

UNIT - VApplet, GUI Programming with Swings, Accessing Databases with JDBC**Applet:**

- Basics
- Architecture
- Applet Skeleton
- Requesting repainting
- Using the status window
- Passing parameters to applets

GUI Programming with Swings

- The origin and design philosophy of swing,
- Components and containers,
- Layout managers,
- Event handling, using a push button,
- Jtextfield, jlabel and image icon, the swing buttons, jtext field, jscrollpane, jlist, combobox, trees, jtable, An overview of jmenubar, jmenu and jmenuitem, creating a main menu, showmessagedialog, showconfirmdialog, showinputdialog, showoptiondialog, jdialog, create a modeless dialog.

Accessing Databases with JDBC:

- Types of Drivers
- JDBC Architecture
- JDBC classes and Interfaces
- Basic steps in developing JDBC applications
- Creating a new database and table with JDBC.

5.1 Introduction to Applet

Applets are small Java programs that are used in Internet computing. The Applets can be easily transported over the internet from one computer to another computer and can be run using "appletviewer" or java enabled "**Web Browser**".

An Applet, like any application program, can do many things for us. ***It can perform arithmetic operations, display graphics, animations, text, accept the user input and play interactive games.***

Applets are created in the following situations:

1. When we need something to be included dynamically in the web page.
2. When we require some flash output
3. When we want to create a program and make it available on the internet

5.2 Types of Applet

There are two types of Applets.

→ The first type is created is based on the **Applet** class of java.applet package. These applets use the **Abstract Window Toolkit(AWT)** for designing the graphical user interface.

→ The second type of type of the Applets are based on the **Swing** class **JApplet**. The swing Applets use the swing classes to create Graphical User Interface. The JApplet inherits the properties from the Applet, so all the features of the Applet are available in the JApplet.

5.3Applet Architecture

An applet is a window-based program. As such, its architecture is different from the console-based programs. The key concepts are as follow:

- **First, applets are event driven.** An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return.
- **Second, the user initiates interaction with an applet.** These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated

5.4 An Applet Skelton

Most of the applets override a set of methods of the Applet.

Four of these methods,

init()
start()
stop()
destroy()

apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use.

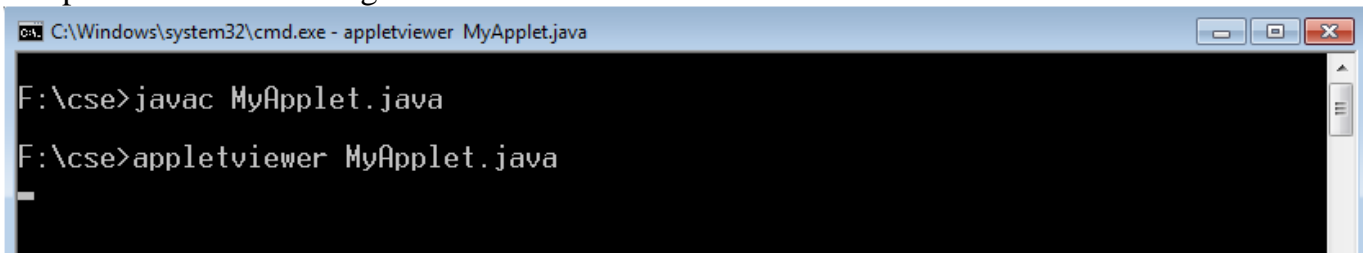
AWT-based applets will also override the **paint()** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed.

These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/* <applet code="AppletSkel" width=300 height=100>
</applet> */

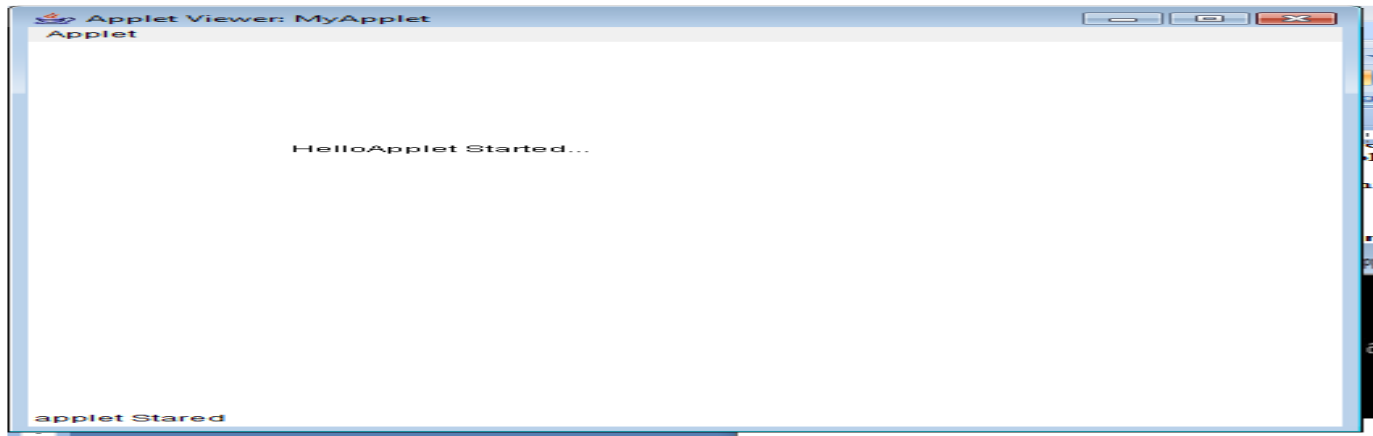
public class AppletSkel extends Applet
{
    // Called first.
    public void init()
    {
        // initialization
    }
    /* Called second, after init(). Also called whenever the applet is restarted. */
    public void start()
    {
        // start or resume execution
    }
    // Called when the applet is stopped.
    public void stop()
    {
        // suspends execution
    }
    /* Called when applet is terminated. This is the last method executed. */
    public void destroy()
    {
        // perform shutdown activities
    }
    // Called when an applet's window must be restored.
    public void paint(Graphics g)
    {
        // redisplay contents of window
    }
}
```

The procedure for Running



```
C:\Windows\system32\cmd.exe - appletviewer MyApplet.java

F:\cse>javac MyApplet.java
F:\cse>appletviewer MyApplet.java
```



Life Cycle of an Applet

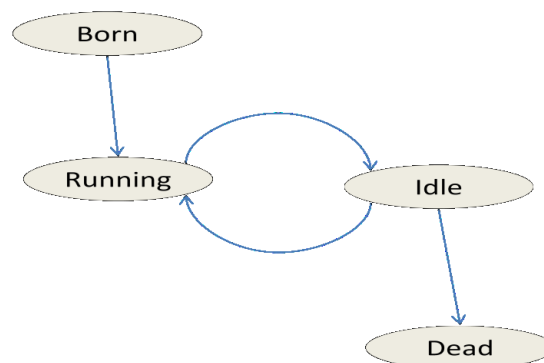
It is important to understand the order in which the various methods shown in the skeleton are called. **When an applet begins, the following methods are called, in this sequence:**

1. **init()**
2. **start()**
3. **paint()**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop()**
2. **destroy()**

Let's look more closely at these methods.



init()

The **init()** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start()

The **start()** method is called after **init()**. It is also called to restart an applet after it has been stopped. Whereas **init()** is called once—the first time an applet is loaded—**start()** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start()**.

paint()

The **paint()** method is called each time your applet's output must be redrawn. This situation can occur for several reasons.

For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.

paint() is also called when the applet begins execution.

Whatever the cause, whenever the applet must redraw its output, **paint()** is called.

The **paint()** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop()

The **stop()** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop()** is called, the applet is probably running. You should use **stop()** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start()** is called if the user returns to the page.

destroy()

The **destroy()** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop()** method is always called before **destroy()**.

5.5 Requesting the repaint() method

Whenever your applet needs to update the information displayed in its window, it simply calls **repaint()**.

The **repaint()** method is defined by the AWT that causes AWT run-time system to execute a call to your applet's **update()** method, which in turn calls **paint()**.

The AWT will then execute a call to **paint()** that will display the stored information. The **repaint()** method has four forms. The simplest version of **repaint()** is:

1. void repaint()

This causes the entire window to be repainted. Other versions that will cause repaint are:

2. void repaint(int left, int top, int width, int height)

If your system is slow or busy, **update()** might not be called immediately. If multiple calls have been made to AWT within a short period of time, then **update()** is not called very frequently. This can be a problem in many situations in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint()**:

3. void repaint(long maxDelay)

4. void repaint(long maxDelay, int x, int y, int width, int height)

Where "maxDelay" is the number milliseconds should be elapsed before **update()** method is called.

5.6 Using the Status Window

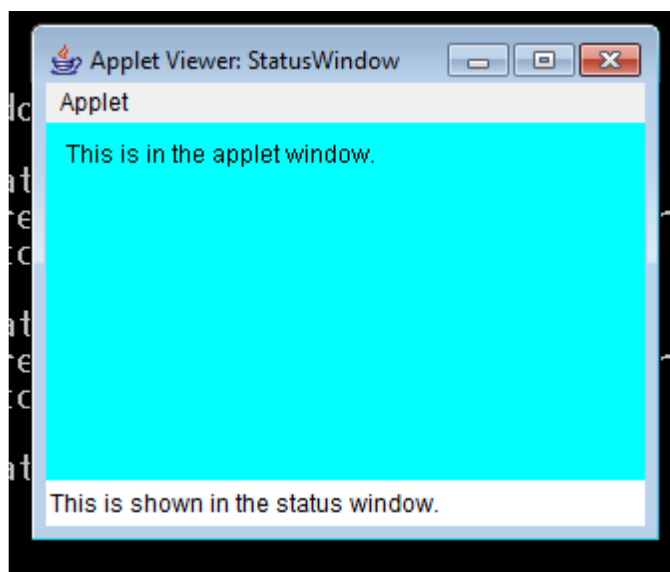
In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus()** with the string that you want displayed.

The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

```
import java.awt.*;
import java.applet.*;
/* <applet code="StatusWindow" width=300 height=50></applet> */
public class StatusWindow extends Applet
{
    public void init()
    {
        setBackground(Color.cyan);
    }
    public void paint(Graphics g)
    {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
    }
}
```

Running applet:

1. javac StatusWindow.java
2. appletviewer StatusWindow.java



Output:

5.7 Passing parameters to Applet

We can supply user defined parameters to an applet using the **<param>** Tag of HTML. Each **<param>** Tag has the **attributes** such as **"name"** , **"value"** to which actual values are assigned. Using this tag we can change the text to be displayed through applet.

We write the <param> Tag as follow:

```
<param name="name1" value="Hello Applet" > </param>
```

→Passing parameters to an Applet is something similar to passing parameters to **main()** method using command line arguments.

To set up and handle parameters, we need to do two things:

1. **Include appropriate <param> Tag in the HTML file**
2. **Provide the code in the applet to take these parameters.**

ParaPassing.java

```
import java.applet.*;
import java.awt.*;
public class ParaPassing extends Applet
{
    String str;
    public void init()
    {
        str=getParameter("name1");
        if(str==null)
            str="Java";
        str="Hello "+str;
    }
    public void paint(Graphics g)
    {
        g.drawString(str,50,50);
    }
}
```

para.html

```
<html>
<head> <title> Passing Parameters</title> </head>
<body bgcolor=pink >
<applet code="ParaPassing.class" width=400 height=400 >
<param name="name1" value="Example Applet for Passing the Parameters" >
</param>
</applet>
</body>
</html>
```

Running the Program:

1. Compile the "ParaPassing.java" using the "javac" command, which generates the "ParaPassing.class" file
2. Use "ParaPassing.class" file to code attribute of the <applet> Tag and save it as "para.html"

3. Give "para.html" as input to the "**appletviewer**" to see the output or open the file using the applet enabled web browser.

5.8 Getting the Input from the User

→ Applets work in the graphical environment. Therefore, applets treat inputs as text strings. we must first create an area of the screen in which user can type and edit input items.

→ We can do this by using the **TextField** class of the applet package. Once text fields are created, user can enter and edit the content.

→ Next step is to retrieve the contents from the text fields for display of calculations, if any. The text fields contain the item in the form of String. They need to be converted to the right form, before they are used in any computations.

Example Program:

```
import java.applet.*;
import java.awt.*;
public class InputApplet extends Applet
{
    TextField text1, text2;
    Label l1, l2;
    public void init()
    {
        text1 = new TextField(8);
        l1 = new Label("Enter First No");
        text2 = new TextField(8);
        l2 = new Label("Enter second No");
        add(l1);
        add(text1);
        add(l2);
        add(text2);
        text1.setText("0");
        text2.setText("0");
    }
    public void paint(Graphics g)
    {
        int x=0, y=0, z=0;
        String s1, s2, res;
        g.drawString("Enter a number in each text box", 50, 100);
        try
        {
            s1 = text1.getText();
            x = Integer.parseInt(s1);
            s2 = text2.getText();
            y = Integer.parseInt(s2);
        }
        catch (Exception e)
        {

```



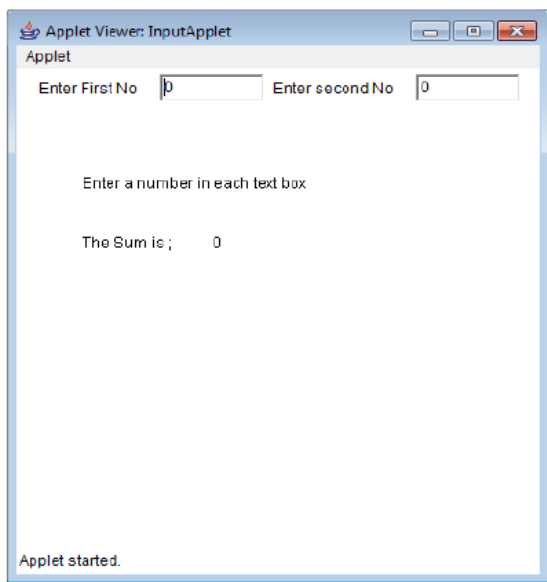
```
    }
    z=x+y;
    res=String.valueOf(z);
    g.drawString("The Sum is :",50,150);
    g.drawString(res,150,150);
}
public boolean action(Event e,Object obj)
{
    repaint();
    return true;
}
}
```

sum.html

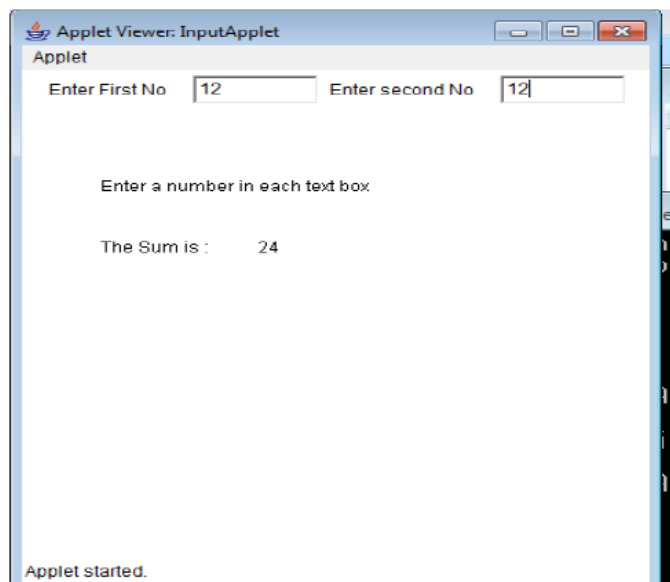
```
<html>
  <head>
    <title>Geting input from the User </title>
  </head>
  <body>
    <applet code="InputApplet.class" width=400 height=400 ></applet>
  </body>
</html>
```

Output:

Before Entering the Input



After Entering the input



Swings: The origin and design philosophy of swing, Components and containers, Layout managers, Event handling, Using a push button, jtextfield, jlabel and image icon, The swing buttons, Trees, An overview of jmenubar, jmenu and jmenuitem, Creating a main menu, Add mnemonics and accelerators to Menu items, showmessagedialog, showconfirmdialog, showinputdialog, showoptiondialog, jdialog, Create a modeless dialog.

6.1 The origin and design philosophy of swing

→ Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the **Abstract Window Toolkit**.

→ The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.

→ The use of native peers (platform-specific equivalents) led to several problems.

→ First, because of variations between operating systems, a component might look differently on different platforms. This potential variability threatened the overarching philosophy of **java: write once, run anywhere**.

→ Second, the **look and feel** of each component was fixed (because it is defined by the platform) and could not be (easily) changed.

→ Third, the use of heavyweight components caused some frustrating restrictions.

→ The solution to the limitations and restrictions present in the AWT was Swing.

→ **Swing is built on the foundation of the AWT**. This is why the AWT is still a crucial part of Java.

→ Swing also uses the same event handling mechanism as the AWT.

→ Swing addresses the limitations associated with AWT's components through two key features:

→ **lightweight components and a pluggable look and feel.**

→ With very few exceptions, **Swing components are lightweight**.

→ This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible.

The MVC Connection

→ In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

→ Swing uses a modified version of a classic component architecture called: **Model-View-Controller**, (MVC).

→ In MVC terminology, the **model** corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.

→ The **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.

→ The **controller** determines how the component reacts to the user.

For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated.

→ Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the **UI delegate**. For this reason, Swing's approach is called either the **Model-Delegate** architecture or the **Separable Model** architecture.

Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

	Java AWT	Java Swing
1)	AWT components are platform -dependent .	Java swing components are platform-independent .
2)	AWT components are heavyweight .	Swing components are lightweight .
3)	AWT doesn't support pluggable look and feel .	Swing supports pluggable look and feel .
4)	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedPane etc.
5)	AWT doesn't follows MVC (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC

6.2 Components and Containers

→ A Swing GUI consists of two key items: **components and containers** . However, this distinction is mostly conceptual because all containers are also components.

→ The difference between the two is found in their intended purpose:

→ A *component* is an independent visual control, such as a push button or slider.

→ A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components.

→ Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy* , at the top of which must be a *top-level container* .

6.2.1 Components

→ In general, Swing components are derived from the **JComponent** class.

→ **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes **Container** and **Component**. Thus, a Swing component is built on and compatible with an AWT component.

→ All of Swing's components are represented by classes defined within the package **javax.swing** . The following table shows the class names for Swing components (including those used as containers).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
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	JWindow		

6.2.2 Containers

→ Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**.

→ **The top-level containers are heavyweight.**

→ A top-level container is not contained within any other container.

→ Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is **JFrame**. The one used for applets is **JApplet**.

→ **The second type of containers supported by Swing are lightweight containers.**

→ Lightweight containers *do* inherit **JComponent**.

An example of a lightweight container is **JPanel**, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.

6.2.3 The 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 the glass pane, the content pane, and the layered pane**.

→ The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of **JPanel**. The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component, for example.

6.3 LAYOUT MANAGERS:

→ The layout manager controls the position of components within a container.

→ Java offers several layout managers. Many are provided by the AWT and Swing adds some of its own.

→ All layout managers are instances of a class that implements the **LayoutManager** interface.

FlowLayout	A simple layout that positions components left-to-right, top-to-bottom.
BorderLayout	Positions components within the center or the borders of the container. This is the default layout for a content pane.

GridLayout	Lays out components within a grid
GridBagLayout	Lays out different size components within a flexible grid
BoxLayout	Lays out components vertically or horizontally within a box
SpringLayout	Lays out components subject to a set of constraints.

A FIRST SIMPLE SWING PROGRAM:

→The following program shows one way to write a Swing application. In the process, it demonstrates several key features of Swing.

→It uses two Swing components: **JFrame** and **JLabel**. **JFrame** is the top-level container that is commonly used for Swing applications .

→**JLabel** is the Swing component that creates a label, which is a component that displays information.

→The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output.

→The program uses a **JFrame** container to hold an instance of a **JLabel**. The label displays a short text message.

```
import javax.swing.*;
class SwingDemo extends JFrame
{
    SwingDemo()
    {
        setSize(275,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel jlab = new JLabel(" Swing defines the modern J ava GUI.");
        add(jlab);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        SwingDemo sd=new SwingDemo();
    }
}
```

Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

javac SwingDemo.java

To run the program, use this command line:

java SwingDemo

When the program is run, it will produce the window shown in Figure



6.4 Event Handling

Types of Event Handling Mechanisms

There are two types of Event handling mechanisms supported by Java.

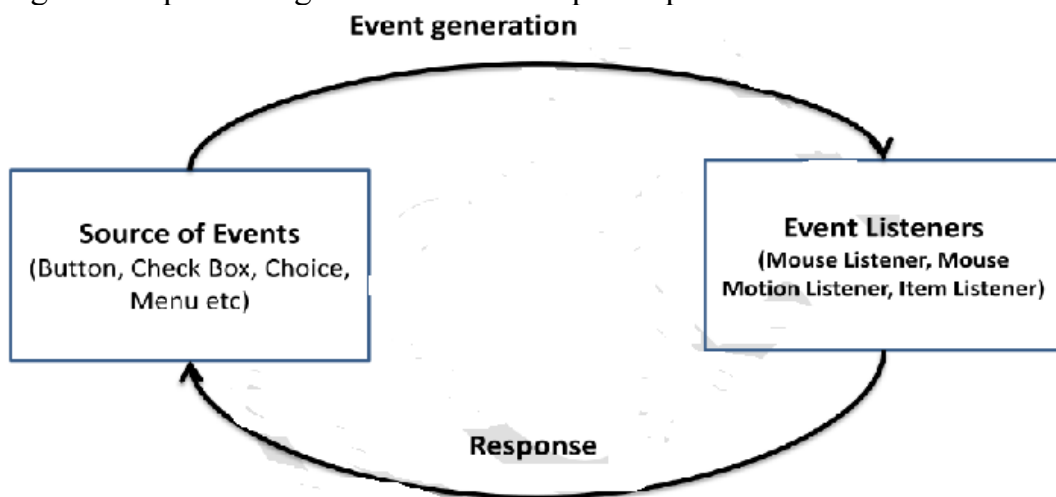
→First, the approach supported by Java 1.0, where the generated event is given hierarchically to the objects until it was handled. This can be also called as ***Hierarchical Event Handling Model***.

→Second, the approach supported by Java 1.1, which registers the listeners to the source of the event and the registered listener processes the event and returns response to the source. This is called "***Delegation Event Model***"

6.4.1 The Delegation Event Model

→The modern approach to handling events is based on the ***delegation event model***, which defines standard and consistent mechanisms to generate and process ***events (1)***. Its concept is quite simple: a ***source (2)*** generates an event and sends it to one or more ***listeners (3)***. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns response.

The ***advantage*** of this design is that the ***application logic*** that processes events is cleanly separated from the user ***interface logic*** that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.



In the delegation event model, listeners must ***register*** with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

1. What is an Event?

In the delegation model, an ***event*** is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

Examples: Pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse etc..

2. Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void addTypeListener(TypeListener el)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener()**. The method that registers a mouse motion listener is called **addMouseMotionListener()**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as **multicasting** the event. In all cases, notifications are sent only to listeners that register to receive them.

3. Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements.

→ First, it must have been registered with one or more sources to receive notifications about specific types of events.

→ Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved.

→ The package **java.awt.event** defines many types of events that are generated by various user interface elements.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MouseEventsdemo extends JFrame implements
MouseListener, MouseMotionListener
{
    JLabel jl = new JLabel("mouse operation");
    String msg = "";
```

```

MouseEventsdemo()
{
    setVisible(true);
    setSize(400,400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle("MOuse events");
    addMouseListener(this);
    addMouseMotionListener(this);
    add(jl);
}
public void mouseClicked(MouseEvent e)
{
    jl.setText("Mouse is clicked");
}
public void mouseEntered(MouseEvent me)
{
    jl.setText("Mouse is Entered");
}
public void mouseExited(MouseEvent me)
{
    jl.setText("Mouse is Exited");
}
public void mousePressed(MouseEvent me)
{
    jl.setText("Mouse is Pressed");
}
public void mouseReleased(MouseEvent me)
{
    jl.setText("Mouse is Released");
}
public void mouseDragged(MouseEvent me)
{
    jl.setText("Mouse is Dragged");
}
public void mouseMoved(MouseEvent me)
{
    jl.setText("Mouse is moved");
}
public static void main(String args[])
{
    MouseEventsdemo me=new MouseEventsdemo();
}
}
// Action LISTENER
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Ballcolor extends JFrame implements ActionListener
{
    JButton b1,b2,b3;
    Ballcolor()
    {
        setSize(300,300);
        setLayout(null);
        setVisible(true);
        setLayout(new FlowLayout());
    }
}

```



```

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        b1=new JButton("Red");
        b2=new JButton("Green");
        b3=new JButton("Yellow");
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        add(b1);
        add(b2);
        add(b3);
    }
    public void actionPerformed(ActionEvent e)
    {
        Graphics g=getGraphics();
        if(e.getSource()==b1)
        {
            g.setColor(Color.RED);
            g.fillOval(100,200,100,100);
        }
        if(e.getSource()==b2)
        {
            g.setColor(Color.GREEN);
            g.fillOval(100,200,100,100);
        }
        if(e.getSource()==b3)
        {
            g.setColor(Color.YELLOW);
            g.fillOval(100,200,100,100);
        }
    }
    public static void main(String[] args)
    {
        new Ballcolor();
    }
}

```

6.5 USING A PUSH BUTTON

→A push button is an instance of **JButton**. **JButton** inherits the abstract class **AbstractButton**, which defines the functionality common to all buttons.

→**JButton** allows an icon, a string, or both to be associated with the push button.

Three of its constructors are shown here:

`JButton(Icon icon)`

`JButton(String str)`

`JButton(String str , Icon icon)`

Here, *str* and *icon* are the string and icon used for the button.

→When the button is pressed, an **ActionEvent** is generated. Thus **JButton** provides the following methods which are used to add or remove an action listener

void addActionListener(ActionListener al) void removeActionListener(ActionListener al)

→ ActionListener interface defines only one method: actionPerformed().

void actionPerformed(ActionEvent ae)

→ This method is called when a button is pressed.

→ Using the **ActionEvent** object passed to the **actionPerformed()** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button.

→ The action command is obtained by calling **setActionCommand()** on the button. **String getActionCommand()**

→ The action command identifies 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.

```
// Demonstrate a button.import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class ButtonDemo extends Frame implements ActionListener
{
    JLabel jlab;
    ButtonDemo()
    {
        setLayout(new FlowLayout());
        setSize(220, 90);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JButton jbtnFirst = new JButton("First");
        JButton jbtnSecond = new JButton("Second");
        addActionListener( this );
        addActionListener( this );
        add(jbtnFirst);
        add(jbtnSecond);
        jlab = new JLabel("Press a button.");
        add(jlab);
        setVisible( true );
    }
}
```

```
public void actionPerformed(ActionEvent ae)
{
    if(ae.getActionCommand().equals("First"))
        jlab.setText("First button was pressed.");
    else
        jlab.setText("Second button was pressed. ");
}
public static void main(String[] args)
{
    ButtonDemo b = new ButtonDemo();
}
}
```

JTextField:

JTextField is the simplest Swing text component. It is also probably its most widely used text component. **JTextField** 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 **JTextField**'s constructors are shown here:

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

Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field.

JTextField generates events in response to user interaction. For example, 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()**.

JLabel and ImageIcon

JLabel is Swing's easiest-to-use component.

JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors.

Here are three of them:

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

→ Here, *str* and *icon* are the text and icon used for the label. The *align* argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: **LEFT**, **RIGHT**, **CENTER**, **LEADING**, or **TRAILING**.

→ The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL.

→ Here is the **ImageIcon** constructor used by the example in this section: **ImageIcon(String filename)**

→ It obtains the image in the file named *filename*.

THE SWING BUTTONS

→ Swing defines four types of buttons:

```
JButton,
JToggleButton
JCheckBox
JRadioButton .
```

→ All are subclasses of the **AbstractButton** class, which extends **JComponent**.

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

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. Three of its constructors are shown here:

```

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

```

Here, *str* and *icon* are the string and icon used for the button.

→ When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed()** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling **setActionCommand()** on the button. You can obtain the action command by calling **getActionCommand()** on the event object. It is declared like this:

```
String getActionCommand()
```

- ✓ The action command identifies 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 useful variation on the push button is called a *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

→ Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**.

- ✓ **JToggleButton** defines several constructors.

```
JToggleButton(String str)
```

- ✓ This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position.
- ✓ **JToggleButton** also generates an item event. This event is used by those components that support the concept of selection. When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected.
- ✓ 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. It is shown here:

```
Object getItem()
```

The easiest way to determine a toggle button's state is by calling the **isSelected()** method (inherited from **AbstractButton**) on the button that generated the event. It returns **true** if the button is selected and **false** otherwise.

Check Boxes

- ✓ The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. **JCheckBox** defines several constructors.

```
JCheckBox(String str)
```

- ✓ It creates a check box that has the text specified by *str* as a label.

- ✓ When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem()** on the **ItemEvent** passed to the **itemStateChanged()** method defined by **ItemListener**. The easiest way to determine the selected state of a check box is to call **isSelected()** on the **JCheckBox** instance.

Radio Buttons

- ✓ 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** provides several constructors.
`JRadioButton(String str)`
- ✓ Here, *str* is the label for the button.
- ✓ In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:
`void add(AbstractButton ab)`
- ✓ A **JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the **ActionListener** interface.

JList

- ✓ In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list.
- ✓ **JList** was made generic and is declared like this:
`class JList<E>`
Here, **E** represents the type of the items in the list.
- ✓ **JList** provides several constructors. The one used here is `JList(E[] items)`
This creates a **JList** that contains the items in the array specified by *items*.
- ✓ **JList** is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved. The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.
- ✓ A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged()**, which is shown here:
`void valueChanged(ListSelectionEvent le)`
- ✓ By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode()**, which is defined by **JList**.
It is shown here:
`void setSelectionMode(int mode)`
- ✓ Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel**:

ADD MNEMONICS AND ACCELERATORS TO MENU ITEMS

→ In real applications a menu usually includes support for keyboard shortcuts. These come in two forms: mnemonics and accelerators.

→ A mnemonic defines a key that lets you select an item from an active menu by typing a key.

→ An accelerator is a key that lets you select an item without having to activate the menu first.

→ A mnemonic can be specified for both JMenuItem and JMenu objects.

There are two ways to set the mnemonic for JMenuItem.

→ An accelerator can be associated with a JMenuItem object. It is specified by calling `setAccelerator()`

```
void setAccelerator(KeyStroke ks)
```

→ KeyStroke is a class that contains several factory methods that construct various types of keystroke accelerators.

```
static KeyStroke getKeyStroke(int ch, int modifier)
```

JOption Pane:

- ✓ Swing provides extensive built-in support for dialogs through the JOptionPane class.
- ✓ JOptionPane is an easy-to-use dialog class that offers solutions to many common dialog-based problems. JOptionPane supports the four basic types of dialogs:



Message



Confirmation



Input



Option

- ✓ A message dialog displays a message and then waits until the user presses the OK button. This dialog provides an easy and effective way to ensure that the user is aware of some piece of information.
- ✓ A confirmation dialog asks the user a question that typically has a Yes/No answer, and then waits for a response.
- ✓ An input dialog allows the user to enter a string or select an item from a list.
- ✓ An option dialog lets you specify a list of options from which the user will choose.
- ✓ All dialogs created by JOptionPane are modal. A modal dialog demands a response before the program will continue. As a result you cannot refocus input to another part of the application without first closing the dialog. Thus a modal dialog stops the program until the user responds.
- ✓ A modeless (also called nonmodal) dialog does not prevent other parts of the program from being used. Thus, the rest of the program remains active and input can be refocused to other windows.

→ JOptionPane defines the following four factory methods that create standard.

→ JDialog based dialogs: **showConfirmDialog()**, **showInputDialog()**, **showMessageDialog()**, and **showOptionDialog()**.

showMessageDialog ()

→ The **showMessageDialog ()** method creates the simplest dialog that can be constructed. `static void showMessageDialog(Component parent, Object msg)` throws `HeadlessException`

→ Here parent specifies the component relative to which the dialog is displayed. If this argument is null, the dialog is usually displayed in the center of the screen. msg- message to be displayed.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MyFrame2 extends JFrame implements ActionListener
{
    JRadioButton button1,button2;
    JButton button;
    public MyFrame2()
    {
        setVisible(true);
        setTitle("MyFrame");
        setSize(300, 300);
        setLayout(new FlowLayout());
        button1 = new JRadioButton("Male");
        button2 = new JRadioButton("Female");
        button = new JButton("click");
        ButtonGroup group = new ButtonGroup();
        group.add( button1);
        group.add( button2);
        add(button1); add(button2); add(button);
        button.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        if(button1.isSelected())
        {
            JOptionPane.showMessageDialog(this, "U r selected Male option thankyou");
        }
        if(button2.isSelected())
        {
            JOptionPane.showMessageDialog(this, "U r selected Female option thankyou");
        }
    }
    public static void main(String[] args)
    {
        MyFrame2 frame2 = new MyFrame2();
    }
}

```

Accessing Databases with JDBC:

Types of Drivers, JDBC Architecture, JDBC classes and Interfaces, Basic steps in developing JDBC applications, Creating a new database and table with JDBC.

What is JDBC Driver?

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The *Java.sql* package that ships with JDK, contains various classes with their behaviours defined and their actual implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

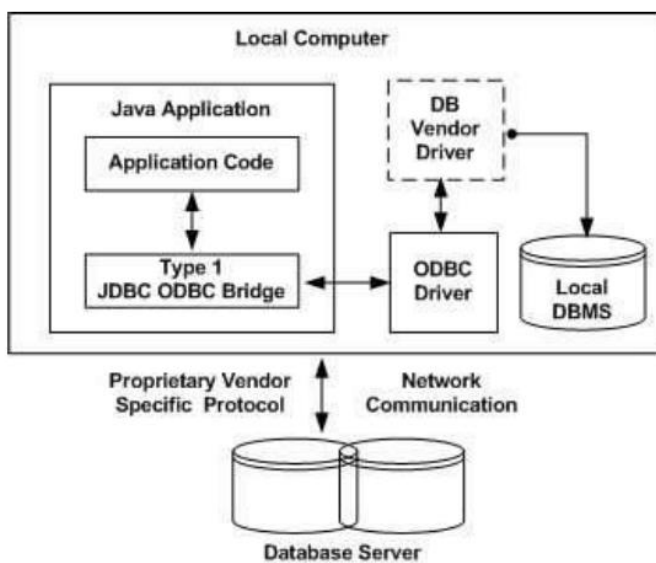
JDBC Drivers Types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below –

Type 1: JDBC-ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

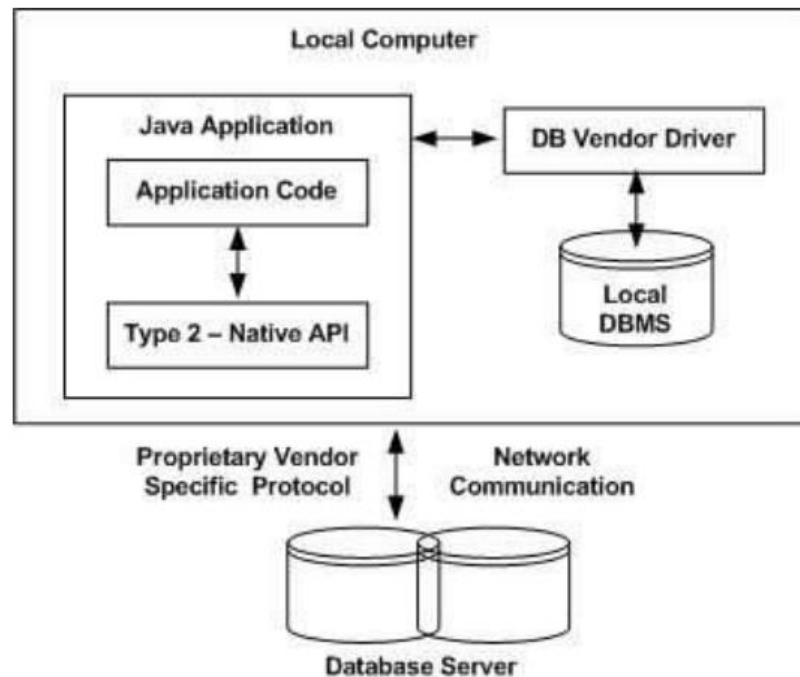


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.



The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

MySQL's Connector/J driver is a **Type 4 driver**. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

Example to connect to the mysql database in java

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySQL as the database. So we need to know following informations

for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

1. create database sonoo;
2. use sonoo;
3. create table emp(id **int**(10),name varchar(40),age **int**(3));

Example to Connect Java Application with mysql database

In this example, sonoo is the database name, root is the username and password.

```
import java.sql.*;
class MysqlCon
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection con=DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/sonoo","root","root");
            //here sonoo is database name, root is username and password
            Statement stmt=con.createStatement();
            ResultSet rs=stmt.executeQuery("select * from emp");
            while(rs.next())
                System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
            con.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

The above example will fetch all the records of emp table.

To connect java application with the mysql database mysqlconnector.jar file is required to be loaded.

Two ways to load the jar file:

1. paste the mysqlconnector.jar file in jre/lib/ext folder
2. set classpath

1) paste the mysqlconnector.jar file in JRE/lib/ext folder:

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

1. temporary 2. permanent

How to set the temporary classpath

open command prompt and write:

1. C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.

How to set the permanent classpath

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;. as C:\folder\mysql-connector-java-5.0.8-bin.jar;

JDBC-Result Sets

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories –

- ☐ **Navigational methods:** Used to move the cursor around.
- ☐ **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- ☐ **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet –

- ☐ **createStatement(int RSType, int RSConcurrency);**
- ☐ **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- ☐ **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency Description

ResultSet.CONCUR_READ_ONLY Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE Creates an updateable result set.

Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions

- ☐ One that takes in a column name.
- ☐ One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –

S.N. Methods & Description

1 public int getInt(String columnName) throws SQLException

Returns the int in the current row in the column named columnName.

2 public int getInt(int columnIndex) throws SQLException

Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL. There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

For a better understanding, let us study Viewing - Example Code.

Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type –

- ☐ One that takes in a column name.
- ☐ One that takes in a column index.