton class.

Multiple sources

A listener can be plugged into several sources. This will be explained in the next example.

```
import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.BorderFactory;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.border.EtchedBorder;




public class MultipleSources extends JFrame {


    JLabel statusbar;


    public MultipleSources() {
```

```
        setTitle("Multiple Sources");

        JPanel panel = new JPanel();

        statusbar = new JLabel(" ZetCode");


        statusbar.setBorder(BorderFactory.createEtchedBorder(

                EtchedBorder.RAISED));


        panel.setLayout(null);


        JButton close = new JButton("Close");

        close.setBounds(40, 30, 80, 25);

        close.addActionListener(new ButtonListener());


        JButton open = new JButton("Open");

        open.setBounds(40, 80, 80, 25);

        open.addActionListener(new ButtonListener());


        JButton find = new JButton("Find");

        find.setBounds(40, 130, 80, 25);

        find.addActionListener(new ButtonListener());
```

```
        JButton save = new JButton("Save");
        save.setBounds(40, 180, 80, 25);
        save.addActionListener(new ButtonListener());

        panel.add(close);
        panel.add(open);
        panel.add(find);
        panel.add(save);

        add(panel);
        add(statusbar, BorderLayout.SOUTH);

        setSize(400, 300);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e)
        {
```

```
            JButton o = (JButton) e.getSource();

            String label = o.getText();

            statusbar.setText(" " + label + " button clicked");

        }

      }



    public static void main(String[] args) {

        new MultipleSources();

    }

}
```

We create four buttons and a statusbar. The statusbar will display an informative message upon clicking on the button.

```
close.addActionListener(new ButtonListener());

...

open.addActionListener(new ButtonListener());

...
```

Each button will be registered against a **ButtonListener** class.

```
JButton o = (JButton) e.getSource();

String label = o.getText();
```

Here we determine, which button was pressed.

```
statusbar.setText(" " + label + " button clicked")
```

We update the statusbar.

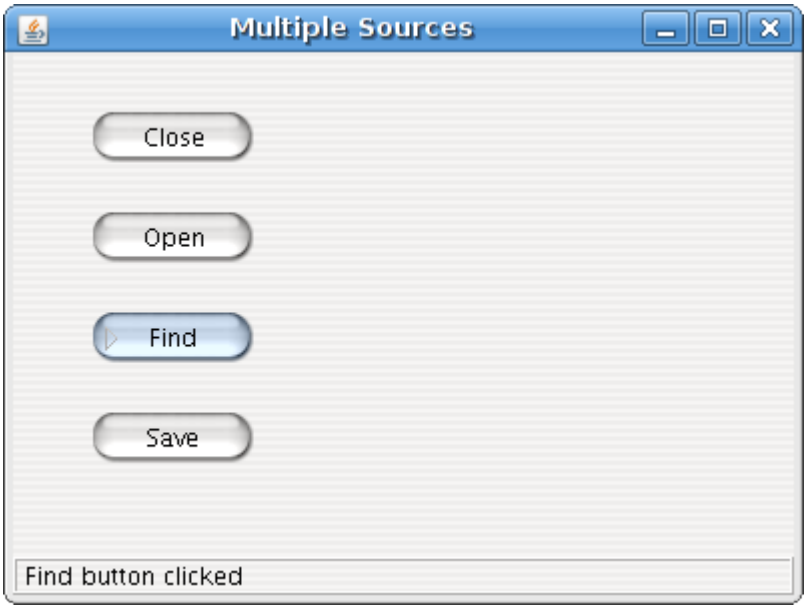


Figure: Multiple Sources

Multiple listeners

We can register several listeners for one event.

```
import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import java.util.Calendar;



import javax.swing.BorderFactory;
```

```
import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JSpinner;

import javax.swing.SpinnerModel;

import javax.swing.SpinnerNumberModel;

import javax.swing.border.EtchedBorder;




public class MultipleListeners extends JFrame {


    private JLabel statusbar;

    private JSpinner spinner;

    private static int count = 0;


    public MultipleListeners() {


        setTitle("Multiple Listeners");

        JPanel panel = new JPanel();

        statusbar = new JLabel("0");
```

```
        statusbar.setBorder(BorderFactory.createEtchedBorder(
                EtchedBorder.RAISED));


        panel.setLayout(null);


        JButton add = new JButton("+");
        add.setBounds(40, 30, 80, 25);
        add.addActionListener(new ButtonListener1());
        add.addActionListener(new ButtonListener2());


        Calendar calendar = Calendar.getInstance();
        int currentYear = calendar.get(Calendar.YEAR);


        SpinnerModel yearModel = new SpinnerNumberModel(currentYear,
                                    currentYear - 100,
                                    currentYear + 100,
                                    1);


        spinner = new JSpinner(yearModel);
        spinner.setEditor(new JSpinner.NumberEditor(spinner, "#"));
```

```
        spinner.setBounds(190, 30, 80, 25);


        panel.add(add);
        panel.add(spinner);


        add(panel);
        add(statusbar, BorderLayout.SOUTH);


        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    class ButtonListener1 implements ActionListener {
        public void actionPerformed(ActionEvent e)
        {
            Integer val = (Integer) spinner.getValue();
            spinner.setValue(++val);
        }
```

```
       }


     class ButtonListener2 implements ActionListener {
         public void actionPerformed(ActionEvent e)
         {
             statusbar.setText(Integer.toString(++count));
         }
       }



     public static void main(String[] args) {
         new MultipleListeners();
     }
 }
```

In this example, we have a button, spinner and a statusbar. We use two button listeners for one event. One click of a button will add one year to the spinner component and update the statusbar. The statusbar will show, how many times we have clicked on the button.

```
add.addActionListener(new ButtonListener1());

add.addActionListener(new ButtonListener2());
```

We register two button listeners.

```
SpinnerModel yearModel = new SpinnerNumberModel(currentYear,

                            currentYear - 100,
```

```
                                currentYear + 100,

                                1);

spinner = new JSpinner(yearModel);
```

Here we create the spinner component. We use a year model for the spinner.

The **SpinnerNumberModel** arguments are initial value, min, max values and the step.

```
spinner.setEditor(new JSpinner.NumberEditor(spinner, "#"));
```

We remove the thousands separator.

```
Integer val = (Integer) spinner.getValue();

spinner.setValue(++val);
```

Here we increase the year number.

Figure: Multiple Listeners

Removing listeners

The Java Swing toolkit enables us to remove the registered listeners.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;
```

```
import java.awt.event.ItemEvent;

import java.awt.event.ItemListener;



import javax.swing.JButton;

import javax.swing.JCheckBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;




public class RemoveListener extends JFrame {



    private JLabel text;

    private JButton add;

    private JCheckBox active;

    private ButtonListener buttonlistener;

    private static int count = 0;



    public RemoveListener() {



        setTitle("Remove listener");
```

```
        JPanel panel = new JPanel();


        panel.setLayout(null);


        add = new JButton("+");
        add.setBounds(40, 30, 80, 25);
        buttonlistener = new ButtonListener();


        active = new JCheckBox("Active listener");
        active.setBounds(160, 30, 140, 25);


        active.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent event) {
                if (active.isSelected()) {
                  add.addActionListener(buttonlistener);}
                else {
                  add.removeActionListener(buttonlistener);
                }
            }
        });
```

```
        text = new JLabel("0");
        text.setBounds(40, 80, 80, 25);


        panel.add(add);
        panel.add(active);
        panel.add(text);


        add(panel);


        setSize(310, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e)
        {
            text.setText(Integer.toString(++count));
        }
      }
```

```
    public static void main(String[] args) {

        new RemoveListener();

    }

}
```

We have three components on the panel. A button, check box and a label. By toggling the check box, we add or remove the listener for a button.

```
buttonlistener = new ButtonListener();
```

We have to create a non anonymous listener, if we want to later remove it. We need a reference to it.

```
if (active.isSelected()) {

    add.addActionListener(buttonlistener);}

else {

    add.removeActionListener(buttonlistener);

}
```

We determine, whether the check box is selected. Then we add or remove the listener.

Figure: Remove listener

Moving a window

The following example will look for a position of a window on the screen.

```
import java.awt.Font;

import java.awt.event.ComponentEvent;

import java.awt.event.ComponentListener;



import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;




public class MovingWindow extends JFrame implements ComponentListener {
```

```
    private JLabel labelx;
    private JLabel labely;


    public MovingWindow() {


        setTitle("Moving window");


        JPanel panel = new JPanel();
        panel.setLayout(null);


        labelx = new JLabel("x: ");
        labelx.setFont(new Font("Serif", Font.BOLD, 14));
        labelx.setBounds(20, 20, 60, 25);


        labely = new JLabel("y: ");
        labely.setFont(new Font("Serif", Font.BOLD, 14));
        labely.setBounds(20, 45, 60, 25);


        panel.add(labelx);
        panel.add(labely);
```

```
        add(panel);


        addComponentListener(this);


        setSize(310, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    public void componentResized(ComponentEvent e) {
    }


    public void componentMoved(ComponentEvent e) {
        int x = e.getComponent().getX();
        int y = e.getComponent().getY();
        labelx.setText("x: " + x);
        labely.setText("y: " + y);
    }


    public void componentShown(ComponentEvent e) {
```

```
    }


    public void componentHidden(ComponentEvent e) {

    }



    public static void main(String[] args) {

        new MovingWindow();

    }

}
```

The example shows the current window coordinates on the panel. To get the window position, we use the**ComponentListener**

```
labelx.setFont(new Font("Serif", Font.BOLD, 14));
```

We make the font bigger, the default one is a bit small.

```
int x = e.getComponent().getX();


int y = e.getComponent().getY();
```

Here we get the x and the y positions.

Notice, that we have to implement all four methods, that are available in the **ComponentListener**. Even, if we do not use them.

Figure: Moving a window

Adapters

Adapters are convenient classes. In the previous code example, we had to implement all four methods of a**ComponentListener** class. Even if we did not use them. To avoid unnecessary coding, we can use adapters. Adapter is a class that implements all necessary methods. They are empty. We then use only those methods, that we actually need. There is no adapter for a button click event. Because there we have only one method to implement. The**actionPerformed()** method. We can use adapters in situations, where we have more than one method to implement.

The following example is a rewrite of the previous one, using a **ComponentAdapter**.

```
import java.awt.Font;

import java.awt.event.ComponentAdapter;

import java.awt.event.ComponentEvent;



import javax.swing.JFrame;

import javax.swing.JLabel;
```

```
import javax.swing.JPanel;



public class Adapter extends JFrame {


    private JLabel labelx;

    private JLabel labely;


    public Adapter() {


        setTitle("Adapter");


        JPanel panel = new JPanel();

        panel.setLayout(null);


        labelx = new JLabel("x: ");

        labelx.setFont(new Font("Serif", Font.BOLD, 14));

        labelx.setBounds(20, 20, 60, 25);


        labely = new JLabel("y: ");

        labely.setFont(new Font("Serif", Font.BOLD, 14));
```

```
        labely.setBounds(20, 45, 60, 25);




        panel.add(labelx);

        panel.add(labely);


        add(panel);

        addComponentListener(new MoveAdapter());


        setSize(310, 200);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }




    class MoveAdapter extends ComponentAdapter {

      public void componentMoved(ComponentEvent e) {

          int x = e.getComponent().getX();

          int y = e.getComponent().getY();

          labelx.setText("x: " + x);
```

```
            labely.setText("y: " + y);

        }

       }



     public static void main(String[] args) {

         new Adapter();

     }

 }
```

This example is a rewrite of the previous one. Here we use the **ComponentAdapter**.

```
addComponentListener(new MoveAdapter());
```

Here we register the component listener.

```
class MoveAdapter extends ComponentAdapter {


    public void componentMoved(ComponentEvent e) {


        int x = e.getComponent().getX();


        int y = e.getComponent().getY();


        labelx.setText("x: " + x);


        labely.setText("y: " + y);


    }


}
```

Inside the MoveAdapter inner class, we define the **componentMoved()** method. All the other methods are left empty.

Java Swing dialogs

Dialog windows or dialogs are an indispensable part of most modern GUI applications. A dialog is defined as a conversation between two or more persons. In a computer application a dialog is a window which is used to "talk" to the application. A dialog is used to input data, modify data, change the application settings etc. Dialogs are important means of communication between a user and a computer program.

In Java Swing toolkit, we can create two kinds of dialogs. Custom dialogs and standard dialogs. **Custom dialogs** are dialogs, created by the programmer. They are based on the **JDialog** class. **Standard dialogs** preedefined dialogs available in the Swing toolkit. For example **JColorChooser** or **JFileChooser**. These are dialogs for common programming tasks like showing text, receiving input , loading and saving files etc. They save programmer's time and enhance using some standard behaviour.

There are two basic types of dialogs. Modal and modeless. **Modal** dialogs block input to other top level windows.**Modeless** dialogs allow input to other windows. What type of dialog to use, depends on the circumstances. An open file dialog is a good example of a modal dialog. While choosing a file to open, no other operation should be permitted. A typical modeless dialog is a find text dialog. (Like in Eclipse IDE.) It is handy to have the ability to move the cursor in the text control and define, where to start the finding of the particular text.

A simple custom dialog

In the following example we create a simple custom dialog. It is a sample about dialog, found in most GUI applications, usually located in the help menu.

```
import java.awt.Dimension;

import java.awt.Font;
```

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.KeyEvent;



import javax.swing.Box;

import javax.swing.BoxLayout;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JDialog;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;




class AboutDialog extends JDialog {



    public AboutDialog() {



        setTitle("About Notes");
```

```
        setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));


        add(Box.createRigidArea(new Dimension(0, 10)));


        ImageIcon icon = new ImageIcon("notes.png");
        JLabel label = new JLabel(icon);
        label.setAlignmentX(0.5f);
        add(label);


        add(Box.createRigidArea(new Dimension(0, 10)));


        JLabel name = new JLabel("Notes, 1.23");
        name.setFont(new Font("Serif", Font.BOLD, 13));
        name.setAlignmentX(0.5f);
        add(name);


        add(Box.createRigidArea(new Dimension(0, 50)));


        JButton close = new JButton("Close");
        close.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent event) {

                dispose();

            }

        });



        close.setAlignmentX(0.5f);

        add(close);



        setModalityType(ModalityType.APPLICATION_MODAL);



        setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        setLocationRelativeTo(null);

        setSize(300, 200);



    }



}



public class SimpleDialog extends JFrame {
```

```
public SimpleDialog() {

    setTitle("Simple Dialog");

    JMenuBar menubar = new JMenuBar();

    JMenu file = new JMenu("File");
    file.setMnemonic(KeyEvent.VK_F);

    JMenu help = new JMenu("Help");
    help.setMnemonic(KeyEvent.VK_H);

    JMenuItem about = new JMenuItem("About");
    help.add(about);

    about.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            AboutDialog ad = new AboutDialog();
            ad.setVisible(true);
        }
```

```
        });


        menubar.add(file);

        menubar.add(help);

        setJMenuBar(menubar);


        setSize(300, 200);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }


    public static void main(String[] args) {

        new SimpleDialog();

    }

}
```

The sample code will popup a small dialog box. The dialog will display an icon a text and one close button.

```
class AboutDialog extends JDialog {
```

The custom dialog is based on the **JDialog** class.

```
setModalityType(ModalityType.APPLICATION_MODAL);
```

Here we make the dialog modal.

```
setDefaultCloseOperation(DISPOSE_ON_CLOSE);
```

Here we set the defaul close operation.

```
AboutDialog ad = new AboutDialog();


ad.setVisible(true);
```

Here we display the about dialog, from the menu of the main frame.

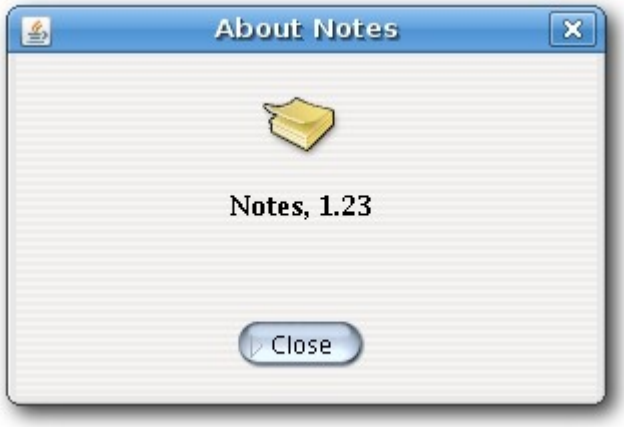


Figure: Simple custom dialog

Message boxes

Message boxes provide information to the user.

```
import java.awt.GridLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;
```

```
import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JOptionPane;

import javax.swing.JPanel;



public class MessageBoxes extends JFrame {



    private JPanel panel;

    public MessageBoxes() {

        setTitle("Message Boxes");

        panel = new JPanel();

        panel.setLayout(new GridLayout(2, 2));

        JButton error = new JButton("Error");

        JButton warning = new JButton("Warning");

        JButton question = new JButton("Question");
```

```
        JButton information = new JButton("Information");



        error.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                JOptionPane.showMessageDialog(panel, "Could not open
file",

                    "Error", JOptionPane.ERROR_MESSAGE);

            }



        });



        warning.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                JOptionPane.showMessageDialog(panel, "A deprecated call",

                    "Warning", JOptionPane.WARNING_MESSAGE);

            }



        });



        question.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {
```

```
                JOptionPane.showMessageDialog(panel, "Are you sure to
quit?",
                    "Question", JOptionPane.QUESTION_MESSAGE);
            }


        });


        information.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                JOptionPane.showMessageDialog(panel, "Download
completed",
                    "Question", JOptionPane.INFORMATION_MESSAGE);
            }


        });


        panel.add(error);
        panel.add(warning);
        panel.add(question);
        panel.add(information);


        add(panel);
```

```
        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    public static void main(String[] args) {
        new MessageBoxes();
    }
}
```

The example shows an error, question, warning and information message boxes.

```
panel.setLayout(new GridLayout(2, 2));
```

We use a **GridLayout** layout manager to organize buttons, that will popup message boxes.

```
JButton error = new JButton("Error");

JButton warning = new JButton("Warning");

JButton question = new JButton("Question");

JButton information = new JButton("Information");
```

Here are the four buttons, that we will use.

```
JOptionPane.showMessageDialog(panel, "Could not open file",
```

```
    "Error", JOptionPane.ERROR_MESSAGE);
```

To create a message box, we call the **showMessageDialog** static method of the **JOptionPane** class. We provide the component name, message text, title and a message type. The message type is determined by the constant we choose. Available constants are:

- ERROR_MESSAGE
- WARNING_MESSAGE
- QUESTION_MESSAGE
- INFORMATION_MESSAGE

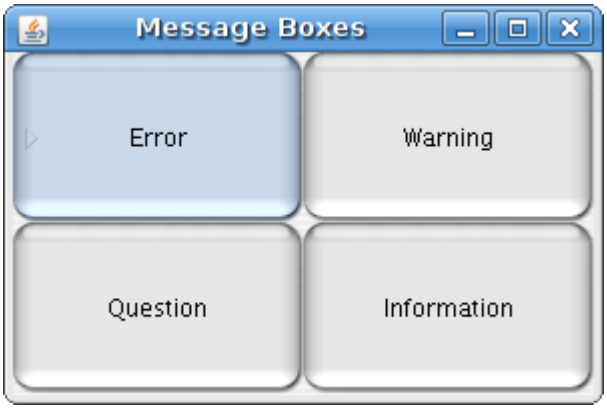


Figure: Message boxes

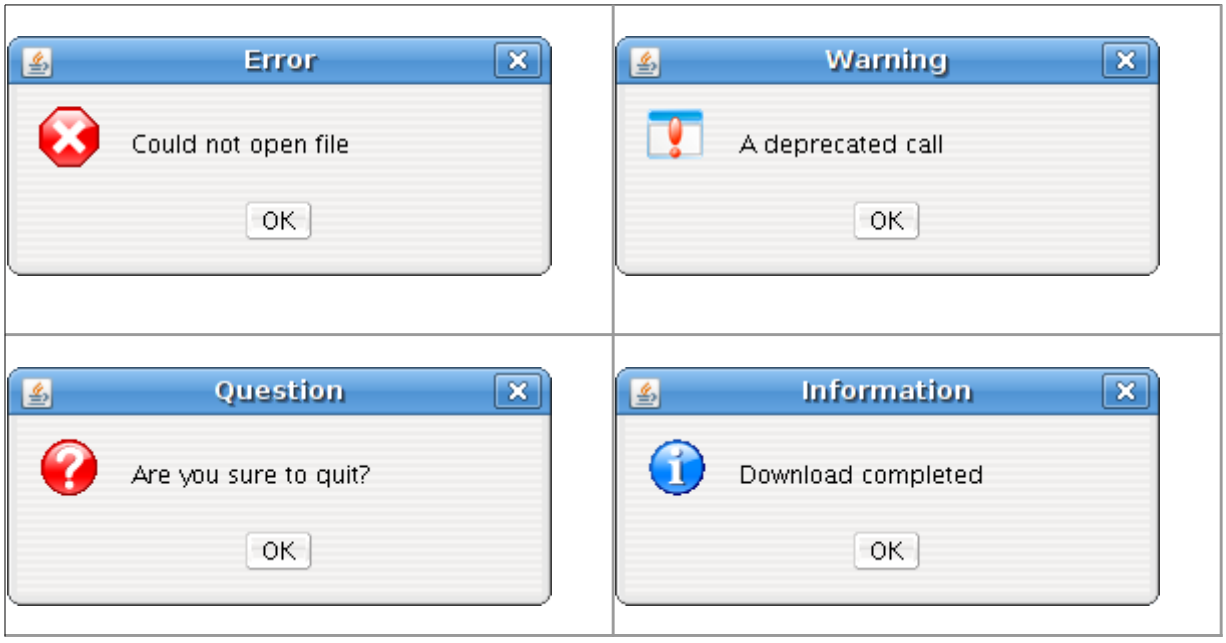


Figure: Message boxes

JFileChooser

JFileChooser is a standard dialog for selecting a file from the file system.

```
import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;



import javax.swing.BorderFactory;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JFileChooser;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JScrollPane;
```

```
import javax.swing.JTextArea;

import javax.swing.JToolBar;

import javax.swing.filechooser.FileFilter;

import javax.swing.filechooser.FileNameExtensionFilter;



public class FileChooserDialog extends JFrame {


    private JPanel panel;

    private JTextArea area;


    public FileChooserDialog() {


        setTitle("FileChooserDialog");


        panel = new JPanel();

        panel.setLayout(new BorderLayout());

        ImageIcon open = new ImageIcon("open.png");


        JToolBar toolbar = new JToolBar();

        JButton openb = new JButton(open);
```

```
        openb.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                JFileChooser fileopen = new JFileChooser();

                FileFilter filter = new FileNameExtensionFilter("c
files", "c");

                fileopen.addChoosableFileFilter(filter);



                int ret = fileopen.showDialog(panel, "Open file");



                if (ret == JFileChooser.APPROVE_OPTION) {

                    File file = fileopen.getSelectedFile();

                    String text = readFile(file);

                    area.setText(text);

                }



            }

        });



        toolbar.add(openb);
```

```
        area = new JTextArea();
        area.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        JScrollPane pane = new JScrollPane();
        pane.getViewport().add(area);

        panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        panel.add(pane);
        add(panel);

        add(toolbar, BorderLayout.NORTH);

        setSize(400, 300);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public String readFile (File file) {

     StringBuffer fileBuffer = null;
```

```
      String fileString = null;

      String line = null;



      try {

        FileReader in = new FileReader (file);

        BufferedReader brd = new BufferedReader (in);

        fileBuffer = new StringBuffer() ;



        while ((line = brd.readLine()) != null) {

              fileBuffer.append(line +
System.getProperty("line.separator"));

        }



        in.close();

        fileString = fileBuffer.toString();

      }

      catch  (IOException e ) {

        return null;

      }

      return fileString;

     }
```

```
    public static void main(String[] args) {

        new FileChooserDialog();

    }

}
```

The code example will demonstrate how to use a file chooser dialog in order to load file contents into the text area component.

```
JFileChooser fileopen = new JFileChooser();
```

This is the constructor of the file chooser dialog.

```
FileFilter filter = new FileNameExtensionFilter("c files", "c");

fileopen.addChoosableFileFilter(filter);
```

Here we define the file filter. In our case, we will have c files with extension .c. We have also the default All files option.

```
int ret = fileopen.showDialog(panel, "Open file");
```

Here we show the file chooser dialog. Upon clicking on the open file button, the return value is equal to**JFileChooser.APPROVE_OPTION**.

```
if (ret == JFileChooser.APPROVE_OPTION) {

    File file = fileopen.getSelectedFile();

    String text = readFile(file);

    area.setText(text);
```

```
}
```

Here we get the name of the selected file. We read the contents of the file and set the text into the textarea.

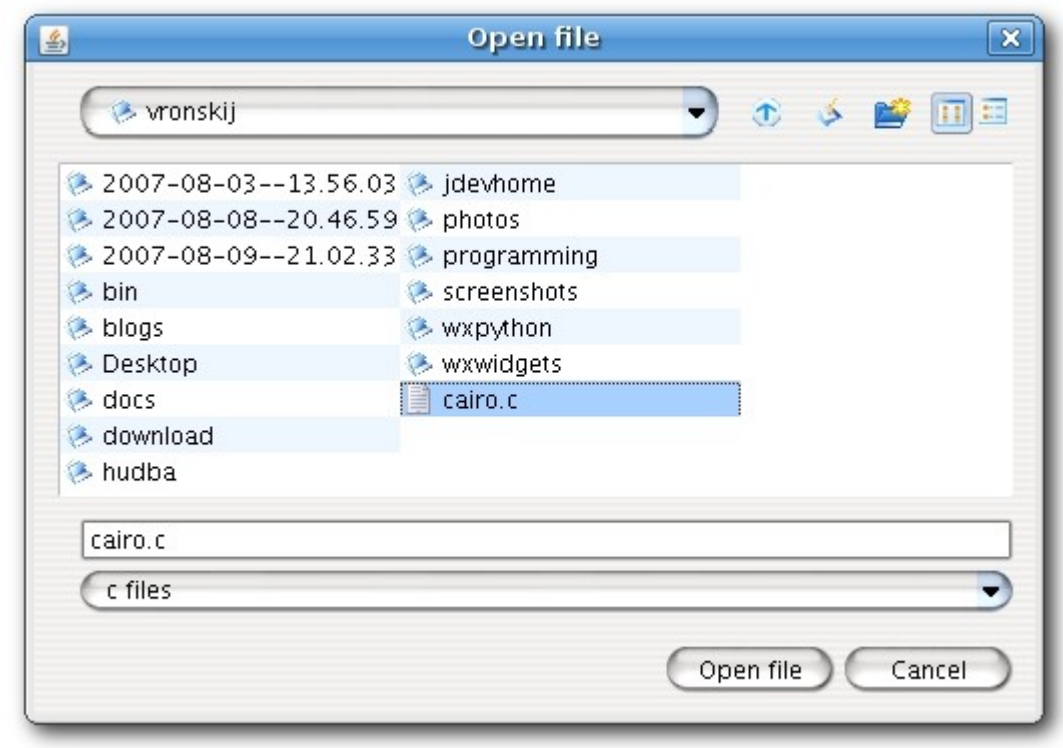


Figure: JFileChooser dialog

JColorChooser

JColorChooser is a standard dialog for selecting a color.

```
import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;
```

```
import javax.swing.BorderFactory;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JColorChooser;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JToolBar;



public class ColorChooserDialog extends JFrame {


    private JPanel panel;

    private JPanel display;


    public ColorChooserDialog() {


        setTitle("ColorChooserDialog");


        panel = new JPanel();

        panel.setLayout(new BorderLayout());

        ImageIcon open = new ImageIcon("color.png");
```

```
        JToolBar toolbar = new JToolBar();
        JButton openb = new JButton(open);


        openb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                JColorChooser clr = new JColorChooser();
                Color color = clr.showDialog(panel, "Choose Color",
Color.white);
                display.setBackground(color);
            }
        });


        toolbar.add(openb);


        display = new JPanel();
        display.setBackground(Color.WHITE);


        panel.setBorder(BorderFactory.createEmptyBorder(30, 50, 30, 50));
        panel.add(display);
        add(panel);
```

```
        add(toolbar, BorderLayout.NORTH);


        setSize(400, 300);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    public static void main(String[] args) {
        new ColorChooserDialog();
    }
}
```

In the example, we have a white panel. We will change the background color of the panel by selecting a color from the color chooser dialog.

```
JColorChooser clr = new JColorChooser();

Color color = clr.showDialog(panel, "Choose Color", Color.white);

display.setBackground(color);
```

This code shows a color chooser dialog. The **showDialog()** method returns the selected color value. We change the display panel background to the newly selected color.

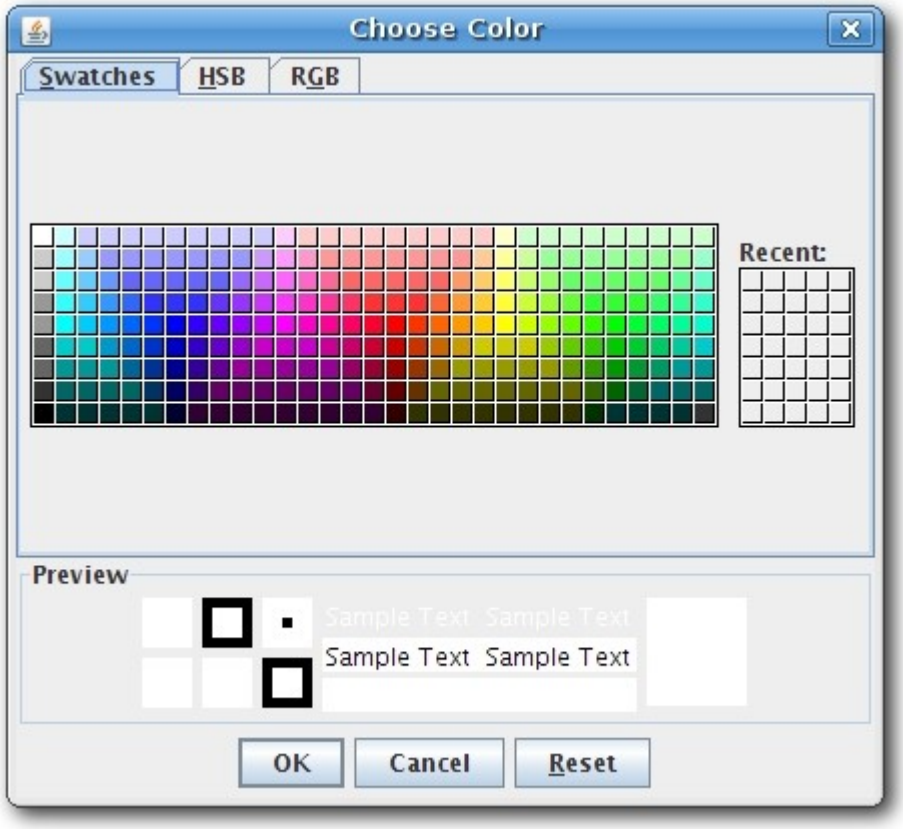


Figure: JColorChooser dialog

Basic Swing components

Swing components are basic building blocks of an application. Swing toolkit has a wide range of various widgets. Buttons, check boxes,sliders, list boxes etc. Everything a programmer needs for his job. In this section of the tutorial, we will describe several useful components.

JLabel Component

JLabel is a simple component for displaying text, images or both. It does not react to input events.

```
import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.Toolkit;



import javax.swing.BorderFactory;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;




public class MyLabel extends JFrame {


    private Toolkit toolkit;


    public MyLabel() {


      setTitle("No Sleep");


        String lyrics =  "<html>It's way too late to think of<br>" +
```

```
        "Someone I would call now<br>" +
        "And neon signs got tired<br>" +
        "Red eye flights help the stars out<br>" +
        "I'm safe in a corner<br>" +
        "Just hours before me<br>" +
        "<br>" +
        "I'm waking with the roaches<br>" +
        "The world has surrendered<br>" +
        "I'm dating ancient ghosts<br>" +
        "The ones I made friends with<br>" +
        "The comfort of fireflies<br>" +
        "Long gone before daylight<br>" +
        "<br>" +
        "And if I had one wishful field tonight<br>" +
        "I'd ask for the sun to never rise<br>" +
        "If God leant his voice for me to speak<br>" +
        "I'd say go to bed, world<br>" +
        "<br>" +
        "I've always been too late<br>" +
        "To see what's before me<br>" +
        "And I know nothing sweeter than<br>" +
```

```
        "Champaign from last New Years<br>" +
        "Sweet music in my ears<br>" +
        "And a night full of no fears<br>" +
        "<br>" +
        "But if I had one wishful field tonight<br>" +
        "I'd ask for the sun to never rise<br>" +
        "If God passed a mic to me to speak<br>" +
        "I'd say stay in bed, world<br>" +
        "Sleep in peace</html>";

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout(10, 10));

        JLabel label = new JLabel(lyrics);
        label.setFont(new Font("Georgia", Font.PLAIN, 14));
        label.setForeground(new Color(50, 50, 25));

        panel.add(label, BorderLayout.CENTER);
        panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        add(panel);
```

```
        pack();



        toolkit = getToolkit();

        Dimension screensize = toolkit.getScreenSize();

        setLocation((screensize.width - getWidth())/2,

            (screensize.height - getHeight())/2);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

    }



    public static void main(String[] args) {



        MyLabel mylabel = new MyLabel();

        mylabel.setVisible(true);



    }

}
```

In our example, we show lyrics of no sleep song from cardigans. We can use html tags in **JLabel** component. We use the
 tag to separate lines.

```
JPanel panel = new JPanel();


panel.setLayout(new BorderLayout(10, 10));
```

We create a panel and set a **BorderLayout** manager.

```
JLabel label = new JLabel(lyrics);

label.setFont(new Font("Georgia", Font.PLAIN, 14));

label.setForeground(new Color(50, 50, 25));
```

Here we create the label component. We set it's font to plain georgia, 14 px tall. We also change the foreground color.

```
panel.add(label, BorderLayout.CENTER);

panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
```

We put the label into the center of the panel. We put 10px around the label.

```
add(panel);

pack();
```

The panel is added to the frame component. We call the **pack()** method, which will resize the window, so that all components are visible.

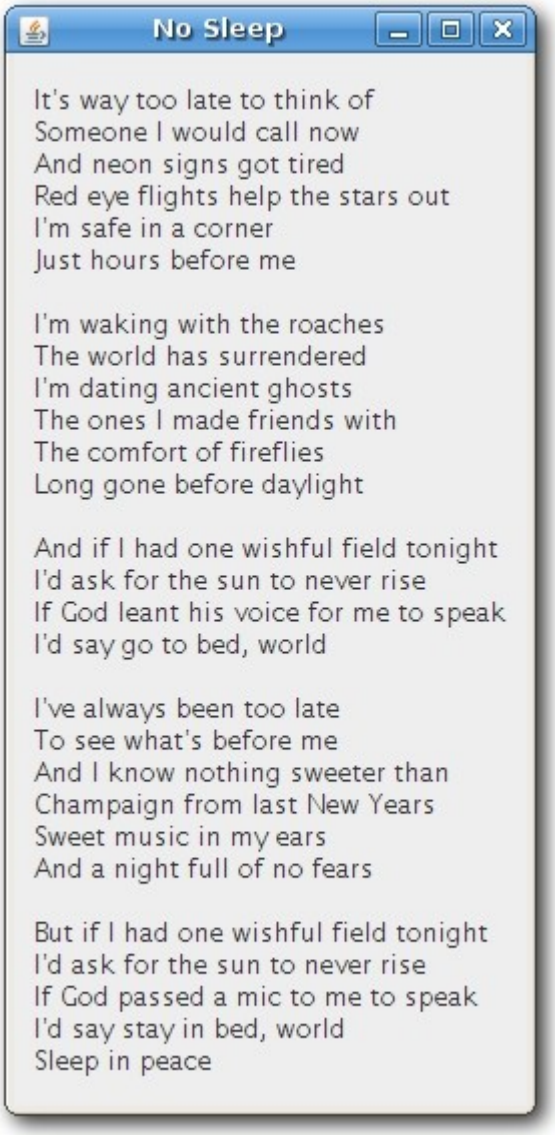


Figure: JLabel

JCheckBox

JCheckBox is a widget that has two states. On and Off. It is a box with a label. If the checkbox is checked, it is represented by a tick in a box. A checkbox can be used to show/hide splashscreen at startup, toggle visibility of a toolbar etc.

```
import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.Box;

import javax.swing.BoxLayout;

import javax.swing.JCheckBox;

import javax.swing.JFrame;




public class CheckBox extends JFrame implements ActionListener {


    public CheckBox() {


        setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));

        add(Box.createRigidArea(new Dimension(15, 20)));


        JCheckBox checkbox = new JCheckBox("Show Title", true);

        checkbox.setFocusable(false);

        checkbox.addActionListener(this);

        add(checkbox);
```

```
        setSize(280, 200);
        setTitle("CheckBox example");
        setLocationRelativeTo(null);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {

        new CheckBox();

    }


    public void actionPerformed(ActionEvent e) {
        if (this.getTitle() == "") {
            this.setTitle("Checkbox example");
        } else {
            this.setTitle("");
```

```
          }


     }

 }
```

Our code example shows or hides the title of the window depending on it's state.

```
setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));

add(Box.createRigidArea(new Dimension(15, 20)));
```

In this example, we use a **BoxLayout** layout manager. We put some space there, so that the checkbox is not too close to the corner.

```
JCheckBox checkbox = new JCheckBox("Show Title", true);
```

Here we have a constructor for the checkbox. We provide text and state.

```
checkbox.setFocusable(false);
```

We have disabled the focus for the checkbox. The rectangle around the text looks ugly, so we go without the focus.

```
if (this.getTitle() == "") {

    this.setTitle("Checkbox example");

} else {

    this.setTitle("");

}
```

Here we toggle the title of the window.

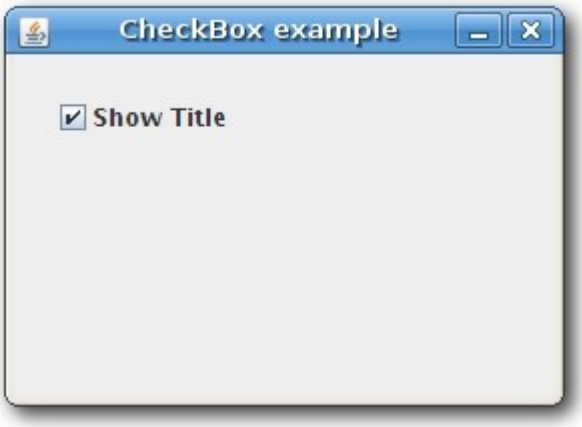


Figure: JCheckBox

JSlider

JSlider is a component that lets the user graphically select a value by sliding a knob within a bounded interval. Our example will show a volume control.

```
import java.awt.BorderLayout;

import java.awt.Dimension;



import javax.swing.BorderFactory;

import javax.swing.Box;

import javax.swing.BoxLayout;

import javax.swing.ImageIcon;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;
```

```
import javax.swing.JSlider;

import javax.swing.event.ChangeEvent;

import javax.swing.event.ChangeListener;




public class Slider extends JFrame {



    private JSlider slider;

    private JLabel label;



    ImageIcon mute = new
ImageIcon(ClassLoader.getSystemResource("mute.png"));

    ImageIcon min = new
ImageIcon(ClassLoader.getSystemResource("min.png"));

    ImageIcon med = new
ImageIcon(ClassLoader.getSystemResource("med.png"));

    ImageIcon max = new
ImageIcon(ClassLoader.getSystemResource("max.png"));



    public Slider() {



        setTitle("JSlider");

        setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));
        panel.setBorder(BorderFactory.createEmptyBorder(40, 40, 40, 40));
        setLayout(new BorderLayout());


        panel.add(Box.createHorizontalGlue());
        slider = new JSlider(0, 150, 0);


        slider.setPreferredSize(new Dimension(150, 30));


        slider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent event) {
                int value = slider.getValue();
                if (value == 0) {
                    label.setIcon(mute);
                } else if (value > 0 && value <= 30) {
                    label.setIcon(min);
                } else if (value > 30 && value < 80) {
                    label.setIcon(med);
                } else {
```

```
                label.setIcon(max);

            }

        }

    });


    panel.add(slider);

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


    label = new JLabel(mute, JLabel.CENTER);

    panel.add(label);

    panel.add(Box.createHorizontalGlue());

    add(panel, BorderLayout.CENTER);


    pack();

    setLocationRelativeTo(null);


}



public static void main(String[] args) {
```

```
        Slider button = new Slider();

        button.setVisible(true);

    }

}
```

In the code example, we show a **JSlider** and a **JLabel**. By dragging the slider, we change the icon on the label component.

```
ImageIcon mute = new ImageIcon(ClassLoader.getSystemResource("mute.png"));
```

Here we create an image icon.

```
panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));
```

Panel component has a horizontal **BoxLayout**.

```
panel.setBorder(BorderFactory.createEmptyBorder(40, 40, 40, 40));
```

We creare a 40px border around the panel.

```
panel.add(Box.createHorizontalGlue());
```

We put resizable space to bo both sides, left and right. It is to prevent **JSlider** from growing to unnatural sizes.

```
slider = new JSlider(0, 150, 0);
```

This is a **JSlider** constructor. The parameters are minimum value, maximum value and current value.

```
slider.addChangeListener(new ChangeListener() {

...

});
```

We add a **ChangeListener** to the slider. Inside the listener, we determine the current slider value and update the label accordingly.

```
panel.add(Box.createRigidArea(new Dimension(5, 0)));
```

We place a 5px rigid space between the two components. They are too close to each other, when the slider is at the end position.

Figure: JSlider

JComboBox

Combobox is a component that combines a button or editable field and a drop-down list. The user can select a value from the drop-down list, which appears at the user's request. If you make the combo box editable, then the combo box includes an editable field into which the user can type a value.

```
import java.awt.Component;

import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.ItemEvent;

import java.awt.event.ItemListener;
```

```
import javax.swing.Box;

import javax.swing.BoxLayout;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JComboBox;

import javax.swing.JDialog;

import javax.swing.JLabel;

import javax.swing.border.LineBorder;



public class ComboBox extends JDialog implements

             ActionListener, ItemListener {


    final String[] authors = {

      "Leo Tolstoy", "John Galsworthy",

         "Honore de Balzac", "Stefan Zweig",

      "Boris Pasternak", "Tom Wolfe"

    };


    final String[] images = {
```

```
  "tolstoy.jpg", "galsworthy.jpg", "balzac.jpg",
     "zweig.jpg", "pasternak.jpg", "wolfe.jpg"
};

private JLabel display = null;
private JComboBox combobox = null;
private JButton button = null;

ImageIcon icon = new ImageIcon(
          ClassLoader.getSystemResource("balzac.jpg"));

public ComboBox() {

    setLayout(new BoxLayout(getContentPane(),
          BoxLayout.Y_AXIS));
    add(Box.createRigidArea(new Dimension(0, 35)));

    display = new JLabel();
    display.setPreferredSize(new Dimension(100, 127));
    display.setMaximumSize(new Dimension(100, 127));
    display.setAlignmentX(Component.CENTER_ALIGNMENT);
```

```
        display.setBorder(LineBorder.createGrayLineBorder());
        add(display);

        add(Box.createRigidArea(new Dimension(0, 15)));

        combobox = new JComboBox(authors);
        combobox.setSelectedIndex(-1);
        combobox.setPreferredSize(new Dimension(140, 22));
        combobox.setMaximumSize(new Dimension(140, 22));
        combobox.addItemListener(this);
        add(combobox);

        add(Box.createRigidArea(new Dimension(0, 15)));

        button = new JButton("Close");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        button.addActionListener(this);
        add(button);

        setTitle("JComboBox");
        setSize(300, 300);
```

```
        setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String[] args) {
        new ComboBox();
    }

    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }

    public void itemStateChanged(ItemEvent e) {

        if (e.getStateChange() == ItemEvent.SELECTED) {
            JComboBox combo = (JComboBox) e.getSource();
            int index = combo.getSelectedIndex();
            display.setIcon(new ImageIcon(
                ClassLoader.getSystemResource(images[index])));
```

```
            }



        }

    }
```

In our example, we have three components. A label, a combobox and a button. The button closes the window. We have six names of famous novelists in our combobox. If we select a name, an image is displayed in the label.

```
public class ComboBox extends JDialog implements


                ActionListener, ItemListener {
```

This is a dialog based application example.

```
display = new JLabel();
```

The display area is a simple **JLabel**.

```
combobox = new JComboBox(authors);


combobox.setSelectedIndex(-1);
```

The constructor of the **JComboBox** takes a string array of novelists. If we provide -1 as an argument in the**setSelectedIndex()** method, no item to be selected.

```
combobox.addItemListener(this);
```

We add an **ItemListener** to our combobox. In the event handler, we get the selected index of the combobox and set an appropriate icon for the label. The selected item is an index to the array of images.

```

```

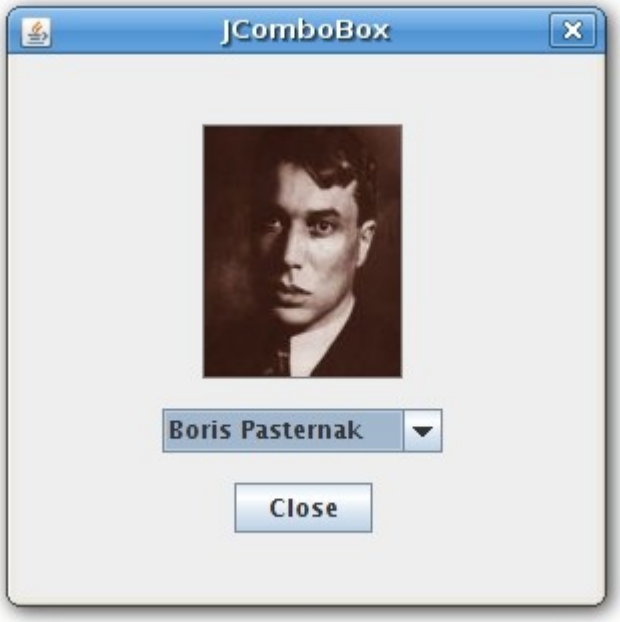


Figure: JComboBox

JProgressBar

A progress bar is a widget that is used, when we process lengthy tasks. It is animated so that the user knows, that our task is progressing. The **JProgressBar** widget provides a horizontal or vertical progress bar. The initial and minimum values are 0, and the maximum is 100.

```
import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.BorderFactory;

import javax.swing.Box;
```

```
import javax.swing.BoxLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JProgressBar;

import javax.swing.Timer;




public class ProgressBar extends JFrame {



    ActionListener updateProBar;

    Timer timer;

    JProgressBar progressBar;

    JButton button;



    public ProgressBar() {



        setTitle("JProgressBar");



        JPanel panel = new JPanel();

        panel.setBorder(BorderFactory.createEmptyBorder(40, 40, 40, 40));
```

```
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));




        progressBar = new JProgressBar();


        progressBar.setMaximumSize(new Dimension(150, 20));

        progressBar.setMinimumSize(new Dimension(150, 20));

        progressBar.setPreferredSize(new Dimension(150, 20));


        progressBar.setAlignmentX(0f);


        panel.add(progressBar);

        panel.add(Box.createRigidArea(new Dimension(0, 20)));


        button = new JButton("Start");

        button.setFocusable(false);

        button.setMaximumSize(button.getPreferredSize());


        updateProBar = new ActionListener() {

            public void actionPerformed(ActionEvent actionEvent) {
```

```
                int val = progressBar.getValue();
                if (val >= 100) {
                    timer.stop();
                    button.setText("End");
                    return;
                }

                progressBar.setValue(++val);
            }
        };

        timer = new Timer(50, updateProBar);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (timer.isRunning()) {
                    timer.stop();
                    button.setText("Start");

                } else if (button.getText() != "End") {
                    timer.start();
```

```
                button.setText("Stop");

            }


        }

    });



    panel.add(button);

    add(panel);


    pack();

    setDefaultCloseOperation(EXIT_ON_CLOSE);

    setResizable(false);

    setLocationRelativeTo(null);

    setVisible(true);


}


public static void main(String[] args) {


    new ProgressBar();
```

```
    }

}
```

The example displays a progress bar and a button. The button starts and stops the progress.

```
progressBar = new JProgressBar();
```

Here we create the **JProgressBar**. The minimum value is 0, maximum 100 and the initial value is 0. These are the default values.

```
progressBar.setMaximumSize(new Dimension(150, 20));

progressBar.setMinimumSize(new Dimension(150, 20));

progressBar.setPreferredSize(new Dimension(150, 20));
```

These lines are for design purposes only. I want my examples to look nice. The default height on my box was only 14px which looked bad.

```
 progressBar.setAlignmentX(0f);
```

This line aligns both progress bar with the button. To the left.

```
panel.add(Box.createRigidArea(new Dimension(0, 20)));
```

Here we put some rigid space between the two components.

```
button.setFocusable(false);
```

Focus looks bad, better disable it.

```
timer = new Timer(50, updateProBar);
```

The timer object launches updateProBar listener every 50ms. Inside that listener, we check, if the progress bar reached the value 100 and stop the timer, or update the progress bar.

```
if (timer.isRunning()) {
```

```
    timer.stop();

    button.setText("Start");



} else if (button.getText() != "End") {

    timer.start();

    button.setText("Stop");

}
```

Clicking on the button starts or stops the progress. The text of the button is updated dynamically. It can have Start, Stop or End String values.

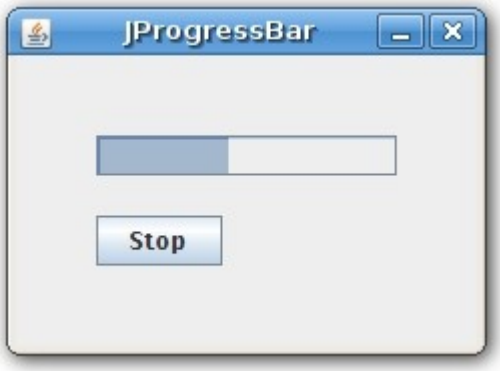


Figure: JProgressBar

JToggleButton

JToggleButton is a button that has two states. Pressed and not pressed. You toggle between these two states by clicking on it. There are situations where this functionality fits well.

```
import java.awt.Color;
```

```
import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.BorderFactory;

import javax.swing.Box;

import javax.swing.BoxLayout;

import javax.swing.JDialog;

import javax.swing.JPanel;

import javax.swing.JToggleButton;

import javax.swing.border.LineBorder;




public class ToggleButton extends JDialog implements ActionListener {



    private JToggleButton red;

    private JToggleButton green;

    private JToggleButton blue;

    private JPanel display;



    public ToggleButton() {
```

```
        setTitle("JToggleButton");


        JPanel bottom = new JPanel();
        bottom.setLayout(new BoxLayout(bottom, BoxLayout.X_AXIS));
        bottom.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));


        JPanel leftPanel = new JPanel();
        leftPanel.setLayout(new BoxLayout(leftPanel, BoxLayout.Y_AXIS));


        red = new JToggleButton("red");
        red.addActionListener(this);


        green = new JToggleButton("green");
        green.addActionListener(this);


        blue = new JToggleButton("blue");
        blue.addActionListener(this);


        blue.setMaximumSize(green.getMaximumSize());
        red.setMaximumSize(green.getMaximumSize());
```

```
        leftPanel.add(red);
        leftPanel.add(Box.createRigidArea(new Dimension(25, 7)));
        leftPanel.add(green);
        leftPanel.add(Box.createRigidArea(new Dimension(25, 7)));
        leftPanel.add(blue);



        bottom.add(leftPanel);
        bottom.add(Box.createRigidArea(new Dimension(20, 0)));

        display = new JPanel();
        display.setPreferredSize(new Dimension(110, 110));
        display.setBorder(LineBorder.createGrayLineBorder());
        display.setBackground(Color.black);

        bottom.add(display);
        add(bottom);

        pack();
        setResizable(false);
```

```
        setLocationRelativeTo(null);

        setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);

        setVisible(true);

    }




    public static void main(String[] args) {

        new ToggleButton();

    }



    public void actionPerformed(ActionEvent e) {



        Color color = display.getBackground();

        int red = color.getRed();

        int green = color.getGreen();

        int blue = color.getBlue();



        if (e.getActionCommand() == "red") {

            if (red == 0) {

                red = 255;
```

```
            } else {
                red = 0;
            }
        }

        if (e.getActionCommand() == "green") {
            if (green == 0) {
                green = 255;
            } else {
                green = 0;
            }
        }

        if (e.getActionCommand() == "blue") {
            if (blue == 0) {
                blue = 255;
            } else {
                blue = 0;
            }
        }
```

```
        Color setCol = new Color(red, green, blue);

        display.setBackground(setCol);

    }

}
```

The example has three panels and three toggle buttons. Panels are bottom panel, left panel and display panel. The bottom panel is use to organize the left and display panels. For this, we use horizontal **BoxLayout** manager. The left panel will holt three toggle buttons. This time we use vertical **BoxLayout** manager. We set the background color of the display panel to black. The toggle buttons will toggle the red, green and blue parts of the color value. The background color will depend on which togglebuttons we have pressed.

```
red = new JToggleButton("red");


red.addActionListener(this);
```

Here we create a toggle button and set an action listener to it.

```
blue.setMaximumSize(green.getMaximumSize());


red.setMaximumSize(green.getMaximumSize());
```

We make all three buttons of equal size.

```
Color color = display.getBackground();


int red = color.getRed();


int green = color.getGreen();


int blue = color.getBlue();
```

In the **actionPerformed** method, we determine the current red, green, blue parts of the display background color.

```
if (e.getActionCommand() == "red") {

    if (red == 0) {

        red = 255;

    } else {

       red = 0;

    }

}
```

We determine, which button was toggled, and update the color part of the RGB value accordingly.

```
Color setCol = new Color(red, green, blue);

display.setBackground(setCol);
```

Here a new color is created and the display panel is updated to a new color.

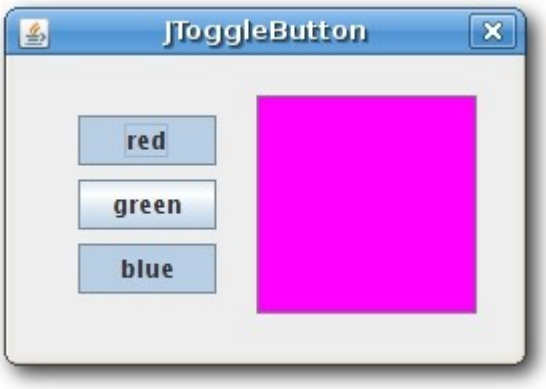


Figure: JToggleButton

Basic Swing components II

Swing components are basic building blocks of an application. Swing toolkit has a wide range of various widgets. Buttons, check boxes,sliders, list boxes etc. Everything a programmer needs for his job. In this section of the tutorial, we will describe several useful components.

JList Component

JList is a component that displays a list of objects. It allows the user to select one or more items.

```
import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.GraphicsEnvironment;



import javax.swing.BorderFactory;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JList;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.UIManager;

import javax.swing.event.ListSelectionEvent;
```

```
import javax.swing.event.ListSelectionListener;



public class List extends JFrame {


    private JLabel label;
    private JList list;



    public List() {


        setTitle("List");
        setDefaultCloseOperation(EXIT_ON_CLOSE);


        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));


        GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
```

```
        String[] fonts = ge.getAvailableFontFamilyNames();


        list = new JList(fonts);
        list.addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent e) {
                if (!e.getValueIsAdjusting()) {
                    String name = (String) list.getSelectedValue();
                    Font font = new Font(name, Font.PLAIN, 12);
                    label.setFont(font);
                }
            }
        });


        JScrollPane pane = new JScrollPane();
        pane.getViewport().add(list);
        pane.setPreferredSize(new Dimension(250, 200));
        panel.add(pane);


        label = new JLabel("Aguirre, der Zorn Gottes");
        label.setFont(new Font("Serif", Font.PLAIN, 12));
        add(label, BorderLayout.SOUTH);
```

```
        add(panel);


        pack();
        setLocationRelativeTo(null);
        setVisible(true);


    }


    public static void main(String[] args) {
        new List();
    }
}
```

In our example, we will display a **JList** and a **JLabel** components. The list component contains a list of all available font family names on our system. If we select an item from the list, the label will be displayed in a font, we have chosen.

```
GraphicsEnvironment ge =

    GraphicsEnvironment.getLocalGraphicsEnvironment();
```

```
String[] fonts = ge.getAvailableFontFamilyNames();
```

Here we obtain all possible font family names on our system.

```
list = new JList(fonts);
```

We create a **JList** component.

```
public void valueChanged(ListSelectionEvent e) {


    if (!e.getValueIsAdjusting()) {
```

This code is quite confusing. Events in list selection are grouped. We receive events for both selecting and deselecting. To filter only the selecting events, we use the **getValueIsAdjusting()** method. Why this weird name? No idea.

```
String name = (String) list.getSelectedValue();


Font font = new Font(name, Font.PLAIN, 12);


label.setFont(font);
```

We get the selected item and set a new font for the label.

```
JScrollPane pane = new JScrollPane();


pane.getViewport().add(list);
```

Interesingly, JLabel component is not scrollable by default. We must put the list into the JScrollPane to make it scrollable.

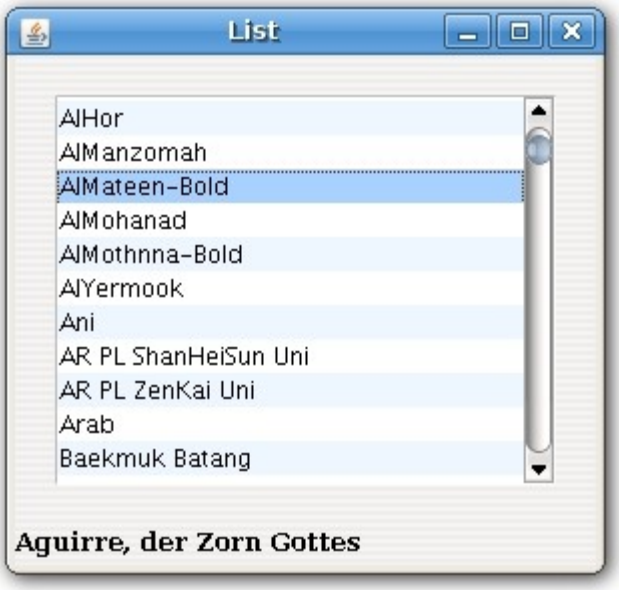


Figure: JList

JTextArea component

A **JTextArea** is a multi-line text area that displays plain text. It is lightweight component for working with text. The component does not handle scrolling. For this task, we use **JScrollPane** component.

```
import java.awt.BorderLayout;

import java.awt.Dimension;



import javax.swing.BorderFactory;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JScrollPane;
```

```
import javax.swing.JTextArea;



public class TextArea extends JFrame {


    public TextArea() {


        setTitle("JTextArea");

        setDefaultCloseOperation(EXIT_ON_CLOSE);


        JPanel panel = new JPanel();

        panel.setLayout(new BorderLayout());

        panel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));


        JScrollPane pane = new JScrollPane();

        JTextArea area = new JTextArea();


        area.setLineWrap(true);

        area.setWrapStyleWord(true);

        area.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));
```

```
        pane.getViewport().add(area);

        panel.add(pane);



        add(panel);

        setSize(new Dimension(350, 300));



        setLocationRelativeTo(null);

        setVisible(true);



    }



    public static void main(String[] args) {



        new TextArea();



    }

}
```

The example shows a simple **JTextArea** component with an excerpt from Martin Luther King speech.

```
JTextArea area = new JTextArea();
```

This is the constructor of the **JTextArea** component.

```
area.setLineWrap(true);
```

Make the lines wrapped, if they are too long to fit the width.

```
area.setWrapStyleWord(true);
```

Here we specify, how is line going to be wrapped. In our case, lines will be wrapped at word boundaries, whitespaces.

```
area.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));
```

We put some border around the text in the component.

```
pane.getViewport().add(area);
```

To make the text scrollable, we put the **JTextArea** component into the **JScrollPane** component.

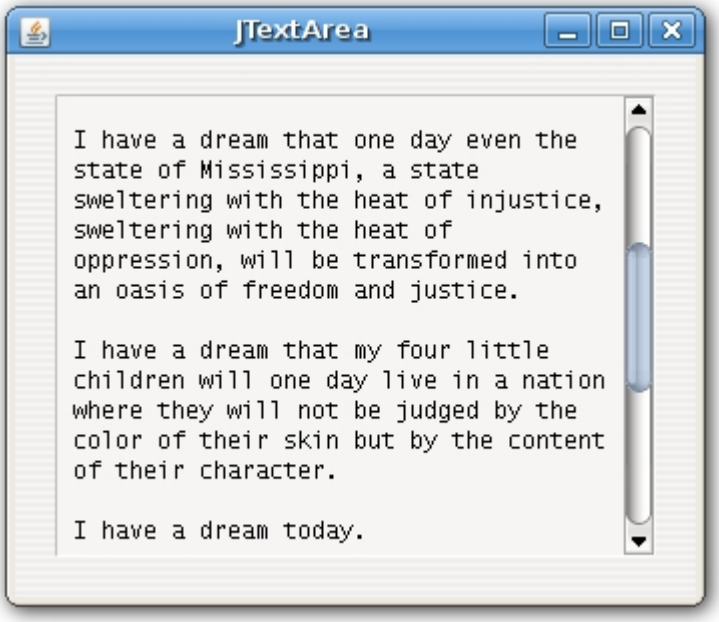


Figure: JTextAra

JTextPane component

JTextPane component is a more advanced component for working with text. The component can do some complex formatting operations over the text. It can display also html documents.

```
import java.awt.BorderLayout;

import java.awt.Dimension;



import java.io.IOException;



import javax.swing.BorderFactory;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTextPane;




public class TextPane extends JFrame {


    public TextPane() {


        setTitle("JTexPane");

        setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
JPanel panel = new JPanel();

panel.setLayout(new BorderLayout());

panel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));


JScrollPane pane = new JScrollPane();

JTextPane textpane = new JTextPane();


textpane.setContentType("text/html");

textpane.setEditable(false);


String cd = System.getProperty("user.dir") + "/";


try {

    textpane.setPage("File:///" + cd + "test.html");

} catch (IOException e) {

    System.out.println("Exception: " + e);

}


textpane.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));
```

```
        pane.getViewport().add(textpane);

        panel.add(pane);



        add(panel);

        setSize(new Dimension(380, 320));



        setLocationRelativeTo(null);

        setVisible(true);



    }



    public static void main(String[] args) {



        new TextPane();



    }

}
```

This is the html code, that we are loading into the **JTextPane** component. The component does not handle scrolling.

```
<html>

<head>
```

```
<title>A simple html document</title>

</head>

<body>



<h2>A simple html document</h2>







<p>

<b>JTextPane</b> can display html documents.

</p>



<br>



<pre>

 JScrollPane pane = new JScrollPane();

 JTextPane textpane = new JTextPane();



 textpane.setContentType("text/html");

 textpane.setEditable(false);

</pre>
```

```
<br>

<small>The Java Swing tutorial, 2007</small>



</body>



</html>
```

In our example we show a **JTextPane** component and load a html document. Example shows formatting capabilities of the component.

```
JTextPane textpane = new JTextPane();




textpane.setContentType("text/html");

textpane.setEditable(false);
```

We create a **JTextPane** component, set the content of the component to be a html document and disable editing.

```
String cd = System.getProperty("user.dir") + "/";
```

Here we determine the current working directory of the user. The html document is located there.

```
try {

    textpane.setPage("File:///" + cd + "test.html");

} catch (IOException e) {

    System.out.println("Exception: " + e);
```

```
}
```

We load a html document into the pane.

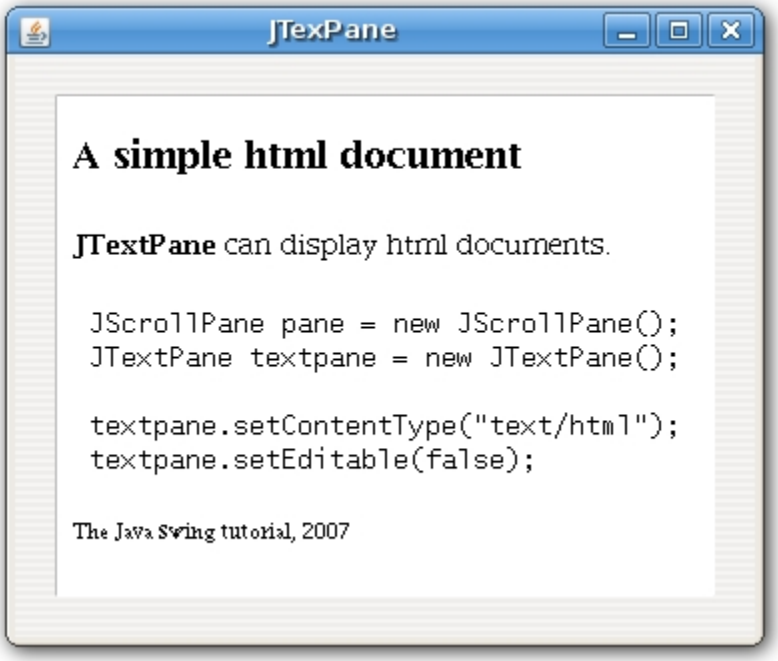


Figure: JTextPane

Java Swing model architecture

Swing engineers created the Swing toolkit implementing a modified Model View Controller design pattern. This enables efficient handling of data and using pluggable look and feel at runtime.

The traditional MVC pattern divides an application into three parts. A model, a view and a cotroller. The model represents the data in the application. The view is the visual representation of the data. And finally the controller processes and responds to events, typically user actions, and may invoke changes on the model. The idea is to separate the data access and business logic from data presentation and user interaction, by introducing an intermediate component: the controller.

The Swing toolkit uses a modified MVC design pattern. The Swing has single **UI object** for both the view and the controller. This modified MVC is sometimes called a **separable model architecture**.

In the Swing toolkit, every component has it's model. Even the basic ones like buttons. There are two kinds of models in Swing toolkit.

- state models
- data models

The state models handle the state of the component. For example the model keeps track whether the component is selected or pressed. The data models handle data, they work with. A list component keeps a list of items, it is displaying.

For Swing developer it means, that we often need to get a model instance in order to manipulate the data in the component. But there are exceptions. For convenience, there are some methods that return data without the model.

```
public int getValue() {

    return getModel().getValue();

}
```

For example the **getValue()** method of the **JSlider** component. The developer does not need to work with the model directly. Instead, the access to the model is done behind the scenes. It would be an overkill to work with models directly in such simple situations. Because of this, the Swing tookit provides some convenience methods like the previous one.

To query the state of the model, we have two kinds of notifications.

- lightweight notification
- stateful notification

The lightweight notification uses a **ChangeListener** class. We have only one single event (**ChangeEvent)** for all notifications coming from the component. For more complicated components, the stateful notification is used. For such notifications, we have different kinds of events. For example the **JList** component has **ListDataEvent** and**ListSelectionEvent**.

If we do not set a model for a component, a default one is created. For example the button component has**DefaultButtonModel** model

```
 public JButton(String text, Icon icon) {

 // Create the model

 setModel(new DefaultButtonModel());



 // initialize

 init(text, icon);

}
```

If we look at the JButton.java source file, we find out, that the default model is created at the construction of the component.

ButtonModel

The model is used for various kinds of buttons like push buttons, check boxes, radio boxes and for menu items. The following example illustrates the model for a **JButton**. We can manage only the state of the button, because no data can be associated with a push button.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.DefaultButtonModel;

import javax.swing.JButton;

import javax.swing.JCheckBox;

import javax.swing.JFrame;

import javax.swing.JLabel;
```

```
import javax.swing.JPanel;

import javax.swing.event.ChangeEvent;

import javax.swing.event.ChangeListener;




public class ButtonModel extends JFrame {



    private JButton ok;

    private JLabel enabled;

    private JLabel pressed;

    private JLabel armed;



    public ButtonModel() {



        setTitle("ButtonModel");



        JPanel panel = new JPanel();

        panel.setLayout(null);



        ok = new JButton("ok");

        JCheckBox cb = new JCheckBox("Enabled", true);
```

```
        ok.setBounds(40, 30, 80, 25);
        ok.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {

            DefaultButtonModel model = (DefaultButtonModel) ok.getModel();
            if (model.isEnabled())
                enabled.setText("Enabled: true");
            else
                enabled.setText("Enabled: false");

            if (model.isArmed())
                armed.setText("Armed: true");
            else
                armed.setText("Armed: false");

            if (model.isPressed())
                pressed.setText("Pressed: true");
            else
                pressed.setText("Pressed: false");
            }
```

```
        });


        cb.addActionListener(new ActionListener() {


            public void actionPerformed(ActionEvent e) {
                if (ok.isEnabled())
                    ok.setEnabled(false);
                else
                    ok.setEnabled(true);
            }
        });


        cb.setBounds(180, 30, 100, 25);


        enabled = new JLabel("Enabled: true");
        enabled.setBounds(40, 90, 90, 25);
        pressed = new JLabel("Pressed: false");
        pressed.setBounds(40, 120, 90, 25);
        armed  = new JLabel("Armed: false");
        armed.setBounds(40, 150, 90, 25);
```

```
        panel.add(ok);
        panel.add(cb);
        panel.add(enabled);
        panel.add(pressed);
        panel.add(armed);

        add(panel);

        setSize(350, 250);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        new ButtonModel();
    }
}
```

In our example, we have a button, check box and three labels. The labels represent three properties of the button. Whether it is pressed, disabled or armed.

```
ok.addChangeListener(new ChangeListener() {
```

We use a lightweight **ChangeListener** to listen for button state changes.

```
DefaultButtonModel model = (DefaultButtonModel) ok.getModel();
```

Here we get the default button model.

```
if (model.isEnabled())

    enabled.setText("Enabled: true");

else

    enabled.setText("Enabled: false");
```

We query the model, whether the button is enabled or not. We update the label accordingly.

```
if (ok.isEnabled())

    ok.setEnabled(false);

else

    ok.setEnabled(true);
```

The check box enables or disables the button. To enable the ok button, we call the **setEnable()** method. So we change the state of the button. Where is the model? The answer lies in the AbstractButton.java file.

```
  public void setEnabled(boolean b) {

      if (!b && model.isRollover()) {

          model.setRollover(false);

      }
```

```
        super.setEnabled(b);

        model.setEnabled(b);

    }
```

The answer is, that internally, we the Swing toolkit works with a model. The **setEnable()** is another convenience method for programmers.

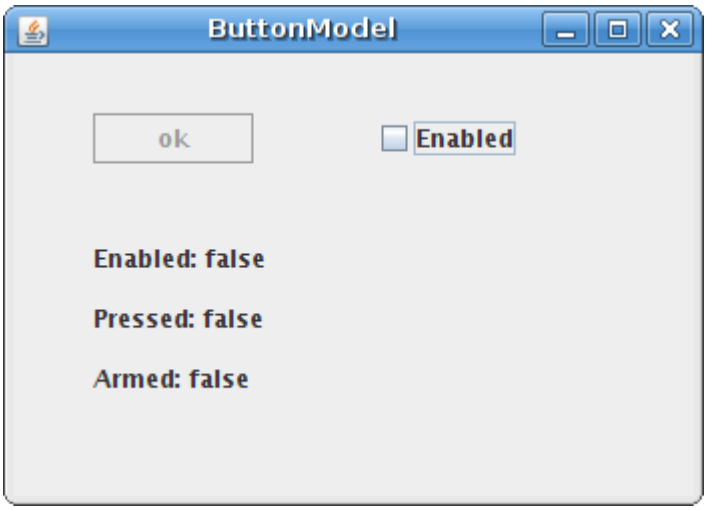


Figure: ButtonModel

Custom ButtonModel

In the previous example, we used a default button model. In the following code example we will use our own button model.

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.ButtonModel;

import javax.swing.DefaultButtonModel;
```

```
import javax.swing.JButton;

import javax.swing.JCheckBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;




public class ButtonModel2 extends JFrame {



    private JButton ok;

    private JLabel enabled;

    private JLabel pressed;

    private JLabel armed;



    public ButtonModel2() {



        setTitle("ButtonModel");



        JPanel panel = new JPanel();

        panel.setLayout(null);
```

```
        ok = new JButton("ok");
        JCheckBox cb = new JCheckBox("Enabled", true);


        ok.setBounds(40, 30, 80, 25);


        cb.addActionListener(new ActionListener() {


            public void actionPerformed(ActionEvent e) {
                if (ok.isEnabled())
                    ok.setEnabled(false);
                else
                    ok.setEnabled(true);
            }
        });


        cb.setBounds(180, 30, 100, 25);


        enabled = new JLabel("Enabled: true");
        enabled.setBounds(40, 90, 90, 25);
        pressed = new JLabel("Pressed: false");
        pressed.setBounds(40, 120, 90, 25);
```

```
        armed  = new JLabel("Armed: false");
        armed.setBounds(40, 150, 90, 25);


        ButtonModel model = new DefaultButtonModel() {
            public void setEnabled(boolean b) {
                if (b)
                    enabled.setText("Pressed: true");
                else
                    enabled.setText("Pressed: false");

                super.setEnabled(b);
            }

            public void setArmed(boolean b) {
                if (b)
                    armed.setText("Armed: true");
                else
                    armed.setText("Armed: false");

                super.setArmed(b);
            }
```

```
            public void setPressed(boolean b) {
                if (b)
                    pressed.setText("Pressed: true");
                else
                    pressed.setText("Pressed: false");

                super.setPressed(b);
            }

        };

        ok.setModel(model);

        panel.add(ok);
        panel.add(cb);
        panel.add(enabled);
        panel.add(pressed);
        panel.add(armed);

        add(panel);
```

```
        setSize(350, 250);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        new ButtonModel2();
    }
}
```

This example does the same thing as the previous one. The difference is that we don't use a change listener and we use a custom button model.

```
ButtonModel model = new DefaultButtonModel() {
```

We create a button model and overwrite the necessary methods.

```
public void setEnabled(boolean b) {

    if (b)

        enabled.setText("Pressed: true");

    else

        enabled.setText("Pressed: false");
```

```
    super.setEnabled(b);

}
```

We overwrite the **setEnabled()** method and add some functionality there. We must not forget to call the parent method as well to procede with the processing.

```
ok.setModel(model);
```

We set the custom model for the button.

JList models

Several components have two models. The **JList** component has the following models: **ListModel** and**ListSelectionModel**. The ListModel handles data. And the ListSelectionModel works with the GUI. The following example shows both models.

```
import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;



import javax.swing.BorderFactory;

import javax.swing.Box;

import javax.swing.BoxLayout;
```

```
import javax.swing.DefaultListModel;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JList;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.ListSelectionModel;




public class List extends JFrame {


    private DefaultListModel model;

    private JList list;


    public List() {


        setTitle("JList models");


        model = new DefaultListModel();

        model.addElement("Amelie");
```

```
        model.addElement("Aguirre, der Zorn Gottes");

        model.addElement("Fargo");

        model.addElement("Exorcist");

        model.addElement("Schindler list");


        JPanel panel = new JPanel();

        panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));


        JPanel leftPanel = new JPanel();

        JPanel rightPanel = new JPanel();


        leftPanel.setLayout(new BorderLayout());

        rightPanel.setLayout(new BoxLayout(rightPanel, BoxLayout.Y_AXIS));


        list = new JList(model);

        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        list.setBorder(BorderFactory.createEmptyBorder(2, 2, 2, 2));


        list.addMouseListener(new MouseAdapter() {


            public void mouseClicked(MouseEvent e) {
```

```
                if(e.getClickCount() == 2){

                  int index = list.locationToIndex(e.getPoint());

                  Object item = model.getElementAt(index);

                  String text = JOptionPane.showInputDialog("Rename
item", item);

                  String newitem = null;

                  if (text != null)

                     newitem = text.trim();

                  else

                     return;



                  if (!newitem.isEmpty()) {

                    model.remove(index);

                    model.add(index, newitem);

                    ListSelectionModel selmodel =
list.getSelectionModel();

                    selmodel.setLeadSelectionIndex(index);

                  }

                }

            }



        });
```

```
        JScrollPane pane = new JScrollPane();
        pane.getViewport().add(list);
        leftPanel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20,
20));

        leftPanel.add(pane);

        JButton removeall = new JButton("Remove All");
        JButton add = new JButton("Add");
        add.setMaximumSize(removeall.getMaximumSize());
        JButton rename = new JButton("Rename");
        rename.setMaximumSize(removeall.getMaximumSize());
        JButton delete = new JButton("Delete");
        delete.setMaximumSize(removeall.getMaximumSize());

        add.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String text = JOptionPane.showInputDialog("Add a new
item");
                String item = null;
```

```
                if (text != null)
                    item = text.trim();
                else
                    return;

                if (!item.isEmpty())
                    model.addElement(item);
            }
        });

        delete.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                ListSelectionModel selmodel = list.getSelectionModel();
                int index = selmodel.getMinSelectionIndex();
                if (index >= 0)
                    model.remove(index);
            }

        });

        rename.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                ListSelectionModel selmodel = list.getSelectionModel();
                int index = selmodel.getMinSelectionIndex();
                if (index == -1) return;
                Object item = model.getElementAt(index);
                String text = JOptionPane.showInputDialog("Rename item",
item);
                String newitem = null;

                if (text != null) {
                    newitem = text.trim();
                } else
                    return;

                if (!newitem.isEmpty()) {
                    model.remove(index);
                    model.add(index, newitem);
                }
            }
        });
```

```
        removeall.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    model.clear();
                }
        });

        rightPanel.add(add);
        rightPanel.add(Box.createRigidArea(new Dimension(0,4)));
        rightPanel.add(rename);
        rightPanel.add(Box.createRigidArea(new Dimension(0,4)));
        rightPanel.add(delete);
        rightPanel.add(Box.createRigidArea(new Dimension(0,4)));
        rightPanel.add(removeall);

        rightPanel.setBorder(BorderFactory.createEmptyBorder(0, 0, 0,
20));

        panel.add(leftPanel);
        panel.add(rightPanel);
```

```
        add(panel);



        setSize(350, 250);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setVisible(true);

    }



    public static void main(String[] args) {

        new List();

    }

}
```

The example shows a list component and four buttons. The buttons control the data in the list component. The example is a bit larger, because we did some additional checks there. We do not allow to input empty spaces into the list component.

```
model = new DefaultListModel();


model.addElement("Amelie");


model.addElement("Aguirre, der Zorn Gottes");


...
```

We create a list model and add elements into it.

```
list = new JList(model);
```

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

list.setBorder(BorderFactory.createEmptyBorder(2, 2, 2, 2));
```

We create a list component. The parameter of the constructor is the model, we have created. We put the list into the single selection mode. We also put some space around the list.

```
if (text != null)

    item = text.trim();

else

    return;



if (!item.isEmpty())

    model.addElement(item);
```

We add only items that are not equal to null and are not empty. e.g. that contain at least one character other than white space. It makes no sense to add white spaces or null values into the list.

```
ListSelectionModel selmodel = list.getSelectionModel();

int index = selmodel.getMinSelectionIndex();

if (index >= 0)

    model.remove(index);
```

This is the code, that runs when we press the delete button. In order to delete an item from the list, it must be selected. So we must figure out the currently selected item. For this, we call the **getSelectionModel()** method This is a GUI work, so we use

a **ListSelectionModel**. Removing an item is working with data. For that we use the list data model.

So, in our example we used both list models. We called add(), remove() and clear() methods of the list data model to work with our data. And we used a list selection model in order to find out the selected item, which is a GUI job.

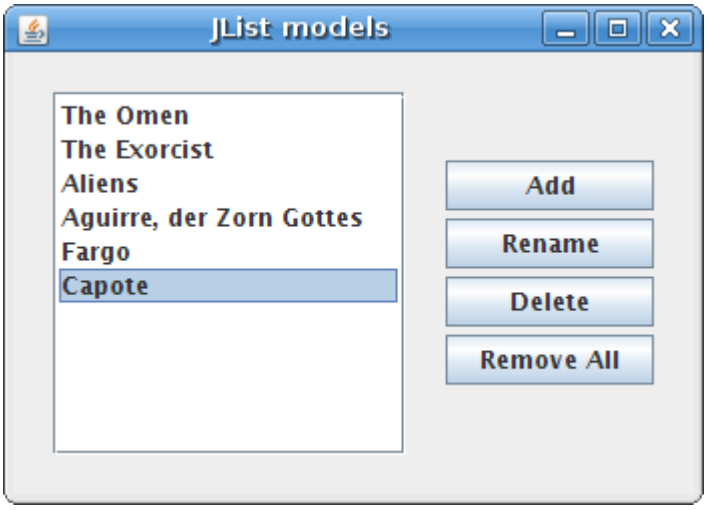


Figure: List Models

A document model

This is an excellent example of a separation of a data from the visual representation. In a **JTextPane** component, we have a **StyledDocument** for setting the style of the text data.

```
import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.BorderFactory;
```

```
import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTextPane;

import javax.swing.JToolBar;

import javax.swing.text.Style;

import javax.swing.text.StyleConstants;

import javax.swing.text.StyledDocument;



public class DocumentModel extends JFrame {

    private StyledDocument doc;

    private JTextPane textpane;

    public DocumentModel() {

        setTitle("Document Model");
```

```
        JToolBar toolbar = new JToolBar();


        ImageIcon bold = new ImageIcon("bold.png");

        ImageIcon italic = new ImageIcon("italic.png");

        ImageIcon strike = new ImageIcon("strike.png");

        ImageIcon underline = new ImageIcon("underline.png");


        JButton boldb = new JButton(bold);

        JButton italb = new JButton(italic);

        JButton strib = new JButton(strike);

        JButton undeb = new JButton(underline);


        toolbar.add(boldb);

        toolbar.add(italb);

        toolbar.add(strib);

        toolbar.add(undeb);


        add(toolbar, BorderLayout.NORTH);


        JPanel panel = new JPanel();

        panel.setLayout(new BorderLayout());
```

```
        panel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));



        JScrollPane pane = new JScrollPane();

        textpane = new JTextPane();

        textpane.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));



        doc = textpane.getStyledDocument();



        Style style = textpane.addStyle("Bold", null);

        StyleConstants.setBold(style, true);



        style = textpane.addStyle("Italic", null);

        StyleConstants.setItalic(style, true);



        style = textpane.addStyle("Underline", null);

        StyleConstants.setUnderline(style, true);



        style = textpane.addStyle("Strike", null);

        StyleConstants.setStrikeThrough(style, true);
```

```
        boldb.addActionListener(new ActionListener() {



            public void actionPerformed(ActionEvent e) {

                doc.setCharacterAttributes(textpane.getSelectionStart(),

                    textpane.getSelectionEnd() -
textpane.getSelectionStart(),

                    textpane.getStyle("Bold"), false);

            }

        });



        italb.addActionListener(new ActionListener() {



            public void actionPerformed(ActionEvent e) {

                doc.setCharacterAttributes(textpane.getSelectionStart(),

                    textpane.getSelectionEnd() -
textpane.getSelectionStart(),

                    textpane.getStyle("Italic"), false);

            }



        });



        strib.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {

                doc.setCharacterAttributes(textpane.getSelectionStart(),

                    textpane.getSelectionEnd() -
textpane.getSelectionStart(),

                    textpane.getStyle("Strike"), false);

            }



        });



        undeb.addActionListener(new ActionListener() {



            public void actionPerformed(ActionEvent e) {

                doc.setCharacterAttributes(textpane.getSelectionStart(),

                    textpane.getSelectionEnd() -
textpane.getSelectionStart(),

                    textpane.getStyle("Underline"), false);

            }

        });



        pane.getViewport().add(textpane);

        panel.add(pane);
```

```
        add(panel);


        setSize(new Dimension(380, 320));
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }


    public static void main(String[] args) {
        new DocumentModel();
    }
}
```

The example has a text pane and a toolbar. In the toolbar, we have four buttons, that change attributes of the text.

```
doc = textpane.getStyledDocument();
```

Here we get the styled document, which is a model for the text pane component.

```
Style style = textpane.addStyle("Bold", null);

StyleConstants.setBold(style, true);
```

A style is a set of text attributes, such as color, size. Here we register a bold style for the text pane component. The registered styles can be retrieved at any time.

```
doc.setCharacterAttributes(textpane.getSelectionStart(),

    textpane.getSelectionEnd() - textpane.getSelectionStart(),

    textpane.getStyle("Bold"), false);
```

Here we change the attributes of the text. The parameters are the offset, length of the selection, the style and the boolean value replace. The offset is the beginning of the text, where we apply the bold text. We get the length value by substracting the selection end and selection start values. Boolean value false means, we are not replacing an old style with a new one, but we merge them. This means, if the text is underlined and we make it bold, the result is an underlined bold text.

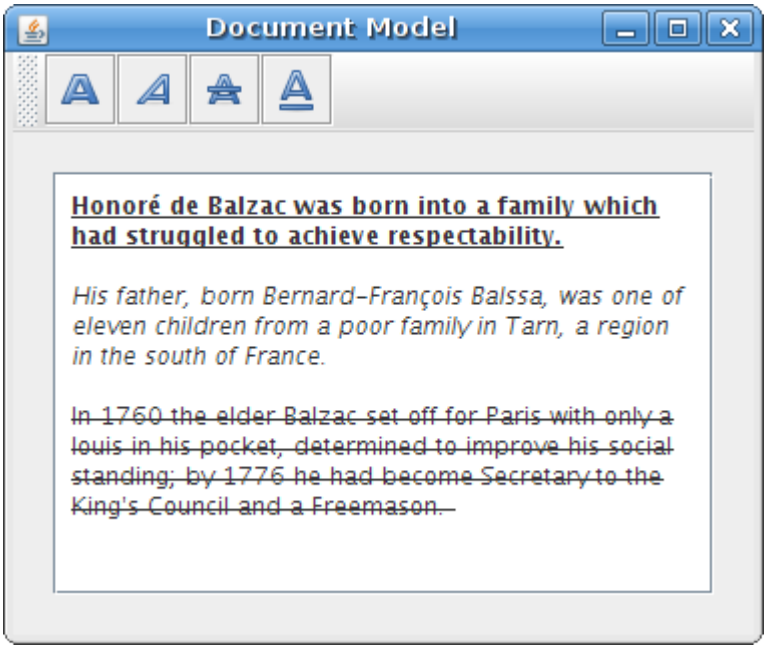


Figure: Document model

Drag and Drop in Swing

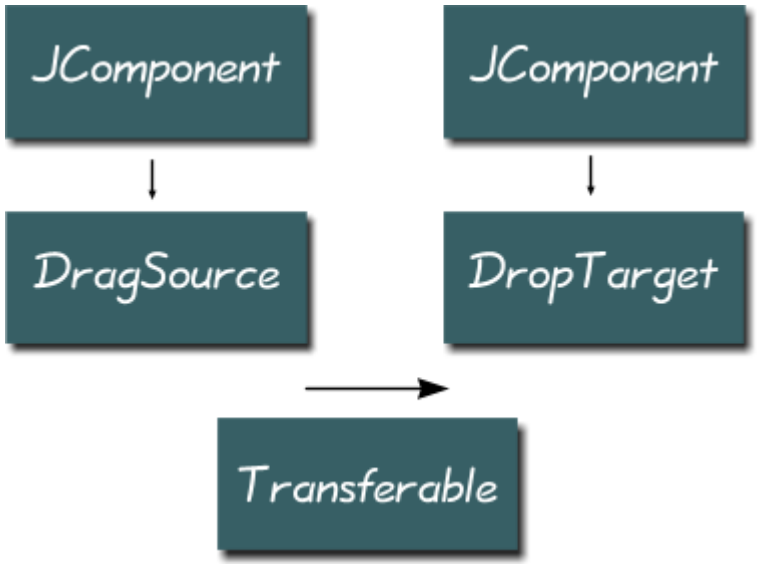
In computer graphical user interfaces, drag-and-drop is the action of (or support for the action of) clicking on a virtual object and dragging it to a different location or onto another

virtual object. In general, it can be used to invoke many kinds of actions, or create various types of associations between two abstract objects. (Wikipedia)

Drag and drop functionality is one of the most visible aspects of the graphical user interface. Drag and drop operation enables users to do complex things intuitively.

Usually, we can drag and drop two things. Data or some graphical objects. If we drag an image from one application to another, we drag and drop binary data. If we drag a tab in Firefox and move it to another place, we drag and drop a graphical component.

The sheer amount of various classes involved with drag and drop operations in Java Swing toolkit might be overwhelming. The best way how to cope with this complexity is to create a small example for all situations. And slowly make it to more complex examples.



The component, where the drag operation begins must have a **DragSource** object registered. A **DropTarget** is an object responsible for accepting drops in an drag and drop operation. A **Transferable** encapsulates data being transferred. The transferred data can be of various type. A **DataFlavor** object provides information about the data being transferred.

Several Swing components have already a built-in support for drag and drop operations. In such cases, a Swing programmer uses a **TransferHandler** to manage the drag and drop

functionality. In situations, where there is no built-in support, the programmer has to create everything from scratch.

A simple drag and drop example

We will demonstrate a simple drag and drop example. We will work with built-in drag and drop support. We will utilize a**TransferHandler** class.

```
import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JTextField;

import javax.swing.TransferHandler;




public class SimpleDnD extends JFrame {


    JTextField field;

    JButton button;


    public SimpleDnD() {


        setTitle("Simple Drag & Drop");


        setLayout(null);
```

```
        button = new JButton("Button");
        button.setBounds(200, 50, 90, 25);

        field = new JTextField();
        field.setBounds(30, 50, 150, 25);

        add(button);
        add(field);

        field.setDragEnabled(true);
        button.setTransferHandler(new TransferHandler("text"));

        setSize(330, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String[] args) {
        new SimpleDnD();
```

```
    }

}
```

In our example we have a text field and a button. We can drag a text from the field and drop it onto the button.

```
field.setDragEnabled(true);
```

The text field has a built in support for dragging. We must enable it.

```
button.setTransferHandler(new TransferHandler("text"));
```

The **TransferHandler** is a class responsible for transfering data between components. The constructor takes a property name as a parameter.

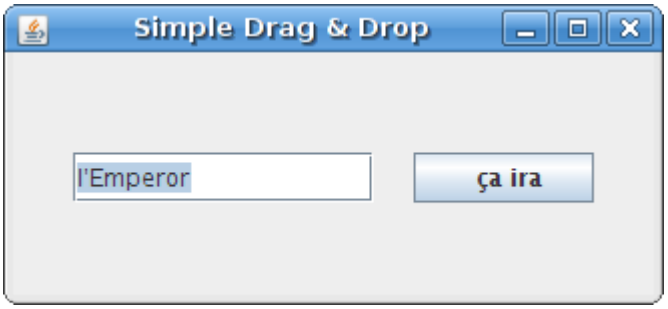


Figure: Simple drag & drop example

Icon drag & drop

Some of the Java Swing components do not have built in drag support. **JLabel** component is such a component. We have to code the drag functionality ourselves.

We will drag and drop icons. In the previous example, we used a text property. This time we will use an icon property.

```
import java.awt.FlowLayout;

import java.awt.event.MouseAdapter;
```

```
import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;



import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JComponent;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.TransferHandler;




public class IconDnD extends JFrame {




    public IconDnD() {


        setTitle("Icon Drag & Drop");


        JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 50,
15));
```

```
        ImageIcon icon1 = new ImageIcon("sad.png");

        ImageIcon icon2 = new ImageIcon("plain.png");

        ImageIcon icon3 = new ImageIcon("crying.png");


        JButton button = new JButton(icon2);

        button.setFocusable(false);


        JLabel label1  = new JLabel(icon1, JLabel.CENTER);

        JLabel label2  = new JLabel(icon3, JLabel.CENTER);


        MouseListener listener = new DragMouseAdapter();

        label1.addMouseListener(listener);

        label2.addMouseListener(listener);


        label1.setTransferHandler(new TransferHandler("icon"));

        button.setTransferHandler(new TransferHandler("icon"));

        label2.setTransferHandler(new TransferHandler("icon"));


        panel.add(label1);

        panel.add(button);
```

```
        panel.add(label2);

        add(panel);



        pack();

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLocationRelativeTo(null);

        setVisible(true);

    }



    class DragMouseAdapter extends MouseAdapter {

        public void mousePressed(MouseEvent e) {

            JComponent c = (JComponent) e.getSource();

            TransferHandler handler = c.getTransferHandler();

            handler.exportAsDrag(c, e, TransferHandler.COPY);

        }

    }



    public static void main(String[] args) {

        new IconDnD();

    }

}
```

In the code example, we have two labels and a button. Each component displays an icon. The two labels enable drag gestures, the button accepts a drop gesture.

```
MouseListener listener = new DragMouseAdapter();

label1.addMouseListener(listener);

label2.addMouseListener(listener);
```

The drag support is not enabled by default for the label. We register a custom mouse adapter for both labels.

```
label1.setTransferHandler(new TransferHandler("icon"));

button.setTransferHandler(new TransferHandler("icon"));

label2.setTransferHandler(new TransferHandler("icon"));
```

Each of the three components has a **TransferHandler** class for an icon property. The **TransferHandler** is needed for both drag sources and drag targets as well.

```
JComponent c = (JComponent) e.getSource();

TransferHandler handler = c.getTransferHandler();

handler.exportAsDrag(c, e, TransferHandler.COPY);
```

These code lines initiate the drag support. We get the drag source. In our case it is a label instance. We get it's transfer handler object. And finally initiate the drag support with the **exportAsDrag()** method call.

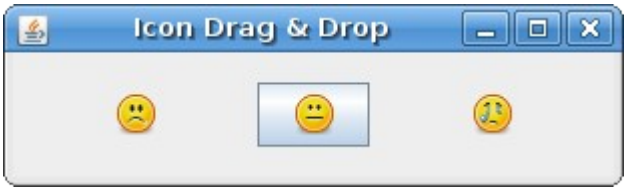


Figure: Icon drag & drop example

Custom JList drop example

Some components do not have a default drop support. One of them is a JList component. There is a good reason for this. We don't know, if the data will be inserted into one row, or two or more rows. So we must implement manually the drop support for the list component.

The comma separated text will be inserted into two or more rows. Text without a comma will go into one row.

```
import java.awt.Dimension;

import java.awt.FlowLayout;

import java.awt.datatransfer.DataFlavor;

import java.awt.datatransfer.Transferable;



import javax.swing.DefaultListModel;

import javax.swing.DropMode;

import javax.swing.JFrame;

import javax.swing.JList;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTextField;

import javax.swing.ListSelectionModel;

import javax.swing.TransferHandler;
```

```
public class ListDrop extends JFrame {



    JTextField field;

    DefaultListModel model;



    public ListDrop() {



        setTitle("ListDrop");



        JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 15,
15));



        JScrollPane pane = new JScrollPane();

        pane.setPreferredSize(new Dimension(180, 150));



        model = new DefaultListModel();

        JList list = new JList(model);



        list.setDropMode(DropMode.INSERT);

        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        list.setTransferHandler(new ListHandler());
```

```
        field = new JTextField("");
        field.setPreferredSize(new Dimension(150, 25));
        field.setDragEnabled(true);

        panel.add(field);
        pane.getViewport().add(list);
        panel.add(pane);

        add(panel);

        pack();

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }


    private class ListHandler extends TransferHandler {
        public boolean canImport(TransferSupport support) {
```

```
            if (!support.isDrop()) {

                return false;

            }



            return
support.isDataFlavorSupported(DataFlavor.stringFlavor);

        }



        public boolean importData(TransferSupport support) {

            if (!canImport(support)) {

              return false;

            }



            Transferable transferable = support.getTransferable();

            String line;

            try {

              line = (String)
transferable.getTransferData(DataFlavor.stringFlavor);

            } catch (Exception e) {

              return false;

            }
```

```
            JList.DropLocation dl = (JList.DropLocation)
support.getDropLocation();

            int index = dl.getIndex();



            String[] data = line.split(",");

            for (String item: data) {

                if (!item.isEmpty())

                   model.add(index++, item.trim());

            }

            return true;

         }

    }



    public static void main(String[] args) {

        new ListDrop();

    }

}
```

In the above example, we have a text field and a list component. The text in the text field can be dragged and dropped into the list. If the text is comma separated, the words will be split into rows. If not, the text will be inserted into one row.

```
list.setDropMode(DropMode.INSERT);
```

Here we specify a drop mode. The **DropMode.INSERT** specifies, that we are going to insert new items into the list component. If we chose **DropMode.INSERT**, we would drop new items onto the existing ones.

```
list.setTransferHandler(new ListHandler());
```

We set a custom transfer handler class.

```
field.setDragEnabled(true);
```

We enable the drag support for the text field component.

```
public boolean canImport(TransferSupport support) {

    if (!support.isDrop()) {

        return false;

    }

    return support.isDataFlavorSupported(DataFlavor.stringFlavor);

 }
```

This method tests suitability of a drop operation. Here we filter out the clipboard paste operations and allow only String drop operations. If the method returns false, the drop operation is cancelled.

```
public boolean importData(TransferSupport support) {

...

}
```

The **importData()** method transfers the data from the clipboard or from the drag and drop operation to the drop location.

```
Transferable transferable = support.getTransferable();
```

The **Transferable** is the class, where the data is bundled.

```
line = (String) transferable.getTransferData(DataFlavor.stringFlavor);
```

We retrieve our data.

```
JList.DropLocation dl = (JList.DropLocation) support.getDropLocation();

int index = dl.getIndex();
```

We get a drop location for the list. We retrieve the index, where the data will be inserted.

```
String[] data = line.split(",");

for (String item: data) {

    if (!item.isEmpty())

        model.add(index++, item.trim());

}
```

Here we split the text into parts and insert it into one or more rows.

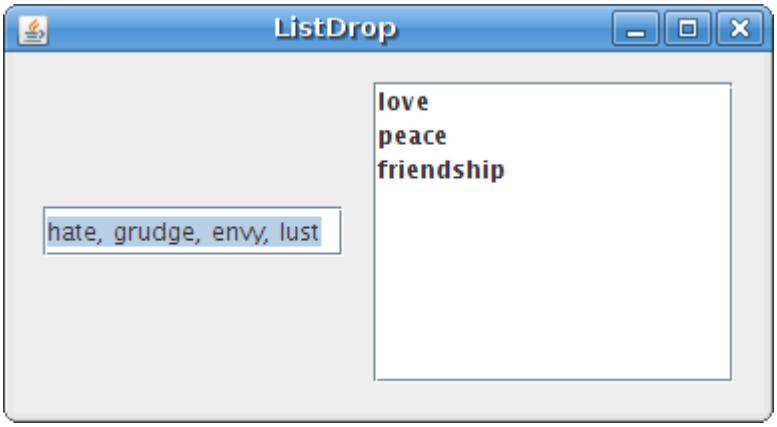


Figure: JList drop example

The previous examples used components with built-in drag and drop support. Next we are going to create a drag and drop functionality from scratch.

Drag Gesture

In the following example we will inspect a simple drag gesture. We will work with several classes needed to create a drag gesture.
A **DragSource**, **DragGestureEvent**, **DragGestureListener**, **Transferable**.

```
import java.awt.Color;

import java.awt.Cursor;

import java.awt.Dimension;

import java.awt.FlowLayout;

import java.awt.datatransfer.DataFlavor;

import java.awt.datatransfer.Transferable;

import java.awt.dnd.DnDConstants;

import java.awt.dnd.DragGestureEvent;

import java.awt.dnd.DragGestureListener;

import java.awt.dnd.DragSource;



import javax.swing.JFrame;

import javax.swing.JPanel;




public class DragGesture extends JFrame implements

    DragGestureListener, Transferable {
```

```
    public DragGesture() {


        setTitle("Drag Gesture");


        JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 50,
15));


        JPanel left = new JPanel();
        left.setBackground(Color.red);
        left.setPreferredSize(new Dimension(120, 120));


        DragSource ds = new DragSource();
        ds.createDefaultDragGestureRecognizer(left,
            DnDConstants.ACTION_COPY, this);


        panel.add(left);
        add(panel);


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

```
        setLocationRelativeTo(null);

        setVisible(true);

    }



    public void dragGestureRecognized(DragGestureEvent event) {

        System.out.println("grag gesture");

        Cursor cursor = null;

        if (event.getDragAction() == DnDConstants.ACTION_COPY) {

            cursor = DragSource.DefaultCopyDrop;

        }

        event.startDrag(cursor, this);

    }



    public static void main(String[] args) {

        new DragGesture();

    }



    public Object getTransferData(DataFlavor flavor) {

        return null;

    }
```

```
    public DataFlavor[] getTransferDataFlavors() {

        return new DataFlavor[0];

    }



    public boolean isDataFlavorSupported(DataFlavor flavor) {

        return false;

    }

}
```

This simple example demostrates a drag gesture. The drag gesture is created, when we click on a component and move a mouse pointer, while the button is pressed. The example will show, how we can create a DragSource for a component.

```
public class DragGesture extends JFrame implements

   DragGestureListener, Transferable {
```

The DragGesture implements two interfaces. The **DragGestureListener** will listen for drag gestures. The **Transferable**handles data for a transfer operation. In the example, we will not transfer any data. We will only demonstrate a drag gesture. So the three necessary methods of the Transferable interface are left unimplemented.

```
DragSource ds = new DragSource();

ds.createDefaultDragGestureRecognizer(left,

    DnDConstants.ACTION_COPY, this);
```

Here we create a **DragSource** object and register it for the left panel. The DragSource is the entity responsible for the initiation of the Drag and Drop operation.
The **createDefaultDragGestureRecognizer()** associates a drag source and**DragGestureListener** with a particular component.

```
public void dragGestureRecognized(DragGestureEvent event) {




}
```

The **dragGestureRecognized()** method responds to a drag gesture.

```
Cursor cursor = null;

if (event.getDragAction() == DnDConstants.ACTION_COPY) {

    cursor = DragSource.DefaultCopyDrop;

}

event.startDrag(cursor, this);
```

The **startDrag()** method of the DragGestureEvent finally starts the drag operation. We will specify two parameters. The cursor type and the Transferable object.

```
public Object getTransferData(DataFlavor flavor) {

    return null;

}




public DataFlavor[] getTransferDataFlavors() {

   return new DataFlavor[0];

}
```

```
public boolean isDataFlavorSupported(DataFlavor flavor) {

    return false;

}
```

The object that implements the Transferable interface must implement these three methods. As I have already mentioned, we left these methods unimplemented for now.

A complex drag and drop example

In the following example, we create a complex drag and drop example. We create a drag source a drop target and a transferable object.

```
import java.awt.Color;

import java.awt.Cursor;

import java.awt.Dimension;

import java.awt.FlowLayout;

import java.awt.datatransfer.DataFlavor;

import java.awt.datatransfer.Transferable;

import java.awt.datatransfer.UnsupportedFlavorException;

import java.awt.dnd.DnDConstants;

import java.awt.dnd.DragGestureEvent;

import java.awt.dnd.DragGestureListener;

import java.awt.dnd.DragSource;

import java.awt.dnd.DropTarget;

import java.awt.dnd.DropTargetAdapter;
```

```
import java.awt.dnd.DropTargetDropEvent;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;



import javax.swing.JButton;

import javax.swing.JColorChooser;

import javax.swing.JFrame;

import javax.swing.JPanel;




public class ComplexExample extends JFrame

    implements DragGestureListener {


    JPanel panel;

    JPanel left;


    public ComplexExample() {


        setTitle("Complex Example");


        panel  = new JPanel(new FlowLayout(FlowLayout.LEFT, 50, 15));
```

```
        JButton openb = new JButton("Choose Color");
        openb.setFocusable(false);

        left = new JPanel();
        left.setBackground(Color.red);
        left.setPreferredSize(new Dimension(100, 100));

        openb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                JColorChooser clr = new JColorChooser();
                Color color = clr.showDialog(panel, "Choose Color",
Color.white);
                left.setBackground(color);
            }
        });

        JPanel right = new JPanel();
        right.setBackground(Color.white);
        right.setPreferredSize(new Dimension(100, 100));
```

```
        new MyDropTargetListener(right);



        DragSource ds = new DragSource();

        ds.createDefaultDragGestureRecognizer(left,

            DnDConstants.ACTION_COPY, this);



        panel.add(openb);

        panel.add(left);

        panel.add(right);

        add(panel);



        pack();

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLocationRelativeTo(null);

        setVisible(true);

    }



    public void dragGestureRecognized(DragGestureEvent event) {

        Cursor cursor = null;

        JPanel panel = (JPanel) event.getComponent();
```

```
        Color color = panel.getBackground();


        if (event.getDragAction() == DnDConstants.ACTION_COPY) {
            cursor = DragSource.DefaultCopyDrop;
        }


        event.startDrag(cursor, new TransferableColor(color));
    }


    class MyDropTargetListener extends DropTargetAdapter {


        private DropTarget dropTarget;
        private JPanel panel;


     public MyDropTargetListener(JPanel panel) {
        this.panel = panel;


        dropTarget = new DropTarget(panel, DnDConstants.ACTION_COPY,
            this, true, null);
      }
```

```
      public void drop(DropTargetDropEvent event) {

        try {



          Transferable tr = event.getTransferable();

          Color color = (Color)
tr.getTransferData(TransferableColor.colorFlavor);



            if
(event.isDataFlavorSupported(TransferableColor.colorFlavor)) {



              event.acceptDrop(DnDConstants.ACTION_COPY);

              this.panel.setBackground(color);

              event.dropComplete(true);

              return;

            }

          event.rejectDrop();

        } catch (Exception e) {

          e.printStackTrace();

          event.rejectDrop();

        }

      }
```

```
    }

    public static void main(String[] args) {
        new ComplexExample();
    }
}

class TransferableColor implements Transferable {

    protected static DataFlavor colorFlavor =
        new DataFlavor(Color.class, "A Color Object");

    protected static DataFlavor[] supportedFlavors = {
        colorFlavor,
        DataFlavor.stringFlavor,
    };

    Color color;

    public TransferableColor(Color color) { this.color = color; }
```

```
    public DataFlavor[] getTransferDataFlavors() { return
supportedFlavors; }



    public boolean isDataFlavorSupported(DataFlavor flavor) {

    if (flavor.equals(colorFlavor) ||

        flavor.equals(DataFlavor.stringFlavor)) return true;

    return false;

  }




   public Object getTransferData(DataFlavor flavor)

        throws UnsupportedFlavorException

   {

     if (flavor.equals(colorFlavor))

         return color;

     else if (flavor.equals(DataFlavor.stringFlavor))

         return color.toString();

     else

         throw new UnsupportedFlavorException(flavor);

   }
```

```
}
```

The code example shows a button and two panels. The button displays a color chooser dialog and sets a color for the first panel. The color can be dragged into the second panel.

This example will enhance the previous one. We will add a drop target and a custom transferable object.

```
new MyDropTargetListener(right);
```

We register a drop target listener with the right panel.

```
 event.startDrag(cursor, new TransferableColor(color));
```

The **startDrag()** method has two parameters. A cursor and a **Transferable** object.

```
public MyDropTargetListener(JPanel panel) {

    this.panel = panel;



    dropTarget = new DropTarget(panel, DnDConstants.ACTION_COPY,

        this, true, null);

}
```

In the MyDropTargetListener we create a drop target object.

```
Transferable tr = event.getTransferable();

Color color = (Color) tr.getTransferData(TransferableColor.colorFlavor);



if (event.isDataFlavorSupported(TransferableColor.colorFlavor)) {
```

```
    event.acceptDrop(DnDConstants.ACTION_COPY);

    this.panel.setBackground(color);

    event.dropComplete(true);

    return;

 }
```

We get the data being transferred. In our case it is a color object. Here we set the color of the right panel.

```
event.rejectDrop();
```

If the conditions for a drag and drop operation are not fulfilled, we reject it.

```
protected static DataFlavor colorFlavor =

    new DataFlavor(Color.class, "A Color Object");
```

In the TransferableColor, we create a new **DataFlavor** object.

```
protected static DataFlavor[] supportedFlavors = {

    colorFlavor,

    DataFlavor.stringFlavor,

};
```

Here we specify, what data flavors we support. In our case it is a custom defined color flavor and a pre-defined**DataFlavor.stringFlavor**.

```
public Object getTransferData(DataFlavor flavor)
```

```
    throws UnsupportedFlavorException

{

  if (flavor.equals(colorFlavor))

      return color;

  else if (flavor.equals(DataFlavor.stringFlavor))

      return color.toString();

  else

      throw new UnsupportedFlavorException(flavor);

}
```

Return an object for a specific data flavor.

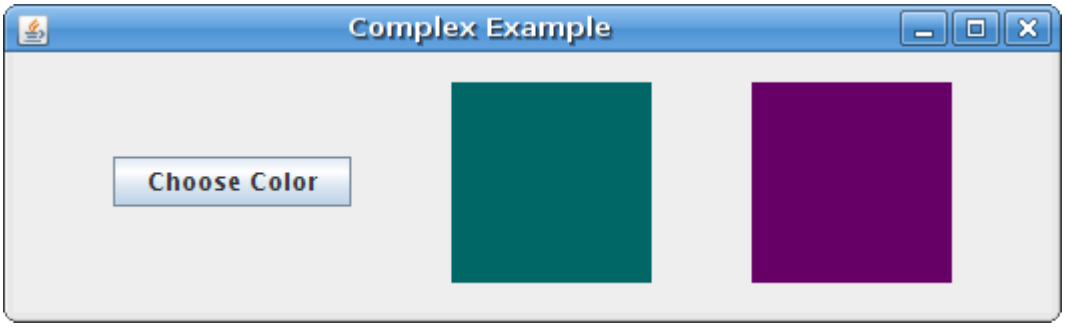


Figure: A complex example

Painting in Swing

Painting is used, when we want to change or enhance an existing widget. Or if we are creating a custom widget from scratch. To do the painting, we use the painting API provided by the Swing toolkit.

The painting is done within the **paintComponent()** method. In the painting process, we use the **Graphics2D** object.

2D Vector Graphics

There are two different computer graphics. Vector and raster graphics. Raster graphics represents images as a collection of pixels. Vector graphics is the use of geometrical primitives such as points, lines, curves or polygons to represent images. These primitives are created using mathematical equations.

Both types of computer graphics have advantages and disadvantages. The advantages of vector graphics over raster are:

- smaller size
- ability to zoom indefinitely
- moving, scaling, filling or rotating does not degrade the quality of an image

Types of primitives

- points
- lines
- polylines
- polygons
- circles
- ellipses
- Splines

Points

The most simple graphics primitive is point. It is a single dot on the window. Interesingly, there is no method to draw a point in Swing. (Or I could not find it.) To draw a point, I used a **drawLine()** method. I used one point twice.

```
import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Insets;



import javax.swing.JPanel;

import javax.swing.JFrame;



import java.util.Random;







public class Points extends JPanel {



    public void paintComponent(Graphics g) {

            super.paintComponent(g);
```

```
            Graphics2D g2d = (Graphics2D) g;


            g2d.setColor(Color.blue);


            for (int i=0; i<=1000; i++) {
                Dimension size = getSize();
                Insets insets = getInsets();


                int w =  size.width - insets.left - insets.right;
                int h =  size.height - insets.top - insets.bottom;


                Random r = new Random();
                int x = Math.abs(r.nextInt()) % w;
                int y = Math.abs(r.nextInt()) % h;
                g2d.drawLine(x, y, x, y);
            }
    }


    public static void main(String[] args) {


        Points points = new Points();
```

```
        JFrame frame = new JFrame("Points");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(points);

        frame.setSize(250, 200);

        frame.setLocationRelativeTo(null);

        frame.setVisible(true);

    }

}
```

One point is difficult to observe. Why not paint 1000 of them. In our example, we do so. We draw 1000 blue points on the panel.

```
g2d.setColor(Color.blue);
```

We will paint our points in blue color.

```
Dimension size = getSize();


Insets insets = getInsets();
```

The size of the window includes borders and titlebar. We don't paint there.

```
int w =  size.width - insets.left - insets.right;


int h =  size.height - insets.top - insets.bottom;
```

Here we calculate the area, where we will effectively paint our points.

```
Random r = new Random();


int x = Math.abs(r.nextInt()) % w;
```

```
int y = Math.abs(r.nextInt()) % h;
```

We get a random number in range of the size of area, that we computed above.

```
g2d.drawLine(x, y, x, y);
```

Here we draw the point. As I already said, we use a **drawLine()** method. We specify the same point twice.

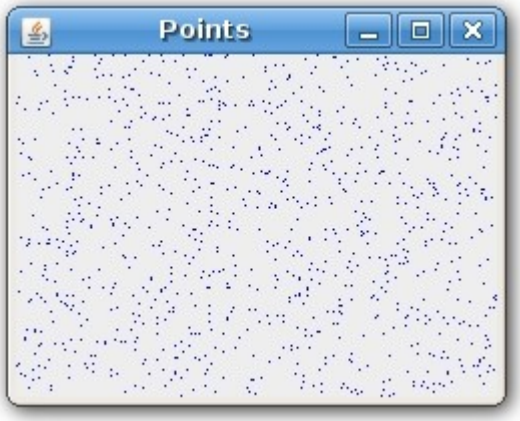


Figure: Points

Lines

A line is a simple graphics primitive. It is drawn using two points.

```
import java.awt.BasicStroke;

import java.awt.Graphics;

import java.awt.Graphics2D;



import javax.swing.JFrame;

import javax.swing.JPanel;
```

```
public class Lines extends JPanel {


    public void paintComponent(Graphics g) {
            super.paintComponent(g);


            Graphics2D g2d = (Graphics2D) g;


            float[] dash1 = { 2f, 0f, 2f };
            float[] dash2 = { 1f, 1f, 1f };
            float[] dash3 = { 4f, 0f, 2f };
            float[] dash4 = { 4f, 4f, 1f };


            g2d.drawLine(20, 40, 250, 40);


            BasicStroke bs1 = new BasicStroke(1, BasicStroke.CAP_BUTT,
                BasicStroke.JOIN_ROUND, 1.0f, dash1, 2f );


            BasicStroke bs2 = new BasicStroke(1, BasicStroke.CAP_BUTT,
                BasicStroke.JOIN_ROUND, 1.0f, dash2, 2f );
```

```
            BasicStroke bs3 = new BasicStroke(1, BasicStroke.CAP_BUTT,
                BasicStroke.JOIN_ROUND, 1.0f, dash3, 2f );

            BasicStroke bs4 = new BasicStroke(1, BasicStroke.CAP_BUTT,
                BasicStroke.JOIN_ROUND, 1.0f, dash4, 2f );

            g2d.setStroke(bs1);
            g2d.drawLine(20, 80, 250, 80);

            g2d.setStroke(bs2);
            g2d.drawLine(20, 120, 250, 120);

            g2d.setStroke(bs3);
            g2d.drawLine(20, 160, 250, 160);

            g2d.setStroke(bs4);
            g2d.drawLine(20, 200, 250, 200);

    }
```

```
    public static void main(String[] args) {



        Lines lines = new Lines();

        JFrame frame = new JFrame("Lines");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(lines);

        frame.setSize(280, 270);

        frame.setLocationRelativeTo(null);

        frame.setVisible(true);



    }

}
```

In the example, we draw five lines. The first line is drawn using the default values. Other will have a different **stroke**. The stroke is created using the **BasicStroke** class. It defines a basic set of rendering attributes for the outlines of graphics primitives.

```
float[] dash1 = { 2f, 0f, 2f };
```

Here we create a dash, that we use in the stroke object.

```
BasicStroke bs1 = new BasicStroke(1, BasicStroke.CAP_BUTT,


    BasicStroke.JOIN_ROUND, 1.0f, dash1, 2f )
```

This code creates a stroke. The stroke defines the line width, end caps, line joins, miter limit, dash and the dash phase.

Figure: Lines

Rectangles

To draw rectangles, we use the **drawRect()** method. To fill rectangles with the current color, we use the **fillRect()**method.

```
import java.awt.Color;

import java.awt.Graphics;

import java.awt.Graphics2D;



import javax.swing.JFrame;

import javax.swing.JPanel;
```

```
public class Rectangles extends JPanel {



    public void paintComponent(Graphics g) {

            super.paintComponent(g);



            Graphics2D g2d = (Graphics2D) g;



            g2d.setColor(new Color(212, 212, 212));

            g2d.drawRect(10, 15, 90, 60);

            g2d.drawRect(130, 15, 90, 60);

            g2d.drawRect(250, 15, 90, 60);

            g2d.drawRect(10, 105, 90, 60);

            g2d.drawRect(130, 105, 90, 60);

            g2d.drawRect(250, 105, 90, 60);

            g2d.drawRect(10, 195, 90, 60);

            g2d.drawRect(130, 195, 90, 60);

            g2d.drawRect(250, 195, 90, 60);



            g2d.setColor(new Color(125, 167, 116));

            g2d.fillRect(10, 15, 90, 60);
```

```
            g2d.setColor(new Color(42, 179, 231));
            g2d.fillRect(130, 15, 90, 60);

            g2d.setColor(new Color(70, 67, 123));
            g2d.fillRect(250, 15, 90, 60);

            g2d.setColor(new Color(130, 100, 84));
            g2d.fillRect(10, 105, 90, 60);

            g2d.setColor(new Color(252, 211, 61));
            g2d.fillRect(130, 105, 90, 60);

            g2d.setColor(new Color(241, 98, 69));
            g2d.fillRect(250, 105, 90, 60);

            g2d.setColor(new Color(217, 146, 54));
            g2d.fillRect(10, 195, 90, 60);

            g2d.setColor(new Color(63, 121, 186));
            g2d.fillRect(130, 195, 90, 60);
```

```
            g2d.setColor(new Color(31, 21, 1));
            g2d.fillRect(250, 195, 90, 60);



    }


    public static void main(String[] args) {


        Rectangles rects = new Rectangles();
        JFrame frame = new JFrame("Rectangles");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(rects);
        frame.setSize(360, 300);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

In the example we draw nine colored rectangles.

```
g2d.setColor(new Color(212, 212, 212));

g2d.drawRect(10, 15, 90, 60);
```

```
...
```

We set the color of the outline of the rectangle to a soft gray color, so that it does not interfere with the fill color. To draw the outline of the rectangle, we use the **drawRect()** method. The first two parameters are the x and y values. The third and fourth are width and height.

```
g2d.fillRect(10, 15, 90, 60);
```

To fill the rectangle with a color, we use the **fillRect()** method.

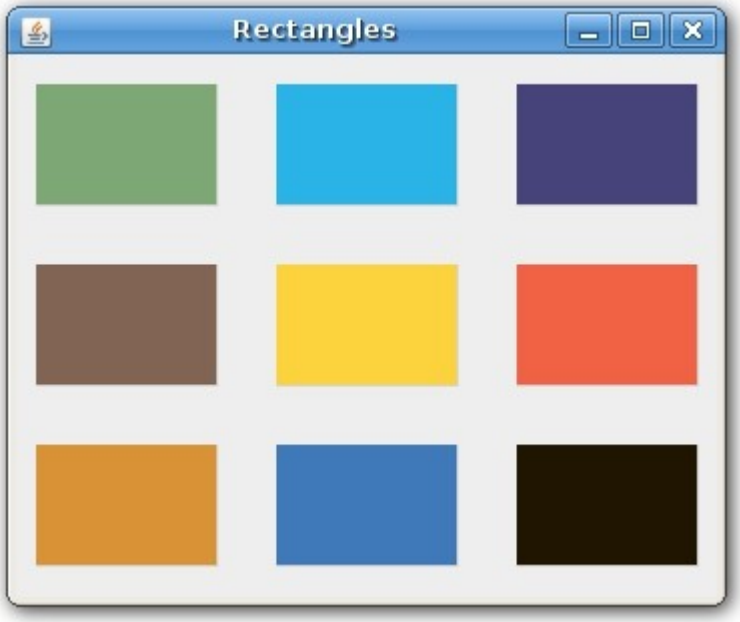


Figure: Rectangles

Textures

A **texture** is a bitmap image applied to the surface in computer graphics. Besides colors and gradients, we can fill our graphics shapes with textures.

```
import java.awt.Color;

import java.awt.Graphics;

import java.awt.Graphics2D;
```

```
import java.awt.Rectangle;

import java.awt.TexturePaint;

import java.awt.image.BufferedImage;



import javax.swing.JFrame;

import javax.swing.JPanel;



import java.io.IOException;

import java.net.URL;

import javax.imageio.ImageIO;




public class Textures extends JPanel {




    public void paintComponent(Graphics g) {

            super.paintComponent(g);


            Graphics2D g2d = (Graphics2D) g;


            g2d.setColor(new Color(212, 212, 212));
```

```
            g2d.drawRect(10, 15, 90, 60);

            g2d.drawRect(130, 15, 90, 60);

            g2d.drawRect(250, 15, 90, 60);

            g2d.drawRect(10, 105, 90, 60);

            g2d.drawRect(130, 105, 90, 60);

            g2d.drawRect(250, 105, 90, 60);



            BufferedImage bimage1 = null;

            BufferedImage bimage2 = null;

            BufferedImage bimage3 = null;

            BufferedImage bimage4 = null;

            BufferedImage bimage5 = null;

            BufferedImage bimage6 = null;



            URL url1 = ClassLoader.getSystemResource("texture1.png");

            URL url2 = ClassLoader.getSystemResource("texture2.png");

            URL url3 = ClassLoader.getSystemResource("texture3.png");

            URL url4 = ClassLoader.getSystemResource("texture4.png");

            URL url5 = ClassLoader.getSystemResource("texture5.png");

            URL url6 = ClassLoader.getSystemResource("texture6.png");
```

```
            try {
                bimage1 = ImageIO.read(url1);
                bimage2 = ImageIO.read(url2);
                bimage3 = ImageIO.read(url3);
                bimage4 = ImageIO.read(url4);
                bimage5 = ImageIO.read(url5);
                bimage6 = ImageIO.read(url6);
            } catch (IOException ioe) {
                ioe.printStackTrace();
            }

            Rectangle rect1 = new Rectangle(0, 0,
                bimage1.getWidth(), bimage1.getHeight());

            Rectangle rect2 = new Rectangle(0, 0,
                bimage2.getWidth(), bimage2.getHeight());

            Rectangle rect3 = new Rectangle(0, 0,
                bimage3.getWidth(), bimage3.getHeight());

            Rectangle rect4 = new Rectangle(0, 0,
```

```
                bimage4.getWidth(), bimage4.getHeight());


            Rectangle rect5 = new Rectangle(0, 0,
                bimage5.getWidth(), bimage5.getHeight());


            Rectangle rect6 = new Rectangle(0, 0,
                bimage6.getWidth(), bimage6.getHeight());


            TexturePaint texture1 = new TexturePaint(bimage1, rect1);
            TexturePaint texture2 = new TexturePaint(bimage2, rect2);
            TexturePaint texture3 = new TexturePaint(bimage3, rect3);
            TexturePaint texture4 = new TexturePaint(bimage4, rect4);
            TexturePaint texture5 = new TexturePaint(bimage5, rect5);
            TexturePaint texture6 = new TexturePaint(bimage6, rect6);


            g2d.setPaint(texture1);
            g2d.fillRect(10, 15, 90, 60);


            g2d.setPaint(texture2);
            g2d.fillRect(130, 15, 90, 60);
```

```
            g2d.setPaint(texture3);
            g2d.fillRect(250, 15, 90, 60);


            g2d.setPaint(texture4);
            g2d.fillRect(10, 105, 90, 60);


            g2d.setPaint(texture5);
            g2d.fillRect(130, 105, 90, 60);


            g2d.setPaint(texture6);
            g2d.fillRect(250, 105, 90, 60);
    }


    public static void main(String[] args) {


        Textures rects = new Textures();
        JFrame frame = new JFrame("Textures");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(rects);
        frame.setSize(360, 210);
        frame.setLocationRelativeTo(null);
```

```
        frame.setVisible(true);



    }

}
```

In our example, we will draw six rectangles filled with different textures. To work with textures, Java Swing has a**TexturePaint** class.

```
BufferedImage bimage1 = null;

...

URL url1 = ClassLoader.getSystemResource("texture1.png");

...

bimage1 = ImageIO.read(url1);
```

We read an image into the memory.

```
Rectangle rect1 = new Rectangle(0, 0,

    bimage1.getWidth(), bimage1.getHeight());
```

We get the size of the texture image.

```
TexturePaint texture1 = new TexturePaint(bimage1, rect1);
```

Here we create a **TexturePaint** object. The parameters are a buffered image and a rectangle of the image. The rectangle is used to anchor and replicate the image. The images are tiled.

```
g2d.setPaint(texture1);

g2d.fillRect(10, 15, 90, 60);
```

Here we apply the texture and fill the rectangle with it.

Figure: Textures

Gradients

In computer graphics, gradient is a smooth blending of shades from light to dark or from one color to another. In 2D drawing programs and paint programs, gradients are used to create colorful backgrounds and special effects as well as to simulate lights and shadows. (answers.com)

```
import java.awt.BasicStroke;

import java.awt.Color;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;



import javax.swing.JFrame;
```

```
import javax.swing.JPanel;



public class Gradients extends JPanel {



    public void paintComponent(Graphics g) {
            super.paintComponent(g);

            Graphics2D g2d = (Graphics2D) g;

            GradientPaint gp1 = new GradientPaint(5, 5,
                Color.red, 20, 20, Color.black, true);

            g2d.setPaint(gp1);
            g2d.fillRect(20, 20, 300, 40);

            GradientPaint gp2 = new GradientPaint(5, 25,
              Color.yellow, 20, 2, Color.black, true);

            g2d.setPaint(gp2);
```

```
            g2d.fillRect(20, 80, 300, 40);


            GradientPaint gp3 = new GradientPaint(5, 25,
             Color.green, 2, 2, Color.black, true);


            g2d.setPaint(gp3);
            g2d.fillRect(20, 140, 300, 40);


            GradientPaint gp4 = new GradientPaint(25, 25,
                Color.blue, 15, 25, Color.black, true);


            g2d.setPaint(gp4);
            g2d.fillRect(20, 200, 300, 40);


            GradientPaint gp5 = new GradientPaint(0, 0,
               Color.orange, 0, 20, Color.black, true);


            g2d.setPaint(gp5);
            g2d.fillRect(20, 260, 300, 40);
    }
```

```
    public static void main(String[] args) {



        Gradients gradients = new Gradients();

        JFrame frame = new JFrame("Gradients");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(gradients);

        frame.setSize(350, 350);

        frame.setLocationRelativeTo(null);

        frame.setVisible(true);



    }
}
```

Our code example presents five rectangles with gradients.

```
GradientPaint gp4 = new GradientPaint(25, 25,

    Color.blue, 15, 25, Color.black, true);
```

To work with gradients, we use Java Swing's **GradientPaint** class.By manipulating the color values and the starting end ending points, we can get interesting results.

```
g2d.setPaint(gp5);
```

The gradient is activated calling the **setPaint()** method.

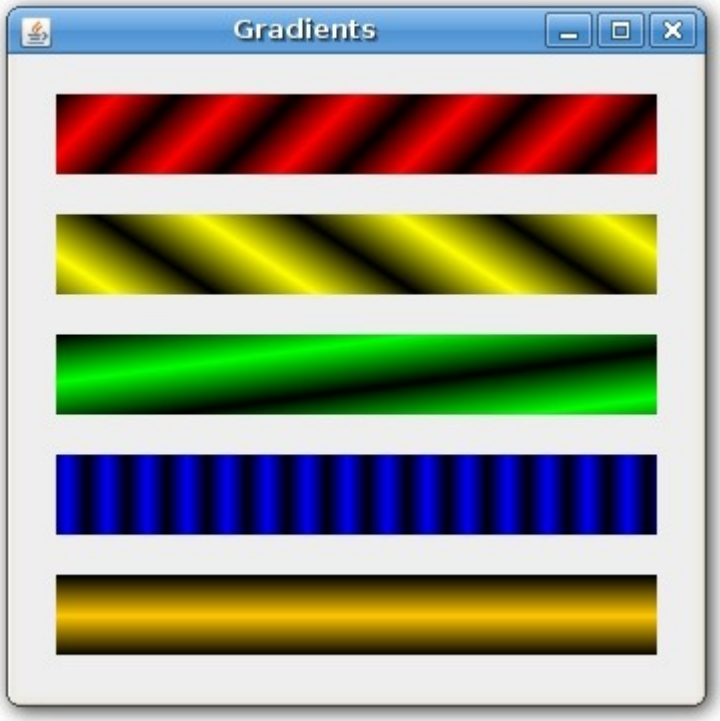


Figure: Gradients

Drawing text

Drawing is done with the **drawString()** method. We specify the string we want to draw and the position of the text on the window area.

```
import java.awt.Font;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.RenderingHints;



import javax.swing.JFrame;
```

```
import javax.swing.JPanel;



public class Text extends JPanel {


    public void paintComponent(Graphics g) {
        super.paintComponent(g);


        Graphics2D g2d = (Graphics2D) g;



        RenderingHints rh = new RenderingHints(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);


        rh.put(RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);


        g2d.setRenderingHints(rh);


        Font font = new Font("URW Chancery L", Font.BOLD, 21);
```

```
        g2d.setFont(font);



        g2d.drawString("Not marble, nor the gilded monuments", 20, 30);

        g2d.drawString("Of princes, shall outlive this powerful rhyme;"

            ,20, 60);

        g2d.drawString("But you shall shine more bright in these
contents",

            20, 90);

        g2d.drawString("Than unswept stone, besmear'd with sluttish
time.",

            20, 120);

        g2d.drawString("When wasteful war shall statues overturn,", 20,
150);

        g2d.drawString("And broils root out the work of masonry,", 20,
180);

        g2d.drawString("Nor Mars his sword, nor war's quick " +

            "fire shall burn", 20, 210);

        g2d.drawString("The living record of your memory.", 20, 240);

        g2d.drawString("'Gainst death, and all oblivious enmity", 20,
270);

        g2d.drawString("Shall you pace forth; your praise shall still " +

            "find room", 20, 300);

        g2d.drawString("Even in the eyes of all posterity", 20, 330);

        g2d.drawString("That wear this world out to the ending doom.",
20, 360);
```

```
        g2d.drawString("So, till the judgment that yourself arise,", 20,
390);

        g2d.drawString("You live in this, and dwell in lovers' eyes.",
20, 420);



    }



    public static void main(String[] args) {



        Text text = new Text();

        JFrame frame = new JFrame("Sonnet 55");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(text);

        frame.setSize(500, 470);

        frame.setLocationRelativeTo(null);

        frame.setVisible(true);

    }

}
```

In our example, we draw a sonnet on the panel component.

```
RenderingHints rh = new RenderingHints(

    RenderingHints.KEY_ANTIALIASING,
```

```
    RenderingHints.VALUE_ANTIALIAS_ON);



rh.put(RenderingHints.KEY_RENDERING,

    RenderingHints.VALUE_RENDER_QUALITY);



g2d.setRenderingHints(rh);
```

This code is to make our text look better. We apply a technique called **antialiasing**.

```
Font font = new Font("URW Chancery L", Font.BOLD, 21);


g2d.setFont(font);
```

We choose a nice font for our text.

```
g2d.drawString("Not marble, nor the gilded monuments", 20, 30);
```

This is the code, that actually draws the text.

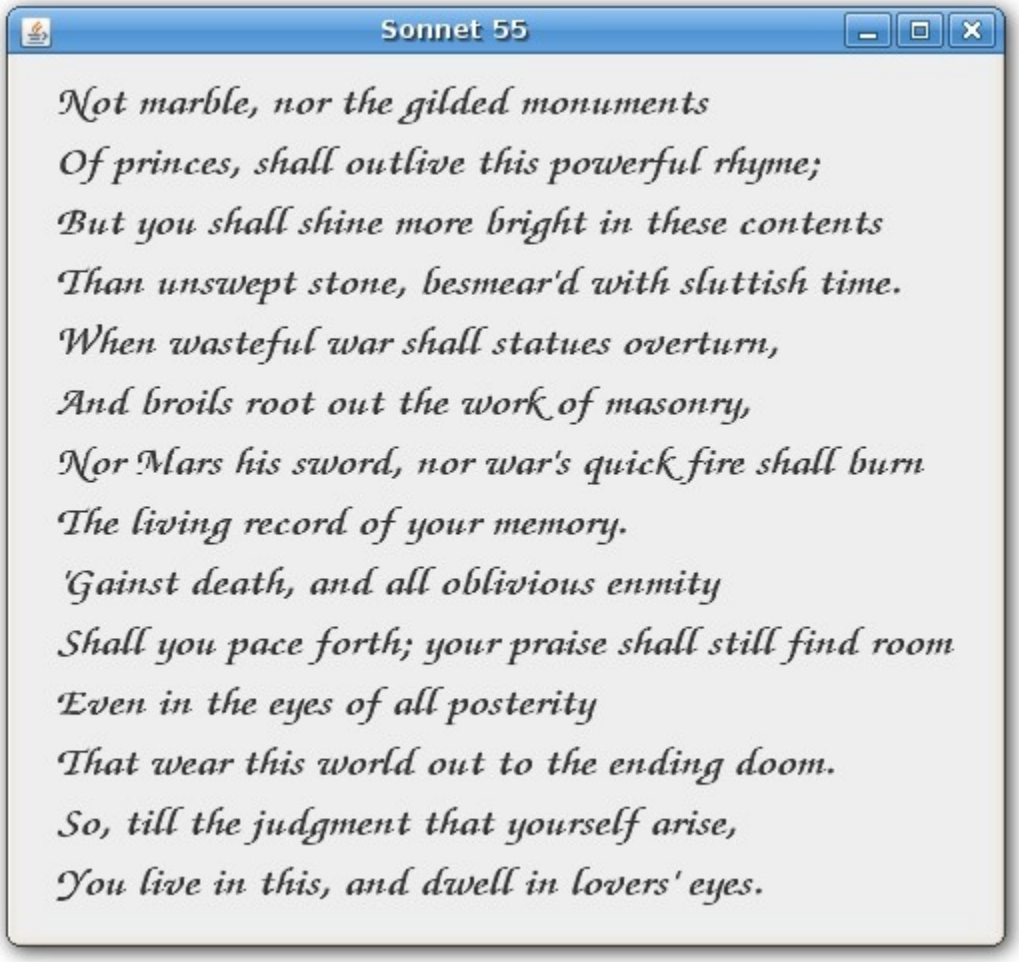


Figure: Sonnet 55

Images

On of the most important capabililies of a toolkit is the ability to display images. An image is an array of pixels. Each pixel represents a color at a given position. We can use components like **JLabel** to display an image, or we can draw it using the **Java 2D API**.

```
import java.awt.Graphics;

import java.awt.Graphics2D;
```

```
import java.awt.Image;



import javax.swing.ImageIcon;

import javax.swing.JFrame;

import javax.swing.JPanel;




public class AnImage extends JPanel {



    public void paintComponent(Graphics g) {

        super.paintComponent(g);



        Graphics2D g2d = (Graphics2D) g;

        Image image = new ImageIcon("dumbier.jpg").getImage();

        g2d.drawImage(image, 10, 10, null);

    }



    public static void main(String[] args) {



        AnImage image = new AnImage();

        JFrame frame = new JFrame("Image");
```

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

        frame.add(image);

        frame.setSize(380, 320);

        frame.setLocationRelativeTo(null);

        frame.setVisible(true);

    }

}
```

This example will draw an image on the panel.

```
Image image = new ImageIcon("dumbier.jpg").getImage();


g2d.drawImage(image, 10, 10, null);
```

These two lines read and draw the image.

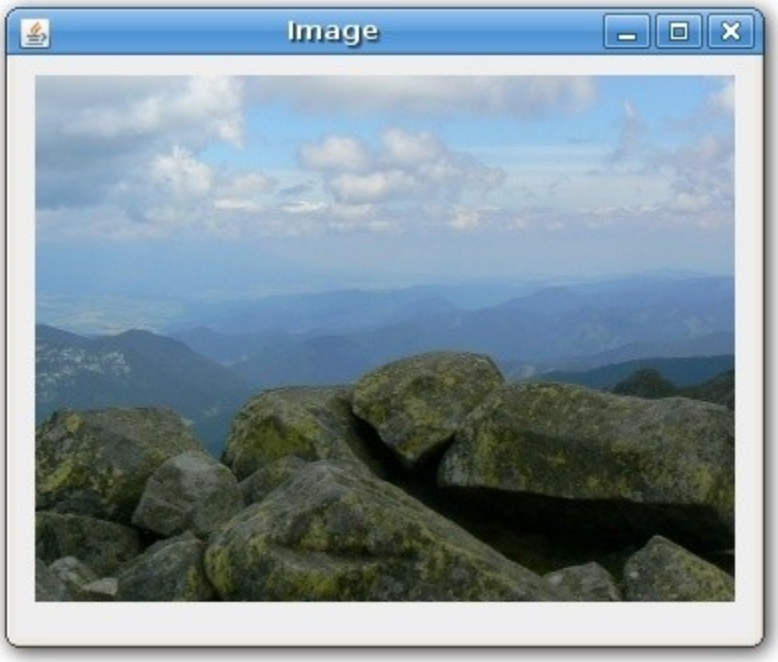


Figure: Image

Resizable components in Java Swing

In this part of the Java Swing tutorial, we will create a resizable component.

Resizable component

Resizable components are most often used when creating charts, diagrams and similar. The most common resizable component is a chart in a spreadsheet application. For example, when we create a chart in a OpenOffice application. The chart can be moved over the grid widget of the application and resized.

In order to create a component that can be freely dragged over a panel, we need a panel with absolute positioning enabled. We must not use a layout manager. In our example, we will create a component (a JPanel) that we can freely move over a parent window and resize.

In order to distinguish which component has a focus, we draw 8 small rectangles on the border of our resizable component. This way we know, that the component has focus. The rectangles serve as a dragging points, where we can draw the component and start resizing. I have learnt to use resizable components from this blog.

```
package resizablecomponent;



import java.awt.Color;

import java.awt.Dimension;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;
```

```
import javax.swing.JFrame;

import javax.swing.JPanel;




/* ResizableComponent.java */


public class ResizableComponent extends JFrame {


  private JPanel panel = new JPanel(null);

  private Resizable resizer;




  public ResizableComponent() {


      add(panel);


      JPanel area = new JPanel();

      area.setBackground(Color.white);

      resizer = new Resizable(area);

      resizer.setBounds(50, 50, 200, 150);

      panel.add(resizer);
```

```
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(new Dimension(350, 300));
      setTitle("Resizable Component");
      setLocationRelativeTo(null);

      addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent me) {

          requestFocus();
          resizer.repaint();
        }
      });
  }

  public static void main(String[] args) {
      ResizableComponent rc = new ResizableComponent();
      rc.setVisible(true);
  }
}
```

The **ResizableComponent** sets up the panel and the component.

```
private JPanel panel = new JPanel(null);
```

We have already mentioned, that we cannot use any layout manager. We must use absolute positioning for resizable component. By providing null to the constructor, we create a panel with absolute positioning.

```
addMouseListener(new MouseAdapter() {

  public void mousePressed(MouseEvent me) {



    requestFocus();

    resizer.repaint();

  }

});
```

If we press on the parent panel, e.g outside the resizable component, we grab focus and repaint the component. The rectangles over the border will disappear.

```
package resizablecomponent;



import java.awt.Color;

import java.awt.Component;

import java.awt.Cursor;

import java.awt.Graphics;
```

```
import java.awt.Insets;

import java.awt.Rectangle;

import java.awt.event.MouseEvent;



import javax.swing.SwingConstants;

import javax.swing.border.Border;



// ResizableBorder.java



public class ResizableBorder implements Border {

  private int dist = 8;



  int locations[] =

  {

    SwingConstants.NORTH, SwingConstants.SOUTH, SwingConstants.WEST,

    SwingConstants.EAST, SwingConstants.NORTH_WEST,

    SwingConstants.NORTH_EAST, SwingConstants.SOUTH_WEST,

    SwingConstants.SOUTH_EAST

  };



  int cursors[] =
```

```
  {

    Cursor.N_RESIZE_CURSOR, Cursor.S_RESIZE_CURSOR,
Cursor.W_RESIZE_CURSOR,

    Cursor.E_RESIZE_CURSOR, Cursor.NW_RESIZE_CURSOR,
Cursor.NE_RESIZE_CURSOR,

    Cursor.SW_RESIZE_CURSOR, Cursor.SE_RESIZE_CURSOR

  };



  public ResizableBorder(int dist) {

    this.dist = dist;

  }



  public Insets getBorderInsets(Component component) {

      return new Insets(dist, dist, dist, dist);

  }



  public boolean isBorderOpaque() {

      return false;

  }



  public void paintBorder(Component component, Graphics g, int x, int y,

                          int w, int h) {
```

```
      g.setColor(Color.black);
      g.drawRect(x + dist / 2, y + dist / 2, w - dist, h - dist);


      if (component.hasFocus()) {


        for (int i = 0; i < locations.length; i++) {
          Rectangle rect = getRectangle(x, y, w, h, locations[i]);
          g.setColor(Color.WHITE);
          g.fillRect(rect.x, rect.y, rect.width - 1, rect.height - 1);
          g.setColor(Color.BLACK);
          g.drawRect(rect.x, rect.y, rect.width - 1, rect.height - 1);
        }
      }
  }


  private Rectangle getRectangle(int x, int y, int w, int h, int
location) {
      switch (location) {
      case SwingConstants.NORTH:
          return new Rectangle(x + w / 2 - dist / 2, y, dist, dist);
```

```
        case SwingConstants.SOUTH:
            return new Rectangle(x + w / 2 - dist / 2, y + h - dist, dist,
                                 dist);
        case SwingConstants.WEST:
            return new Rectangle(x, y + h / 2 - dist / 2, dist, dist);
        case SwingConstants.EAST:
            return new Rectangle(x + w - dist, y + h / 2 - dist / 2, dist,
                                 dist);
        case SwingConstants.NORTH_WEST:
            return new Rectangle(x, y, dist, dist);
        case SwingConstants.NORTH_EAST:
            return new Rectangle(x + w - dist, y, dist, dist);
        case SwingConstants.SOUTH_WEST:
            return new Rectangle(x, y + h - dist, dist, dist);
        case SwingConstants.SOUTH_EAST:
            return new Rectangle(x + w - dist, y + h - dist, dist, dist);
        }
        return null;
    }


    public int getCursor(MouseEvent me) {
```

```
        Component c = me.getComponent();

        int w = c.getWidth();

        int h = c.getHeight();



        for (int i = 0; i < locations.length; i++) {

            Rectangle rect = getRectangle(0, 0, w, h, locations[i]);

            if (rect.contains(me.getPoint()))

                return cursors[i];

        }



        return Cursor.MOVE_CURSOR;

    }

}
```

The **ResizableBorder** is responsible for drawing the border of the component and determining the type of the cursor to use.

```
int locations[] =

{

  SwingConstants.NORTH, SwingConstants.SOUTH, SwingConstants.WEST,

  SwingConstants.EAST, SwingConstants.NORTH_WEST,

  SwingConstants.NORTH_EAST, SwingConstants.SOUTH_WEST,
```

```
    SwingConstants.SOUTH_EAST

  };
```

These are locations, where we will draw rectangles. These locations are grabbing points, where we can grab the component and resize it.

```
g.setColor(Color.black);

g.drawRect(x + dist / 2, y + dist / 2, w - dist, h - dist);
```

In the **paintBorder()** method, we draw the border of the resizable component. The upper code draws the outer border of the component.

```
if (component.hasFocus()) {



  for (int i = 0; i < locations.length; i++) {

    Rectangle rect = getRectangle(x, y, w, h, locations[i]);

    g.setColor(Color.WHITE);

    g.fillRect(rect.x, rect.y, rect.width - 1, rect.height - 1);

    g.setColor(Color.BLACK);

    g.drawRect(rect.x, rect.y, rect.width - 1, rect.height - 1);

  }

}
```

The eight rectangles are drawn only in case that the resizable component has currently focus.

Finally, the **getRectangle()** method gets the coordinates of the rectangles and the **getCursor()** methods gets the cursor type for the grab point in question.

```
package resizablecomponent;



import java.awt.BorderLayout;

import java.awt.Component;

import java.awt.Cursor;

import java.awt.Point;

import java.awt.Rectangle;

import java.awt.event.MouseEvent;



import javax.swing.JComponent;

import javax.swing.event.MouseInputAdapter;

import javax.swing.event.MouseInputListener;



// Resizable.java



public class Resizable extends JComponent {



  public Resizable(Component comp) {

    this(comp, new ResizableBorder(8));
```

```
}


public Resizable(Component comp, ResizableBorder border) {
  setLayout(new BorderLayout());
  add(comp);
  addMouseListener(resizeListener);
  addMouseMotionListener(resizeListener);
  setBorder(border);
}


private void resize() {
    if (getParent() != null) {
      ((JComponent)getParent()).revalidate();
    }
}


MouseInputListener resizeListener = new MouseInputAdapter() {
  public void mouseMoved(MouseEvent me) {
    if (hasFocus()) {
        ResizableBorder border = (ResizableBorder)getBorder();
        setCursor(Cursor.getPredefinedCursor(border.getCursor(me)));
```

```
    }
  }

  public void mouseExited(MouseEvent mouseEvent) {
     setCursor(Cursor.getDefaultCursor());
  }

  private int cursor;
  private Point startPos = null;

  public void mousePressed(MouseEvent me) {
    ResizableBorder border = (ResizableBorder)getBorder();
    cursor = border.getCursor(me);
    startPos = me.getPoint();
    requestFocus();
    repaint();
  }

  public void mouseDragged(MouseEvent me) {

    if (startPos != null) {
```

```
int x = getX();

int y = getY();

int w = getWidth();

int h = getHeight();



int dx = me.getX() - startPos.x;

int dy = me.getY() - startPos.y;



switch (cursor) {

  case Cursor.N_RESIZE_CURSOR:

    if (!(h - dy < 50)) {

      setBounds(x, y + dy, w, h - dy);

      resize();

    }

    break;



  case Cursor.S_RESIZE_CURSOR:

    if (!(h + dy < 50)) {

      setBounds(x, y, w, h + dy);

      startPos = me.getPoint();
```

```
            resize();
          }
          break;

        case Cursor.W_RESIZE_CURSOR:
          if (!(w - dx < 50)) {
            setBounds(x + dx, y, w - dx, h);
            resize();
          }
          break;

        case Cursor.E_RESIZE_CURSOR:
          if (!(w + dx < 50)) {
            setBounds(x, y, w + dx, h);
            startPos = me.getPoint();
            resize();
          }
          break;

        case Cursor.NW_RESIZE_CURSOR:
          if (!(w - dx < 50) && !(h - dy < 50)) {
```

```
            setBounds(x + dx, y + dy, w - dx, h - dy);
            resize();
          }
          break;

        case Cursor.NE_RESIZE_CURSOR:
          if (!(w + dx < 50) && !(h - dy < 50)) {
            setBounds(x, y + dy, w + dx, h - dy);
            startPos = new Point(me.getX(), startPos.y);
            resize();
          }
          break;

        case Cursor.SW_RESIZE_CURSOR:
          if (!(w - dx < 50) && !(h + dy < 50)) {
            setBounds(x + dx, y, w - dx, h + dy);
            startPos = new Point(startPos.x, me.getY());
            resize();
          }
          break;
```

```
          case Cursor.SE_RESIZE_CURSOR:
            if (!(w + dx < 50) && !(h + dy < 50)) {
              setBounds(x, y, w + dx, h + dy);
              startPos = me.getPoint();
              resize();
            }
          break;

          case Cursor.MOVE_CURSOR:
            Rectangle bounds = getBounds();
            bounds.translate(dx, dy);
            setBounds(bounds);
            resize();
          }

          setCursor(Cursor.getPredefinedCursor(cursor));
        }
      }

    public void mouseReleased(MouseEvent mouseEvent) {
```

```
      startPos = null;

     }

   };

 }
```

The **Resizable** class represents the component, that is being resized and moved on the window.

```
 private void resize() {

     if (getParent() != null) {

       ((JComponent)getParent()).revalidate();

     }

 }
```

The **resize()** method is called, after we have resized the component.
The **revalidate()** method will cause the component to be redrawn.

```
MouseInputListener resizeListener = new MouseInputAdapter() {

  public void mouseMoved(MouseEvent me) {

    if (hasFocus()) {

        ResizableBorder border = (ResizableBorder)getBorder();

        setCursor(Cursor.getPredefinedCursor(border.getCursor(me)));

    }
```

```
}
```

We change the cursor type, when we hover the cursor over the grip points. The cursor type changes only if the component has focus.

```
public void mousePressed(MouseEvent me) {

  ResizableBorder border = (ResizableBorder)getBorder();

    cursor = border.getCursor(me);

    startPos = me.getPoint();

    requestFocus();

    repaint();

}
```

If we click on the resizable component, we change the cursor, get the starting point of dragging, give focus to the component and redraw it.

```
int x = getX();

int y = getY();

int w = getWidth();

int h = getHeight();



int dx = me.getX() - startPos.x;

int dy = me.getY() - startPos.y;
```

In the **mouseDragged()** method, we determine the x, y coordinates of the cursor, width and height of the component. We calculate the distances, that we make during the mouse drag event.

```
case Cursor.N_RESIZE_CURSOR:

  if (!(h - dy < 50)) {

   setBounds(x, y + dy, w, h - dy);

   resize();

  }

  break;
```

For all resizing we ensure, that the component is not smaller than 50 px. Otherwise, we could make it so small, that we would eventually hide the component.
The **setBounds()** method relocates and resizes the component.

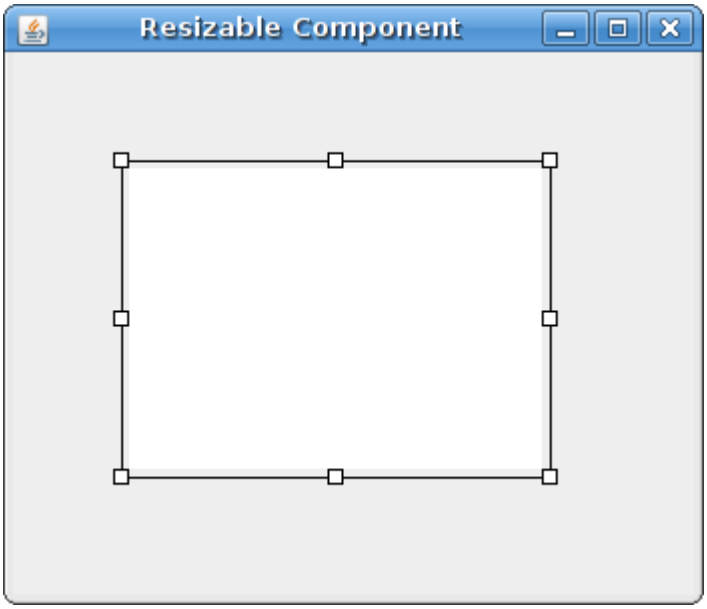


Figure: Resizable component

The Puzzle in Java Swing

In this chapter, we will create a simple puzzle game in Java Swing toolkit.

Puzzle

We have an image of a Sid character from the Ice Age movie. It is cut into 12 pieces. The goal is to form the picture. You can download the picture here.

```
import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.GridLayout;

import java.awt.Image;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.image.CropImageFilter;

import java.awt.image.FilteredImageSource;



import javax.swing.Box;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;
```

```
public class Puzzle extends JFrame implements ActionListener {


    private JPanel centerPanel;

    private JButton button;

    private JLabel label;

    private Image source;

    private Image image;

    int[][] pos;

    int width, height;


    public Puzzle() {


        pos = new int[][] {

                              {0, 1, 2},

                              {3, 4, 5},

                              {6, 7, 8},

                              {9, 10, 11}

                          };
```

```
        centerPanel = new JPanel();

        centerPanel.setLayout(new GridLayout(4, 4, 0, 0));



        ImageIcon sid = new
ImageIcon(Puzzle.class.getResource("icesid.jpg"));

        source = sid.getImage();



        width = sid.getIconWidth();

        height = sid.getIconHeight();




        add(Box.createRigidArea(new Dimension(0, 5)),
BorderLayout.NORTH);

        add(centerPanel, BorderLayout.CENTER);




        for ( int i = 0; i < 4; i++) {

            for ( int j = 0; j < 3; j++) {

                if ( j == 2 && i == 3) {

                    label = new JLabel("");

                    centerPanel.add(label);
```

```
                } else {

                    button = new JButton();

                    button.addActionListener(this);

                    centerPanel.add(button);

                    image = createImage(new
FilteredImageSource(source.getSource(),

                        new CropImageFilter(j*width/3, i*height/4,
(width/3)+1, height/4)));

                    button.setIcon(new ImageIcon(image));

                }

            }

        }



        setSize(325, 275);

        setTitle("Puzzle");

        setResizable(false);

        setLocationRelativeTo(null);

        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        setVisible(true);

    }
```

```
    public static void main(String[] args) {


        new Puzzle();


    }


    public void actionPerformed(ActionEvent e) {
        JButton button = (JButton) e.getSource();
        Dimension size = button.getSize();


        int labelX = label.getX();
        int labelY = label.getY();
        int buttonX = button.getX();
        int buttonY = button.getY();
        int buttonPosX = buttonX / size.width;
        int buttonPosY = buttonY / size.height;
        int buttonIndex = pos[buttonPosY][buttonPosX];




        if (labelX == buttonX && (labelY - buttonY) == size.height ) {
```

```
            int labelIndex = buttonIndex + 3;


            centerPanel.remove(buttonIndex);
            centerPanel.add(label, buttonIndex);
            centerPanel.add(button,labelIndex);
            centerPanel.validate();
        }


        if (labelX == buttonX && (labelY - buttonY) == -size.height ) {


            int labelIndex = buttonIndex - 3;
            centerPanel.remove(labelIndex);
            centerPanel.add(button,labelIndex);
            centerPanel.add(label, buttonIndex);
            centerPanel.validate();
        }


        if (labelY == buttonY && (labelX - buttonX) == size.width ) {


            int labelIndex = buttonIndex + 1;
```

```
            centerPanel.remove(buttonIndex);
            centerPanel.add(label, buttonIndex);
            centerPanel.add(button,labelIndex);
            centerPanel.validate();
        }

        if (labelY == buttonY && (labelX - buttonX) == -size.width ) {

            int labelIndex = buttonIndex - 1;

            centerPanel.remove(buttonIndex);
            centerPanel.add(label, labelIndex);
            centerPanel.add(button,labelIndex);
            centerPanel.validate();
        }
    }
}
```

The goal of this little game is to form the original picture. We move the buttons by clicking on them. Only buttons adjacent to the label can be moved.

```
pos = new int[][] {
```

```
      {0, 1, 2},

      {3, 4, 5},

      {6, 7, 8},

      {9, 10, 11}

};
```

These are the positions of the image parts.

```
ImageIcon sid = new ImageIcon(Puzzle.class.getResource("icesid.jpg"));

source = sid.getImage();
```

We use the **ImageIcon** class to load the image.

```
for ( int i = 0; i < 4; i++) {

    for ( int j = 0; j < 3; j++) {

        if ( j == 2 && i == 3) {

            label = new JLabel("");

            centerPanel.add(label);

        } else {

            button = new JButton();

            button.addActionListener(this);

            centerPanel.add(button);
```

```
            image = createImage(new FilteredImageSource(source.getSource(),

                new CropImageFilter(j*width/3, i*height/4, (width/3)+1,
height/4)));

            button.setIcon(new ImageIcon(image));

        }

    }

}
```

The code creates 11 buttons and one label. We crop the image into pieces and place them on the buttons.

```
int labelX = label.getX();

int labelY = label.getY();

int buttonX = button.getX();

int buttonY = button.getY();
```

We get the x, y coordinates of the button that we hit and an empty label. The x, y coordinates are important in the logic of the program.

```
int buttonPosX = buttonX / size.width;

int buttonPosY = buttonY / size.height;

int buttonIndex = pos[buttonPosY][buttonPosX];
```

Here we get the index of the button in the two dimensional array of the button positions.

```
if (labelX == buttonX && (labelY - buttonY) == size.height ) {
```

```
        int labelIndex = buttonIndex + 3;



        centerPanel.remove(buttonIndex);

        centerPanel.add(label, buttonIndex);

        centerPanel.add(button,labelIndex);

        centerPanel.validate();

    }
```

In this case, we check if we clicked on the button, that is right above the empty label. If it is above the label, they share the x coordinate. If the button is right above the label, the equation `(labelY - buttonY) == size.height` is true.

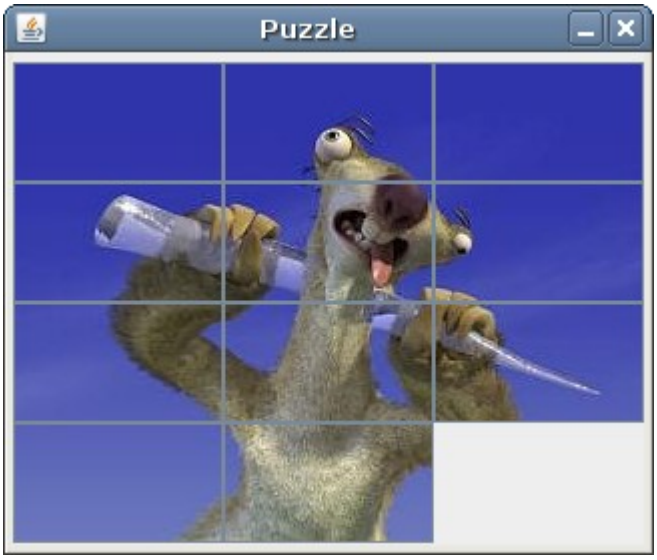


Figure: Puzzle

The Tetris game

In this chapter, we will create a Tetris game clone in Java Swing. The following example is based on the C++ code example available at doc.trolltech.com. It is modified and simplified.

Tetris

The Tetris game is one of the most popular computer games ever created. The original game was designed and programmed by a russian programmer **Alexey Pajitnov** in 1985. Since then, Tetris is available on almost every computer platform in lots of variations. Even my mobile phone has a modified version of the Tetris game.

Tetris is called a falling block puzzle game. In this game, we have seven different shapes called **tetrominoes**. S-shape, Z-shape, T-shape, L-shape, Line-shape, MirroredL-shape and a Square-shape. Each of these shapes is formed with four squares. The shapes are falling down the board. The object of the Tetris game is to move and rotate the shapes, so that they fit as much as possible. If we manage to form a row, the row is destroyed and we score. We play the tetris game until we top out.

Figure: Tetrominoes

The development

We do not have images for our tetris game, we draw the tetrominoes using Swing drawing API. Behind every computer game, there is a mathematical model. So it is in Tetris.

Some ideas behind the game.

- We use a **Timer** class to create a game cycle

- The tetrominoes are drawn
- The shapes move on a square by square basis (not pixel by pixel)
- Mathematically a board is a simple list of numbers

I have simplified the game a bit, so that it is easier to understand. The game starts immediately, after it is launched. We can pause the game by pressing the p key. The space key will drop the tetris piece immediately to the bottom. The d key will drop the piece one line down. (It can be used to speed up the falling a bit.) The game goes at constant speed, no acceleration is implemented. The score is the number of lines, that we have removed.

```
// Tetris.java


package tetris;


import java.awt.BorderLayout;


import javax.swing.JFrame;

import javax.swing.JLabel;




public class Tetris extends JFrame {


    JLabel statusbar;



```

```
    public Tetris() {



        statusbar = new JLabel(" 0");

        add(statusbar, BorderLayout.SOUTH);

        Board board = new Board(this);

        add(board);

        board.start();



        setSize(200, 400);

        setTitle("Tetris");

        setDefaultCloseOperation(EXIT_ON_CLOSE);

    }



    public JLabel getStatusBar() {

        return statusbar;

    }



    public static void main(String[] args) {



        Tetris game = new Tetris();

        game.setLocationRelativeTo(null);
```

```
        game.setVisible(true);



    }

}
```

In the Tetris.java file, we set up the game. We create a board on which we play the game. We create a statusbar.

```
board.start();
```

The **start()** method starts the Tetris game. Immediately, after the window appears on the screen.

```
// Shape.java



package tetris;



import java.util.Random;

import java.lang.Math;







public class Shape {



    enum Tetrominoes { NoShape, ZShape, SShape, LineShape,

                TShape, SquareShape, LShape, MirroredLShape };
```

```
private Tetrominoes pieceShape;

private int coords[][];

private int[][][] coordsTable;



public Shape() {


    coords = new int[4][2];

    setShape(Tetrominoes.NoShape);


}


public void setShape(Tetrominoes shape) {


     coordsTable = new int[][][] {

        { { 0, 0 },   { 0, 0 },   { 0, 0 },   { 0, 0 } },

        { { 0, -1 },  { 0, 0 },   { -1, 0 },  { -1, 1 } },

        { { 0, -1 },  { 0, 0 },   { 1, 0 },   { 1, 1 } },

        { { 0, -1 },  { 0, 0 },   { 0, 1 },   { 0, 2 } },

        { { -1, 0 },  { 0, 0 },   { 1, 0 },   { 0, 1 } },
```

```
            { { 0, 0 },   { 1, 0 },   { 0, 1 },   { 1, 1 } },
            { { -1, -1 }, { 0, -1 },  { 0, 0 },   { 0, 1 } },
            { { 1, -1 },  { 0, -1 },  { 0, 0 },   { 0, 1 } }
        };

        for (int i = 0; i < 4 ; i++) {
            for (int j = 0; j < 2; ++j) {
                coords[i][j] = coordsTable[shape.ordinal()][i][j];
            }
        }
        pieceShape = shape;

    }

    private void setX(int index, int x) { coords[index][0] = x; }
    private void setY(int index, int y) { coords[index][1] = y; }
    public int x(int index) { return coords[index][0]; }
    public int y(int index) { return coords[index][1]; }
    public Tetrominoes getShape()  { return pieceShape; }

    public void setRandomShape()
```

```
    {
        Random r = new Random();
        int x = Math.abs(r.nextInt()) % 7 + 1;
        Tetrominoes[] values = Tetrominoes.values();
        setShape(values[x]);
    }


    public int minX()
    {
      int m = coords[0][0];
      for (int i=0; i < 4; i++) {
          m = Math.min(m, coords[i][0]);
      }
      return m;
    }




    public int minY()
    {
      int m = coords[0][1];
      for (int i=0; i < 4; i++) {
```

```
            m = Math.min(m, coords[i][1]);
        }
        return m;
    }

    public Shape rotateLeft()
    {
        if (pieceShape == Tetrominoes.SquareShape)
            return this;

        Shape result = new Shape();
        result.pieceShape = pieceShape;

        for (int i = 0; i < 4; ++i) {
            result.setX(i, y(i));
            result.setY(i, -x(i));
        }
        return result;
    }

    public Shape rotateRight()
```

```
    {

        if (pieceShape == Tetrominoes.SquareShape)

            return this;



        Shape result = new Shape();

        result.pieceShape = pieceShape;



        for (int i = 0; i < 4; ++i) {

            result.setX(i, -y(i));

            result.setY(i, x(i));

        }

        return result;

    }

}
```

The **Shape** class provides information about a tetris piece.

```
enum Tetrominoes { NoShape, ZShape, SShape, LineShape,


           TShape, SquareShape, LShape, MirroredLShape };
```

The **Tetrominoes** enum holds all seven tetris shapes. Plus the empty shape called here **NoShape**.

```
public Shape() {



    coords = new int[4][2];

    setShape(Tetrominoes.NoShape);



}
```

This is the constructor of the **Shape** class. The **coords** array holds the actual coordinates of a tetris piece.

```
coordsTable = new int[][][] {

   { { 0, 0 },   { 0, 0 },   { 0, 0 },   { 0, 0 } },

   { { 0, -1 },  { 0, 0 },   { -1, 0 },  { -1, 1 } },

   { { 0, -1 },  { 0, 0 },   { 1, 0 },   { 1, 1 } },

   { { 0, -1 },  { 0, 0 },   { 0, 1 },   { 0, 2 } },

   { { -1, 0 },  { 0, 0 },   { 1, 0 },   { 0, 1 } },

   { { 0, 0 },   { 1, 0 },   { 0, 1 },   { 1, 1 } },

   { { -1, -1 }, { 0, -1 },  { 0, 0 },   { 0, 1 } },

   { { 1, -1 },  { 0, -1 },  { 0, 0 },   { 0, 1 } }

};
```

The **coordsTable** array holds all possible coordinate values of our tetris pieces. This is a template from which all pieces take their coordiate values.

```
for (int i = 0; i < 4 ; i++) {

    for (int j = 0; j < 2; ++j) {

        coords[i][j] = coordsTable[shape.ordinal()][i][j];

    }

}
```

Here we put one row of the coordiate values from the **coordsTable** to a **coords** array of a tetris piece. Note the use of the **ordinal()** method. In C++, an enum type is esencially an integer. Unlike in C++, Java enums are full classes. And the ordinal() method returns the current position of the enum type in the enum object.

The following image will help understand the coordinate values a bit more. The coords array saves the coordinates of the tetris piece. For example, numbers { 0, -1 }, { 0, 0 }, { 1, 0 }, { 1, 1 } , represent a rotated S-shape. The following diagram illustrates the shape.

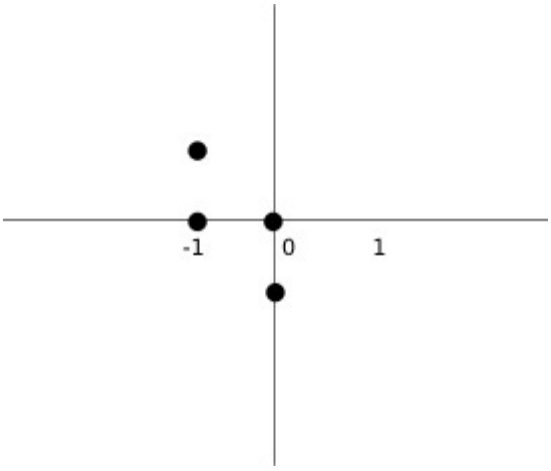


Figure: Tetris

```
public Shape rotateLeft()
```

```
{

    if (pieceShape == Tetrominoes.SquareShape)

        return this;



    Shape result = new Shape();

    result.pieceShape = pieceShape;



    for (int i = 0; i < 4; ++i) {

        result.setX(i, y(i));

        result.setY(i, -x(i));

    }

    return result;

 }
```

This code rotates the piece to the left. The square does not have to be rotated. That's why we simply return the reference to the current object. Looking at the previous image will help to understand the rotation.

```
// Board.java



package tetris;
```

```
import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;



import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.Timer;



import tetris.Shape.Tetrominoes;




public class Board extends JPanel implements ActionListener {




    final int BoardWidth = 10;

    final int BoardHeight = 22;
```

```
    Timer timer;
    boolean isFallingFinished = false;
    boolean isStarted = false;
    boolean isPaused = false;
    int numLinesRemoved = 0;
    int curX = 0;
    int curY = 0;
    JLabel statusbar;
    Shape curPiece;
    Tetrominoes[] board;




    public Board(Tetris parent) {


       setFocusable(true);
       curPiece = new Shape();
       timer = new Timer(400, this);
       timer.start();
```

```
        statusbar =  parent.getStatusBar();
        board = new Tetrominoes[BoardWidth * BoardHeight];
        addKeyListener(new TAdapter());
        clearBoard();
    }


    public void actionPerformed(ActionEvent e) {
        if (isFallingFinished) {
            isFallingFinished = false;
            newPiece();
        } else {
            oneLineDown();
        }
    }



    int squareWidth() { return (int) getSize().getWidth() / BoardWidth; }
    int squareHeight() { return (int) getSize().getHeight() /
BoardHeight; }
    Tetrominoes shapeAt(int x, int y) { return board[(y * BoardWidth) +
x]; }
```

```
    public void start()
    {
        if (isPaused)
            return;

        isStarted = true;
        isFallingFinished = false;
        numLinesRemoved = 0;
        clearBoard();

        newPiece();
        timer.start();
    }

    private void pause()
    {
        if (!isStarted)
            return;

        isPaused = !isPaused;
```

```
        if (isPaused) {

            timer.stop();

            statusbar.setText("paused");

        } else {

            timer.start();

            statusbar.setText(String.valueOf(numLinesRemoved));

        }

        repaint();

    }



    public void paint(Graphics g)

    {

        super.paint(g);



        Dimension size = getSize();

        int boardTop = (int) size.getHeight() - BoardHeight *
squareHeight();




        for (int i = 0; i < BoardHeight; ++i) {

            for (int j = 0; j < BoardWidth; ++j) {
```

```
                Tetrominoes shape = shapeAt(j, BoardHeight - i - 1);
                if (shape != Tetrominoes.NoShape)
                    drawSquare(g, 0 + j * squareWidth(),
                               boardTop + i * squareHeight(), shape);
            }
        }

        if (curPiece.getShape() != Tetrominoes.NoShape) {
            for (int i = 0; i < 4; ++i) {
                int x = curX + curPiece.x(i);
                int y = curY - curPiece.y(i);
                drawSquare(g, 0 + x * squareWidth(),
                           boardTop + (BoardHeight - y - 1) *
squareHeight(),
                           curPiece.getShape());
            }
        }
    }

    private void dropDown()
    {
```

```
        int newY = curY;
        while (newY > 0) {
            if (!tryMove(curPiece, curX, newY - 1))
                break;
            --newY;
        }
        pieceDropped();
    }

    private void oneLineDown()
    {
        if (!tryMove(curPiece, curX, curY - 1))
            pieceDropped();
    }


    private void clearBoard()
    {
        for (int i = 0; i < BoardHeight * BoardWidth; ++i)
            board[i] = Tetrominoes.NoShape;
    }
```

```
private void pieceDropped()
{
    for (int i = 0; i < 4; ++i) {
        int x = curX + curPiece.x(i);
        int y = curY - curPiece.y(i);
        board[(y * BoardWidth) + x] = curPiece.getShape();
    }

    removeFullLines();

    if (!isFallingFinished)
        newPiece();
}

private void newPiece()
{
    curPiece.setRandomShape();
    curX = BoardWidth / 2 + 1;
    curY = BoardHeight - 1 + curPiece.minY();
```

```
        if (!tryMove(curPiece, curX, curY)) {

            curPiece.setShape(Tetrominoes.NoShape);

            timer.stop();

            isStarted = false;

            statusbar.setText("game over");

        }

    }



    private boolean tryMove(Shape newPiece, int newX, int newY)

    {

        for (int i = 0; i < 4; ++i) {

            int x = newX + newPiece.x(i);

            int y = newY - newPiece.y(i);

            if (x < 0 || x >= BoardWidth || y < 0 || y >= BoardHeight)

                return false;

            if (shapeAt(x, y) != Tetrominoes.NoShape)

                return false;

        }



        curPiece = newPiece;

        curX = newX;
```

```
        curY = newY;
        repaint();
        return true;
    }

    private void removeFullLines()
    {
        int numFullLines = 0;

        for (int i = BoardHeight - 1; i >= 0; --i) {
            boolean lineIsFull = true;

            for (int j = 0; j < BoardWidth; ++j) {
                if (shapeAt(j, i) == Tetrominoes.NoShape) {
                    lineIsFull = false;
                    break;
                }
            }

            if (lineIsFull) {
                ++numFullLines;
```

```
                for (int k = i; k < BoardHeight - 1; ++k) {
                    for (int j = 0; j < BoardWidth; ++j)
                        board[(k * BoardWidth) + j] = shapeAt(j, k + 1);
                }
            }
        }

        if (numFullLines > 0) {
            numLinesRemoved += numFullLines;
            statusbar.setText(String.valueOf(numLinesRemoved));
            isFallingFinished = true;
            curPiece.setShape(Tetrominoes.NoShape);
            repaint();
        }
     }

    private void drawSquare(Graphics g, int x, int y, Tetrominoes shape)
    {
        Color colors[] = { new Color(0, 0, 0), new Color(204, 102, 102),
            new Color(102, 204, 102), new Color(102, 102, 204),
            new Color(204, 204, 102), new Color(204, 102, 204),
```

```
            new Color(102, 204, 204), new Color(218, 170, 0)
        };



        Color color = colors[shape.ordinal()];


        g.setColor(color);
        g.fillRect(x + 1, y + 1, squareWidth() - 2, squareHeight() - 2);


        g.setColor(color.brighter());
        g.drawLine(x, y + squareHeight() - 1, x, y);
        g.drawLine(x, y, x + squareWidth() - 1, y);


        g.setColor(color.darker());
        g.drawLine(x + 1, y + squareHeight() - 1,
                        x + squareWidth() - 1, y + squareHeight() - 1);
        g.drawLine(x + squareWidth() - 1, y + squareHeight() - 1,
                        x + squareWidth() - 1, y + 1);


    }
```

```
    class TAdapter extends KeyAdapter {

         public void keyPressed(KeyEvent e) {



             if (!isStarted || curPiece.getShape() ==
Tetrominoes.NoShape) {

                 return;

             }



             int keycode = e.getKeyCode();



             if (keycode == 'p' || keycode == 'P') {

                 pause();

                 return;

             }



             if (isPaused)

                 return;



             switch (keycode) {

             case KeyEvent.VK_LEFT:

                 tryMove(curPiece, curX - 1, curY);
```

```
                break;
            case KeyEvent.VK_RIGHT:
                tryMove(curPiece, curX + 1, curY);
                break;
            case KeyEvent.VK_DOWN:
                tryMove(curPiece.rotateRight(), curX, curY);
                break;
            case KeyEvent.VK_UP:
                tryMove(curPiece.rotateLeft(), curX, curY);
                break;
            case KeyEvent.VK_SPACE:
                dropDown();
                break;
            case 'd':
                oneLineDown();
                break;
            case 'D':
                oneLineDown();
                break;
            }
```

```
            }

        }

    }
```

Finally, we have the Board.java file. This is where the game logic is located.

```
...

isFallingFinished = false;

isStarted = false;

isPaused = false;

numLinesRemoved = 0;

curX = 0;

curY = 0;

...
```

We initialize some important variables. The **isFallingFinished** variable determines, if the tetris shape has finished falling and we then need to create a new shape.
The **numLinesRemoved** counts the number of lines, we have removed so far.
The **curX** and **curY** variables determine the actual position of the falling tetris shape.

```
setFocusable(true);
```

We must explicitely call the **setFocusable()** method. From now, the board has the keyboard input.

```
timer = new Timer(400, this);

timer.start();
```

Timer object fires one or more action events after a specified delay. In our case, the timer calls the**actionPerformed()** method each 400 ms.

```
public void actionPerformed(ActionEvent e) {

    if (isFallingFinished) {

        isFallingFinished = false;

        newPiece();

    } else {

        oneLineDown();

    }

}
```

The **actionPerformed()** method checks if the falling has finished. If so, a new piece is created. If not, the falling tetris piece goes one line down.

Inside the **paint()** method, we draw the all objects on the board. The painting has two steps.

```
for (int i = 0; i < BoardHeight; ++i) {

    for (int j = 0; j < BoardWidth; ++j) {

        Tetrominoes shape = shapeAt(j, BoardHeight - i - 1);

        if (shape != Tetrominoes.NoShape)

            drawSquare(g, 0 + j * squareWidth(),

                       boardTop + i * squareHeight(), shape);
```

```
    }

}
```

In the first step we paint all the shapes, or remains of the shapes, that have been dropped to the bottom of the board. All the squares are rememberd in the board array. We access it using the **shapeAt()** method.

```
if (curPiece.getShape() != Tetrominoes.NoShape) {

    for (int i = 0; i < 4; ++i) {

        int x = curX + curPiece.x(i);

        int y = curY - curPiece.y(i);

        drawSquare(g, 0 + x * squareWidth(),

                   boardTop + (BoardHeight - y - 1) * squareHeight(),

                   curPiece.getShape());

    }

}
```

In the second step, we paint the actual falling piece.

```
private void dropDown()

{

    int newY = curY;

    while (newY > 0) {
```

```
        if (!tryMove(curPiece, curX, newY - 1))

            break;

        --newY;

    }

    pieceDropped();

}
```

If we press the space key, the piece is dropped to the bottom. We simply try to drop the piece one line down until it reaches the bottom or the top of another fallen tetris piece.

```
private void clearBoard()

{

    for (int i = 0; i < BoardHeight * BoardWidth; ++i)

        board[i] = Tetrominoes.NoShape;

}
```

The **clearBoard()** method fills the board with empty NoSpapes. This is later used at collision detection.

```
private void pieceDropped()

{

    for (int i = 0; i < 4; ++i) {

        int x = curX + curPiece.x(i);
```

```
        int y = curY - curPiece.y(i);

        board[(y * BoardWidth) + x] = curPiece.getShape();

    }



    removeFullLines();



    if (!isFallingFinished)

        newPiece();

}
```

The **pieceDropped()** method puts the falling piece into the **board** array. Once again, the board holds all the squares of the pieces and remains of the pieces that has finished falling. When the piece has finished falling, it is time to check, if we can remove some lines off the board. This is the job of the **removeFullLines()** method. Then we create a new piece. More precisely, we try to create a new piece.

```
private void newPiece()

{

    curPiece.setRandomShape();

    curX = BoardWidth / 2 + 1;

    curY = BoardHeight - 1 + curPiece.minY();
```

```
    if (!tryMove(curPiece, curX, curY)) {

        curPiece.setShape(Tetrominoes.NoShape);

        timer.stop();

        isStarted = false;

        statusbar.setText("game over");

    }

}
```

The **newPiece()** method creates a new tetris piece. The piece gets a new random shape. Then we compute the initial**curX** and **curY** values. If we cannot move to the initial positions, the game is over. We top out. The timer is stopped. We put game over string on the statusbar.

```
private boolean tryMove(Shape newPiece, int newX, int newY)

{

    for (int i = 0; i < 4; ++i) {

        int x = newX + newPiece.x(i);

        int y = newY - newPiece.y(i);

        if (x < 0 || x >= BoardWidth || y < 0 || y >= BoardHeight)

            return false;

        if (shapeAt(x, y) != Tetrominoes.NoShape)

            return false;
```

```
    }



    curPiece = newPiece;

    curX = newX;

    curY = newY;

    repaint();

    return true;

}
```

The **tryMove()** method tries to move the tetris piece. The method returns false, if it has reached the board boundaries or it is adjacent to the already fallen tetris pieces.

```
for (int i = BoardHeight - 1; i >= 0; --i) {

    boolean lineIsFull = true;



    for (int j = 0; j < BoardWidth; ++j) {

        if (shapeAt(j, i) == Tetrominoes.NoShape) {

            lineIsFull = false;

            break;

        }

    }
```

```
    if (lineIsFull) {

        ++numFullLines;

        for (int k = i; k < BoardHeight - 1; ++k) {

            for (int j = 0; j < BoardWidth; ++j)

                board[(k * BoardWidth) + j] = shapeAt(j, k + 1);

        }

    }

}
```

Inside the **removeFullLines()** method, we check if there is any full row among all rows in the **board**. If there is at least one full line, it is removed. After finding a full line we increase the counter. We move all the lines above the full row one line down. This way we destroy the full line. Notice, that in our Tetris game, we use so called naive gravity. This means, that the squares may be left floating above empty gaps.

Every tetris piece has four squares. Each of the squares is drawn with the **drawSquare()** method. Tetris pieces have different colors.

```
g.setColor(color.brighter());

g.drawLine(x, y + squareHeight() - 1, x, y);

g.drawLine(x, y, x + squareWidth() - 1, y);
```

The left and top sides of a square are drawn with a brighter color. Similarly, the bottom and right sides are drawn with darker colors. This is to simulate a 3D edge.

We control the game with a keyboard. The control mechanism is implemented with a **KeyAdapter**. This is an inner class that overrides the **keyPressed()** method.

```
case KeyEvent.VK_RIGHT:

    tryMove(curPiece, curX + 1, curY);

    break;
```

If we pressed the left arrow key, we try to move the falling piece one square to the left.

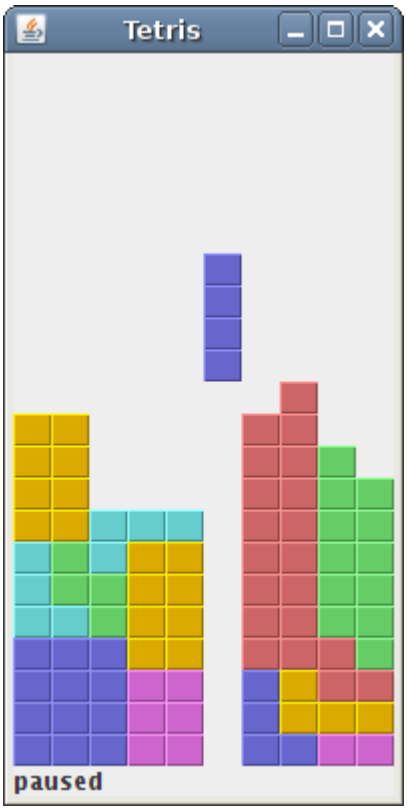


Figure: Tetris