FREE eBook

LEARNING swing

Free unaffiliated eBook created from
Stack Overflow contributors.

#swing

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: swing

It is an unofficial and free swing ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official swing.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with swing

Remarks

Swing has been superseded by JavaFX. Oracle generally recommends developing new applications with JavaFX. Still: Swing will be supported in Java for the foreseeable future. JavaFX also integrates well with Swing, to allow transitioning applications smoothly.

It is strongly recommended to have most of your Swing components on the Event Dispatch Thread. It's easy to forget to bundle your GUI setup into a `invokeLater` call. From the Java Documentation:

> Swing event handling code runs on a special thread known as the event dispatch thread. Most code that invokes Swing methods also runs on this thread. This is necessary because most Swing object methods are not "thread safe": invoking them from multiple threads risks thread interference or memory consistency errors. Some Swing component methods are labelled "thread safe" in the API specification; these can be safely invoked from any thread. All other Swing component methods must be invoked from the event dispatch thread. Programs that ignore this rule may function correctly most of the time, but are subject to unpredictable errors that are difficult to reproduce.

Also, unless for good reason, *always* make sure that you called `setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)` or else you might possibly have to deal with a memory leak if you forget to destroy the JVM.

Examples

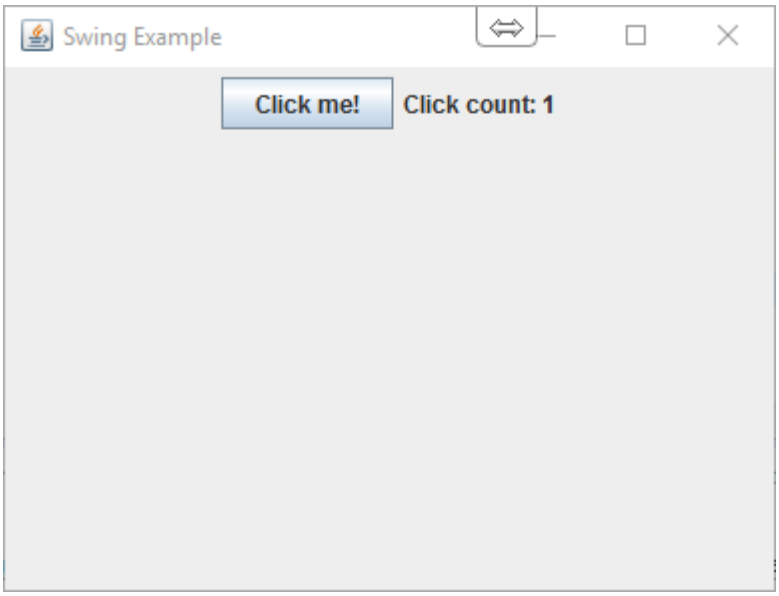
Incrementing with a button

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;

/**
 * A very simple Swing example.
 */
public class SwingExample {
    /**
     * The number of times the user has clicked the button.
     */
    private long clickCount;

    /**
     * The main method: starting point of this application.
     *
```

```
     * @param arguments the unused command-line arguments.
     */
    public static void main(final String[] arguments) {
        new SwingExample().run();
    }

    /**
     * Schedule a job for the event-dispatching thread: create and show this
     * application's GUI.
     */
    private void run() {
        SwingUtilities.invokeLater(this::createAndShowGui);
    }

    /**
     * Create the simple GUI for this application and make it visible.
     */
    private void createAndShowGui() {
        // Create the frame and make sure the application exits when the user closes
        // the frame.
        JFrame mainFrame = new JFrame("Counter");
        mainFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        // Add a simple button and label.
        JPanel panel = new JPanel();
        JButton button = new JButton("Click me!");
        JLabel label = new JLabel("Click count: " + clickCount);
        panel.add(button);
        panel.add(label);
        mainFrame.getContentPane().add(panel);

        // Add an action listener to the button to increment the count displayed by
        // the label.
        button.addActionListener(actionEvent -> {
            clickCount++;
            label.setText("Click count: " + clickCount);
        });

        // Size the frame.
        mainFrame.setBounds(80, 60, 400, 300);
        //Center on screen
        mainFrame.setLocationRelativeTo(null);
        //Display frame
        mainFrame.setVisible(true);
    }
}
```

Result

As the button labeled "Click me!" is pressed the click count will increase by one:



"Hello World!" on window title with lambda

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;

public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame("Hello World!");
            frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
            frame.setSize(200, 100);
            frame.setVisible(true);
        });
    }
}
```

Inside the `main` method:
On the first line `SwingUtilities.invokeLater` is called and a lambda expression with a block of code `() -> {...}` is passed to it. This executes the passed lambda expression on the EDT, which is short for Event Dispatch Thread, instead of the main thread. This is necessary, because inside the lambda expression's code block, there are Swing components going to be created and updated.

Inside the code block of the lambda expression:
On the first line, a new `JFrame` instance called `frame` is created using `new JFrame("Hello World!")`. This creates a window instance with "Hello World!" on its title. Afterwards on the second line the `frame` is configured to `EXIT_ON_CLOSE`. Otherwise the window will just be closed, but the execution of the program is going to remain active. The third line configures the `frame` instance to be 200 pixels in width and 100 pixels in height using the `setSize` method. Until now the execution won't show up anything at all. Only after calling `setVisible(true)` on the fourth line, the `frame` instance is configured to appear on the screen.

"Hello World!" on window title with compatibility

Using `java.lang.Runnable` we make our "Hello World!" example available to Java users with

versions dating all the way back to the 1.2 release:

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;

public class Main {
    public static void main(String[] args){
        SwingUtilities.invokeLater(new Runnable(){

            @Override
            public void run(){
                JFrame frame = new JFrame("Hello World!");
                frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
                frame.setSize(200, 100);
                frame.setVisible(true);
            }
        });
    }
}
```

Read Getting started with swing online: https://riptutorial.com/swing/topic/2191/getting-started-with-swing

Chapter 2: Basics

Examples

Delay a UI task for a specific period

All Swing-related operations happen on a dedicated thread (the EDT - **E**vent **D**ispatch **T**hread). If this thread gets blocked, the UI becomes non-responsive.

Therefore, if you want to delay an operation you cannot use `Thread.sleep`. Use a `javax.swing.Timer` instead. For example the following `Timer` will reverse the text of on a `JLabel`

```
  int delay = 2000;//specify the delay for the timer
  Timer timer = new Timer( delay, e -> {
    //The following code will be executed once the delay is reached
    String revertedText = new StringBuilder( label.getText() ).reverse().toString();
    label.setText( revertedText );
  } );
  timer.setRepeats( false );//make sure the timer only runs once
```

A complete runnable example which uses this `Timer` is given below: the UI contains a button and a label. Pressing the button will reverse the text of the label after a 2 second delay

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

public final class DelayedExecutionExample {

  public static void main( String[] args ) {
    EventQueue.invokeLater( () -> showUI() );
  }

  private static void showUI(){
    JFrame frame = new JFrame( "Delayed execution example" );

    JLabel label = new JLabel( "Hello world" );
    JButton button = new JButton( "Reverse text with delay" );
    button.addActionListener( event -> {
      button.setEnabled( false );
      //Instead of directly updating the label, we use a timer
      //This allows to introduce a delay, while keeping the EDT free
      int delay = 2000;
      Timer timer = new Timer( delay, e -> {
        String revertedText = new StringBuilder( label.getText() ).reverse().toString();
        label.setText( revertedText );
        button.setEnabled( true );
      } );
      timer.setRepeats( false );//make sure the timer only runs once
      timer.start();
    } );

    frame.add( label, BorderLayout.CENTER );
    frame.add( button, BorderLayout.SOUTH );
    frame.pack();
```

```
    frame.setDefaultCloseOperation( WindowConstants.EXIT_ON_CLOSE );
    frame.setVisible( true );
  }
}
```

Repeat a UI task at a fixed interval

Updating the state of a Swing component must happen on the Event Dispatch Thread (the EDT). The `javax.swing.Timer` triggers its `ActionListener` on the EDT, making it a good choice to perform Swing operations.

The following example updates the text of a `JLabel` each two seconds:

```
//Use a timer to update the label at a fixed interval
int delay = 2000;
Timer timer = new Timer( delay, e -> {
  String revertedText = new StringBuilder( label.getText() ).reverse().toString();
  label.setText( revertedText );
} );
timer.start();
```

A complete runnable example which uses this `Timer` is given below: the UI contains a label, and the text of the label will be reverted each two seconds.

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

public final class RepeatTaskFixedIntervalExample {
  public static void main( String[] args ) {
    EventQueue.invokeLater( () -> showUI() );
  }
  private static void showUI(){
    JFrame frame = new JFrame( "Repeated task example" );
    JLabel label = new JLabel( "Hello world" );

    //Use a timer to update the label at a fixed interval
    int delay = 2000;
    Timer timer = new Timer( delay, e -> {
      String revertedText = new StringBuilder( label.getText() ).reverse().toString();
      label.setText( revertedText );
    } );
    timer.start();

    frame.add( label, BorderLayout.CENTER );
    frame.pack();
    frame.setDefaultCloseOperation( WindowConstants.EXIT_ON_CLOSE );
    frame.setVisible( true );
  }
}
```

Running a UI task a fixed number of times

In the `ActionListener` attached to a `javax.swing.Timer`, you can keep track of the number of times the `Timer` executed the `ActionListener`. Once the required number of times is reached, you can use

the `Timer#stop()` method to stop the `Timer`.

```
Timer timer = new Timer( delay, new ActionListener() {
  private int counter = 0;
  @Override
  public void actionPerformed( ActionEvent e ) {
    counter++;//keep track of the number of times the Timer executed
    label.setText( counter + "" );
    if ( counter == 5 ){
      ( ( Timer ) e.getSource() ).stop();
    }
  }
});
```

A complete runnable example which uses this `Timer` is given below: it shows a UI where the text of the label will count from zero to five. Once five is reached, the `Timer` is stopped.

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

public final class RepeatFixedNumberOfTimes {
  public static void main( String[] args ) {
    EventQueue.invokeLater( () -> showUI() );
  }
  private static void showUI(){
    JFrame frame = new JFrame( "Repeated fixed number of times example" );
    JLabel label = new JLabel( "0" );

    int delay = 2000;
    Timer timer = new Timer( delay, new ActionListener() {
      private int counter = 0;
      @Override
      public void actionPerformed( ActionEvent e ) {
        counter++;//keep track of the number of times the Timer executed
        label.setText( counter + "" );
        if ( counter == 5 ){
          //stop the Timer when we reach 5
          ( ( Timer ) e.getSource() ).stop();
        }
      }
    });
    timer.setInitialDelay( delay );
    timer.start();

    frame.add( label, BorderLayout.CENTER );
    frame.pack();
    frame.setDefaultCloseOperation( WindowConstants.EXIT_ON_CLOSE );
    frame.setVisible( true );
  }
}
```

Creating Your First JFrame

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
```

```
public class FrameCreator {

    public static void main(String args[]) {
        //All Swing actions should be run on the Event Dispatch Thread (EDT)
        //Calling SwingUtilities.invokeLater makes sure that happens.
        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame();
            //JFrames will not display without size being set
            frame.setSize(500, 500);

            JLabel label = new JLabel("Hello World");
            frame.add(label);

            frame.setVisible(true);
        });
    }

}
```

As you may notice if you run this code, the label is position in a very bad place. This is difficult to change in a good manner using the `add` method. To allow more dynamic and flexible placing check out Swing Layout Managers.

Creating JFrame Sub-class

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;

public class CustomFrame extends JFrame {

    private static CustomFrame statFrame;

    public CustomFrame(String labelText) {
        setSize(500, 500);

        //See link below for more info on FlowLayout
        this.setLayout(new FlowLayout());

        JLabel label = new JLabel(labelText);
        add(label);

        //Tells the JFrame what to do when it's closed
        //In this case, we're saying to "Dispose" on remove all resources
        //associated with the frame on close
        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    }

    public void addLabel(String labelText) {
        JLabel label = new JLabel(labelText);
        add(label);
        this.validate();
    }

    public static void main(String args[]) {
        //All Swing actions should be run on the Event Dispatch Thread (EDT)
```

```
        //Calling SwingUtilities.invokeLater makes sure that happens.
        SwingUtilities.invokeLater(() -> {
            CustomFrame frame = new CustomFrame("Hello Jungle");
            //This is simply being done so it can be accessed later
            statFrame = frame;
            frame.setVisible(true);
        });

        try {
            Thread.sleep(5000);
        } catch (InterruptedException ex) {
            //Handle error
        }

        SwingUtilities.invokeLater(() -> statFrame.addLabel("Oh, hello world too."));
    }

}
```

For more information on FlowLayout here.

Listening to an Event

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class CustomFrame extends JFrame {

    public CustomFrame(String labelText) {
        setSize(500, 500);

        //See link below for more info on FlowLayout
        this.setLayout(new FlowLayout());

        //Tells the JFrame what to do when it's closed
        //In this case, we're saying to "Dispose" on remove all resources
        //associated with the frame on close
        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        //Add a button
        JButton btn = new JButton("Hello button");
        //And a textbox
        JTextField field = new JTextField("Name");
        field.setSize(150, 50);
        //This next block of code executes whenever the button is clicked.
        btn.addActionListener((evt) -> {
            JLabel helloLbl = new JLabel("Hello " + field.getText());
            add(helloLbl);
            validate();
        });
        add(btn);
        add(field);
    }

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

```
        //All Swing actions should be run on the Event Dispatch Thread (EDT)
        //Calling SwingUtilities.invokeLater makes sure that happens.
        SwingUtilities.invokeLater(() -> {
            CustomFrame frame = new CustomFrame("Hello Jungle");
            //This is simply being done so it can be accessed later
            frame.setVisible(true);
        });
    }

}
```

Create a "Please wait..." popup

This code can be added to any event like a listener, button, etc. A blocking `JDialog` will appear and will remain until the process is complete.

```
final JDialog loading = new JDialog(parentComponent);
JPanel p1 = new JPanel(new BorderLayout());
p1.add(new JLabel("Please wait..."), BorderLayout.CENTER);
loading.setUndecorated(true);
loading.getContentPane().add(p1);
loading.pack();
loading.setLocationRelativeTo(parentComponent);
loading.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
loading.setModal(true);

SwingWorker<String, Void> worker = new SwingWorker<String, Void>() {
    @Override
    protected String doInBackground() throws InterruptedException
        /** Execute some operation */
    }
    @Override
    protected void done() {
        loading.dispose();
    }
};
worker.execute(); //here the process thread initiates
loading.setVisible(true);
try {
    worker.get(); //here the parent thread waits for completion
} catch (Exception e1) {
    e1.printStackTrace();
}
```

Adding JButtons (Hello World Pt.2)

Assuming that you have successfully created a JFrame and that Swing has been imported...

You can import Swing entirely

```
import javax.Swing.*;
```

or You can import the Swing Components/Frame that you intend to use

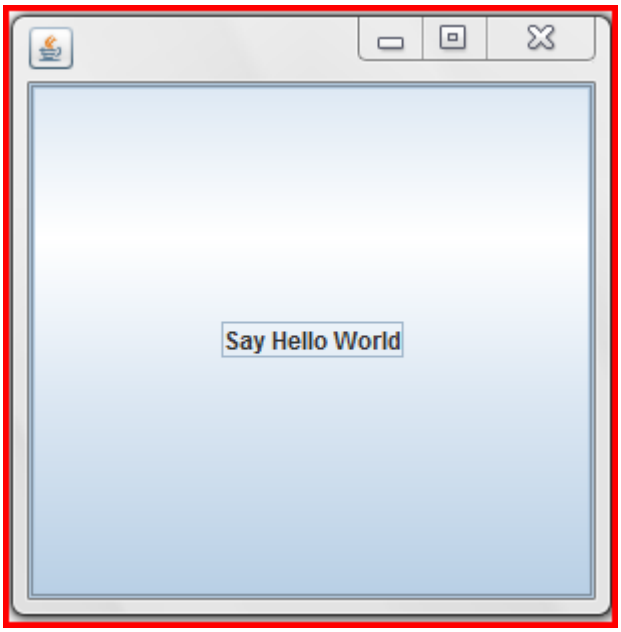
```
import javax.Swing.Jframe;
```

```
import javax.Swing.JButton;
```

Now down to adding the Jbutton...

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

    JFrame frame = new JFrame(); //creates the frame
    frame.setSize(300, 300);
    frame.setVisible(true);

    ////////////////////////ADDING BUTTON BELOW//////////////////////////////
    JButton B = new JButton("Say Hello World");
    B.addMouseListener(new MouseAdapter() {

        public void mouseReleased(MouseEvent arg0) {
            System.out.println("Hello World");
        }

    });
    B.setBounds(0, 0,frame.getHeight(), frame.getWidth());
    B.setVisible(true);
    frame.add(B);
    ////////////////////////////////////////////////////////////////////////
}
```

By Executing/Compiling this code you should get something like this...



When the button is clicked... "Hello World" should also appear in your console.

Read Basics online: https://riptutorial.com/swing/topic/5415/basics

Chapter 3: Graphics

Examples

Using the Graphics class

Intro

The `Graphics` class allows you to draw onto java components such as a `Jpanel`, it can be used to draw strings, lines, shapes and images. This is done by *overriding* the `paintComponent(Graphics g)` method of the `JComponent` you are drawing on using the `Graphics` object received as argument to do the drawing:

class `Board`

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

public class Board extends JPanel{

    public Board() {
        setBackground(Color.WHITE);
    }

    @override
    public Dimension getPreferredSize() {
        return new Dimension(400, 400);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // draws a line diagonally across the screen
        g.drawLine(0, 0, 400, 400);
        // draws a rectangle around "hello there!"
        g.drawRect(140, 180, 115, 25);
    }
}
```

wrapper class `DrawingCanvas`

```
import javax.swing.*;

public class DrawingCanvas extends JFrame {

    public DrawingCanvas() {

        Board board = new Board();

        add(board); // adds the Board to our JFrame
```

```
        pack(); // sets JFrame dimension to contain subcomponents

        setResizable(false);
        setTitle("Graphics Test");
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        setLocationRelativeTo(null); // centers window on screen
    }

    public static void main(String[] args) {
        DrawingCanvas canvas = new DrawingCanvas();
        canvas.setVisible(true);
    }
}
```

Colors

To draw shapes with different colors you must set the color of the `Graphics` object before each draw call using `setColor`:

```
g.setColor(Color.BLUE);  // draws a blue square
g.fillRect(10, 110, 100, 100);

g.setColor(Color.RED);  // draws a red circle
g.fillOval(10, 10, 100, 100);

g.setColor(Color.GREEN);  // draws a green triangle
int[] xPoints = {0, 200, 100};
int[] yPoints = {100, 100, 280};
g.fillPolygon(xPoints, yPoints, 3);
```

Drawing images

Images can be drawn onto a `JComponent` using the `drawImage` method of class `Graphics`:

loading an image

```
BufferedImage img;
try {
    img = ImageIO.read(new File("stackoverflow.jpg"));
} catch (IOException e) {
    throw new RuntimeException("Could not load image", e);
}
```

drawing the image

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    int x = 0;
```

```
    int y = 0;

    g.drawImage(img, x, y, this);
}
```

The `x` and `y` specify the location of the **top-left** of the image.

Using the Repaint Method to Create Basic Animation

The MyFrame class the extends JFrame and also contains the main method

```
import javax.swing.JFrame;


public class MyFrame extends JFrame{

    //main method called on startup
    public static void main(String[] args) throws InterruptedException {

        //creates a frame window
        MyFrame frame = new MyFrame();

        //very basic game loop where the graphics are re-rendered
        while(true){
            frame.getPanel().repaint();

            //The program waits a while before rerendering
            Thread.sleep(12);
        }
    }

    //the MyPanel is the other class and it extends JPanel
    private MyPanel panel;

    //constructor that sets some basic staring values
    public MyFrame(){
        this.setSize(500, 500);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //creates the MyPanel with paramaters of x=0 and y=0
        panel = new MyPanel(0,0);
        //adds the panel (which is a JComponent because it extends JPanel)
        //into the frame
        this.add(panel);
        //shows the frame window
        this.setVisible(true);
    }

    //gets the panel
    public MyPanel getPanel(){
        return panel;
    }

}
```

The MyPanel class that extends JPanel and has the paintComponent method

```
import java.awt.Graphics;
import javax.swing.JPanel;


public class MyPanel extends JPanel{

    //two int variables to store the x and y coordinate
    private int x;
    private int y;

    //construcor of the MyPanel class
    public MyPanel(int x, int y){
        this.x = x;
        this.y = y;
    }

    /*the method that deals with the graphics
        this method is called when the component is first loaded,
         when the component is resized and when the repaint() method is
        called for this component
    */
    @Override
    public void paintComponent(Graphics g){
        super.paintComponent(g);

        //changes the x and y varible values
        x++;
        y++;

        //draws a rectangle at the x and y values
        g.fillRect(x, y, 50, 50);
    }

}
```

Read Graphics online: https://riptutorial.com/swing/topic/5153/graphics

Chapter 4: GridBag Layout

Syntax

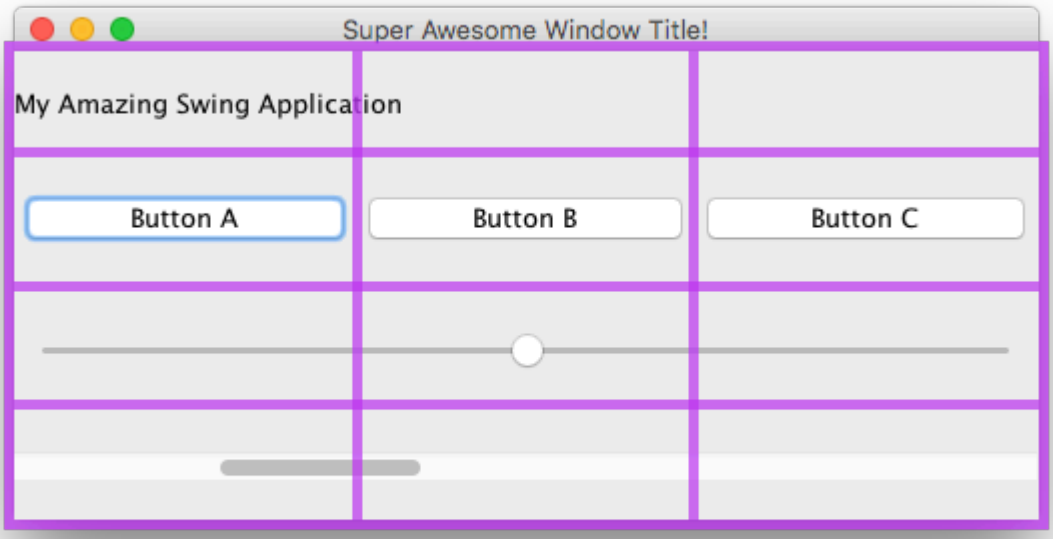
- frame.setLayout(new GridBagLayout()); //Set GridBagLayout for frame
- pane.setLayout(new GridBagLayout()); //Set GridBagLayout for Panel
- JPanel pane = new JPanel(new GridBagLayout()); //Set GridBagLayout for Panel
- GridBagConstraints c = new GridBagConstraints() //Initialize a GridBagConstraint

Examples

How does GridBagLayout work?

Layouts are used whenever you want your components to not just be displayed next to each other. The `GridBagLayout` is a useful one, as it divides your window into rows and columns, and you decide which row and column to put components into, as well as how many rows and colums big the component is.

Let's take this window as an example. Grid lines have been marked on to show the layout.



Here, I have created 6 components, laid out using a GridBagLayout.

Component	Position	Size
`JLabel`: "My Amazing Swing Application"	`0, 0`	`3, 1`
`JButton`: "Button A"	`0, 1`	`1, 1`

Component	Position	Size
`JButton`: "Button B"	`1, 1`	`1, 1`
`JButton`: "Button C"	`2, 1`	`1, 1`
`JSlider`	`0, 2`	`3, 1`
`JScrollBar`	`0, 3`	`3, 1`

Note that position `0, 0` is at the top left: `x` (column) values increase from left to right, `y` (row) values increase from top to bottom.

To start laying out components in a `GridBagLayout`, first set the layout of your `JFrame` or content pane.

```
frame.setLayout(new GridBagLayout());
//OR
pane.setLayout(new GridBagLayout());
//OR
JPanel pane = new JPanel(new GridBagLayout()); //Add the layout when creating your content
pane
```

Note that you never define the size of the grid. This is done automatically as you add your components.

Afterwards, you will need to create a `GridBagConstraints` object.

```
GridBagConstraints c = new GridBagConstraints();
```

To make sure that your components fill up the size of the window, you may want to set the weight of all componets to 1. Weight is used to determine how to distribute space among columns and rows.

```
c.weightx = 1;
c.weighty = 1;
```

Another thing that you may want to do is make sure that components take up as much horizontal space as they can.

```
c.fill = GridBagConstraints.HORIZONTAL;
```

You can also set other fill options if you wish.

```
GridBagConstraints.NONE //Don't fill components at all
GridBagConstraints.HORIZONTAL //Fill components horizontally
GridBagConstraints.VERTICAL //Fill components vertically
GridBagConstraints.BOTH //Fill components horizontally and vertically
```

When creating components, you will want to set where on the grid it should go, and how many grid tiles it should use. For example, to place a button in the 3rd row in the 2nd column, and take up a

5 x 5 grid space, do the following. Keep in mind that the grid starts at `0, 0`, not `1, 1`.

```
JButton button = new JButton("Fancy Button!");
c.gridx = 2;
c.gridy = 1;
c.gridwidth = 5;
c.gridheight = 5;
pane.add(buttonA, c);
```

When adding components to your window, remember to pass the constraints as a parameter. This can be seen in the last line in the code example above.

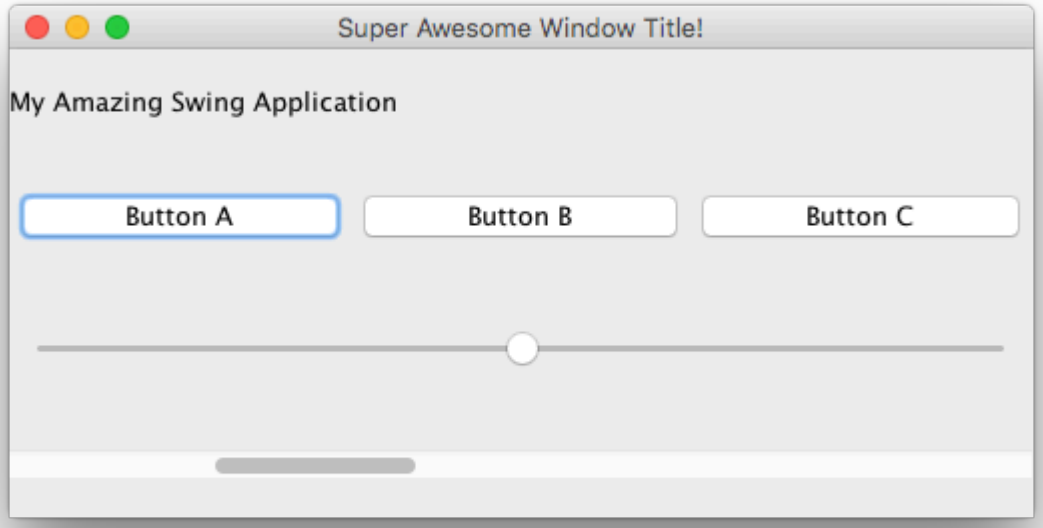
You can reuse the same `GridBagConstraints` for every component - changing it after adding a component doesn't change the previously added component.

Example

Here's the code for the example at the start of this section.

```
JFrame frame = new JFrame("Super Awesome Window Title!"); //Create the JFrame and give it a
title
frame.setSize(512, 256); //512 x 256px size
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE); //Quit the application when the
JFrame is closed

JPanel pane = new JPanel(new GridBagLayout()); //Create a pane to house all content, and give
it a GridBagLayout
frame.setContentPane(pane);

GridBagConstraints c = new GridBagConstraints();
c.weightx = 1;
c.weighty = 1;
c.fill = GridBagConstraints.HORIZONTAL;

JLabel headerLabel = new JLabel("My Amazing Swing Application");
c.gridx = 0;
c.gridwidth = 3;
c.gridy = 0;
pane.add(headerLabel, c);

JButton buttonA = new JButton("Button A");
c.gridx = 0;
c.gridwidth = 1;
c.gridy = 1;
pane.add(buttonA, c);

JButton buttonB = new JButton("Button B");
c.gridx = 1;
c.gridwidth = 1;
c.gridy = 1;
pane.add(buttonB, c);

JButton buttonC = new JButton("Button C");
c.gridx = 2;
c.gridwidth = 1;
c.gridy = 1;
```

```
pane.add(buttonC, c);

JSlider slider = new JSlider(0, 100);
c.gridx = 0;
c.gridwidth = 3;
c.gridy = 2;
pane.add(slider, c);

JScrollBar scrollBar = new JScrollBar(JScrollBar.HORIZONTAL, 20, 20, 0, 100);
c.gridx = 0;
c.gridwidth = 3;
c.gridy = 3;
pane.add(scrollBar, c);

frame.setVisible(true); //Show the window
```



Read GridBag Layout online: https://riptutorial.com/swing/topic/3698/gridbag-layout

Chapter 5: GridLayout

Examples

How GridLayout works

A `GridLayout` is a layout manager which places components inside a grid with equal cell sizes. You can set the number of rows, columns, the horizontal gap and the vertical gap using the following methods:

- `setRows(int rows)`
- `setColumns(int columns)`
- `setHgap(int hgap)`
- `setVgap(int vgap)`

or you can set them with the following constructors:

- `GridLayout(int rows, int columns)`
- `GridLayout(int rows, int columns, int hgap, int vgap)`

If the number of rows or columns is unknown, you can set the respective variable to `0`. For example:

```
new GridLayout(0, 3)
```

This will cause the `GridLayout` to have 3 columns and as many rows as needed.

The following example demonstrates how a `GridLayout` lays out components with different values for rows, columns, horizontal gap, vertical gap and screen size.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.EventQueue;
import java.awt.GridLayout;

import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSpinner;
import javax.swing.SpinnerNumberModel;
import javax.swing.WindowConstants;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class GridLayoutExample {

    private GridLayout gridLayout;
    private JPanel gridPanel, contentPane;
    private JSpinner rowsSpinner, columnsSpinner, hgapSpinner, vgapSpinner;
```

```
    public void createAndShowGUI() {
        gridLayout = new GridLayout(5, 5, 3, 3);

        gridPanel = new JPanel(gridLayout);

        final ChangeListener rowsColumnsListener = new ChangeListener() {
            @Override
            public void stateChanged(ChangeEvent e) {
                gridLayout.setRows((int) rowsSpinner.getValue());
                gridLayout.setColumns((int) columnsSpinner.getValue());
                fillGrid();
            }
        };

        final ChangeListener gapListener = new ChangeListener() {
            @Override
            public void stateChanged(ChangeEvent e) {
                gridLayout.setHgap((int) hgapSpinner.getValue());
                gridLayout.setVgap((int) vgapSpinner.getValue());
                gridLayout.layoutContainer(gridPanel);
                contentPane.revalidate();
                contentPane.repaint();
            }
        };

        rowsSpinner = new JSpinner(new SpinnerNumberModel(gridLayout.getRows(), 1, 10, 1));
        rowsSpinner.addChangeListener(rowsColumnsListener);

        columnsSpinner = new JSpinner(new SpinnerNumberModel(gridLayout.getColumns(), 1, 10,
1));
        columnsSpinner.addChangeListener(rowsColumnsListener);

        hgapSpinner = new JSpinner(new SpinnerNumberModel(gridLayout.getHgap(), 0, 50, 1));
        hgapSpinner.addChangeListener(gapListener);

        vgapSpinner = new JSpinner(new SpinnerNumberModel(gridLayout.getVgap(), 0, 50, 1));
        vgapSpinner.addChangeListener(gapListener);

        JPanel actionPanel = new JPanel();
        actionPanel.add(new JLabel("Rows:"));
        actionPanel.add(rowsSpinner);
        actionPanel.add(Box.createHorizontalStrut(10));
        actionPanel.add(new JLabel("Columns:"));
        actionPanel.add(columnsSpinner);
        actionPanel.add(Box.createHorizontalStrut(10));
        actionPanel.add(new JLabel("Horizontal gap:"));
        actionPanel.add(hgapSpinner);
        actionPanel.add(Box.createHorizontalStrut(10));
        actionPanel.add(new JLabel("Vertical gap:"));
        actionPanel.add(vgapSpinner);

        contentPane = new JPanel(new BorderLayout(0, 10));
        contentPane.add(gridPanel);
        contentPane.add(actionPanel, BorderLayout.SOUTH);

        fillGrid();

        JFrame frame = new JFrame("GridLayout Example");
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        frame.setContentPane(contentPane);
        frame.setSize(640, 480);
```

```
        frame.setLocationByPlatform(true);
        frame.setVisible(true);
    }

    private void fillGrid() {
        gridPanel.removeAll();
        for (int row = 0; row < gridLayout.getRows(); row++) {
            for (int col = 0; col < gridLayout.getColumns(); col++) {
                JLabel label = new JLabel("Row: " + row + " Column: " + col);
                label.setHorizontalAlignment(JLabel.CENTER);
                label.setBorder(BorderFactory.createLineBorder(Color.GRAY));
                gridPanel.add(label);
            }
        }
        contentPane.revalidate();
        contentPane.repaint();
    }

    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                new GridLayoutExample().createAndShowGUI();
            }
        });
    }

}
```

Read GridLayout online: https://riptutorial.com/swing/topic/2780/gridlayout

Chapter 6: JList

Examples

Modify the selected elements in a JList

Given a `JList` like

```
JList myList = new JList(items);
```

the selected items in the list can be modified through the `ListSelectionModel` of the `JList`:

```
ListSelectionModel sm = myList.getSelectionModel();
sm.clearSelection();                      // clears the selection
sm.setSelectionInterval(index, index);    // Sets a selection interval
                                          // (single element, in this case)
```

Alternatively, `JList` also provides some convenient methods to directly manipulate the selected indexes:

```
myList.setSelectionIndex(index);            // sets one selected index
                                            // could be used to define the Default Selection

myList.setSelectedIndices(arrayOfIndexes);  // sets all indexes contained in
                                            // the array as selected
```

Read JList online: https://riptutorial.com/swing/topic/5413/jlist

Chapter 7: Layout management

Examples

Border layout

```
import static java.awt.BorderLayout.*;
import javax.swing.*;
import java.awt.BorderLayout;

JPanel root = new JPanel(new BorderLayout());

root.add(new JButton("East"), EAST);
root.add(new JButton("West"), WEST);
root.add(new JButton("North"), NORTH);
root.add(new JButton("South"), SOUTH);
root.add(new JButton("Center"), CENTER);

JFrame frame = new JFrame();
frame.setContentPane(root);
frame.pack();
frame.setVisible(true);
```

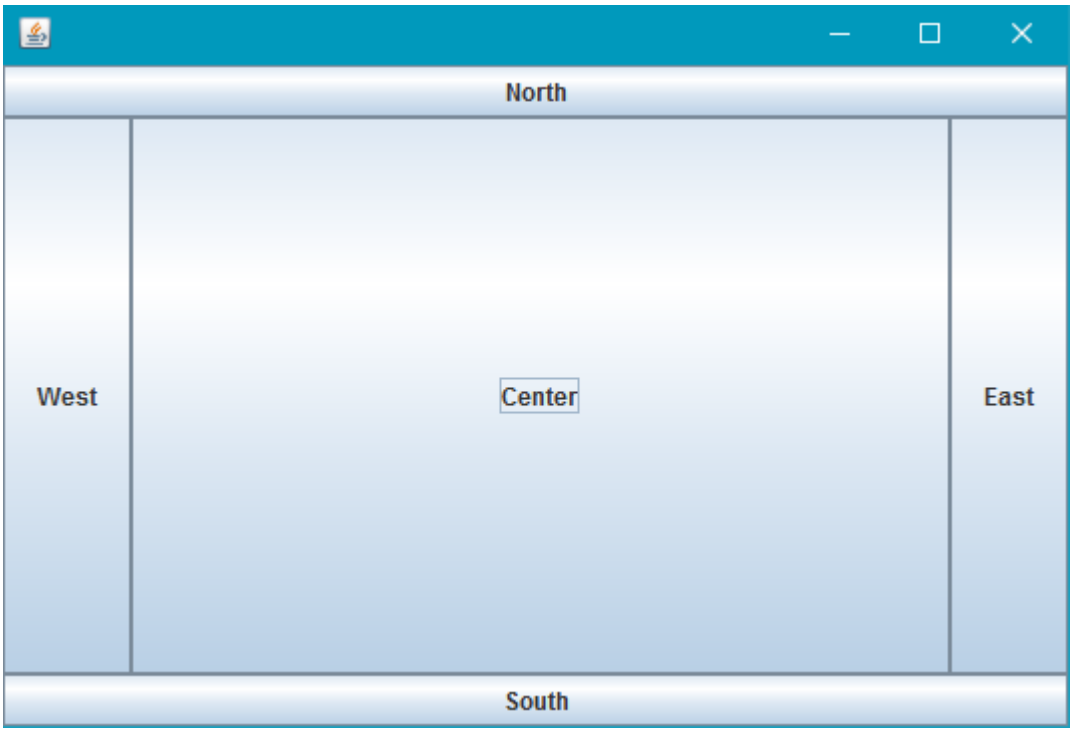
Border layout is one of the simplest layout managers. The way to use a layout manager is to set the manager of a `JPanel`.

Border Layout slots follow the following rules:

- North & South: preferred height
- East & West: preferred width
- Center: maximum remaining space

In `BorderLayout` slots can also be empty. The layout manager will automatically compensate for any empty spaces, resizing when needed.

Here is what this example looks like:



Flow layout

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

public class FlowExample {
    public static void main(String[] args){
        SwingUtilities.invokeLater(new Runnable(){

            @Override
            public void run(){
                JPanel panel = new JPanel();
                panel.setLayout(new FlowLayout());

                panel.add(new JButton("One"));
                panel.add(new JButton("Two"));
                panel.add(new JButton("Three"));
                panel.add(new JButton("Four"));
                panel.add(new JButton("Five"));

                JFrame frame = new JFrame();
                frame.setContentPane(Panel);
                frame.pack();
                frame.setVisible(true);
            }
        });
    }
}
```

Flow layout is the simplest layout manager that Swing has to offer. Flow layout tries to put everything on one line, and if the layout overflows the width, it will wrap the line. The order is specified by the order you add components to your panel.

Screenshots:

Grid layout

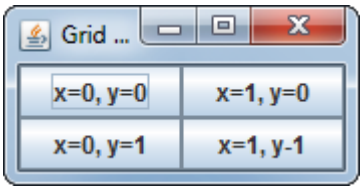
The `GridLayout` allows you to arrange components in the form of a grid.

You pass the number of rows and columns you want the grid to have to the `GridLayout`'s constructor, for example `new GridLayout(3, 2)` will create a `GridLayout` with 3 rows and 2 columns.

When adding components to a container with the `GridLayout`, the components will be added row by row, from left to right:

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

public class Example {
    public static void main(String[] args){
        SwingUtilities.invokeLater(Example::createAndShowJFrame);
    }

    private static void createAndShowJFrame(){
        JFrame jFrame = new JFrame("Grid Layout Example");

        // Create layout and add buttons to show restraints
        JPanel jPanel = new JPanel(new GridLayout(2, 2));
        jPanel.add(new JButton("x=0, y=0"));
        jPanel.add(new JButton("x=1, y=0"));
        jPanel.add(new JButton("x=0, y=1"));
        jPanel.add(new JButton("x=1, y-1"));

        jFrame.setContentPane(jPanel);
        jFrame.pack();
        jFrame.setLocationRelativeTo(null);
        jFrame.setVisible(true);
    }
}
```

This creates and shows a `JFrame` that looks like:



A more detailed description is available: GridLayout

Read Layout management online: https://riptutorial.com/swing/topic/5417/layout-management

Chapter 8: MigLayout

Examples

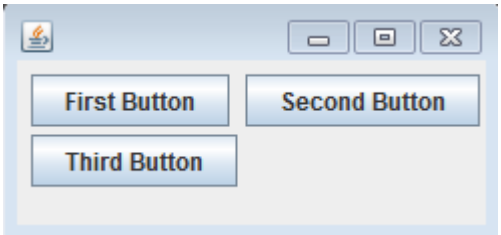
Wrapping elements

This example demonstrates how to place 3 buttons in total with 2 buttons being in the first row. Then a wrap occurs, so the last button is in a new row.

The constraints are simple strings, in this case "wrap" while placing the component.

```
public class ShowMigLayout {

    // Create the elements
    private final JFrame demo = new JFrame();
    private final JPanel panel = new JPanel();
    private final JButton button1 = new JButton("First Button");
    private final JButton button2 = new JButton("Second Button");
    private final JButton button3 = new JButton("Third Button");

    public static void main(String[] args) {
        ShowMigLayout showMigLayout = new ShowMigLayout();
        SwingUtilities.invokeLater(showMigLayout::createAndShowGui);
    }

    public void createAndShowGui() {
        // Set the position and the size of the frame
        demo.setBounds(400, 400, 250, 120);

        // Tell the panel to use the MigLayout as layout manager
        panel.setLayout(new MigLayout());

        panel.add(button1);
        // Notice the wrapping
        panel.add(button2, "wrap");
        panel.add(button3);

        demo.add(panel);
        demo.setVisible(true);
    }
}
```

Output:



Read MigLayout online: https://riptutorial.com/swing/topic/2966/miglayout

Chapter 9: MVP Pattern

Examples

Simple MVP Example

To illustrate a simple example usage of the MVP pattern, consider the following code which creates a simple UI with only a button and a label. When the button is clicked, the label updates with the number of times the button has been clicked.

We have 5 classes:

- Model - The POJO to maintain state (M in MVP)
- View - The class with UI code (V in MVP)
- ViewListener - Interface providing methods to responding to actions in the view
- Presenter - Responds to input, and updates the view (P in MVP)
- Application - The "main" class to pull everything together and launch the app

A minimal "model" class which just maintains a single `count` variable.

```
/**
 * A minimal class to maintain some state
 */
public class Model {
    private int count = 0;

    public void addOneToCount() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

A minimal interface to notify the listeners:

```
/**
 * Provides methods to notify on user interaction
 */
public interface ViewListener {
    public void onButtonClicked();
}
```

The view class constructs all UI elements. The view, and *only* the view, should have reference to UI elements (ie. no buttons, text fields, etc. in the presenter or other classes).

```
/**
 * Provides the UI elements
 */
```

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.WindowConstants;

public class View {
    // A list of listeners subscribed to this view
    private final ArrayList<ViewListener> listeners;
    private final JLabel label;

    public View() {
        final JFrame frame = new JFrame();
        frame.setSize(200, 100);
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        frame.setLayout(new GridLayout());

        final JButton button = new JButton("Hello, world!");

        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(final ActionEvent e) {
                notifyListenersOnButtonClicked();
            }
        });
        frame.add(button);

        label = new JLabel();
        frame.add(label);

        this.listeners = new ArrayList<ViewListener>();

        frame.setVisible(true);
    }

    // Iterate through the list, notifying each listner individualy
    private void notifyListenersOnButtonClicked() {
        for (final ViewListener listener : listeners) {
            listener.onButtonClicked();
        }
    }

    // Subscribe a listener
    public void addListener(final ViewListener listener) {
        listeners.add(listener);
    }

    public void setLabelText(final String text) {
        label.setText(text);
    }
}
```

The notification logic may also be coded like this in Java8:

```
        ...
```

```
        final Button button = new Button("Hello, world!");
        // In order to do so, our interface must be changed to accept the event parametre
        button.addActionListener((event) -> {
            notifyListeners(ViewListener::onButtonClicked, event);
            // Example of calling methodThatTakesALong, would be the same as callying:
            // notifyListeners((listener, long)->listener.methodThatTakesALong(long), 10L)
            notifyListeners(ViewListener::methodThatTakesALong, 10L);
        });
        frame.add(button);
        ...

/**
 * Iterates through the subscribed listeneres notifying each listener individually.
 * Note: the {@literal '<T>' in private <T> void} is a Bounded Type Parametre.
 *
 * @param <T>      Any Reference Type (basically a class).
 *
 * @param consumer A method with two parameters and no return,
 *                 the 1st parametre is a ViewListner,
 *                 the 2nd parametre is value of type T.
 *
 * @param data     The value used as parametre for the second argument of the
 *                 method described by the parametre consumer.
 */
private <T> void notifyListeners(final BiConsumer<ViewListener, T> consumer, final T data) {
    // Iterate through the list, notifying each listener, java8 style
    listeners.forEach((listener) -> {

        // Calls the funcion described by the object consumer.
        consumer.accept(listener, data);

        // When this method is called using ViewListener::onButtonClicked
        // the line: consumer.accept(listener,data); can be read as:
        // void accept(ViewListener listener, ActionEvent data) {
        //     listener.onButtonClicked(data);
        // }

    });
}
```

The interface must be refactored in order to take the ActionEvent as a parametre:

```
public interface ViewListener {
    public void onButtonClicked(ActionEvent evt);
    // Example of methodThatTakesALong signature
    public void methodThatTakesALong(long );
}
```

Here only one notify-method is needed, the actual listener method and its parameter are passed on as parameters. In case needed this can also be used for something a little less nifty than actual event handling, it all works as long as there is a method in the interface, e.g.:

```
        notifyListeners(ViewListener::methodThatTakesALong, -1L);
```

The presenter can take in the view and add itself as a listener. When the button is clicked in the view, the view notifies all listeners (including the presenter). Now that the presenter is notified, it

can take appropriate action to update the model (ie. the state of the application), and then update the view accordingly.

```
/**
 * Responsible to responding to user interaction and updating the view
 */
public class Presenter implements ViewListener {
    private final View view;
    private final Model model;

    public Presenter(final View view, final Model model) {
        this.view = view;
        view.addListener(this);
        this.model = model;
    }

    @Override
    public void onButtonClicked() {
        // Update the model (ie. the state of the application)
        model.addOneToCount();
        // Update the view
        view.setLabelText(String.valueOf(model.getCount()));
    }
}
```

To put everything together, the view can be created and injected into the presenter. Similarly, an initial model can be created and injected. While both *can* be created in the presenter, injecting them into the constructor allows for much simpler testing.

```
public class Application {
    public Application() {
        final View view = new View();
        final Model model = new Model();
        new Presenter(view, model);
    }

    public static void main(String... args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new Application();
            }
        });
    }
}
```

Read MVP Pattern online: https://riptutorial.com/swing/topic/5154/mvp-pattern

Chapter 10: StyledDocument

Syntax

- doc.insertString(index, text, attributes); //attributes should be a AttributeSet

Examples

Creating a DefaultStyledDocument

```
try {
    StyledDocument doc = new DefaultStyledDocument();
    doc.insertString(0, "This is the beginning text", null);
    doc.insertString(doc.getLength(), "\nInserting new line at end of doc", null);
    MutableAttributeSet attrs = new SimpleAttributeSet();
    StyleConstants.setBold(attrs, true);
    doc.insertString(5, "This is bold text after 'this'", attrs);
} catch (BadLocationException ex) {
    //handle error
}
```

DefaultStyledDocuments will probably be your most used resources. They can be created directly, and subclass the `StyledDocument` abstract class.

Adding StyledDocument to JTextPane

```
try {
        JTextPane pane = new JTextPane();
        StyledDocument doc = new DefaultStyledDocument();
        doc.insertString(0, "Some text", null);
        pane.setDocument(doc); //Technically takes any subclass of Document
    } catch (BadLocationException ex) {
        //handle error
    }
```

The JTextPane can then be added to any Swing GUI form.

Copying DefaultStyledDocument

`StyledDocuments` generally do not implement clone, and so have to copy them in a different way if that is necessary.

```
try {
        //Initialization
        DefaultStyledDocument sourceDoc = new DefaultStyledDocument();
        DefaultStyledDocument destDoc = new DefaultStyledDocument();
        MutableAttributeSet bold = new SimpleAttributeSet();
        StyleConstants.setBold(bold, true);
        MutableAttributeSet italic = new SimpleAttributeSet();
```

```
        StyleConstants.setItalic(italic, true);
        sourceDoc.insertString(0, "Some bold text. ", bold);
        sourceDoc.insertString(sourceDoc.getLength(), "Some italic text", italic);

        //This does the actual copying
        String text = sourceDoc.getText(0, sourceDoc.getLength()); //This copies text, but
loses formatting.
        for (int i = 0; i < text.length(); i++) {
            Element e = destDoc.getCharacterElement(i); //A Elment describes a particular part
of a document, in this case a character
            AttributeSet attr = e.getAttributes(); //Gets the attributes for the character
            destDoc.insertString(destDoc.getLength(), text.substring(i, i+1), attr); //Gets
the single character and sets its attributes from the element
        }
    } catch (BadLocationException ex) {
        //handle error
    }
```

Serializing a DefaultStyledDocument to RTF

Using the AdvancedRTFEditorKit library you can serialize a `DefaultStyledDocument` to an RTF string.

```
try {
   DefaultStyledDocument writeDoc = new DefaultStyledDocument();
   writeDoc.insertString(0, "Test string", null);

   AdvancedRTFEditorKit kit = new AdvancedRTFEditorKit();
   //Other writers, such as a FileWriter, may be used
   //OutputStreams are also an option
   Writer writer = new StringWriter();
   //You can write just a portion of the document by modifying the start
   //and end indexes
   kit.write(writer, writeDoc, 0, writeDoc.getLength());
   //This is the RTF String
   String rtfDoc = writer.toString();

   //As above this may be a different kind of reader or an InputStream
   StringReader reader = new StringReader(rtfDoc);
   //AdvancedRTFDocument extends DefaultStyledDocument and can generally
   //be used wherever DefaultStyledDocument can be.
   //However for reading, AdvancedRTFDocument must be used
   DefaultStyledDocument readDoc = new AdvancedRTFDocument();
   //You can insert at different values by changing the "0"
   kit.read(reader, readDoc, 0);
   //readDoc is now the same as writeDoc
} catch (BadLocationException | IOException ex) {
    //Handle exception
    ex.printStackTrace();
}
```

Read StyledDocument online: https://riptutorial.com/swing/topic/5416/styleddocument

Chapter 11: Swing Workers and the EDT

Syntax

- public abstract class SwingWorker<T,V>
- T - the result type returned by this SwingWorker's doInBackground and get methods.
- V - the type used for carrying out intermediate results by this SwingWorker's publish and process methods.
- T doInBackground() - The abstract function that must be overridden.Return type is T.

Examples

Main and event dispatch thread

Like any other java program, every swing program starts with a main method. The main method is initiated by the main thread. However, Swing components need to be created and updated on the event dispatch thread (or short: EDT). To illustrate the dynamic between the main thread and the EDT take a look at this Hello World! example.

The main thread is just used to delegate the creation of the window to the EDT. If the EDT is not initiated yet, the first call to `SwingUtilities.invokeLater` will setup the necessary infrastructure for processing Swing components. Furthermore, the EDT remains active in the background. The main thread is going to die directly after initiating the EDT setup, but the EDT will remain active until the user exits the program. This can be achieved by hitting the close box on the visible `JFrame` instance. This will shutdown the EDT and the program's process is going to entirely.

Find the first N even numbers and display the results in a JTextArea where computations are done in background.

```
import java.awt.EventQueue;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.SwingWorker;

class PrimeNumbersTask extends SwingWorker<List<Integer>, Integer> {
    private final int numbersToFind;

    private final JTextArea textArea;
```

```
    PrimeNumbersTask(JTextArea textArea, int numbersToFind) {
        this.numbersToFind = numbersToFind;
        this.textArea = textArea;
    }

    @Override
    public List<Integer> doInBackground() {
        final List<Integer> result = new ArrayList<>();
        boolean interrupted = false;
        for (int i = 0; !interrupted && (i < numbersToFind); i += 2) {
            interrupted = doIntenseComputing();
            result.add(i);
            publish(i); // sends data to process function
        }
        return result;
    }

    private boolean doIntenseComputing() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            return true;
        }
        return false;
    }

    @Override
    protected void process(List<Integer> chunks) {
        for (int number : chunks) {
            // the process method will be called on the EDT
            // thus UI elementes may be updated in here
            textArea.append(number + "\n");
        }
    }
}

public class SwingWorkerExample extends JFrame {
    private JTextArea textArea;

    public SwingWorkerExample() {
        super("Java SwingWorker Example");
        init();
    }

    private void init() {
        setSize(400, 400);
        setLayout(new GridLayout(1, 1));
        textArea = new JTextArea();
        add(textArea);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
    }

    public static void main(String args[]) throws Exception {

        SwingWorkerExample ui = new SwingWorkerExample();
```

```
        EventQueue.invokeLater(() -> {
            ui.setVisible(true);
        });

        int n = 100;
        PrimeNumbersTask task = new PrimeNumbersTask(ui.textArea, n);
        task.execute(); // run async worker which will do long running task on a
        // different thread
        System.out.println(task.get());
    }
}
```

Read Swing Workers and the EDT online: https://riptutorial.com/swing/topic/3431/swing-workers-and-the-edt

Chapter 12: timer in JFrame

Examples

Timer In JFrame

Suppose you have a button in your Java program that counts down a time. Here is the code for 10 minutes timer.

```
private final static long REFRESH_LIST_PERIOD=10 * 60 * 1000; //10 minutes

Timer timer = new Timer(1000, new ActionListener() {

   @Override
   public void actionPerformed(ActionEvent e) {
    if (cnt > 0) {
     cnt = cnt - 1000;
     btnCounter.setText("Remained (" + format.format(new Date(cnt)) + ")");
    } else {
     cnt = REFRESH_LIST_PERIOD;
     //TODO
    }

   }
  });

  timer.start();
```

Read timer in JFrame online: https://riptutorial.com/swing/topic/6745/timer-in-jframe

Chapter 13: Using Look and Feel

Examples

Using system L&F

Swing supports quite a few native L&Fs.
You can always easily install one without calling for a specific L&F class:

```
public class SystemLookAndFeel
{
    public static void main ( final String[] args )
    {
        // L&F installation should be performed within EDT (Event Dispatch Thread)
        // This is important to avoid any UI issues, exceptions or even deadlocks
        SwingUtilities.invokeLater ( new Runnable ()
        {
            @Override
            public void run ()
            {
                // Process of L&F installation might throw multiple exceptions
                // It is always up to you whether to handle or ignore them
                // In most common cases you would never encounter any of those
                try
                {
                    // Installing native L&F as a current application L&F
                    // We do not know what exactly L&F class is, it is provided by the
UIManager
                    UIManager.setLookAndFeel ( UIManager.getSystemLookAndFeelClassName () );
                }
                catch ( final ClassNotFoundException e )
                {
                    // L&F class was not found
                    e.printStackTrace ();
                }
                catch ( final InstantiationException e )
                {
                    // Exception while instantiating L&F class
                    e.printStackTrace ();
                }
                catch ( final IllegalAccessException e )
                {
                    // Class or initializer isn't accessible
                    e.printStackTrace ();
                }
                catch ( final UnsupportedLookAndFeelException e )
                {
                    // L&F is not supported on the current system
                    e.printStackTrace ();
                }

                // Now we can create some natively-looking UI
                // This is just a small sample frame with a single button on it
                final JFrame frame = new JFrame ();
                final JPanel content = new JPanel ( new FlowLayout () );
                content.setBorder ( BorderFactory.createEmptyBorder ( 50, 50, 50, 50 ) );
```

```
                content.add ( new JButton ( "Native-looking button" ) );
                frame.setContentPane ( content );
                frame.setDefaultCloseOperation ( WindowConstants.EXIT_ON_CLOSE );
                frame.pack ();
                frame.setLocationRelativeTo ( null );
                frame.setVisible ( true );
            }
        } );
    }
}
```

These are the native L&Fs JDK supports (OS -> L&F):

OS	L&F name	L&F class
Solaris, Linux with GTK+	GTK+	com.sun.java.swing.plaf.gtk.GTKLookAndFeel
Other Solaris, Linux	Motif	com.sun.java.swing.plaf.motif.MotifLookAndFeel
Classic Windows	Windows	com.sun.java.swing.plaf.windows.WindowsLookAndFeel
Windows XP	Windows XP	com.sun.java.swing.plaf.windows.WindowsLookAndFeel
Windows Vista	Windows Vista	com.sun.java.swing.plaf.windows.WindowsLookAndFeel
Macintosh	Macintosh	com.apple.laf.AquaLookAndFeel *
IBM UNIX	IBM	javax.swing.plaf.synth.SynthLookAndFeel *
HP UX	HP	javax.swing.plaf.synth.SynthLookAndFeel *

* these L&Fs are supplied by system vendor and actual L&F class name might vary

Using custom L&F

```
public class CustomLookAndFeel
{
    public static void main ( final String[] args )
    {
        // L&F installation should be performed within EDT (Event Dispatch Thread)
        // This is important to avoid any UI issues, exceptions or even deadlocks
        SwingUtilities.invokeLater ( new Runnable ()
        {
            @Override
            public void run ()
            {
                // Process of L&F installation might throw multiple exceptions
                // It is always up to you whether to handle or ignore them
                // In most common cases you would never encounter any of those
                try
```

```
                {
                    // Installing custom L&F as a current application L&F
                    UIManager.setLookAndFeel ( "javax.swing.plaf.metal.MetalLookAndFeel" );
                }
                catch ( final ClassNotFoundException e )
                {
                    // L&F class was not found
                    e.printStackTrace ();
                }
                catch ( final InstantiationException e )
                {
                    // Exception while instantiating L&F class
                    e.printStackTrace ();
                }
                catch ( final IllegalAccessException e )
                {
                    // Class or initializer isn't accessible
                    e.printStackTrace ();
                }
                catch ( final UnsupportedLookAndFeelException e )
                {
                    // L&F is not supported on the current system
                    e.printStackTrace ();
                }

                // Now we can create some pretty-looking UI
                // This is just a small sample frame with a single button on it
                final JFrame frame = new JFrame ();
                final JPanel content = new JPanel ( new FlowLayout () );
                content.setBorder ( BorderFactory.createEmptyBorder ( 50, 50, 50, 50 ) );
                content.add ( new JButton ( "Metal button" ) );
                frame.setContentPane ( content );
                frame.setDefaultCloseOperation ( WindowConstants.EXIT_ON_CLOSE );
                frame.pack ();
                frame.setLocationRelativeTo ( null );
                frame.setVisible ( true );
            }
        } );
    }
}
```

You can find a huge list of available Swing L&Fs in the topic here: Java Look and Feel (L&F)
Keep in mind that some of those L&Fs might be quite outdated at this point.

Read Using Look and Feel online: https://riptutorial.com/swing/topic/3627/using-look-and-feel

Chapter 14: Using Swing for Graphical User Interfaces

Remarks

Quitting the application on window close

It's easy to forget to quit the application when the window is closed. Remember to add the following line.

```
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE); //Quit the application when the JFrame is closed
```

Examples

Creating an Empty Window (JFrame)

Creating the JFrame

Creating a window is easy. You just have to create a `JFrame`.

```
JFrame frame = new JFrame();
```

Titling the Window

You may wish to give your window a title. You can so do by passing a string when creating the `JFrame`, or by calling `frame.setTitle(String title)`.

```
JFrame frame = new JFrame("Super Awesome Window Title!");
//OR
frame.setTitle("Super Awesome Window Title!");
```

Setting the Window Size

The window will be as small as possible when it has been created. To make it bigger, you can set its size explicitly:

```
frame.setSize(512, 256);
```

Or you can have the frame size itself based on the size of its contents with the `pack()` method.

```
frame.pack();
```

The `setSize()` and `pack()` methods are mutually exclusive, so use one or the other.

What to do on Window Close

Note that the application will **not** quit when the window has been closed. You can quit the application after the window has been closed by telling the `JFrame` to do that.

```
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
```

Alternatively, you can tell the window to do something else when it is closed.

```
WindowConstants.DISPOSE_ON_CLOSE //Get rid of the window
WindowConstants.EXIT_ON_CLOSE //Quit the application
WindowConstants.DO_NOTHING_ON_CLOSE //Don't even close the window
WindowConstants.HIDE_ON_CLOSE //Hides the window - This is the default action
```

Creating a Content Pane

An optional step is to create a content pane for your window. This is not needed, but if you want to do so, create a `JPanel` and call `frame.setContentPane(Component component)`.

```
JPanel pane = new JPanel();
frame.setContentPane(pane);
```

Showing the Window

After creating it, you will want to create your components, then show the window. Showing the window is done as such.

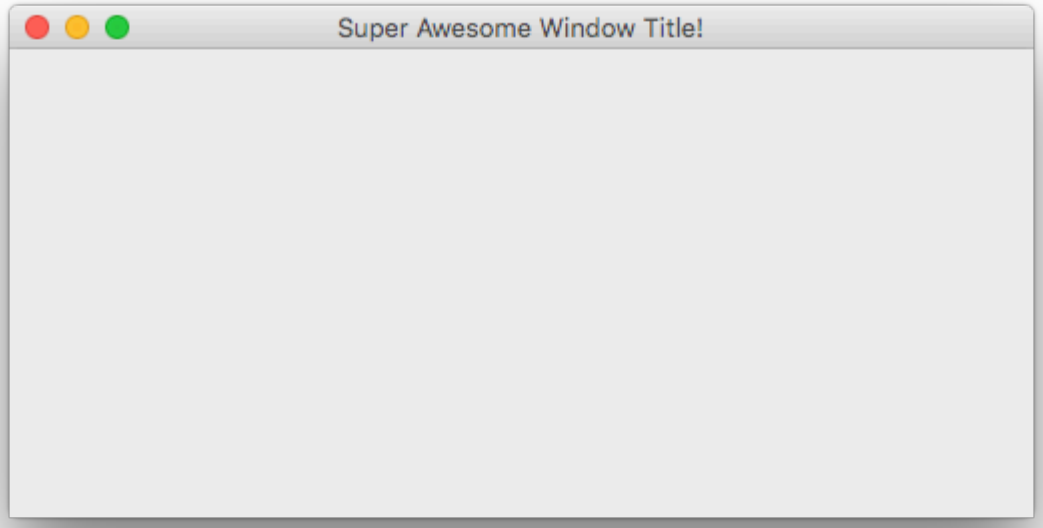
```
frame.setVisible(true);
```

Example

For those of you who like to copy and paste, here's some example code.

```
JFrame frame = new JFrame("Super Awesome Window Title!"); //Create the JFrame and give it a
title
frame.setSize(512, 256); //512 x 256px size
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE); //Quit the application when the
```

```
JFrame is closed

JPanel pane = new JPanel(); //Create the content pane
frame.setContentPane(pane); //Set the content pane

frame.setVisible(true); //Show the window
```



Adding Components

A component is some sort of user interface element, such as a button or a text field.

Creating a Component

Creating components is near identical to creating a window. Instead of creating a `JFrame` however, you create that component. For example, to create a `JButton`, you do the following.

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

Many components can have parameters passed to them when created. For example, a button can be given some text to display.

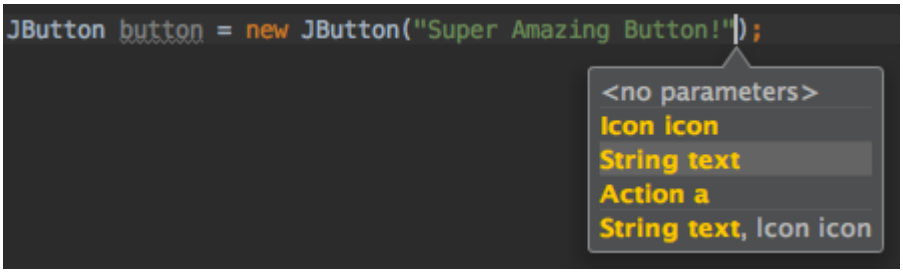
```
JButton button = new JButton("Super Amazing Button!");
```

If you don't want to create a button, a list of common components can be found in another example on this page.

The parameters that can be passed to them vary from component to component. A good way of checking what they can accept is by looking at the paramters within your IDE (If you use one). The

default shortcuts are listed below.

- IntelliJ IDEA - Windows / Linux: `CTRL + P`
- IntelliJ IDEA - OS X / macOS: `CMD + P`
- Eclipse: `CTRL + SHIFT + Space`
- NetBeans: `CTRL + P`



Showing the Component

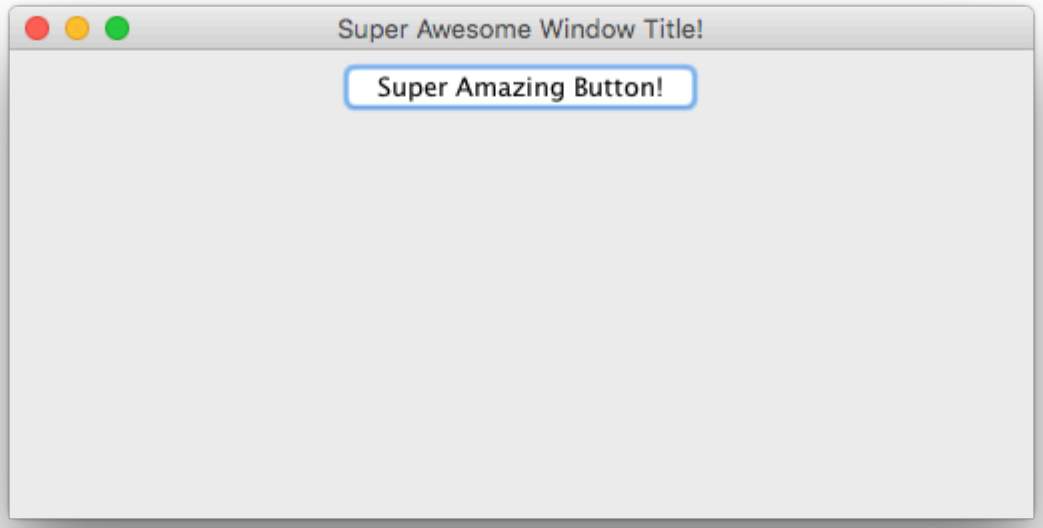
After a component has been created, you would typically set its parameters. After than, you need to put it somewhere, such as on your `JFrame`, or on your content pane if you created one.

```
frame.add(button); //Add to your JFrame
//OR
pane.add(button); //Add to your content pane
//OR
myComponent.add(button); //Add to whatever
```

Example

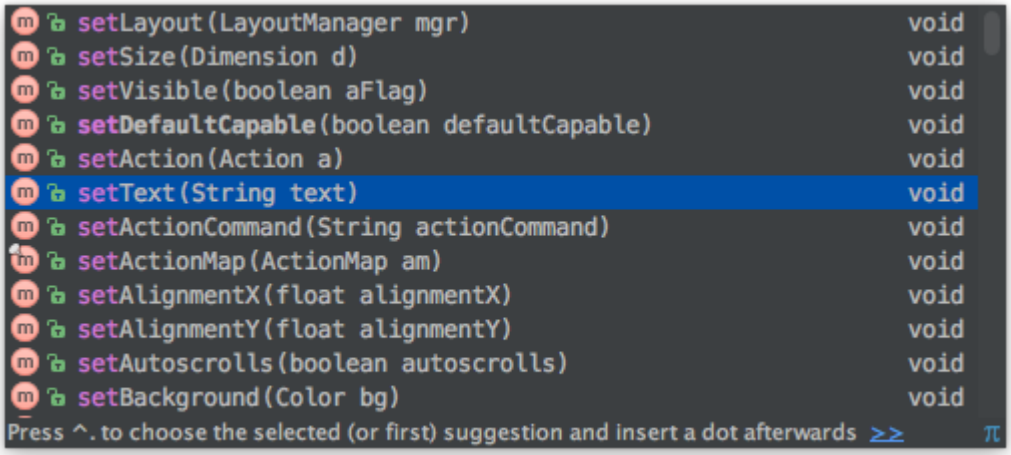
Here's an example of creating a window, setting a content pane, and adding a button to it.

```
JFrame frame = new JFrame("Super Awesome Window Title!"); //Create the JFrame and give it a
title
frame.setSize(512, 256); //512 x 256px size
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE); //Quit the application when the
JFrame is closed

JPanel pane = new JPanel(); //Create the content pane
frame.setContentPane(pane); //Set the content pane

JButton button = new JButton("Super Amazing Button!"); //Create the button
pane.add(button); //Add the button to the content pane

frame.setVisible(true); //Show the window
```



Setting Parameters for Components

Components have various parameters that can be set for them. They vary from component to component, so a good way to see what parameters can be set for components is to start typing `componentName.set`, and let your IDE's autocomplete (If you use an IDE) suggest methods. The default shortcut in many IDEs, if it doesn't show up automatically, is `CTRL + Space`.



Common parameters that are shared between all components

Description	Method
Sets the smallest size that the component can be (only if the layout manager honors the minimumSize property)	`setMinimumSize(Dimension minimumSize)`

Description	Method
Sets the biggest size that the component can be (only if the layout manager honors the maximumSize property)	`setMaximumSize(Dimension maximumSize)`
Sets the perferred size of the component (only if the layout manager honors the preferredSize property)	`setPreferredSize(Dimension preferredSize)`
Shows or hides the component	`setVisible(boolean aFlag)`
Sets whether the component should respond to user input	`setEnabled(boolean enabled)`
Sets the font of text	`setFont(Font font)`
Sets the text of the tooltip	`setToolTipText(String text)`
Sets the Backgroundcolor of the component	`setBackground(Color bg)`
Sets the Foregroundcolor (font color) of the component	`setForeground(Color bg)`

Common parameters in other components

Common Components	Description	Method
`JLabel`, `JButton`, `JCheckBox`, `JRadioButton`, `JToggleButton`, `JMenu`, `JMenuItem`, `JTextArea`, `JTextField`	Sets the text displayed	`setText(String text)`
`JProgressBar`, `JScrollBar`, `JSlider`, `JSpinner`	Set's a numerical value between the component's min and max values	`setValue(int n)`
`JProgressBar`, `JScrollBar`, `JSlider`, `JSpinner`	Set's the smallest possible value that the `value` property can be	`setMinimum(int n)`
`JProgressBar`, `JScrollBar`, `JSlider`, `JSpinner`	Set's the biggest possible value that the `value` property can be	`setMaxmimum(int n)`
`JCheckBox`, `JToggleBox`	Set's whether the value is true or false (Eg: Should a checkbox be checked?)	`setSelected(boolean b)`

Common Components

Description	Class
Button	`JButton`

Description	Class
Checkbox	`JCheckBox`
Drop down menu / Combo box	`JComboBox`
Label	`JLabel`
List	`JList`
Menu bar	`JMenuBar`
Menu in a menu bar	`JMenu`
Item in a menu	`JMenuItem`
Panel	`JPanel`
Progress bar	`JProgressBar`
Radio button	`JRadioButton`
Scroll bar	`JScrollBar`
Slider	`JSlider`
Spinner / Number picker	`JSpinner`
Table	`JTable`
Tree	`JTree`
Text area / Multiline text field	`JTextArea`
Text field	`JTextField`
Tool bar	`JToolBar`

Making Interactive User Interfaces

Having a button there is all well and good, but what's the point if clicking it does nothing? `ActionListener`s are used to tell your button, or other component to do something when it is activated.

Adding `ActionListener`s is done as such.

```
buttonA.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        //Code goes here...
        System.out.println("You clicked the button!");
```

```
    }
});
```

Or, if you're using Java 8 or above...

```
buttonA.addActionListener(e -> {
    //Code
    System.out.println("You clicked the button!");
});
```

Example (Java 8 and above)

```
JFrame frame = new JFrame("Super Awesome Window Title!"); //Create the JFrame and give it a
title
frame.setSize(512, 256); //512 x 256px size
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE); //Quit the application when the
JFrame is closed

JPanel pane = new JPanel(); //Create a pane to house all content
frame.setContentPane(pane);

JButton button = new JButton("Click me - I know you want to.");
button.addActionListener(e -> {
    //Code goes here
    System.out.println("You clicked me! Ouch.");
});
pane.add(buttonA);

frame.setVisible(true); //Show the window
```

Organizing Component Layout

Adding components one after another results in a UI that's hard to use, because the components all are **somewhere**. The components are ordered from top to bottom, each component in a separate "row".

To remedy this and provide you as developer with a possibility to layout components easily Swing has `LayoutManager`s.

These LayoutManagers are covered more extensively in Introduction to Layout Managers as well as the separate Layout Manager topics:

- Grid Layout
- GridBag Layout

Read Using Swing for Graphical User Interfaces online:
https://riptutorial.com/swing/topic/2982/using-swing-for-graphical-user-interfaces

Credits

S. No	Chapters	Contributors
1	Getting started with swing	Community, Freek de Bruijn, Petter Friberg, Vogel612, XavCo7
2	Basics	DarkV1, DonyorM, elias, Robin, Squidward
3	Graphics	Adel Khial, Ashlyn Campbell, Squidward
4	GridBag Layout	CraftedCart, Enwired, mayha, Vogel612
5	GridLayout	Lukas Rotter, user6653173
6	JList	Andreas Fester, Squidward, user6653173
7	Layout management	explv, J Atkin, mayha, pietrocalzini, recke96, Squidward, XavCo7
8	MigLayout	hamena314, keuleJ
9	MVP Pattern	avojak, ehzawad, Leonardo Pina, sjngm, Squidward
10	StyledDocument	DonyorM, Squidward
11	Swing Workers and the EDT	dpr, isaias-b, rahul tyagi
12	timer in JFrame	SSD
13	Using Look and Feel	Mikle Garin
14	Using Swing for Graphical User Interfaces	CraftedCart, mayha, Michael, Vogel612, Winter