

**SIMATS ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICALAND TECHNICAL SCIENCES**

**JAVA ASSIGNMENT**

**REPORT SUBMITTED BY**

MURALI KRISHNAN S-192321093

**COURSE CODE / NAME**

CSA0986 / PROGRAMMING IN JAVA FOR API

SLOT B

**Operators in Java**

Operator in Java is a symbol that is used to perform operations. For example: +, -, \*, / etc.

There are many types of operators in Java which are given below:

**Unary Operator,**

**Arithmetic Operator,**

**Shift Operator,**

**Relational Operator,**

**Bitwise Operator,**

**Logical Operator,**

**Ternary Operator and**

**Assignment Operator.**

**Unary Operator**

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

* incrementing/decrementing a value by one
* negating an expression
* inverting the value of a Boolean

Java Unary Operator Example: ++ and --

public class OperatorExample{

public static void main(String args[]){

int x=10;

System.out.println(x++);//10 (11)

System.out.println(++x);//12

System.out.println(x--);//12 (11)

System.out.println(--x);//10 }}

Output:

10

12

12

10

**Arithmetic Operators**

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Arithmetic Operator Example

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a\*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}

**Output:**

15

5

50

2

0

**SHIFT OPERATOR**

**Left Shift Operator**

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

public class OperatorExample{

public static void main(String args[]){

System.out.println(10<<2);//10\*2^2=10\*4=40

System.out.println(10<<3);//10\*2^3=10\*8=80

System.out.println(20<<2);//20\*2^2=20\*4=80

System.out.println(15<<4);//15\*2^4=15\*16=240

}}

Output:

40

80

80

240

**Right Shift Operator**

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

public OperatorExample{

public static void main(String args[]){

System.out.println(10>>2);//10/2^2=10/4=2

System.out.println(20>>2);//20/2^2=20/4=5

System.out.println(20>>3);//20/2^3=20/8=2 }}

Output:

2

5

2

**AND Operator Example: Logical && and Bitwise &**

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

1. public **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}

Output:

false

false

**AND Operator Example: Logical && vs Bitwise &**

public class OperatorExample{

public static void main(String args[]){

int a=10;

int b=5;

int c=20;

System.out.println(a<b&&a++<c);//false && true = false

System.out.println(a);//10 because second condition is not checked

System.out.println(a<b&a++<c);//false && true = false

System.out.println(a);//11 because second condition is checked

}}

Output:

false

10

false

11

**OR Operator Example: Logical || and Bitwise |**

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

public class OperatorExample{

public static void main(String args[]){

int a=10;

int b=5;

int c=20;

System.out.println(a>b||a<c);//true || true = true

System.out.println(a>b|a<c);//true | true = true

//|| vs |

System.out.println(a>b||a++<c);//true || true = true

System.out.println(a);//10 because second condition is not checked

System.out.println(a>b|a++<c);//true | true = true

System.out.println(a);//11 because second condition is checked

}}

Output:

true

true

true

10

true

11

**Ternary Operator**

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Java Ternary Operator Example

public class OperatorExample{

public static void main(String args[]){

int a=2;

int b=5;

int min=(a<b)?a:b;

System.out.println(min);

}}

Output:

2

**Assignment Operator**

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

public class OperatorExample{

public static void main(String args[]){

int a=10;

int b=20;

a+=4;//a=a+4 (a=10+4)

b-=4;//b=b-4 (b=20-4)

System.out.println(a);

System.out.println(b);

}}

Output:

14

16

**CONDITIONAL STATEMENTS**

Conditional statements in Java allow the program to make decisions based on certain conditions. These statements help control the flow of execution in a program depending on whether the condition evaluates to true or false.

**Types of Conditional Statements**

1. **if Statement**
2. **if-else Statement**
3. **if-else-if Ladder**
4. **switch Statement**

**1. if Statement**

The if statement evaluates a condition, and if it’s true, the block of code inside the if statement is executed.

**Syntax:**

java

Copy code

if (condition) {

// Code to execute if condition is true

}

**Example:**

java

Copy code

int age = 20;

if (age >= 18) {

System.out.println("You are eligible to vote.");

}

**Explanation**: Here, the program checks if age is 18 or above. If true, it prints "You are eligible to vote."

**2. if-else Statement**

The if-else statement allows two possible paths: if the condition is true, it executes one block of code, and if it’s false, it executes another.

**Syntax:**

java

Copy code

if (condition) {

// Code to execute if condition is true

} else {

// Code to execute if condition is false

}

**Example:**

java

Copy code

int age = 16;

if (age >= 18) {

System.out.println("You are eligible to vote.");

} else {

System.out.println("You are not eligible to vote.");

}

**Explanation**: If age is less than 18, it will print "You are not eligible to vote."

**3. if-else-if Ladder**

The if-else-if ladder allows multiple conditions to be checked in sequence. As soon as one condition is true, the corresponding block of code executes, and the ladder ends.

**Syntax:**

java

Copy code

if (condition1) {

// Code to execute if condition1 is true

} else if (condition2) {

// Code to execute if condition2 is true

} else if (condition3) {

// Code to execute if condition3 is true

} else {

// Code to execute if none of the conditions are true

}

**Example:**

java

Copy code

int marks = 85;

if (marks >= 90) {

System.out.println("Grade: A");

} else if (marks >= 80) {

System.out.println("Grade: B");

} else if (marks >= 70) {

System.out.println("Grade: C");

} else {

System.out.println("Grade: D");

}

**Explanation**: This code assigns grades based on marks. If marks are between 80 and 89, it assigns a "B" grade, and so on.

**4. switch Statement**

The switch statement allows you to execute one block of code among many options based on the value of an expression. It is commonly used when you have multiple options for a single variable.

**Syntax:**

java

Copy code

switch (expression) {

case value1:

// Code to execute if expression == value1

break;

case value2:

// Code to execute if expression == value2

break;

// more cases...

default:

// Code to execute if no case matches

}

**Example:**

java

Copy code

int day = 3;

String dayName;

switch (day) {

case 1:

dayName = "Sunday";

break;

case 2:

dayName = "Monday";

break;

case 3:

dayName = "Tuesday";

break;

case 4:

dayName = "Wednesday";

break;

case 5:

dayName = "Thursday";

break;

case 6:

dayName = "Friday";

break;

case 7:

dayName = "Saturday";

break;

default:

dayName = "Invalid day";

break;

}

System.out.println("Day: " + dayName);

**Explanation**: This code assigns the day name based on the integer value of day. If day is 3, it assigns "Tuesday" to dayName.

**LOOPING STATEMENTS**

**1. for Loop**

The for loop is used when the number of iterations is known beforehand. It includes an initialization, a condition, and an increment/decrement statement, all in one line.

**Syntax:**

java

Copy code

for (initialization; condition; update) {

// Code to execute in each iteration

}

**Example:**

java

Copy code

for (int i = 1; i <= 5; i++) {

System.out.println("Iteration " + i);

}

**Explanation**: Here, the loop initializes i to 1, checks if i <= 5, and increments i after each iteration. This will print "Iteration 1" to "Iteration 5".

**2. while Loop**

The while loop is used when the number of iterations is not known in advance. It keeps executing as long as the condition is true.

**Syntax:**

java

Copy code

while (condition) {

// Code to execute in each iteration

}

**Example:**

java

Copy code

int i = 1;

while (i <= 5) {

System.out.println("Iteration " + i);

i++;

}

**Explanation**: This loop initializes i to 1 and keeps printing "Iteration X" as long as i is less than or equal to 5, incrementing i in each iteration.

**3. do-while Loop**

The do-while loop is similar to the while loop, but it guarantees at least one execution of the loop body, regardless of the condition, since the condition is checked at the end.

**Syntax:**

java

Copy code

do {

// Code to execute in each iteration

} while (condition);

**Example:**

java

Copy code

int i = 1;

do {

System.out.println("Iteration " + i);

i++;

} while (i <= 5);

**Explanation**: This loop executes the block first and then checks if i is less than or equal to 5. It prints "Iteration X" from 1 to 5. Even if the condition were false initially, the loop would run once.

**INHERITANCE AND ITS TYPES**

In Java, **inheritance** is a key feature of object-oriented programming (OOP) that allows a class (subclass) to inherit fields and methods from another class (superclass). This enables code reusability, logical class hierarchies, and polymorphism, where an object can take on multiple forms.

**Key Concepts of Inheritance**

* **Superclass**: The parent class from which properties and behaviors are inherited.
* **Subclass**: The child class that inherits from the superclass.
* **extends Keyword**: Used to establish inheritance between classes in Java.
* **Method Overriding**: Allows a subclass to provide a specific implementation for a method already defined in its superclass.

**Types of Inheritance in Java**

Java supports three main types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**

**Note**: Java does not support **multiple inheritance** (a class inheriting from more than one superclass) to avoid ambiguity and complexity. However, multiple inheritance can be achieved using interfaces.

**1. Single Inheritance**

In single inheritance, a class inherits from only one superclass. This is the most straightforward inheritance type, where a subclass extends the behavior of a single parent class.

**Example of Single Inheritance**

java

Copy code

// Superclass

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

// Subclass

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

public class SingleInheritanceExample {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.eat(); // Inherited method

myDog.bark(); // Dog's own method

}

}

**Explanation**: Dog inherits from Animal and can use both the eat method (from Animal) and its own bark method.

**2. Multilevel Inheritance**

In multilevel inheritance, a class is derived from another class, which is itself derived from another class, creating a chain of inheritance.

**Example of Multilevel Inheritance**

java

Copy code

// Base class (grandparent class)

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

// Derived class (parent class)

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

// Derived class (child class)

class Puppy extends Dog {

void weep() {

System.out.println("The puppy weeps.");

}

}

public class MultilevelInheritanceExample {

public static void main(String[] args) {

Puppy myPuppy = new Puppy();

myPuppy.eat(); // Inherited from Animal

myPuppy.bark(); // Inherited from Dog

myPuppy.weep(); // Puppy’s own method

}

}

**Explanation**: Puppy inherits from Dog, which in turn inherits from Animal. Puppy has access to methods from both Animal and Dog.

1. **Hierarchical Inheritance**

In hierarchical inheritance, multiple classes inherit from a single superclass. Each subclass can define its own methods while sharing behavior from the common superclass.

**Example of Hierarchical Inheritance**

java

Copy code

// Superclass

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

// First subclass

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

// Second subclass

class Cat extends Animal {

void meow() {

System.out.println("The cat meows.");

}

}

public class HierarchicalInheritanceExample {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.eat(); // Inherited from Animal

myDog.bark(); // Dog’s own method

Cat myCat = new Cat();

myCat.eat(); // Inherited from Animal

myCat.meow(); // Cat’s own method

}

}

**Explanation**: Dog and Cat both inherit from Animal, sharing the eat method but each having its own unique behavior (bark for Dog, meow for Cat).

**APPLETE IN JAVA**

An **applet** is a small Java program that can be embedded in web pages and executed within a browser or an applet viewer. Applets are primarily used for internet-based applications that need to run on a client’s machine, offering interactive features such as graphics, animations, or simple games. However, with the decline of support for Java applets in modern web browsers, applets are less commonly used today.

**Key Features of Java Applets**

* **Platform Independence**: Applets are platform-independent, like other Java applications.
* **Security**: Applets run in a restricted environment known as the “sandbox,” which prevents them from performing unauthorized actions.
* **GUI Components**: Applets can use AWT or Swing libraries to add graphical components like buttons, text fields, etc.

**Lifecycle of an Applet**

An applet has a specific lifecycle, which includes the following methods:

1. **init()**: Initializes the applet; called only once.
2. **start()**: Called each time the applet becomes active.
3. **stop()**: Called each time the applet is inactive.
4. **destroy()**: Called when the applet is about to be destroyed.

**Types of Applet Examples**

Here are examples of the different types of tasks applets can perform:

**1. Basic Applet with init() and paint()**

This example demonstrates a basic applet that displays text on the screen. The paint() method is used to draw on the applet.

java

Copy code

import java.applet.Applet;

import java.awt.Graphics;

/\*

<applet code="HelloApplet.class" width="300" height="200">

</applet>

\*/

public class HelloApplet extends Applet {

public void init() {

System.out.println("Applet initialized.");

}

public void paint(Graphics g) {

g.drawString("Hello, Applet!", 50, 100);

}

}

**Explanation**: This applet initializes using init() and then draws "Hello, Applet!" on the screen using the paint() method.

**2. Interactive Applet with Buttons**

In this example, the applet has buttons, and clicking the buttons changes the text displayed. This demonstrates how to handle user interactions.

java

Copy code

import java.applet.Applet;

import java.awt.\*;

import java.awt.event.\*;

/\*

<applet code="ButtonApplet.class" width="300" height="200">

</applet>

\*/

public class ButtonApplet extends Applet implements ActionListener {

Button button1, button2;

String message = "";

public void init() {

button1 = new Button("Button 1");

button2 = new Button("Button 2");

add(button1);

add(button2);

button1.addActionListener(this);

button2.addActionListener(this);

}

public void actionPerformed(ActionEvent e) {

if (e.getSource() == button1) {

message = "Button 1 pressed";

} else if (e.getSource() == button2) {

message = "Button 2 pressed";

}

repaint();

}

public void paint(Graphics g) {

g.drawString(message, 50, 100);

}

}

**Explanation**: This applet has two buttons. When either button is clicked, actionPerformed is called, and a message is displayed indicating which button was pressed.

**3. Drawing Shapes in an Applet**

This applet demonstrates the use of basic graphics to draw shapes like rectangles, circles, and lines.

java

Copy code

import java.applet.Applet;

import java.awt.Graphics;

/\*

<applet code="ShapeApplet.class" width="300" height="200">

</applet>

\*/

public class ShapeApplet extends Applet {

public void paint(Graphics g) {

g.drawRect(50, 50, 100, 50); // Draws a rectangle

g.drawOval(200, 50, 50, 50); // Draws a circle

g.drawLine(50, 150, 250, 150); // Draws a line

}

}

**Explanation**: This applet draws a rectangle, a circle, and a line using the paint() method.

**4. Animation in an Applet**

This example shows a simple animation where a circle moves horizontally across the applet area.

java

Copy code

import java.applet.Applet;

import java.awt.\*;

/\*

<applet code="AnimationApplet.class" width="400" height="200">

</applet>

\*/

public class AnimationApplet extends Applet implements Runnable {

int xPosition = 0;

Thread animator;

public void init() {

animator = new Thread(this);

animator.start();

}

public void run() {

while (true) {

xPosition += 5;

if (xPosition > getWidth()) {

xPosition = 0;

}

repaint();

try {

Thread.sleep(50);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

public void paint(Graphics g) {

g.setColor(Color.RED);

g.fillOval(xPosition, 100, 30, 30); // Draws a moving circle

}

}

**Explanation**: This applet animates a red circle moving from left to right. It uses a thread to repeatedly update the circle’s position and redraw it.

**5. Applet with User Input (TextField)**

This applet accepts user input through a TextField and displays it.

java

Copy code

import java.applet.Applet;

import java.awt.\*;

import java.awt.event.\*;

/\*

<applet code="InputApplet.class" width="300" height="200">

</applet>

\*/

public class InputApplet extends Applet implements ActionListener {

TextField inputField;

String message = "";

public void init() {

inputField = new TextField(20);

add(inputField);

inputField.addActionListener(this);

}

public void actionPerformed(ActionEvent e) {

message = inputField.getText();

repaint();

}

public void paint(Graphics g) {

g.drawString("You entered: " + message, 50, 100);

}

}

**Explanation**: This applet includes a TextField to accept input from the user. When the user presses Enter, the entered text is displayed.

**JAVAFX**

JavaFX is a powerful framework in Java used for creating rich Internet applications (RIAs) with a focus on graphical user interface (GUI) design. It provides a set of Java libraries that allow developers to build desktop applications and integrate them with web-based applications. JavaFX includes capabilities for 2D and 3D graphics, animations, media playback, and more, making it ideal for interactive applications.

Here’s how to create an animation in JavaFX for a circle with the following effects:

1. **Rotation**: The circle will rotate around its center.
2. **Revolution**: The circle will move in a circular orbit around a specified center point.
3. **Outer Layer Color Change**: The outer stroke color of the circle changes over time.
4. **Inner Color Varying from Low to High Intensity**: The fill color of the circle varies gradually.

**Code for JavaFX Circle Animation**

java

Copy code

import javafx.animation.\*;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.layout.Pane;

import javafx.scene.paint.Color;

import javafx.scene.shape.Circle;

import javafx.stage.Stage;

import javafx.util.Duration;

public class CircleAnimation extends Application {

@Override

public void start(Stage primaryStage) {

// Create the circle to be animated

Circle circle = new Circle(50, Color.BLUE);

circle.setStroke(Color.BLACK);

circle.setStrokeWidth(3);

circle.setCenterX(100);

circle.setCenterY(100);

// Create a Pane to hold the circle

Pane pane = new Pane();

pane.getChildren().add(circle);

Scene scene = new Scene(pane, 400, 400);

// Animation for Rotation around its own center

RotateTransition rotateTransition = new RotateTransition(Duration.seconds(2), circle);

rotateTransition.setByAngle(360); // Rotate by 360 degrees

rotateTransition.setCycleCount(Animation.INDEFINITE);

rotateTransition.setInterpolator(Interpolator.LINEAR);

rotateTransition.play();

// Animation for Revolution around a point

PathTransition pathTransition = new PathTransition();

Circle orbit = new Circle(150); // Orbit radius

orbit.setCenterX(200);

orbit.setCenterY(200);

pathTransition.setPath(orbit);

pathTransition.setNode(circle);

pathTransition.setInterpolator(Interpolator.LINEAR);

pathTransition.setDuration(Duration.seconds(5));

pathTransition.setCycleCount(Animation.INDEFINITE);

pathTransition.play();

// Outer layer color change animation

FillTransition strokeColorTransition = new FillTransition(Duration.seconds(3), circle);

strokeColorTransition.setFromValue(Color.BLACK);

strokeColorTransition.setToValue(Color.RED);

strokeColorTransition.setCycleCount(Animation.INDEFINITE);

strokeColorTransition.setAutoReverse(true);

strokeColorTransition.play();

// Inner color intensity change animation

FillTransition fillTransition = new FillTransition(Duration.seconds(2), circle);

fillTransition.setFromValue(Color.LIGHTBLUE);

fillTransition.setToValue(Color.DARKBLUE);

fillTransition.setCycleCount(Animation.INDEFINITE);

fillTransition.setAutoReverse(true);

fillTransition.play();

// Set the scene and show the stage

primaryStage.setTitle("Circle Animation");

primaryStage.setScene(scene);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

**Explanation of the Code**

1. **Rotation**: A RotateTransition is applied to the circle itself, making it rotate around its center.
2. **Revolution**: A PathTransition makes the circle move along the path of another larger, invisible circle (orbit), creating a revolution effect.
3. **Outer Layer Color Change**: A FillTransition changes the stroke color of the circle from black to red, which continuously alternates.
4. **Inner Color Varying**: Another FillTransition changes the fill color from a lighter blue to a darker blue, providing a gradient effect that goes from low to high intensity.

This code requires JavaFX to be properly set up in your development environment. Let me know if you need additional help with JavaFX setup or further customization.

**JAVA DATABASE CONNECTIVITY** (**JDBC)**

**JDBC** (Java Database Connectivity) is a Java API that allows Java applications to interact with a database. JDBC provides a standard interface to connect to a database, execute SQL queries, and retrieve results, enabling Java applications to communicate with various databases (like MySQL, Oracle, PostgreSQL) in a database-independent manner.

**Key Components of JDBC**

1. **JDBC Driver**: A software component that enables Java applications to interact with the database. Different types of JDBC drivers exist, including:
   * Type-1 (JDBC-ODBC Bridge Driver)
   * Type-2 (Native API Driver)
   * Type-3 (Network Protocol Driver)
   * Type-4 (Thin Driver, most commonly used)
2. **Connection Interface**: Represents a connection with a specific database. It is used to establish a session with the database.
3. **Statement Interface**: Represents SQL statements. It’s used to execute SQL queries on the database.
4. **ResultSet Interface**: Represents the result of executing a SQL query. It provides methods to iterate over the results and retrieve data.
5. **DriverManager Class**: Manages the list of database drivers and establishes a connection between the Java application and the database.

**Steps to Connect to a Database Using JDBC**

1. **Load the JDBC Driver**: Load the JDBC driver class.
2. **Establish a Connection**: Use DriverManager to establish a connection to the database.
3. **Create a Statement**: Create a Statement object to execute SQL queries.
4. **Execute Queries**: Use Statement to run queries and retrieve results in ResultSet.
5. **Process Results**: Process data from ResultSet.
6. **Close Connection**: Close the database connection.

**Example: Connecting to a MySQL Database Using JDBC**

The following example demonstrates how to connect to a MySQL database, execute a SQL query, and retrieve data.

**1. Setting Up the MySQL Database**

Assuming you have a database called school with a table students as follows:

sql

Copy code

CREATE DATABASE school;

USE school;

CREATE TABLE students (

id INT PRIMARY KEY,

name VARCHAR(50),

age INT,

grade CHAR(1)

);

INSERT INTO students (id, name, age, grade) VALUES

(1, 'Alice', 14, 'A'),

(2, 'Bob', 15, 'B'),

(3, 'Charlie', 14, 'A');

**2. Java Code to Connect to the Database and Retrieve Data**

java

Copy code

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

public class JDBCExample {

public static void main(String[] args) {

// Database URL, Username, and Password

String url = "jdbc:mysql://localhost:3306/school";

String username = "root";

String password = "your\_password";

Connection connection = null;

Statement statement = null;

ResultSet resultSet = null;

try {

// Load MySQL JDBC Driver

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

// Establish Connection to Database

connection = DriverManager.getConnection(url, username, password);

System.out.println("Connected to the database successfully.");

// Create a Statement

statement = connection.createStatement();

// Execute a Query

String sql = "SELECT \* FROM students";

resultSet = statement.executeQuery(sql);

// Process the ResultSet

System.out.println("ID\tName\t\tAge\tGrade");

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

int age = resultSet.getInt("age");

String grade = resultSet.getString("grade");

System.out.println(id + "\t" + name + "\t\t" + age + "\t" + grade);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

// Close resources

try {

if (resultSet != null) resultSet.close();

if (statement != null) statement.close();

if (connection != null) connection.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

}

**Explanation of the Code**

1. **Load JDBC Driver**: Class.forName("com.mysql.cj.jdbc.Driver"); loads the MySQL JDBC driver, allowing Java to connect to a MySQL database.
2. **Establish Connection**: DriverManager.getConnection(url, username, password); establishes a connection to the database using the URL, username, and password provided.
3. **Create Statement**: connection.createStatement(); creates a Statement object, which will be used to execute SQL queries.
4. **Execute Query**: statement.executeQuery(sql); executes the SQL query (SELECT \* FROM students) and returns a ResultSet object.
5. **Process ResultSet**: The while loop iterates through the ResultSet, retrieving and printing each record's id, name, age, and grade fields.
6. **Close Connection**: Finally, the ResultSet, Statement, and Connection objects are closed to release resources.

**Sample Output**

If the students table contains three records, the output would be:

css

Copy code

Connected to the database successfully.

ID Name Age Grade

1 Alice 14 A

2 Bob 15 B

3 Charlie 14 A

**JDBC Connection URL Format**

The JDBC connection URL format for MySQL databases is typically:

bash

Copy code

jdbc:mysql://hostname:port/databasename

In this example:

* hostname is localhost (your local machine).
* port is 3306, the default MySQL port.
* databasename is school.

**Advantages of JDBC**

1. **Database Independence**: Provides a standard API for interaction with various databases.
2. **Java Integration**: JDBC integrates seamlessly with Java applications, allowing for powerful, data-driven applications.
3. **Rich SQL Support**: Allows execution of any SQL statements, including queries, updates, and stored procedures.

**Common JDBC Classes and Interfaces**

* **DriverManager**: Manages JDBC drivers and handles database connections.
* **Connection**: Represents a session with a specific database.
* **Statement**: Used for executing static SQL statements.
* **PreparedStatement**: A subclass of Statement, used for executing precompiled SQL statements with parameters.
* **CallableStatement**: Used to call stored procedures.
* **ResultSet**: Represents the result of a query, with methods to access each row and column.

JDBC provides a flexible and robust way to connect Java applications to various databases, making it an essential tool for developing data-driven applications in Java.

Top of Form

Bottom of Form