OOP-JAVA
CST 205

Graphical User Interface and Database support of Java

MODULE 5

Author – Milan George Mathew

Syllabus

Swings fundamentals - Swing Key Features, Model View Controller (MVC), Swing Controls, Components and Containers, Swing Packages, Event Handling in Swings, Swing Layout Managers, Exploring Swings –JFrame, JLabel, The Swing Buttons, JTextField.

Java DataBase Connectivity (JDBC) - JDBC overview, Creating and Executing Queries – create table, delete, insert, select.

Contents

6.3. JDBC
6.4. Data Base Connectivity Steps
    6.4.1. Load and Register Drivers
    6.4.2. Establish Connection
    6.4.3. Creating a Statement
    6.4.4. Executing Statements
    6.4.5. Processing Result
    6.4.6. Closing

Swing: Key Features

Swing is a package in Java containing components for a **Graphical User Interface (GUI)** of which most are extended from **AWT (Abstract Window Toolkit)** package. Swing is entirely written in Java which makes it **platform independent** and **light weight**. Swing components are **not implemented** by **platform specific code** unlike the AWT components which varies with the **operating system** as it uses native code to render components. Swing provides more **powerful and flexible** functionalities than standard AWT components. Swing classes are defined by javax.swing package and its subpackages. Two key features of swing are:

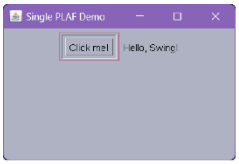
- Swing components are **Light-weight**
- Swing supports **Pluggable Look and Feel (PLAF)**

Light weight

Swing components are lightweight because they are written **entirely in Java** and does not map directly to **platform-specific peers** that were written by Java API developers to interface with **native objects**. Lightweight components do not call the **native operating system** for drawing the **graphical user interface (GUI)** components. They are rendered using **graphical primitives**. The components can be transparent allowing non rectangular components too. The look and feel of each of the component is determined by **Swing** not by the underlying operating system.

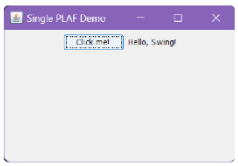
Pluggable Look and Feel

Swing supports a **pluggable look and feel (PLAF)**. Because each Swing component is rendered by **Java code not by native peers**, the look and feel of a component is under the **control of Swing.** It is possible to **separate the look and feel** of a component from the **logic of the component**. Just a single line of code can change the look and feel of the components and swing has some predefined look and feel plugins provided in different classes. It is possible to "**plug in**" a new look and feel for any given component without creating any side effects in the code that uses that component.

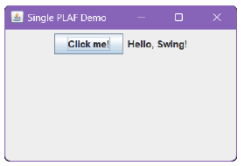


Motif

Nimbus



Windows



Metal

`javax.swing.plaf.metal.MetalLookAndFeel`

`javax.swing.plaf.nimbus.NimbusLookAndFeel`

It is possible to define entire sets of look-and-feels that represent different GUI styles. It is possible to define a look and feel that is consistent across all platforms. It is possible to create a look and feel that acts like a specific platform.

Swing	AWT
Swing components are not platform dependent.	AWT components are platform dependent
Swing provides several additional components such as scroll panes, trees etc in addition to other standard components	The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
Swing is written entirely in Java. So swing components are light-weight	AWT components use native code. So they are heavy weight
Swing supports a pluggable look and feel (PLAF) that can be dynamically changed at run-time depending on environment	In AWT look and feel of each component is fixed and it is difficult to change its look and feel.
Swing follow MVC	AWT does not follow MVC

Swing vs AWT

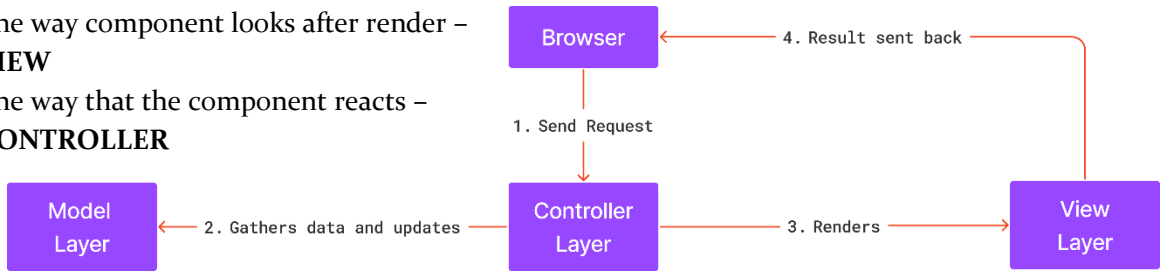
Swing	Applet
Swing is light weight Component.	Applet is heavy weight Component.
Swing needs main method to execute the program.	Applet does not need main method to execute.
Swing follows MVC (Model view Controller)	Applet does not follow MVC.
Swing have its own Layout like most popular Box Layout.	Applet uses various layouts like Flow Layout
Swing uses for stand alone Applications.	Applet need HTML code to Run.
To execute Swing, the browser is not needed.	To execute Applet program we need browsers like browser Appletviewer, web etc

Swing vs Applet

Model View Controller (MVC)

A visual component is a composite of three different aspects

- The state information associated with the component - **MODEL**
- The way component looks after render – **VIEW**
- The way that the component reacts – **CONTROLLER**



MVC (Model View Controller) architecture is successful because each piece of the design corresponds to an aspect of a component.The **model** contains simple Java classes that is gathered by the **controller** on a server request from the browser and then is passed on to the **view** layer to be **rendered** to visual components.

Terminology

Model

Corresponds to the state information associated with the component.

View

Determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.

Controller

Determines how the component reacts to the user.

By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. Swing uses a **modified version of MVC** that **combines view and the controller** them into a single logical entity called the **UI delegate**. So, Swing's approach is called either the **Model-Delegate architecture** or the **Separable Model architecture**. The PLAF in swing is enabled by this architecture, as view and model are separated from the controller layer.

Swing Controls

To support **Model-Delegate** architecture, most Swing components contain two objects

- **Model** – defined by interfaces
- **UI Delegate** – classes that inherit **ComponentUI**

Swing controls plays major role in swing applications, every application will have some components and their corresponding event listener. Swing controls will inherit the properties from following classes.

- **Component**
- **Container**
- **JComponent**

Some swing controls are

- Container Control (Parent control) – it holds other controls like child controls or components
    - JFrame
    - JPanel
- Child Control – it holds controls inside its container only. They can be components or containers
    - JTextArea

- JLabel
- JButton

When **container control is deleted** then its **child controls are also deleted.**

Containers and Components

A Swing GUI consists of two key items: **components** and **containers.**

Container – is a **special type** of component that is **designed to hold other components**. To display a component is must first be held in a container. So, all Swing GUIs have at least one container. As they are components by itself **one container can be held within another container**.

Components – derived from the **JComponent** Class. **JComponent** provides the functionality that is common to all components. **JComponent** supports the **pluggable look and feel**. **JComponent** inherits the **AWT classes Container** and **Component**. So, a Swing component is built on and compatible with an AWT component.

The class names for swing components are

- JApplet
- JButton
- JCheckBox
- JCheckBoxMenuItem
- JColorChooser
- JComboBox
- JComponent
- JDesktopPane
- JDialog
- JEditorPane
- JFileChooser
- JFormattedTextField
- JFrame
- JInternalFrame
- JLabel
- JLayeredPane
- JList
- JMenu
- JMenuBar
- JMenuItem
- JOptionPane
- JPanel
- JPasswordField
- JPopupMenu
- JProgressBar

- JRadioButton
- JRadioButtonMenuItem
- JRootPane
- JScrollBar
- JScrollPane
- JSeparator
- JSlider
- JSpinner
- JSplitPane
- JTabbedPane
- JTable
- JTextArea
- JTextField
- JTextPane
- JToggleButton
- JToolBar
- JToolTip
- JTree
- JViewport

Swing defines **two types** of containers:

- **Top Level containers** - heavy weight containers, does not inherit **JComponent** but inherits **Component** and **Container.** They are not contained within any other container and is to be at the top of the containment hierarchy. **JFrame** is the most common used for applications and **JApplet** for applets.
    - **JFrame**
    - **JApplet**
    - **JWindow**
    - **JDialog**
- **Light-weight containers** – inherits **JComponent**. They are often used to organize and manage groups of related components because a light-weight container can be contained within another container.
    - **JPanel**

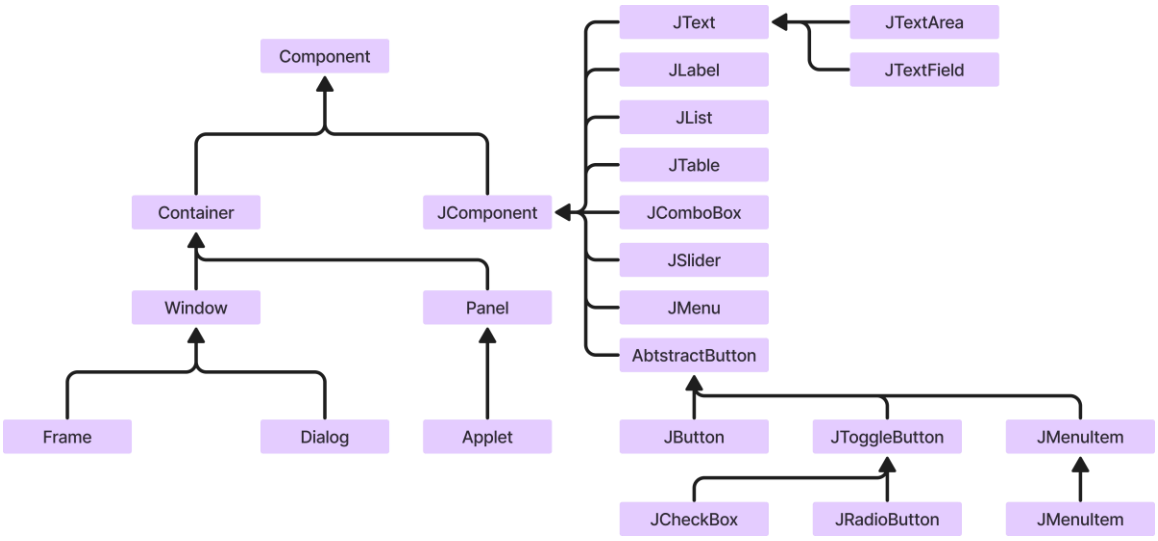
Top-level Container Panes

Each top-level container defines a set of panes. At the **top** of the hierarchy is an instance of **JRootPane**. **JRootPane** is a **lightweight container** whose purpose is to **manage the other panes**. It also helps manage the optional menu bar.

The panes that comprise the root pane are called:

- **Glass Pane**
    - It is a **top-level container**

    - It sits above and **completely covers** all other panes.
    - By default, it is a transparent instance of **JPanel**.
- **Layered Pane**
    - The layered pane is an instance of **JLayeredPane**.
    - The layered pane allows components to be given a **depth value**.
    - This value determines which component overlays another.
- **Content Pane**
    - The pane with which your **application will interact the most** is the content pane
    - When we **add a component**, such as a button, to a **top- level container**, we will add it to the **content pane**.
    - By default, the content pane is an opaque instance of **JPanel**.



Swing Packages

Swing is a very large subsystem and makes use of many packages. These are the packages used by Swing that are defined by **Java SE6**. The javax.swing package contains all the basic classes such as push buttons, labels, and check boxes used to implement Swing GUIs and thus it must be imported in a every program using Swing.

Some of its subpackages are

- `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`

View a simple Swing program implementation here

Swing Demo

This is a simple Java program that demonstrates the use of the Swing package to create a JFrame and a JLabel.

```java
package module_5.M5_p2_swing_demo;
// simple program to demonstrate the Swing package using a JFrame and
JLabel
import javax.swing.*;

public class Swing_demo {
    Swing_demo() {

        // Set title to the frame
        JFrame jfrm = new JFrame("A Simple Swing");

        // Set the size of the JFrame
        jfrm.setSize(275, 100);

         // Set the default close operation for the JFrame
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a JLabel with the text message
        JLabel jlab = new JLabel("Hello welcome to a simple swing
program");

         // Add the JLabel to the JFrame
        jfrm.add(jlab);

        // Make the JFrame visible
        jfrm.setVisible(true);
    }
    public static void main(String args[]) {
        // Schedule a job for the event-dispatching thread
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                 // Create a new instance of the Swing_demo class
                new Swing_demo();
            }
        });
    }
}
```

Step by Step Explanation

1. Import the necessary packages

```
import javax.swing.*;
```

- This line imports the Swing package, which contains the classes and interfaces needed to create graphical user interfaces (GUIs) in Java.

2. Create a JFrame

```
JFrame jfrm = new JFrame("A Simple Swing");
```

- This line creates a new JFrame object, which is the main window of a Swing application. The title of the frame is set to "A Simple Swing".

3. Set the size of the JFrame

```
jfrm.setSize(275, 100);
```

- This line sets the size of the JFrame to 275 pixels wide and 100 pixels high.

4. Set the default close operation for the JFrame

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

- This line sets the default close operation for the JFrame to EXIT_ON_CLOSE, which means that the application will exit when the user clicks the close button on the frame.

5. Create a JLabel

```
JLabel jlab = new JLabel("Hello welcome to a simple swing program");
```

- This line creates a new JLabel object, which is used to display text on the frame. The text of the label is set to "Hello welcome to a simple swing program". ***JLabel is the simplest*** of all components of Swing as it has ***no interaction with the user*** and is ***passive***

6. Add the JLabel to the JFrame

```
jfrm.add(jlab);
```

- This line adds the JLabel to the JFrame.

7. Make the JFrame visible

```
jfrm.setVisible(true);
```

- This line makes the JFrame visible on the screen.

8. Schedule a job for the event-dispatching thread

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new Swing_demo();
    }
});
```

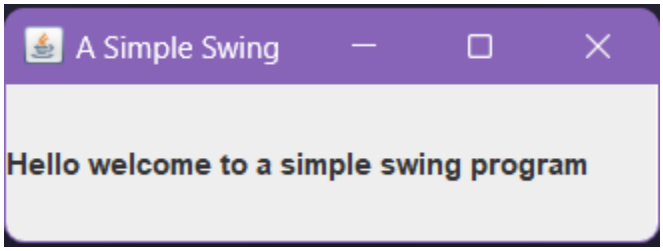
- This line schedules a job for the event-dispatching thread, which is responsible for handling all GUI events. The job is to create a new instance of the Swing_demo class.

Conclusion

- This is a simple example of how to use the Swing package to create a JFrame and a JLabel. By following the steps in this

Output

- The Output of the program may look like this



Event Handling in Swing

Event handling is done in Swing through adding an eventListener to the desired component and then creating an event handler that implements the corresponding events' member methods. This is implemented by the Delegation event model and the events are packaged in java.awt.event and javax.swing .event.

View an Event handling Swing program implementation here

Event Demo

This is a simple Java program that demonstrates the event handling mechanism of swing using two buttons and a label

```
package module_5.M5_p2_swing_event_handle;
// this is a program in Swing that is used to demonstrate the event
handling mechanism of swing using two buttons and a label
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EventDemo extends JFrame {
    JLabel jlab;

    // Main constructor of the EventDemo class
    EventDemo() {
        // Set the frame title and set it to close when clicked on the
close button
        JFrame jfrm = new JFrame("An Event example");
        // Set the frame layout to FlowLayout
        jfrm.setLayout(new FlowLayout());
        // Set the frame size to 220x90
        jfrm.setSize(220, 90);
```

```
        // Set the default operation when the frame is closed to exit the
program
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Create an OK button
        JButton jbtOK = new JButton("OK");

        // Create a Cancel button
        JButton jbtCancel = new JButton("Cancel");

        // Add an ActionListener to the OK button to perform an action
when the button is clicked
        jbtOK.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jlab.setText("OK Button is clicked.");
            }
        });

        // Add an ActionListener to the Cancel button to perform an action
when the button is clicked
        jbtCancel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jlab.setText("Cancel Button is clicked.");
            }
        });

        // Add the buttons to the frame
        jfrm.add(jbtOK);
        jfrm.add(jbtCancel);

        // Create a label and add it to the frame
        jlab = new JLabel("Press a button!");
        jfrm.add(jlab);

        // Display the frame
        jfrm.setVisible(true);
    }

    // Main method
    public static void main(String[] args) {
        // Create an instance of the EventDemo class
        new EventDemo();
    }
}
```

Step-by-Step Explanation

1. Import the necessary packages

```
import javax.swing.*;

import java.awt.*;

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

- This line imports the Swing package, which contains the classes and interfaces needed to create graphical user interfaces (GUIs) in Java.

2. Create a JFrame

```
JFrame jfrm = new JFrame("An Event example");
```

- This line creates a new JFrame object, which is the main window of a Swing application. The title of the frame is set to "An Event example".

3. Set the size of the JFrame

```
jfrm.setSize(220, 90);
```

- This line sets the size of the JFrame to 220 pixels wide and 90 pixels high.

4. Set the default close operation for the JFrame

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

- This line sets the default close operation for the JFrame to EXIT_ON_CLOSE, which means that the application will exit when the user clicks the close button on the frame.

5. Create two JButtons

```
// Create an OK button
    JButton jbtOK = new JButton("OK");


// Create a Cancel button
    JButton jbtCancel = new JButton("Cancel");
```

- These lines creates an OK button and a Cancel button using JButton class.

6. Adding and implementing Action Listeners

```
    jbtOK.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jlab.setText("OK Button is clicked.");
```

```
            }
        });
    jbtCancel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jlab.setText("Cancel Button is clicked.");
            }
        });
```

- These lines add Action Listeners to the buttons that implements the one and only one method in the ActionListener interface *actionPerformed* and then sets the label to show a specific text as message

7. Add the Button to the Frame

```
jfrm.add(jbtOK);
jfrm.add(jbtCancel);
```

- These lines add the buttons to the JFrame

8. Initialize the label and add it to Frame

```
jlab = new JLabel("Press a button!");
jfrm.add(jlab);
```

- These lines initialisez the label with a text and then displays adds the label to fram

9. Make the JFrame visible

```
jfrm.setVisible(true);
```

- This line makes the JFrame visible on the screen.

8. Schedule a job for the event-dispatching thread

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new EventDemo();
    }
});
```

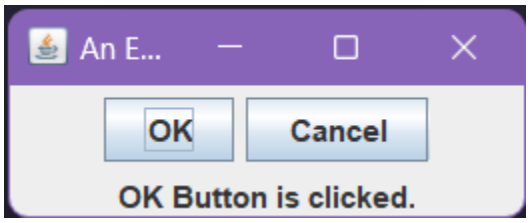
- This line schedules a job for the event-dispatching thread, which is responsible for handling all GUI events. The job is to create a new instance of the EventDemo class.

Conclusion

- This is a simple example of how to setup Event Handling using Swing package.

Output

- The Output of the program may look like this



Event-Dispatching Thread

To enable the GUI code to be created on the event dispatching thread, we must use one of two methods that are defined by the **SwingUtilities** class.

- invokeLater() - asynchronous
- invokeAndWait() - synchronous

```
static void invokeLater(Runnable obj)
```

```
static void invokeAndWait(Runnable obj) throws InterruptedException,
InvocationTargetException
```

invokeLater: Suitable for most cases where you want to update the GUI asynchronously. It's commonly used for tasks that don't require waiting for the GUI update to complete. It returns immediately.

invokeAndWait: Use when you need to wait for a specific GUI update to complete before proceeding. This is less common than **invokeLater** because it can lead to responsiveness issues if used inappropriately. It waits for the Runnable to return.

Swing Layout Managers

A layout manager automatically **arranges our controls within a window** by using some type of algorithm. Each **Container object has a layout manager** associated with it. A **layout manager** is an instance of any class that implements the **LayoutManager interface.** The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used.

```
void setLayout(LayoutManager layoutObj)
```

The **layout manager is notified** each time we **add a component** to a container. Each layout manager keeps track of a list of components that are stored by their names. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize( )** and **preferredLayoutSize( )** methods.

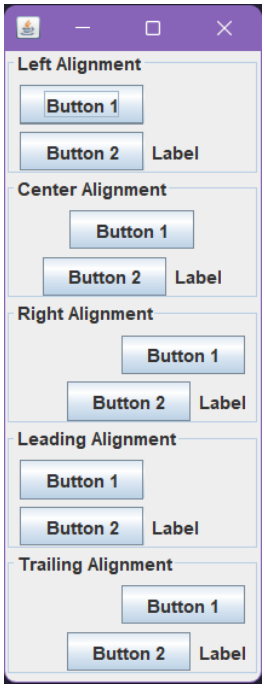
There are several LayoutManager Classes for Swing:

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

FlowLayout

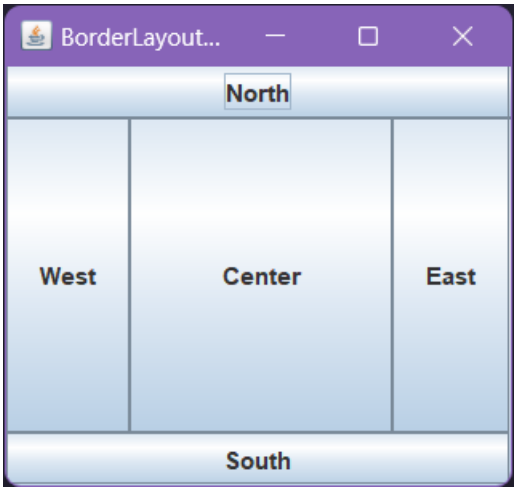
FlowLayout is a layout manager in Java Swing that arranges components in a **left-to-right flow**, placing them in a row until the **row is full**, and then moving on to the **next row**. It is one of the **simplest and most straightforward** layout managers and is suitable for scenarios where you want components to be displayed in a **natural flow.** The default direction is from **left-to-right and top-to-bottom.** FlowLayout has three constructors defined:

```
FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```



- **how** – specifies how each line is to be aligned. The valid values for this argument are:
    - FlowLayout.**LEFT**
    - FlowLayout.**CENTER**
    - FlowLayout.**RIGHT**
    - FlowLayout.**LEADING**
    - FlowLayout.**TRAILING**
- **horz** and **vert** – specifies the space horizontally and vertically between the components, which is 5 pixels by default.

BorderLayout



The content pane associated with **JFrame** by default uses the **BorderLayout** for arranging the components with respect to regions defined by boundaries.

The edge regions are narrow and the center is large area. The sides are referred to as:

- **North**
- **South**
- **East**
- **West**

and the middle area as **Center**

It has two constructors:

```
BorderLayout( )
```

```
BorderLayout(int horz, int vert)
```

- **horz** and **vert** – specifies the space horizontally and vertically between the components.

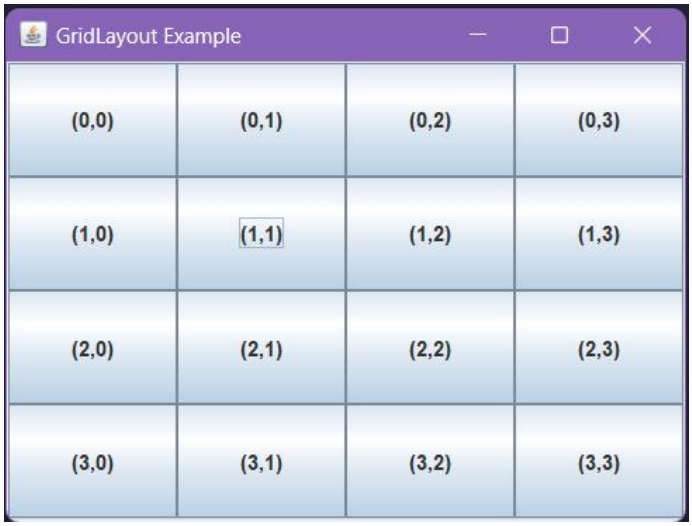
Components are added to the Layout by the add() method given by:

```
void add(Component compObj, Object region)
```

the regions are defined by:

- BorderLayout.**CENTER**
- BorderLayout.**SOUTH**
- BorderLayout.**EAST**
- BorderLayout.**WEST**
- BorderLayout.**NORTH**

GridLayout



GridLayout lays out components in a **two-dimensional** grid. We can define the number of **rows and column.** It defines three constructors:

```
GridLayout( )
GridLayout(int numRows, int numColumns)
GridLayolayoutut(int numRows, int numColumns, int horz, int vert)
```

- **numRows** and **numColumns** – creates a grid layout with **numRows*numColumns**, by default it is one so the default contructor creates a **single column grid**
- **horz** and **vert** – specifies the space horizontally and vertically between the components.

CardLayout

The **CardLayout** class is **unique** among the other layout managers in that it **stores several different layouts**. Each layout can be thought of as being on a **separate index card** in a deck that can be shuffled so that **any card is on top** at a given time. This can be useful for user interfaces with **optional components** that can be **dynamically enabled and disabled upon user input**. CardLayout provides two constructors:

```
CardLayout( )
CardLayout(int horz, int vert)
```

- **horz** and **vert** – specifies the space horizontally and vertically between the components.

The cards are typically held in an object of type **JPanel.** This panel must have **CardLayout selected as its manager**. The cards that form the deck are also typically objects of type **JPanel.** The cards are added to the deck using **add()** method.

```
void add(Component panelObj, Object name)
```

Once these steps are complete, we must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

After creating deck the cards are accessed and shown by the following methods

```
void first(Container deck)
```

```
void last(Container deck)
```

```
void next(Container deck)
```

```
void previous(Container deck)
```

```
void show(Container deck, String cardName)
```

GridBagLayout

The **GridBagLayout** is a powerful and flexible layout manager in Java Swing. We can specify the **relative placement** of components by **specifying their positions** within cells inside a grid using **GridBagLayout.** The key to the grid bag is that each component can be a different size, and each row number of columns in the grid can have a different, thus it is called a **Grid Bag.** It's a **collection of small grids** joined together. The **location and size of each component** in a grid bag are determined by a **set of constraints** linked to it.The general procedure for using this layout is

- create a **GridBagLayout** object with the same manager
- set constraints of each component and add them to the layout

There is only one defined constructor:

```
GridBagLayout( )
```

GridBagLayout defines several methods, of which many are protected and not for general use.

One of the methods are **setConstraints():**

```
void setConstraints(Component comp, GridBagConstraints cons)
```

GridBagConstraints is a subclass of GridBagLayout which is used to initialize constraints to the object and then pass it to **add()** or **setConstraints()** methodsIt provides several fields to govern the size, placement and spacing of container like

- **int** anchor
- **int** fill
- **int** gridheight
- **int** gridwidth
- **int** gridy
- **int** gridx
- **int** ipady
- **int** ipadx
- **double** wieghtx
- **double** weighty

GridBagConstraints also defines several static fields that contain standard constraint values, such as:

- GridBagConstraints.**CENTER**
- GridBagConstraints.**VERTICAL**

When a component is smaller than its cell, you can use the anchor field to specify where within the cell the **component's top-left** corner will be locate:

- GridBagConstraints.**CENTER**
- GridBagConstraints.**EAST**
- GridBagConstraints.**NORTH**
- GridBagConstraints.**NORTHEAST**
- GridBagConstraints.**NORTHWEST**
- GridBagConstraints.**SOUTH**
- GridBagConstraints.**SOUTHEAST**
- GridBagConstraints.**SOUTHWEST**
- GridBagConstraints.**WEST**

The second type of values that can be given to anchor is relative, which means the values are relative to the **container's orientation**:

- GridBagConstraints.**FIRST_LINE_END**
- GridBagConstraints.**FIRST_LINE_START**
- GridBagConstraints.**LAST_LINE_END**
- GridBagConstraints.**LAST_LINE_START**
- GridBagConstraints.**LINE_END**
- GridBagConstraints.**LINE_START**
- GridBagConstraints.**PAGE_END**
- GridBagConstraints.**PAGE_START**

Insets

Sometimes we may want to **leave a small amount of space** between the **container** that holds the **components** and the **window** that contains it.

To do this, **override** the **getInsets( )** method that is defined by Container. **getInsets( )** method returns an **Insets object** that contains the **top, bottom, left, and right inset** to be used when the **container is displayed**. These values are used by the **layout manager** to inset the components **when it lays out the window.**

```
Insets getInsets( )
```

The **constructor** for Insets is shown here:

```
Insets(int top, int left, int bottom, int right)
```

The values passed in top, left, bottom, and right specify the amount of space between the container and its enclosing window.

Swing Components

All of the components of Swing are derived from the **JComponent Class.** The components are all lightweight. Some of the swing components are:

- JButton
- JCheckBox
- JComboBox
- JLabel
- JList
- JRadioButton
- JScrollPane
- JTabbedPane
- Jtable
- JTextField
- JToggleButton
- JTree

JFrame

Every containment hierarchy must begin with a top-level container. **JFrame** is a top-level container that is commonly used for Swing applications. It doesn't inherit **JComponent.** But inherits the AWT classes Component and Container. **JFrame** is a heavy-weight component. The most commonly used constructor of **JFrame** is:

```
JFrame(String title)
```

This can create a container that defines a **rectangular window** complete with a **title bar, close, minimize, maximize, and restore buttons and a system menu**. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

The **setSize( )** method (which is inherited by **JFrame from the AWT class Component**) sets the **dimensions of the window**, which are specified in **pixels**:

```
void setSize(int width, int height)
```

If we want the **entire application to terminate** when its top level window is closed the easiest way is to call **setDefaultCloseOperation( )**

```
void setDefaultCloseOperation(int what)
```

- what is the operation to be performed given by integer constants:
    - JFrame**.EXIT_ON_CLOSE**
    - JFrame**.DISPOSE_ON_CLOSE**
    - JFrame**.HIDE_ON_CLOSE**
    - JFrame**.DO_NOTHING_ON_CLOSE**

The content pane can be obtained by calling **getContentPane( )** on a **JFrame instance**.

```
Container getContentPane( )
```

The **setVisible( )** method is inherited from the AWT Component class. It determines the visibility of the **JFrame,** which is by default false that sets it invisible. So the setVisible() method is used to set it to visible:

```
void setVisible(boolean aFlag)
```

- aFlag is to be set true to make the **JFrame** visible

JLabel

JLabel is the **easiest-to-use** component of Swing. It creates a Label that can **display text** and/or an **icon.** It is a **passive component** because it **does not respond** to user input.

It has several constructors of which following are the most commonly used:

```
JLabel(Icon icon)
```

```
JLabel(String str)
```

```
JLabel(String str, Icon icon, int align)
```

- **icon** is the object of **Icon** class that has to be displayed
- **str** is the text to be displayed
- **align** argument sets how the text and/or icon should aling within the label. The possible values for the argument are:
    - **LEFT**
    - **RIGHT**
    - **CENTER**
    - **LEADING**
    - **TRAILING**

The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** class **implements Icon** interface and encapsulates an image.

```
ImageIcon(String filename)
```

Where file name is the file path of icon.

The icon and text associated with the label can be obtained by the following methods:

```
Icon getIcon( )
```

```
String getText( )
```

The icon and text associated with a label can be set by these methods:

```
void setIcon(Icon icon)
```

```
void setText(String str)
```

Swing Buttons

Swing defines **four** types of buttons which are the subclass of **AbstractButton** class which extends **Jcomponent**. They are:

- JButton
- JToggleButton
- JCheckBox
- JRadioButton

AbstractButton contains many methods that allow you to control the **behavior of buttons**.

For example, there are different methods to set icon for different states of a button by the methods:

```
void setDisabledIcon(Icon di)
```

```
void setPressedIcon(Icon pi)
```

```
void setSelectedIcon(Icon si)
```

```
void setRolloverIcon(Icon ri)
```

We can get the text associated with a button using:

```
String getText( )
```

We can modify the text associated with a button using:

```
void setText(String str)
```

The model used by all buttons is defined by the **ButtonModel interface**.

JButton

The **JButton** class provides the functionality of a **push button**. **JButton** allows an **icon, a string, or both** to be associated with the **push button**.

There are three constructors for **JButton:**

```
JButton(Icon icon)
```

```
JButton(String str)
```

```
JButton(String str, Icon icon)
```

The buttons generates **ActionEvent** when it is pressed, thus it has to be registered with a **ActionListener** which implements the **actionPerformed()** method.

We can set the action command by calling **setActionCommand( )** on the button.

```
void setActionCommand(String command)
```

We can obtain the action command by calling the **getActionCommand():**

```
String getActionCommand()
```

The **action command** helps to identify the button. Thus, when using **two or more buttons within the same application**, the **action command** gives you an **easy way to determine which button was pressed**.

JToggleButton

A toggle button looks just like a push button, but has two states:

- **Pushed** - selected
- **Released** - deselected

When we press a toggle button, it **stays pressed**. It **does not pop back up** as a regular push button. When we press the **toggle button a second time, it releases.** Each time a toggle button is pushed, it **toggles between its two states. JToggleButton** is a **superclass** for **JCheckBox** and **JRadioButton.** JToggleButton defines several constructors of which is:

```
JToggleButton(String str)
```

This creates a toggle button that contains the text passed in String str. **By default**, the button is in the **off position**. **JToggleButton** uses a model defined by a **nested class** called **JToggleButton.ToggleButtonModel**. JToggleButton generates an ActionEvent when pressed or released.

To handle item events, we must implement the **ItemListener** interface. Each time an item event is generated, it is passed to the **itemStateChanged( )** method defined by **ItemListener**.

Inside **itemStateChanged( )**, the **getItem( )** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event.

```
Object getItem()
```

The easiest way to determine a toggle button's state is by calling the **isSelected( )** method.

```
boolean isSelected( )
```

JCheckBox

JCheckBox class provides the functionality of a **check box**. Its immediate superclass is **JToggleButton.** JCheckBox defines several constructors of which is:

```
JCheckBox(String str)
```

When the user, selects or deselects a check box, an **ItemEvent** is generated. Inside the **itemStateChanged( )** method, **getItem( )** is called on **ItemEvent** object to obtain a **reference** to the **JCheckBox** object that **generated the event.**To determine the selected state of a check box is to call **isSelected( )** on the **JCheckBox** instance.

JRadioButton

Radio buttons are a group of **mutually exclusive buttons**, in which **only one button can be selected at any one time**. They are supported by the **JRadioButton** class, which extends **JToggleButton**. JRadioButton defines several constructors of which is:

```
JRadioButton(String str)
```

Radio button are made into a group by adding it to a button group created using **ButtonGroup** class. So that only one of the button in a group is selected.

```
void add(AbstractButton ab)
```

A **JRadioButton** generates **action events**, **item events**, and **change.**

We normally implement the **ActionListener** interface with method **actionPerformed( )**. Inside this method we can check its **action command** by calling **getActionCommand( ).**

By default, the **action command** is the same as the **button label**, but we can set the action command to something else by calling **setActionCommand( )** on the radio button. We can call **getSource( )** on the **ActionEvent** object and check that reference against the buttons.

We can simply check each radio button to find out which one is currently selected by calling **isSelected( )** on each button.

JTextField

JTextField is the **simplest Swing text component**. It just allows you to edit **one line** of text. It is derived from **JTextComponent,** which provides the basic functionality common to **Swing text components.** Three of the TextField's constructors are:

```
JTextField(int cols)
```

```
JTextField(String str, int cols)
```

```
JTextField(String str)
```

- **str** is the thread to be initially presented in the text field, if no String is specified then it shows an empty text field and it would be initially empty
- **cols** is the number of columns of the text field, if not provided it takes the size to fit the initital text str.

JTextField generates events in response to user interaction.

- An **ActionEvent** is fired when the user presses **ENTER**.
- A **CaretEvent** is fired each time the **caret** (i.e., the cursor) **changes position**.

CaretEvent is packaged in **javax.swing.event.**
To obtain the text currently in the text field, call **getText( ).**

```
String getText()
```

JList

In Swing, the basic list class is called **JList**. **JList** provides several constructors of which is:

```
JList(Object[ ] Items)
```

A **JList** generates a **ListSelectionEvent** when the user **makes or changes a selection** or deselects an item. It is handled by implementing **ListSelectionListener.**
ListSelectionListener interface specifies only one method, called **valueChanged()**

```
void valueChanged(ListSelectionEvent le)
```

We can change this behavior by calling **setSelectionMode( )**,

```
void setSelectionMode(int mode)
```

here mode can be

- SINGLE_SELECTION
- SINGLE_INTERVAL_SELECTION
- MUTIPLE_INTERVAL_SELECTION

We can obtain the index of the item selected from list by calling **getSelectedIndex( )**

```
int getSelectedIndex( )
```

We can obtain the value associated with the selection by calling **getSelectedValue( )**

```
Object getSelectedValue( )
```

JComboBox

Swing provides a combo box **(a combination of a text field and a drop-down list)** through the **JComboBox** class. Its constructor is

```
JComboBox(Object[ ] items)
```

Items can also be dynamically added to the list of choices via the **addItem( )** method

```
void addItem(Object obj)
```

To obtain the item selected in the list is to call **getSelectedItem( )** on the combo box.

```
Object getSelectedItem( )
```

A demo program using all these components is provided here.

Java Database Connectivity

Introduction

Java is a Programming language that is meant for coding and developing interafaces. But it can't store persistent data in a arranged and centralized manner on itself. There comes the significance of Database.

A database is a structured collection of data that is organized in a way to be easily accessed, managed, and updated. It is designed to efficiently store and retrieve information, making it an essential component of many software applications and systems. Some of the commonly used database services are

- MySQL
- Oracle
- PostgreSQL

There are different types of databases like relational database, object-based database etc.

Relational Databases

A relational database is a type of database that organizes data into tables, where each table consists of rows and columns. The concept of a relational database is based on the principles of relational algebra and set theory. It provides a structured and efficient way to store, manage, and retrieve data. Some of the relational database services are Oracle, Microsoft Access, MySQL, PostgreSQL, MongoDB.

SQL – Structured Query Language

SQL, which stands for **Structured Query Language**, is a standardized programming language designed for **managing and manipulating relational databases**. SQL provides a set of commands for defining and manipulating the structure of a relational database, as well as for performing operations on the data stored within the database.

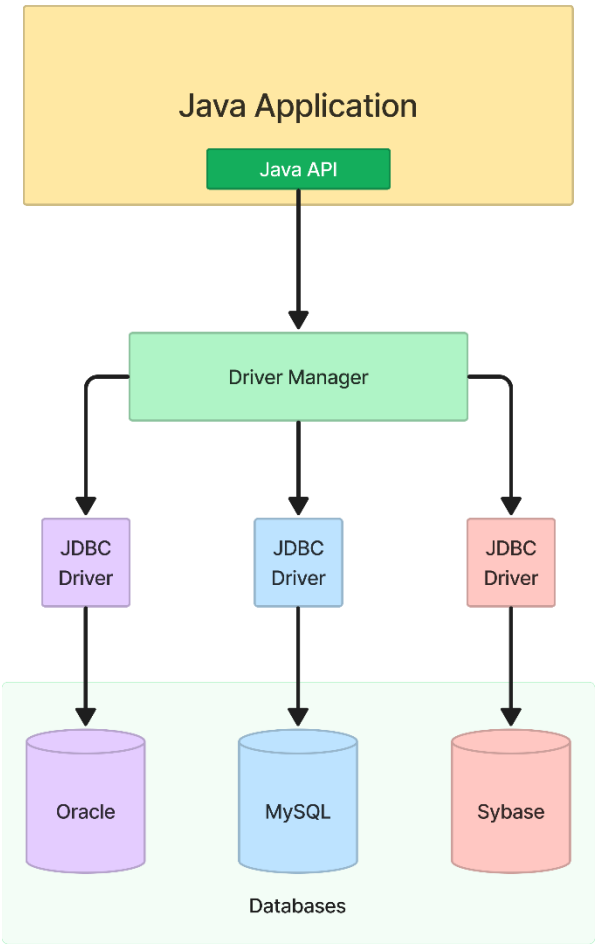
There are different types of query statements like

- **Data Query Language (DQL):** used to retrieve data – **SELECT**
- **Data Definition Language (DDL):** used to modify database – **CREATE , ALTER , DROP**
- **Data Manipulation Language (DML):** used to manipulate data in database – **INSERT, UPDATE, DELETE**
  And many more.

Java DataBase Connectivity (JDBC)

JDBC API (**A**pplication **P**rogramming **I**nterface) is a Java API that **can access any kind of tabular data**, especially data stored in a **Relational database. JDBC** is used for executing **SQL statements** from Java program. With the help of JDBC API, we can **insert, update, delete and fetch data** from the database.

Before JDBC, **ODBC** (**O**pen **D**ata**B**ase **C**onnectivity) API whose driver is written in **C language** was used to connect and execute queries with database. As its used **C language** it became platform dependent and insecure. This made Java define its own database connectivity API , **JDBC.** If our application is using **JDBC API** to interact with the **database**, then **we need not change much** in our code even if we **change the database** of our application. It standardizes many operations like:



- Connecting to database
- Querying with database
- Updating the database
- Calling stored procedures

JDBC architecture consist of two parts: **JDBC API, JDBC Drivers**

The **JDBC API** defines a **set of interfaces and classes** that all major database providers follow, so that using **JDBC API**, Java developers can connect to many **Relational Database Management Systems (RDBMS).** The **JDBC API** connects to the database using **JDBC Driver**, a software component that enables a Java application to interact with specific databases. **JDBC API** is comprise of two packages:

- java.sql
- javax.sql

Some of the classes and interfaces of **java.sql** package is:

- **Driver Manager** – Driver manager **class** manages drivers found in JDBC environment and load the most appropriate driver
- **Connection** - Connection **interface** objects which represents connection and it's object also helps in creating object of Statement, PreparedStatement etc.

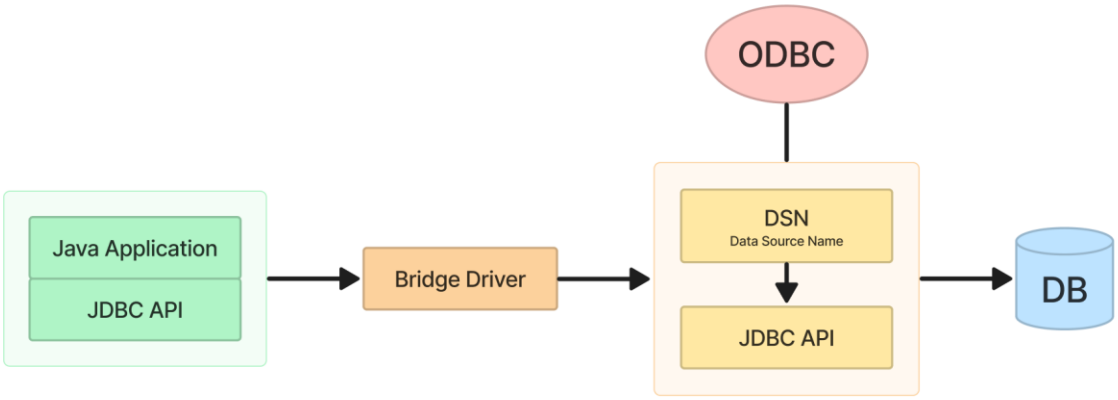
- **Statement** - Statement **interface** object is used to **execute query** and also store it's value to **ResultSet** object.
- **Prepared Statement** – represents a **precompiled SQL statement.**
- **Callable Statement** – supports stored procedure
- **Result Set** – used to store the result of the SQL query. The application gets the result from the ResultSet object
- **SQLException - SQLException** class is used to represent error or warning during access from database or during connectivity.

JDBC drivers contains a set of Java classes that enables to connect to that particular database. All major vendors provide their own JDBC drivers.

There are four types of JDBC Drivers:

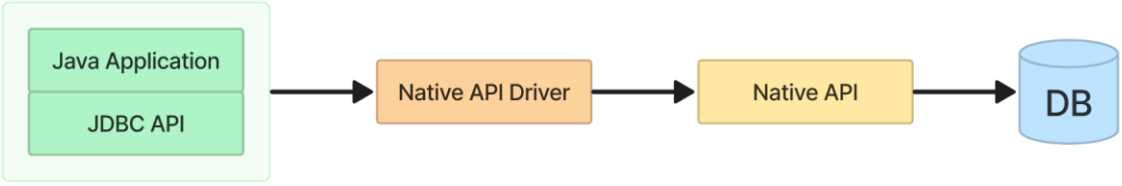
Type 1 JDBC Driver

The Type 1 **(JDBC-ODBC Bridge driver)** driver translates all **JDBC calls into ODBC** calls and sends them to the ODBC driver. The JDBC-ODBC Bridge allows us to access almost any databases, since they are already available.



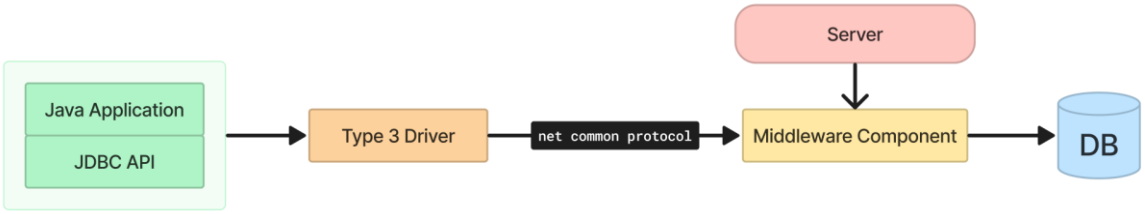
Type 2 JDBC Driver

Type 2 drivers **(Native-API/partly Java driver)** convert JDBC calls into database-specific calls.



Type 3 JDBC Driver

Type 3 database driver **(All Java/Net-protocol driver)** requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. As the driver is written fully in Java and is portable, it is suitable for web.

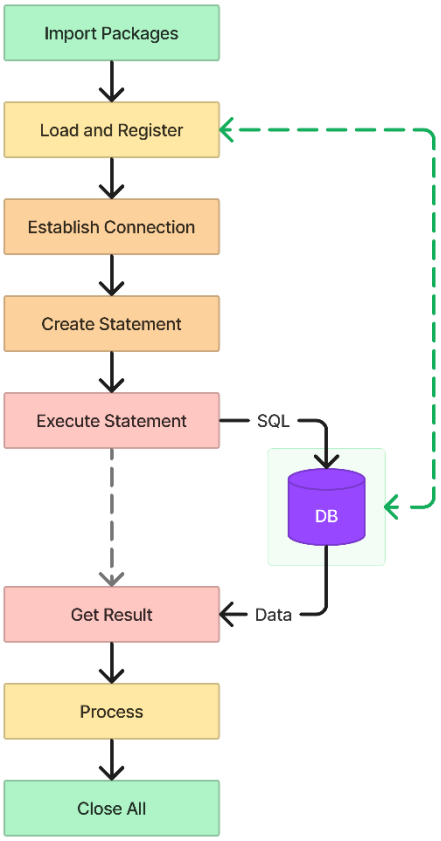


Type 4 JDBC Driver

Type 4 drivers **(Native-protocol/all-Java driver)** uses Java networking libraries to communicate directly with the database server. It is most suitable for web.

DataBase Connectivity Steps

1. **Import packages** – necessary packages like **java.sql, javax.sql** are to be imported
2. **Load and Register Driver** – proper database driver is to be loaded and registered
3. **Establish connection** – connection is made to the database
4. **Create statement** – SQL statement is created
5. **Execute Statement** – the created statement is to be executed
6. **Get the Result Set** – the result is obtained
7. **Process the result** – do necessary functions with the data
8. **Close all** – the Connection and Statement objects are to be closed



Load and Register Drivers

We can load the drivers in Java by two ways:

- Calling **Class.forName()** with the Driver class name as an argument
- Calling **DriverManager.registerDriver()** with constructor of the driver class as argument

Each database has its own drivers the commonly used ones of MySQL are given by the names:

- **com.mysql.jdbc.Driver** – constructor - **com.mysql.jdbc.Driver ()**
- **com.mysql.cj.jdbc.Driver** – constructor - **com.mysql.cj.jdbc.Driver ()**

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

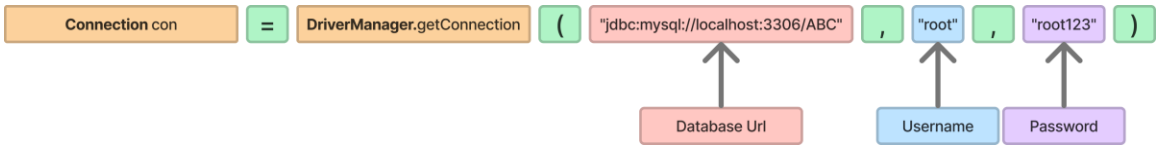
OR

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
```

Establish Connection

The **getConnection()** method of **DriverManager** class is used to establish connection with the database.

```
public static Connection getConnection(String url ,String name,
String password) throws SQLException
```



A **JDBC URL** provides a way of identifying a database so that the appropriate driver will recognize it and establish a connection with it. Its standart syntax is

jdbc:<subprotocol>:<subname>

- **jdbc** – is the protocol
- **subprotcol** – is usually the driver used for connectivity
- **subname** – is the database name

Creating a Statement

Once a connection is established, we can interact with the database. This is done by creating a **Statement object** by **createStatement()** method of the **Connection object**:

```
Statement object= connectionobject.createStatement();
```

Here it would be of the form

```
Statement statement= con.createStatement();
```

The Statement object is used to send and execute SQL statements

Executing Statements

We have two method to execute a statement given by Statement Class:

```
int executeUpdate(String sql)
```

```
ResultSet executeQuery(String sql)
```

Here both of them have different usages

executeUpdate() is used for statements that executes **CREATE,INSERT,DELETE and UPDATE.** It returns the number of rows affected by the statement as integer.

```
statement.executeUpdate("CREATE TABLE sample(Roll int, Name varchar(15))
```

executeQuery() is used for data retrieval statement **SELECT.** It returns an object of the ResultSet class which can be used to access the data.

```
ResultSet result =statement.executeQuery("SELECT Roll, Name from sample");
```

Processing Result

While we execute a select query it returns a ResultSet object which has a collection of each of the data entered in the relation in an ordered form.

Following are some methods to retrieve the values:

- **getInt()** – to retrieve integer value
- **getString()** – to retrieve string values
- **getFloat()** – to retrieve float value

thus the general form of the method is **getXXXX()** where XXXX is the data type

```
public int getXXXX(int columnIndex)
```

```
public int getXXXX(String columnLabel)
```

there is a method to move the cursor to next entry defined as:

```
boolean next() throws SQLException;
```

If there is another row in the result set, the next method **moves the cursor** to that row and returns **true**. If there are **no more rows**, it returns **false**.

Closing

The open connections and the Statement are to be closed at the end of the program using the **close()** method.

For reference of various statements in MySQL executed through Java visit this link.