Unit – V :

Layout Manager: Border, Grid, Flow, Card and Gridbag.

Applets: Concepts of Applets, life cycle of an applet, creating applets, passing parameters toapplets.

JDBC Connectivity: JDBC Type 1 to 4 Drivers, connection establishment, QueryExecution.

A Layout Manager in Java is an interface that defines how components are arranged within a container, such as a Frame or Panel. It determines the size and position of components, facilitating the creation of visually appealing and organized graphical user interfaces (GUIs). Java provides several layout managers to suit various design needs.

Types of Layout Managers

1. FlowLayout

FlowLayout arranges components in a row, left to right, wrapping to the next line as needed. It is ideal for scenarios where components need to maintain their natural sizes and maintain a flow-like structure.

2. BorderLayout

BorderLayout divides the container into five regions: NORTH, SOUTH, EAST, WEST, and CENTER. Components can be added to these regions, and they will occupy the available space accordingly. This layout manager is suitable for creating interfaces with distinct sections, such as a title bar, content area, and status bar.

3. GridLayout

GridLayout arranges components in a grid with a specified number of rows and columns. Each cell in the grid can hold a component. This layout manager is ideal for creating a uniform grid of components, such as a calculator or a game board

4. CardLayout

CardLayout allows components to be stacked on top of each other, like a deck of cards. Only one component is visible at a time, and you can switch between components using methods like `next()` and `previous()`. This layout is useful for creating wizards or multi-step processes

5. GridBagLayout

GridBagLayout is a powerful layout manager that allows you to create complex layouts by specifying constraints for each component. It arranges components in a grid, but unlike GridLayout, it allows components to span multiple rows and columns and have varying sizes

JAVA FLOWLAYOUT

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.

Fields of FlowLayout class

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

6. Constructors of FlowLayout class

FlowLayout(): creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.

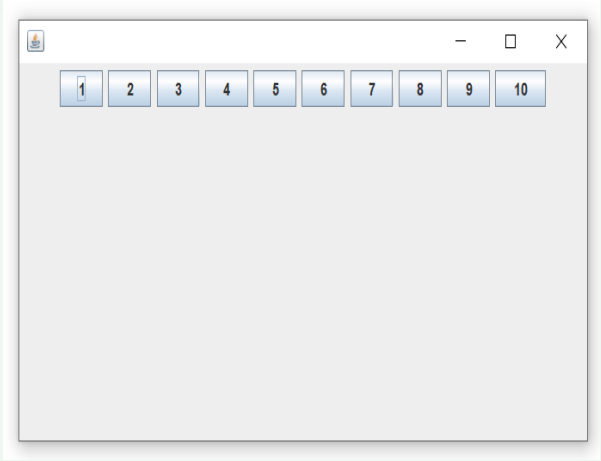
FlowLayout(int align): creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.

FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignment and the given horizontal and vertical gap.

Write a java program to add awt components to the frame using flow layout

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

public class FlowLayoutExample extends Frame
{

Frame frameObj;

// constructor
FlowLayoutExample()
```

```
{
    // creating a frame object
    frameObj = new Frame();

     // creating the buttons
    Button b1 = new Button("1");
    Button b2 = new Button("2");
    Button b3 = new Button("3");
    Button b4 = new Button("4");
    Button b5 = new Button("5");
    Button b6 = new Button("6");
    Button b7 = new Button("7");
    Button b8 = new Button("8");
    Button b9 = new Button("9");
    Button b10 = new Button("10");


    // adding the buttons to frame
    frameObj.add(b1);
    frameObj.add(b2);
     frameObj.add(b3);
     frameObj.add(b4);
    frameObj.add(b5);
     frameObj.add(b6);
     frameObj.add(b7);
     frameObj.add(b8);
     frameObj.add(b9);
     frameObj.add(b10);

    // parameter less constructor is used
    // therefore, alignment is center
    // horizontal as well as the vertical gap is 5 units.
    frameObj.setLayout(new FlowLayout());

    frameObj.setSize(300, 300);
    frameObj.setVisible(true);
}

// main method
public static void main(String argvs[])
{
    new FlowLayoutExample();
}
}
```



JAVA BORDERLAYOUT

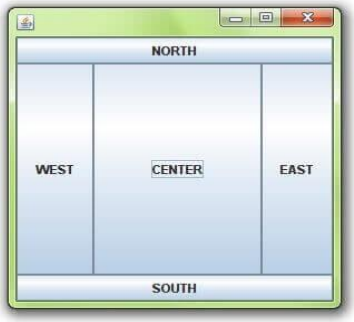
The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

1. public static final int NORTH
2. public static final int SOUTH
3. public static final int EAST
4. public static final int WEST
5. public static final int CENTER

Constructors of BorderLayout class:

BorderLayout(): creates a border layout but with no gaps between the components.
BorderLayout(int hgap, int vgap): creates a border layout with the given horizontal and vertical gaps between the components.

Write a java program to add awt components to the frame using border layout




```
import java.awt.*;

public class Border extends Frame
{
Frame f;
Border()
{
f = new Frame();

// creating buttons
Button b1 = new Button("NORTH");; // the button will be labeled as NORTH
Button b2 = new Button("SOUTH");; // the button will be labeled as SOUTH
Button b3 = new Button("EAST");; // the button will be labeled as EAST
Button b4 = new Button("WEST");; // the button will be labeled as WEST
Button b5 = new Button("CENTER");; // the button will be labeled as CENTER
f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction
f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Direction
f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direction
f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Direction
f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center
f.setSize(300, 300);
f.setVisible(true);
}
public static void main(String[] args) {
new Border();
}  }
```

JAVA GRIDLAYOUT

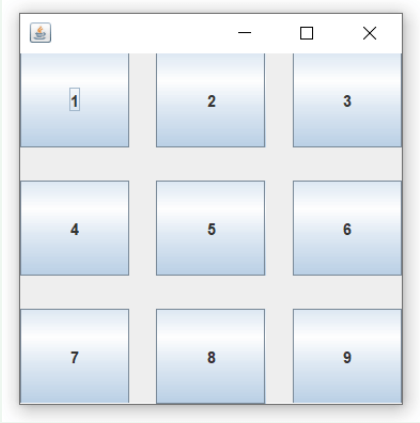
The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

Write a java program to add awt components to the frame using Grid layout

```
import java.awt.*;
public class GridLayoutExample1 extends Frame
{

Frame frameObj;

// constructor
GridLayoutExample1()
{
frameObj = new Frame();

// creating 9 buttons
Button btn1 = new Button("1");
Button btn2 = new Button("2");
Button btn3 = new Button("3");
Button btn4 = new Button("4");
Button btn5 = new Button("5");
Button btn6 = new Button("6");
Button btn7 = new Button("7");
Button btn8 = new Button("8");
Button btn9 = new Button("9");

// adding buttons to the frame
// since, we are using the parameterless constructor, therefore;
// the number of columns is equal to the number of buttons we
// are adding to the frame. The row count remains one.

frameObj.add(btn1); frameObj.add(btn2); frameObj.add(btn3);
```



```
frameObj.add(btn4); frameObj.add(btn5); frameObj.add(btn6);
frameObj.add(btn7); frameObj.add(btn8); frameObj.add(btn9);

// setting the grid layout
// a 3 * 3 grid is created with the horizontal gap 20
// and vertical gap 25

frameObj.setLayout(new GridLayout(3, 3, 20, 25));
frameObj.setSize(300, 300);
frameObj.setVisible(true);
}
// main method
public static void main(String argvs[])
{
new GridLayoutExample();
}
}
```

JAVA CARDLAYOUT

The **Java CardLayout** class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout Class

1. **CardLayout():** creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

Commonly Used Methods of CardLayout Class

- **public void next(Container parent):** is used to flip to the next card of the given container.
- **public void previous(Container parent):** is used to flip to the previous card of the given container.
- **public void first(Container parent):** is used to flip to the first card of the given container.
- **public void last(Container parent):** is used to flip to the last card of the given container.
- **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

Write a java program to add awt components to the frame using Card layout

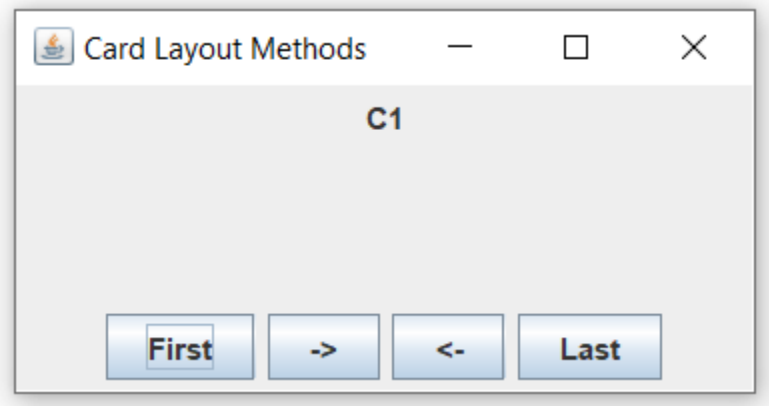
```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class CardLayoutExample3 extends Frame
{

// Initializing the value of currCard to 1 .
private int currCard = 1;

// Declaring of objects of the CardLayout class.
private CardLayout cObjl;

// constructor of the class
public CardLayoutExample3()
{

// Method to set the Title of the Frame
setTitle("Card Layout Methods");

// Method to set the visibility of the Frame
setSize(310, 160);

// Creating an Object of the "panel" class
Panel cPanel = new Panel();

// Initializing of the object "cObjl" of the CardLayout class.
cObjl = new CardLayout();

// setting the layout
cPanel.setLayout(cObjl);

// Initializing the object  Panel1" of the Panel class.
Panel Panel1 = new Panel();

// Initializing the object  Panel2 of the CardLayout class.
Panel Panel2 = new Panel();

// Initializing the object  "Panel3" of the CardLayout class.
Panel jPanel3 = new Panel();
```

```
// Initializing the object
// "Panel4" of the CardLayout class.
Panel Panel4 = new Panel();


// Initializing the object
// "l1" of the Label class.
Label Label1 = new Label("C1");

// Initializing the object
// "Label2" of the Label class.
Label Label2 = new Label("C2");

// Initializing the object
// "Label3" of the Label class.
Label Label3 = new Label("C3");

// Initializing the object
// "Label4" of the Label class.
Label Label4 = new Label("C4");

// Adding Label "Label1" to the Panel "Panel1".
Panel1.add(Label1);

// Adding Label "Label2" to the Panel "Panel2".
Panel2.add(Label2);

// Adding Label "jLabel3" to the Panel "jPanel3".
jPanel3.add(jLabel3);

// Adding Label "Label4" to the Panel "Panel4".
Panel4.add(Label4);

// Add the "Panel1" on cPanel
cPanel.add(Panel1, "1");

// Add the "Panel2" on cPanel
cPanel.add(Panel2, "2");

// Add the "Panel3" on cPanel
cPanel.add(Panel3, "3");

// Add the "Panel4" on cPanel
cPanel.add(Panel4, "4");

// Creating an Object of the "Panel" class
```

```
Panel btnPanel = new Panel();

// Initializing the object  "firstButton" of the Button class.
Button firstButton = new Button("First");

// Initializing the object
// "nextButton" of the Button class.
Button nextButton = new Button("->");

// Initializing the object
// "previousbtn" of Button class.
Button previousButton = new Button("<-");

// Initializing the object
// "lastButton" of the Button class.
Button lastButton = new Button("Last");

// Adding the Button "firstbutton" on the Panel.
btnPanel.add(firstButton);

// Adding the Button "nextButton" on the Panel.
btnPanel.add(nextButton);

// Adding the Button "previousButton" on the Panel.
btnPanel.add(previousButton);

// Adding the Button "lastButton" on the Panel.
btnPanel.add(lastButton);

// adding firstButton in the ActionListener
firstButton.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae)
{

// using the first cObjl CardLayout
cObjl.first(cPanel);

// value of currCard is 1
currCard = 1;
}
});

// add lastButton in ActionListener
lastButton.addActionListener(new ActionListener()
{
```

```
public void actionPerformed(ActionEvent ae)
{

// using the last cObjl CardLayout
cObjl.last(cPanel);

// value of currCard is 4
currCard = 4;
}
});

// add nextButton in ActionListener
nextButton.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae)
{

if (currCard < 4)
{

// increase the value of currCard by 1
currCard = currCard + 1;

// show the value of currCard
cObjl.show(cPanel, "" + (currCard));
}
}
});

// add previousButton in ActionListener
previousButton.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae)
{

if (currCard > 1)
{

// decrease the value
// of currCard by 1
currCard = currCard - 1;

// show the value of currCard
cObjl.show(cPanel, "" + (currCard));
}
}
```

```
});

// using to get the content pane
getContentPane().add(cPanel, BorderLayout.NORTH);

// using to get the content pane
getContentPane().add(btnPanel, BorderLayout.SOUTH);
}

// main method
public static void main(String argvs[])
{

// Creating an object of the CardLayoutExample3 class.
CardLayoutExample3 cll = new CardLayoutExample3();

// method to set the default operation of the Frame.
cll.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// aethod to set the visibility of the JFrame.
cll.setVisible(true);
}
}
```



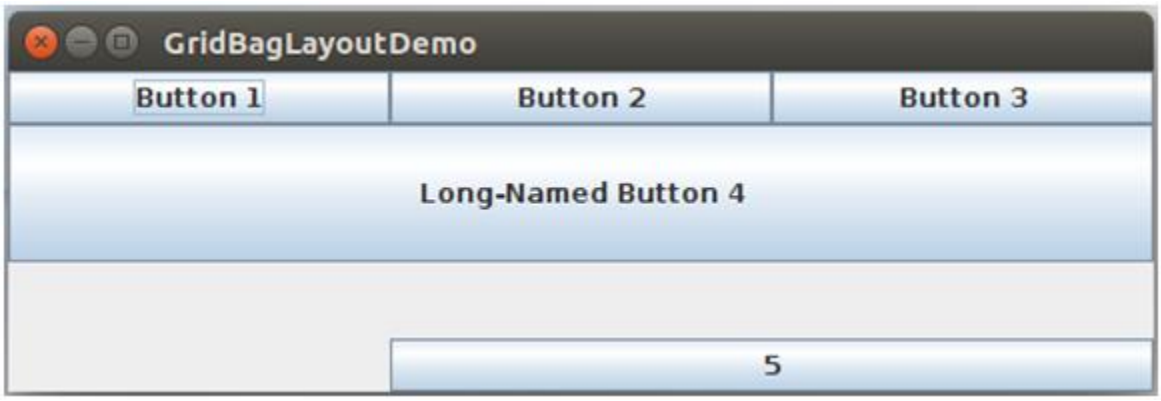
JAVA GRIDBAGLAYOUT

Java, being a versatile programming language, provides developers with numerous tools and techniques to create visually appealing and well-organized user interfaces (UIs). One such tool is the **GridLayout**, which offers a systematic approach to arrange components within a container in rows and columns. In this section, we will delve into the intricacies of **GridLayout in Java**, exploring its functionalities, advantages, and implementation techniques.

Write a java program to add awt components to the frame using Gridbag layout

```
public class GridBagLayoutDemo
 {
final static boolean shouldFill = true;
final static boolean shouldWeightX = true;
final static boolean RIGHT_TO_LEFT = false;

public static void addComponentsToPane(Container pane)
{
if (RIGHT_TO_LEFT)
{
pane.setComponentOrientation(ComponentOrientation.RIGHT_TO_LEFT);
}

Button button;
pane.setLayout(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
if (shouldFill) {
//natural height, maximum width
c.fill = GridBagConstraints.HORIZONTAL;
}

button = new Button("Button 1");
if (shouldWeightX) {
c.weightx = 0.5;
}
c.fill = GridBagConstraints.HORIZONTAL;
c.gridx = 0;
c.gridy = 0;
pane.add(button, c);

button = new JButton("Button 2");
c.fill = GridBagConstraints.HORIZONTAL;
c.weightx = 0.5;
c.gridx = 1;
c.gridy = 0;
pane.add(button, c);

button = new Button("Button 3");
c.fill = GridBagConstraints.HORIZONTAL;
c.weightx = 0.5;
c.gridx = 2;
c.gridy = 0;
pane.add(button, c);
```

```
button = new Button("Long-Named Button 4");
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 40;      //make this component tall
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
pane.add(button, c);

button = new Button("5");
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 0;       //reset to default
c.weighty = 1.0;   //request any extra vertical space
c.anchor = GridBagConstraints.PAGE_END; //bottom of space
c.insets = new Insets(10,0,0,0);  //top padding
c.gridx = 1;       //aligned with button 2
c.gridwidth = 2;   //2 columns wide
c.gridy = 2;       //third row
pane.add(button, c);
}


private static void createAndShowGUI() {
//Create and set up the window.
Frame frame = new Frame("GridBagLayoutDemo");
frame.setDefaultCloseOperation(Frame.EXIT_ON_CLOSE);

//Set up the content pane.
addComponentsToPane(frame.getContentPane());

//Display the window.
frame.pack();
frame.setVisible(true);
}

public static void main(String[] args) {
 {
public void run() {
createAndShowGUI();
}
});
}
}
```


GridBagLayoutDemo
Button 1
Button 2
Button 3
Long-Named Button 4
5

APPLETS

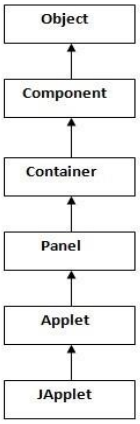
Applets: Concepts of Applets, life cycle of an applet, creating applets, passing parameters toapplets.

https://codingatharva.blogspot.com/search/label/Applet%20and%20Graphics%20in%20Java

Java applets are small programs written in Java that are designed to run within a web browser or an applet viewer. They were a popular way to create interactive web applications in the early days of the internet. However, applets have become largely obsolete due to security concerns and the rise of modern web technologies like HTML5, JavaScript, and CSS.

Key Features of Java Applets:

1. **Platform Independence**: Applets are platform-independent, as they run on the Java Virtual Machine (JVM).
2. **Embedded in Web Pages**: Applets are embedded in HTML pages using the `<applet>` or `<object>` tag (now deprecated).
3. **Security Restrictions**: Applets run in a sandbox environment to prevent unauthorized access to the user's system.
4. **Graphical User Interface (GUI)**: Applets can create interactive GUIs using the Abstract Window Toolkit (AWT) or Swing.



Advantage of Applet

There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

Plugin is required at client browser to execute applet.

SIMPLE APPLET PROGRAM TO DISPLAY HELLO WORLD

```
Import java.applet.*;

import java.awt.Graphics;

public class HelloWorldApplet extends Applet

{

   public void paint(Graphics g)

{

       g.drawString("Hello, World!", 20, 20);

   }

}

/*<applet code="HelloWorldApplet.class" width="300"
height="300"></applet>*/
```

Save the file as HelloWorldApplet.java

Compile:

javac HelloWorldApplet.java

Run

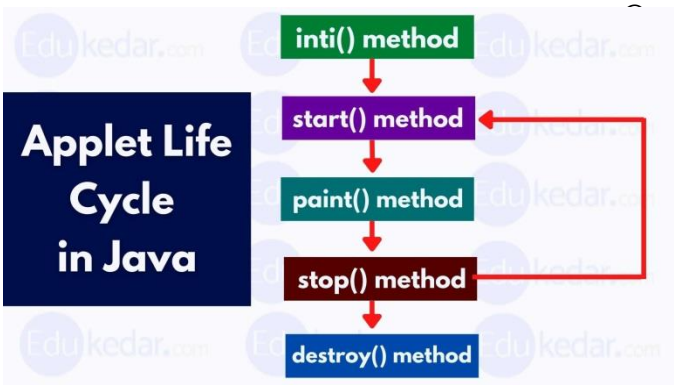
appletviewer HelloWorldApplet

LIFECYCLE OF AN APPLET:

An applet's lifecycle is managed by the browser or applet viewer. It includes the following methods:

1. **init()**: Called once when the applet is initialized.
2. **start()**: Called each time the applet becomes active.
3. **paint(Graphics g)**: Used to draw graphics or update the applet's display.
4. **stop()**: Called when the applet is no longer active.
5. **destroy()**: Called when the applet is being removed from memory.

os through java



The **applet life cycle in Java** consists of several stages, which are managed by the browser. The key methods involved in this life cycle are:

1. **init()**: Initializes the applet; called once when the applet is loaded.
2. **start()**: Starts the applet's execution; called after init() and each time the user returns to the page.
3. **stop()**: Stops the applet's execution; called when the user leaves the page.
4. **paint()**: Responsible for rendering the applet's content; called whenever the applet needs to be redrawn.
5. **destroy()**: Cleans up resources before the applet is destroyed; called when the applet is no longer needed.

Example of Applet Life Cycle

Here is an example demonstrating the applet life cycle:

```
import java.applet.*;
import java.awt.*;

public class MyApplet extends Applet
{
public void init()
 {
// Initialize objects
}

public void start()
{
// Start the applet code
}

public void paint(Graphics g)
```

```
 {
g.drawString("Hello, world!", 50, 25);
}

public void stop()
{
// Stop the applet code
}

public void destroy()
 {
// Destroy the applet
}
}
```

When this applet is loaded into a web browser, the following sequence of events occurs:

1. The browser calls the `init()` method to initialize the applet.
2. The browser calls the `start()` method to start the applet.
3. The browser calls the `paint()` method to draw the applet on the screen.
4. If the user navigates away from the page or the browser is closed, the browser calls the `stop()` method to stop the applet.
5. If the applet is no longer needed, the browser calls the `destroy()` method to release any resources that were allocated by the applet

PARAMETERS PASSING TO APPLET

1. Use the param attribute of tag to pass the parameters.
2. Use the getParameter() method of Applet class to retrieve a parameter's value.
3. Pass parameters in NAME=VALUE pairs in tags between the opening and closing APPLET tags.
4. Read the values passed through the PARAM tags with the getParameter() method of the java.applet.Applet class.
5. Include appropriate tags in the HTML documents.
6. Provide code in the applet to parse these parameters.

PROGRAM TO PRINT PARAMETER OF APPLET IN JAVA

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

public class Parameter extends Applet
{
 String name;

 public void init()
 {
  setBackground(Color.red);
  name = getParameter("Coding");
 }

 public void paint(Graphics g)
 {
  g.drawString("Coding Atharva",25,100);
  g.drawString(name,20,10);
 }

}
/*
 <applet code="Parameter" height=300 width=200>
  <param name="Coding" value="100" />
 </applet>
*/
```

JDBC-JAVA DATABAE CONNECTIVITY

JDBC Connectivity: JDBC Type 1 to 4 Drivers, connection establishment, QueryExecution.

JDBC stands for **Java Database Connectivity**. It's a Java API (Application Programming Interface) that allows Java programs to connect to and interact with databases.

Key Points About JDBC:

- **Purpose:** Enables Java applications to execute SQL statements, retrieve results, and update data in a database.
- **Platform-independent:** JDBC provides a standard interface, so Java programs can interact with many different types of databases (like MySQL, Oracle, PostgreSQL) using the same code, just by switching drivers.
- **Part of Java Standard Edition:** It's included in the Java SE platform, so you don't need extra installations (except the database-specific JDBC driver).
- **Main Components:**
  - **DriverManager:** Manages a list of database drivers.
  - **Connection:** Represents a session with a database.
  - **Statement / PreparedStatement:** Used to send SQL commands.
  - **ResultSet:** Stores the results retrieved from a database query.

How does JDBC work?

1. Your Java program loads the appropriate **JDBC driver** (a database-specific component).
2. Connects to the database via a **Connection** object.
3. Sends SQL commands using **Statement** or **PreparedStatement**.
4. Processes the results through **ResultSet**.
5. Closes the connection when done.

Why use JDBC?

- To **access and manipulate databases** directly from Java programs.
- To provide a **consistent API** for all relational databases.
- To support both **simple queries** and **complex transactions**.

Types of JDBC Drivers

Java Database Connectivity (JDBC) is an API that allows Java applications to interact with databases. JDBC drivers are essential components that enable this interaction by converting Java calls into database-specific calls. There are four types of JDBC drivers, each with its own characteristics and use cases.

Type-1 Driver: JDBC-ODBC Bridge Driver

The Type-1 driver, also known as the JDBC-ODBC bridge driver, uses ODBC drivers to connect to databases. It converts JDBC method calls into ODBC function calls. This driver is easy to use and can connect to any database, but it has several drawbacks, including degraded performance and the need for ODBC driver installation on client machines. Additionally, it is not written in Java, making it non-portable

Type-2 Driver: Native-API Driver

The Type-2 driver, or Native-API driver, uses the client-side libraries of the database. It converts JDBC method calls into native calls of the database API. This driver offers better performance than the Type-1 driver but requires the installation of database-specific client libraries on each client machine. It is also not fully written in Java, which affects its portability.

Type-3 Driver: Network Protocol Driver

The Type-3 driver, also known as the Network Protocol driver, uses middleware to convert JDBC calls into vendor-specific database protocols. This driver is fully written in Java, making it portable. It does not require client-side libraries, as the middleware handles tasks like auditing, load balancing, and logging. However, it requires network support on the client machine and can be costly to maintain due to database-specific coding in the middle tier.

Type-4 Driver: Thin Driver

The Type-4 driver, or Thin driver, interacts directly with the database using a vendor-specific protocol. It is fully written in Java, making it portable and easy to use without requiring any native libraries or middleware. This driver offers the best performance among all JDBC drivers but is database-dependent

Choosing the Right Driver

- **Type-4 Driver**: Preferred for accessing a single type of database, such as Oracle or IBM.

- **Type-3 Driver**: Suitable for applications accessing multiple types of databases simultaneously.
- **Type-2 Driver**: Useful when Type-3 or Type-4 drivers are not available for a specific database.
- **Type-1 Driver**: Generally used for development and testing purposes only.

In summary, the choice of JDBC driver depends on the specific requirements of the application, including performance, portability, and the types of databases being accessed.

To establish a database connection in a Java program, you typically follow these steps:

---

1. Add the JDBC Driver

- First, you need the JDBC driver for the specific database you're using (e.g., MySQL, PostgreSQL, Oracle).
- This driver is usually a .jar file that must be included in your project's classpath.

---

2. Load the JDBC Driver

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

- This line loads the JDBC driver class into memory.
- Since JDBC 4.0, this step is optional if the driver jar has a proper service provider configuration.

---

3. Create a Connection String (URL)

- The connection URL depends on the database.
- For example, MySQL:

```
jdbc:mysql://hostname:port/databasename
String url = "jdbc:mysql://localhost:3306/mydatabase";
```

---

4. Establish the Connection

Use the DriverManager.getConnection() method to connect:

```
Connection conn = DriverManager.getConnection(url, "username", "password");
```

---

5. Use the Connection

- Create a Statement or PreparedStatement to run queries.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
```

---

6. Close the Connection

- Always close ResultSet, Statement, and Connection objects to free resources.

```
rs.close();
stmt.close();
conn.close();
```

Write a java program to establish database connection to a java application

```
import java.sql.*;
public class DatabaseConnectExample
 {
   public static void main(String[] args)
{
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String user = "root";
        String password = "mypassword";

        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        try {
            // Load the JDBC driver (optional for newer JDBC versions)
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish connection
            conn = DriverManager.getConnection(url, user, password);

            // Create statement
            stmt = conn.createStatement();

            // Execute query
            rs = stmt.executeQuery("SELECT * FROM mytable");
```

```
            // Process result set
            while(rs.next()) {
                System.out.println("Column1: " + rs.getString("column1"));
            }
        } catch (ClassNotFoundException e)
 {
            System.out.println("JDBC Driver not found.");
            e.printStackTrace();
        } catch (SQLException e)
{
            System.out.println("Database connection error.");
            e.printStackTrace();
        }
finally
{
            // Close resources
            try
{
 if(rs != null) rs.close();
 }
catch(SQLException e)
 {
 e.printStackTrace();
}
            Try
 {
 if(stmt != null) stmt.close();
}
catch(SQLException e)
 {
e.printStackTrace();
}
            try
{
if(conn != null) conn.close();
 }
catch(SQLException e)
{
e.printStackTrace();
 }
        }
    }
}
```