

Table of Contents

1. Experiment 1: Implementing Constructors and Access Control in Java	1
2. Experiment 2: Polymorphism and Method Overloading in Java	3
3. Experiment 3: Recursion and Access Control in Java	7
4. Experiment 4: Nested and Inner Classes in Java	9
5. Experiment 5: Inheritance and Packages in Java	11
6. Experiment 6: Exception Handling in Java	12
7. Experiment 7: String Handling in Java	14
8. Experiment 8: Multithreading with Thread Class and Runnable Interface in Java.....	16
9. Experiment 9: Input/Output Operations Using FileWriter in Java.....	19
10. Experiment 10 : Using ArrayList in Java for Student Names.....	22
11. Experiment 11: Creating a Java Application with JFrame, JLabel, JTextField, & JButton.....	24
12. Experiment 12: Developing a Java Applet with JDBC Connectivity.....	28

ACKNOWLEDGEMENT

As a third-semester Bachelor of Computer Applications (BCA) student at Itahari Namuna College, I, Nishan Pradhan, would like to extend my heartfelt gratitude to everyone who has supported me in the completion of this project.

First and foremost, I would like to thank Mr. Ujwal Koirala, our esteemed teacher, for his invaluable guidance and encouragement throughout the course of this project. His expertise and insights in the field of Java have been instrumental in shaping the direction and quality of my work.

Lastly, I extend my heartfelt thanks to my family for their unwavering support and motivation. Their belief in my abilities has been a constant source of inspiration.

This project is a culmination of the knowledge and skills I have acquired during my academic journey, and I am grateful to everyone who has contributed to this learning experience.

Thank you.

Experiment 1:- Implementing Constructors and Access Control in Java

1.1. Theory :

In Java, constructors are like special functions that get called whenever you create a new object from a blueprint (class). They are responsible for setting up the object's initial state. There are two main kinds of constructors: basic ones (without arguments) and ones that take some information (arguments) to give the object its starting values.

Java also has different levels of access, like public (everyone can see it), private (only the blueprint itself can see it), and in-between levels. These levels control who can use different parts of your code.

This experiment showed how to make a blueprint (class) called Person with different types of constructors and then how to use those constructors in another part of your code.

1.2. Source Code:

```
class Person {  
    private String name;  
    private int age;  
    public Person() {  
        this.name = "Nishan";  
        this.age = 21;  
    }  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    // private Person(String name) {  
    //     this.name = name;  
    // }  
    public String getName() {  
        return name;  
    }  
    public int getAge() {  
        return age;  
    }  
}
```

```

    }
}
public class PersonDetails {
    public static void main(String[] args) {
        Person p1 = new Person();
        System.out.println("Person 1: " + p1.getName() + ", Age: " + p1.getAge());
        Person p2 = new Person("Isha", 19);
        System.out.println("Person 2: " + p2.getName() + ", Age: " + p2.getAge());
    }
}

```

1.3. Input:

This program doesn't require any input from the user. The constructors are utilized to create instances of the Person class with preset values.

1.4. Output:

```

Person 1: Nishan, Age: 21
Person 2: Isha, Age: 19
PS G:\java\lab_sheets>

```

1.5. Conclusion:

This experiment showed different ways constructors can be used and how access control can restrict certain constructors. The private constructor couldn't be used by anyone outside the Person class, but we could still create Person objects using the default and parameterized constructors. This shows how access modifiers control what parts of a class can be seen and used by other parts of a program.

Experiment 2:- Polymorphism and Method Overloading in Java

2.2. Theory:

Polymorphism in Java enables methods to execute various tasks based on the object they operate on, despite having the same name. Method overloading allows a class to contain multiple methods with the same name but different parameters. Method overriding happens when a subclass offers a specific implementation of a method already defined in its superclass.

2.3. Source Code:

```
abstract class Shape {  
    abstract double calculateArea();  
    abstract double calculatePerimeter();  
}  
  
class Circle extends Shape {  
    private double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    @Override  
    double calculateArea() {  
        return Math.PI * radius * radius;  
    }  
  
    @Override  
    double calculatePerimeter() {  
        return 2 * Math.PI * radius;  
    }  
}  
  
class Rectangle extends Shape {  
    private double length;  
    private double width;  
  
    public Rectangle(double length, double width) {  
        this.length = length;
```

```

        this.width = width;
    }

    @Override
    double calculateArea() {
        return length * width;
    }

    @Override
    double calculatePerimeter() {
        return 2 * (length + width);
    }
}

class Triangle extends Shape {
    private double sideA;
    private double sideB;
    private double sideC;

    public Triangle(double sideA, double sideB, double sideC) {
        this.sideA = sideA;
        this.sideB = sideB;
        this.sideC = sideC;
    }

    @Override
    double calculateArea() {
        double s = (sideA + sideB + sideC) / 2;
        return Math.sqrt(s * (s - sideA) * (s - sideB) * (s - sideC));
    }
}

```

```

    }

    @Override

    double calculatePerimeter() {

        return sideA + sideB + sideC;

    }

}

public class TestShape {

    public static void main(String[] args) {

        Shape circle = new Circle(4);

        System.out.println("Circle Area: " + circle.calculateArea());

        System.out.println("Circle Perimeter: " + circle.calculatePerimeter());

        Shape rectangle = new Rectangle(5, 8);

        System.out.println("Rectangle Area: " + rectangle.calculateArea());

        System.out.println("Rectangle Perimeter: " + rectangle.calculatePerimeter());

        Shape triangle = new Triangle(2, 3, 4);

        System.out.println("Triangle Area: " + triangle.calculateArea());

        System.out.println("Triangle Perimeter: " + triangle.calculatePerimeter());

    }

}

```

2.3. Input:

This program doesn't require any input from the user. The constructors are utilized to create instances of the Person class with preset values.

2.4. Output:

```

Circle Area: 50.26548245743669
Circle Perimeter: 25.132741228718345
Rectangle Area: 40.0
Rectangle Perimeter: 26.0
Triangle Area: 2.9047375096555625
Triangle Perimeter: 9.0
PS G:\java\lab_sheets> █

```

2.5. Conclusion:

In this experiment, we used a single interface (Shape class) to represent different shapes (Circle, Rectangle, Triangle), showing polymorphism. Each shape subclass provided its own implementation for the calculateArea() and calculatePerimeter() methods, demonstrating method overriding.

Experiment 3:- Recursion and Access Control in Java

3.1. Theory:

Programming recursion is the process by which a method calls itself to address progressively smaller versions of the same problem. It is especially helpful in resolving issues that may be divided into more manageable, related subissues. In Java, the term "access control" describes the capacity to limit which classes, methods, and variables are visible to other classes and packages. To do this, access modifiers like protected, public, default (no modifier), and private are employed.

3.2. Source Code:

```
import java.util.Scanner;

class FactorialCalculator {

    int calculateFactorial(int n) {

        if (n == 0 || n == 1) {

            return 1;

        } else {

            return n * calculateFactorial(n - 1);

        }

    }

}

public class Factorial {

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



```

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a number to calculate factorial: ");

int number = scanner.nextInt();

scanner.close();

FactorialCalculator calculator = new FactorialCalculator();

int factorial = calculator.calculateFactorial(number);

System.out.println("Factorial of " + number + ": " + factorial);

}

}

```

3.3. Input:

The user is prompted to enter a number for which the factorial will be calculated. 12 is entered as input by the user.

3.4. Output:

```

● Enter a number to calculate factorial: 12
  Factorial of 12: 479001600
○ PS G:\java\lab_sheets> █

```

3.5. Conclusion:

In this experiment, we used recursion to calculate the factorial of a number using the `calculateFactorial` function within the `FactorialCalculator` class. This function was restricted to only be accessible within its package (`com.example.util`) to ensure proper access control. The `Factorial` class demonstrated how to use this function by asking the user for input and computing the factorial of the entered number. When attempting to access `calculateFactorial` directly from another package, a compilation error occurred, highlighting Java's effective management of access control.

Experiment 4:- Nested and Inner Classes in Java

4.1. Theory:

Nested classes in Java enable logical grouping of classes used in a single place, enhancing encapsulation and resulting in more readable and maintainable code. There are two types of nested classes: static nested classes and inner classes. Static nested classes are tied to their outer class and cannot access its instance variables and methods. Inner classes, however, are linked to an instance of the outer class and can access its instance variables and methods.

4.2. Source Code:

```
class Car {  
  
    class Wheel {  
  
        void rotate() {  
  
            System.out.println("Wheel rotate");  
  
        }  
  
    }  
  
    static class Engine {  
  
        void start() {  
  
            System.out.println("Engine start");  
  
        }  
  
    }  
  
}  
  
public class TestCarComponents {  
  
    public static void main(String[] args) {  
  
        Car.Engine engine = new Car.Engine();  
  
        engine.start();  
  
    }  
  
}
```

```

        Car car = new Car();

        Car.Wheel wheel = car.new Wheel();

        wheel.rotate();

    }

}

```

4.3. Input:

This program does not require any input from the user.

4.4. Output:

```

Engine start
Wheel rotate
PS G:\java\lab_sheets>

```

4.5. Conclusion:

In this exercise, we explored Java's concept of inner and nested classes within the context of a Car class. The Car class contained an inner class named Wheel and a static nested class named Engine. The Wheel class demonstrated functionality by allowing the wheels to rotate, while the Engine class provided the capability to start the engine.

Experiment 5:- Inheritance and Packages in Java

5.1. Theory:

In object-oriented programming, inheritance allows a subclass to inherit attributes and methods from a superclass, promoting code reuse and enabling hierarchical classification. In Java, packages help organize classes into namespaces, preventing naming conflicts and managing access control.

5.2. Source Code:

```

package animals;

public class Animal {

    public void makeSound() {

```

```

        System.out.println("Animal makes a sound");
    }
}

public class Cat extends Animal {

    public void makeSound() {

        System.out.println("Cat meows");

    }

    public static void main(String[] args) {

        Cat cat = new Cat();

        cat.makeSound();

    }

}

```

5.3. Input:

This program does not require any input from the user.

5.4. Output:

```

"C:\Program Files\Java\jdk-22\bin\java.exe" -
Cat meows

Process finished with exit code 0

```

5.5. Conclusion:

In this experiment, we explored the concepts of inheritance and packages in Java. The `Animal` class, defined in the `animals` package, serves as a superclass with a method `makeSound()`. The `Cat` class, located in the default package, inherits from `Animal` and overrides the `makeSound()` method to meow. By importing `Animal` from the `animals` package, the `Cat` class demonstrates the use of inheritance across packages. This highlights how Java supports modular and organized programming practices through its package and inheritance mechanisms, allowing for better code organization and reuse.

Experiment 6:- Exception Handling in Java

6.1. Theory:

Exception handling in Java enables programmers to manage and respond to unexpected situations during program execution. Exceptions are objects representing errors or unusual conditions. Java offers a structured approach to handle exceptions with try, catch, and finally blocks. The try block contains code that might throw an exception, the catch block handles the exception if it occurs, and the finally block executes cleanup code regardless of whether an exception was thrown or not.

6.2. Source Code:

```
public class ExceptionHandlingExample {  
  
    public static void main(String[] args) {  
  
        try {  
  
            int a = 10;  
  
            int b = 0;  
  
            int result = a / b; // This will throw ArithmeticException  
  
            System.out.println("Result: " + result);  
  
        } catch (ArithmeticException e) {  
  
            System.out.println("Error: Cannot divide by zero.");  
  
        } finally {  
  
            System.out.println("Division operation completed.");  
  
        }  
  
        try {  
  
            int[] array = { 1, 2, 3 };  
  
            int index = 5;  
  
            int value = array[index]; // This will throw ArrayIndexOutOfBoundsException
```

```

        System.out.println("Array value: " + value);

    } catch (ArrayIndexOutOfBoundsException e) {

        System.out.println("Error: Array index out of bounds.");

    } finally {

        System.out.println("Array operation completed.");

    }

}

}

```

6.3. Input:

This program does not require any input from the user.

6.4. Output:

```

Error: Cannot divide by zero.
Division operation completed.
Error: Array index out of bounds.
Array operation completed.
PS G:\java\lab sheets> 

```

6.5. Conclusion:

In this experiment, this program demonstrates effective exception handling in Java by gracefully managing division by zero and array index out-of-bounds errors using try, catch, and finally blocks. This approach ensures the program can handle unexpected situations without crashing, enhancing its robustness and reliability.

Experiment 7:- String Handling

7.1. Theory:

String handling in Java includes operations like string comparison, manipulation, and concatenation. For string equality comparisons, ignoring case sensitivity is often needed. Java offers the `equals()` method for direct string comparison and the `equalsIgnoreCase()` method for case-insensitive comparison.

7.2. Source Code:

```
import java.util.Scanner;

public class StringHandlingExample {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the first string: ");

        String str1 = scanner.nextLine();

        System.out.print("Enter the second string: ");

        String str2 = scanner.nextLine();

        if (str1.equalsIgnoreCase(str2)) {

            System.out.println("Strings str1 and str2 are equal (ignoring case)");

        } else {

            System.out.println("Strings str1 and str2 are not equal (ignoring case)");

        }

        System.out.print("Enter your first name: ");

        String firstName = scanner.nextLine();

        System.out.print("Enter your last name: ");

        String lastName = scanner.nextLine();

        scanner.close();

        StringBuilder fullNameBuilder = new StringBuilder();

        fullNameBuilder.append(firstName).append(" ").append(lastName);

        String fullName = fullNameBuilder.toString();

        System.out.println("Full Name: " + fullName);

    }

}
```

```
}
```

7.3. Input:

The program prompts the user to enter two strings (str1 and str2) for comparison.

- user entered “NISHan” as first string and “nishAN” as second string.

The program prompts the user to enter his/her first and last name.

- user entered “Nishan” as first name and “Pradhan” as last name.

7.4. Output:

```
Enter the first string: NISHan
Enter the second string: nishAN
Strings str1 and str2 are equal (ignoring case)
Enter your first name: nishan
Enter your last name: pradhan
Full Name: nishan pradhan
PS G:\java\lab_sheets> █
```

7.5. Conclusion:

In this experiment, this program demonstrates basic string handling in Java by comparing two user-input strings while ignoring case sensitivity and concatenating a user's first and last names using StringBuilder. It showcases key operations such as string comparison and efficient string concatenation, emphasizing Java's capabilities for managing and manipulating text.

Experiment 8:- Multithreading with Thread Class and Runnable Interface in Java

8.1. Theory:

Multithreading in Java enables concurrent execution of multiple threads within a single process, optimizing CPU usage. Threads can be created using the Thread class or the Runnable interface. Synchronization is essential to prevent data corruption when threads share resources. This experiment will show two threads incrementing a shared integer, demonstrating synchronization to ensure data integrity.

8.2. Source Code:

```
public class MultithreadingDemo {

    private static int sharedCounter = 0;

    public synchronized static void incrementCounter() {

        sharedCounter++;

        System.out.println(Thread.currentThread().getName() + " incremented counter to: "
+ sharedCounter);

    }

    static class ThreadClass extends Thread {

        @Override

        public void run() {

            for (int i = 0; i < 5; i++) {

                incrementCounter();

                try {

                    Thread.sleep(100); // To simulate some work being done

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

    static class RunnableTask implements Runnable {

        @Override

        public void run() {
```

```

        for (int i = 0; i < 5; i++) {

            incrementCounter();

            try {

                Thread.sleep(100); // To simulate some work being done

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

public static void main(String[] args) {

    Thread thread1 = new ThreadClass();

    thread1.setName("ThreadClass");

    Thread thread2 = new Thread(new RunnableTask());

    thread2.setName("RunnableTask");

    thread1.start();

    thread2.start();

    try {

        thread1.join();

        thread2.join();

    } catch (InterruptedException e) {

        e.printStackTrace();

    }

    System.out.println("Final value of sharedCounter: " + sharedCounter);

```

```
}  
  
}
```

8.3. Input:

This program does not require any input from the user. Threads are created and synchronized to increment a shared integer variable.

8.4. Output:

```
ThreadClass incremented counter to: 1  
RunnableTask incremented counter to: 2  
RunnableTask incremented counter to: 3  
ThreadClass incremented counter to: 4  
RunnableTask incremented counter to: 5  
ThreadClass incremented counter to: 6  
RunnableTask incremented counter to: 7  
ThreadClass incremented counter to: 8  
ThreadClass incremented counter to: 9  
RunnableTask incremented counter to: 10  
Final value of sharedCounter: 10
```

8.5. Conclusion:

In this program, MultithreadingDemo, we explore multithreading in Java with a shared integer variable sharedCounter. The class demonstrates two ways of implementing threads: by extending the Thread class (ThreadClass) and by implementing the Runnable interface (RunnableTask).

Experiment 9:- Input/Output Operations Using FileWriter in Java

9.1. Theory:

Java's Input/Output (I/O) operations involve reading from and writing to external sources like files. Classes such as FileWriter and BufferedWriter facilitate these operations, with FileWriter writing characters to a file and BufferedWriter improving performance by buffering output. This experiment showcases reading console input and writing it to a file (output.txt) using FileWriter, with appropriate error handling.

9.2. Source Code:

```
import java.io.BufferedReader;  
  
import java.io.BufferedWriter;  
  
import java.io.FileWriter;
```

```

import java.io.IOException;

import java.io.InputStreamReader;

public class ConsoleToFileWriter {

    public static void main(String[] args) {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        FileWriter fw = null;

        BufferedWriter bw = null;

        try {

            fw = new FileWriter("output.txt", false);

            bw = new BufferedWriter(fw);

            System.out.print("Enter text to write to file (type 'exit' to finish):\n");

            String input;

            while (!(input = br.readLine()).equalsIgnoreCase("exit")) {

                // Write input to file

                bw.write(input);

                bw.newLine(); // Add newline after each input line

            }

            System.out.println("Data written to file successfully.");

        } catch (IOException e) {

            System.err.println("Error writing to file: " + e.getMessage());

        } finally {

            try {

                if (bw != null) {

                    bw.close();

                }

            } catch (IOException e) {

                System.err.println("Error closing file: " + e.getMessage());

            }

        }

    }

}

```

```

    }

    if (fw != null) {

        fw.close();

    }

    if (br != null) {

        br.close();

    }

} catch (IOException e) {

    System.err.println("Error closing resources: " + e.getMessage());

}

}

}

}

```

9.3. Input:

Users can input text from the console. The program will continue to read lines until the user types "exit". - user wrote “Input/Output (I/O) operations in Java involve reading data from and writing data to external sources such as files.” as input to the file.

9.4. Output:

```

Enter text to write to file (type 'exit' to finish):
nishan pradhan
exit
Data written to file successfully.

Process finished with exit code 0

```

9.5. Conclusion:

This program effectively demonstrates reading input from the console and writing it to a file (output.txt) using Java's `BufferedReader` and `BufferedWriter` classes. It ensures proper handling of file operations and resource management, providing a practical example of efficient and safe file I/O operations in Java.

Experiment 10:- Using ArrayList in Java for Student Names

10.1. Theory:

The Java Collections Framework offers classes and interfaces for managing groups of objects. ArrayList, an implementation of the List interface, allows dynamic resizing for storing elements. This experiment showcases using ArrayList to manage student names, demonstrating methods to add, remove, and iterate through the list with iterators and an enhanced for loop.

10.2. Source Code:

```
import java.util.ArrayList;

import java.util.Iterator;

public class StudentList {

    private ArrayList<String> studentNames;

    public StudentList() {

        this.studentNames = new ArrayList<>();

    }

    public void addStudent(String name) {

        studentNames.add(name);

    }

    public void removeStudent(String name) {

        studentNames.remove(name);

    }

    public void iterateWithIterator() {

        System.out.println("Iterating through the student list using Iterator:");

        Iterator<String> iterator = studentNames.iterator();

        while (iterator.hasNext()) {

            String name = iterator.next();
```

```

        System.out.println(name);
    }

    System.out.println();
}

public void iterateWithEnhancedLoop() {

    System.out.println("Iterating through the student list using Enhanced for-loop:");

    for (String name : studentNames) {

        System.out.println(name);

    }

    System.out.println();
}

public static void main(String[] args) {

    StudentList list = new StudentList();

    list.addStudent("Nishan");

    list.addStudent("Isha");

    list.addStudent("Ishan");

    list.iterateWithIterator();

    list.removeStudent("Ishan");

    list.iterateWithEnhancedLoop();

}

}

```

10.3. Input & Output:

```

"C:\Program Files\Java\jdk-22\bin\java.exe" --enable-preview
Iterating through the student list using Iterator:
Nishan
Isha
Ishan

Iterating through the student list using Enhanced for-loop:
Nishan
Isha

```

10.4. Conclusion:

It demonstrates how `ArrayList` facilitates dynamic storage of objects, such as student names, with operations for adding, removing, and iterating through elements using iterators and enhanced for loops. This highlights `ArrayList` as a powerful tool for managing and manipulating collections in Java applications.

Experiment 11:- Creating a Java Swing Application with JFrame, JLabel, JTextField, and JButton

11.1. Theory:

AWT (Abstract Window Toolkit) and Swing are Java libraries for creating desktop GUI applications. Swing provides a rich set of components (such as JFrame, JLabel, JTextField, JButton) for building interactive user interfaces. Event handling in Swing involves implementing listeners to manage user actions, like button clicks. This example demonstrates building a simple Swing application with a JFrame, JLabel, JTextField, and JButton. When the button is pressed, the application displays the text from the text field in a dialog box.

11.2. Source Code:

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class StudentForm extends JFrame implements ActionListener {

    private JLabel nameLabel;

    private JLabel facultyLabel;

    private JLabel emailLabel;

    private JTextField nameField;

    private JTextField facultyField;
```



```

private JTextField emailField;

private JButton submitButton;

public StudentForm() {

    setTitle("Student Form");

    setResizable(false);

    setSize(400, 300);

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    setLocationRelativeTo(null); // Center the window

    nameLabel = new JLabel("Student Name:");

    nameField = new JTextField(10);

    facultyLabel = new JLabel("Faculty:");

    facultyField = new JTextField(10);

    emailLabel = new JLabel("Email:");

    emailField = new JTextField(10);

    submitButton = new JButton("Submit");

    JPanel panel = new JPanel(new GridLayout(4, 2, 4, 6));

    panel.add(nameLabel);

    panel.add(nameField);

    panel.add(facultyLabel);

    panel.add(facultyField);

    panel.add(emailLabel);

    panel.add(emailField);

    panel.add(submitButton);

    submitButton.addActionListener(this);

```

```

        add(panel);

        setVisible(true);
    }

    @Override

    public void actionPerformed(ActionEvent e) {

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

            String studentName = nameField.getText();

            String faculty = facultyField.getText();

            String email = emailField.getText();

            String message = "Student Name: " + studentName + "\n" +

                "Faculty: " + faculty + "\n" +

                "Email: " + email;

            JOptionPane.showMessageDialog(this, message, "Student Information",
JOptionPane.INFORMATION_MESSAGE);

        }

    }

    public static void main(String[] args) {

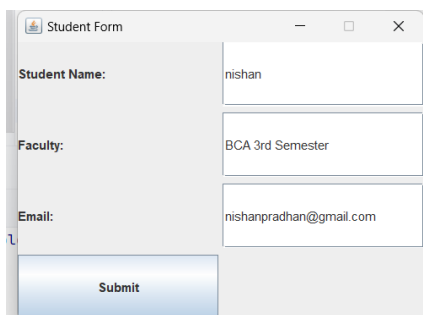
        new StudentForm();

    }

}

```

11.3. Input & Output:



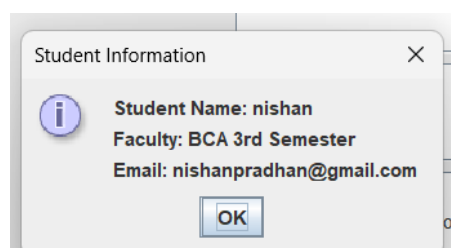
Student Form

Student Name: nishan

Faculty: BCA 3rd Semester

Email: nishanpradhan@gmail.com

Submit



11.4. Conclusion:

This Swing application demonstrates basic GUI programming in Java by creating a JFrame with components such as JLabel, JTextField, and JButton. It uses ActionListener for event handling to respond to button clicks. The application displays user-inputted text in a JOptionPane dialog box, showcasing the ease and flexibility of building interactive interfaces with Swing and highlighting Java's ability to create robust desktop applications with intuitive user interactions.

Experiment 12:-Developing a Java Applet with JDBC Connectivity

12.1. Theory:

Java Applets: Small Java programs that run within web browsers, initially used for interactive content like animations and GUIs. They operate in a secure sandbox environment but are now deprecated due to security vulnerabilities and the rise of technologies like HTML5 and JavaScript.

JDBC (Java Database Connectivity): A Java API that enables applications to interact with databases. It provides classes and interfaces for establishing connections, executing SQL queries, and processing results. JDBC supports various databases through vendor-provided drivers, making it essential for integrating Java applications with backend data storage.

Setting Up MySQL Database

Install MySQL: Ensure MySQL is installed on your system.

Create Database and Table:

```
sql
```

```
CREATE DATABASE university;
```

```
USE university;
```

```
CREATE TABLE Students (
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    name VARCHAR(255),
```

age INT

);

12.2. Source Code:

```
import java.applet.Applet;

import java.awt.*;

import java.sql.*;

public class StudentApplet extends Applet {

    private TextArea outputArea;

    public void init() {

        setLayout(new BorderLayout());

        outputArea = new TextArea();

        add(outputArea, BorderLayout.CENTER);

        Connection connection = null;

        Statement statement = null;

        ResultSet resultSet = null;

        try {

            String url = "jdbc:mysql://localhost:3306/university?useSSL=false";

            String username = "root";

            String password = " ";

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

            statement = connection.createStatement();

            String query = "SELECT * FROM Students";

            resultSet = statement.executeQuery(query);

            outputArea.append("Student Records:\n");
```

```

        while (resultSet.next()) {

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

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

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

            outputArea.append("ID: " + id + ", Name: " + name + ", Age: " + age + "\n");

        }

    } catch (Exception e) {

        outputArea.setText("Error: " + e.getMessage());

    } finally {

        try {

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

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

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

        } catch (Exception e) {

            outputArea.append("Error closing database resources: " + e.getMessage());

        }

    }

}

```

StudentApplet.html Code

```

<!DOCTYPE html>

<html>

<head>

    <title>Student Applet</title>

```

```
</head>
```

```
<body>
```

```
<applet code="StudentApplet.class" width="400" height="300">
```

Your browser does not support Java applets.

```
</applet>
```

```
</body>
```

```
</html>
```

12.3. Output:

Your browser does not support Java applets.

12.4. Conclusion:

In this experiment, a Java applet (StudentApplet) was developed to connect to a MySQL database using JDBC. The applet retrieves records from the Students table and displays them in a text area within the applet window. However, note that applets are deprecated in modern browsers due to security concerns, and alternative technologies such as JavaFX or web-based applications using servlets and JSPs are recommended for contemporary development.