

Department of Computer Science & Applications



JAVA



PRACTICAL FILE OF :

Submitted by:

Taranpreet Singh
(22046995)

Submitted to:

Dr. Rajan Manro

Index

INDEX NO	PRACTICAL NAME	PAGE NO.
1.	Single Inheritance	1-3
2.	Multilevel Inheritance	4-8
3.	TreeInherit	9-11
4.	Dynamic Method Dispatch	12-13
5.	Super Keyword	14-15
6.	Abstract Class	16-17
7.	Array	18-19
8.	Average Marks	20-21
9.	Matrix Addition	22-24
10.	String Function	25-27
11.	String Buffer	28-29
12.	Packages	30-31
13.	Interfaces	32-35
14.	Exception Handling	36-37
15.	Custom Exception	38-40
16.	Banking Transaction File Exception	41-43
17.	Multithreading	44-45
18.	Multiple Threads	46-47
19.	Inter-Thread Communication	48-50
20.	Applet Structure	51-52
21.	AWT Components	53-55
22.	Advanced Layout Managers & Event Handling	56-58
23.	JDBC	59-60
24.	Crud Operations	61-63
25.	Update and Delete	64-65

Git Repository:

Click to open

(<https://github.com/TaranMehra/JAVABASICS>)

Single Inheritance (1)

Objective

The primary objective of this program is to illustrate the concept of single inheritance, where the Derived class inherits the properties and methods of the Base class. It also aims to show how derived classes can extend the functionality of base classes by adding new methods, such as Subtract and show, while reusing methods from the base class, like Add and display.

```
class Base
{
    int a,b,sum;
    void Add(int x, int y){
        a = x;
        b = y;
        sum = a+b;
    }
    void display(){
        System.out.println("\nIn Base Class");
        System.out.println("A =" +a);
        System.out.println("B =" +b);
        System.out.println("Sum =" +sum);
    }
}

class Derived extends Base
{
    int sub;
    void Subtract(int x, int y){
        a = x;
        b = y;
        sub = a-b;
    }
    void show(){
        System.out.println("\nIn Derived Class");
        System.out.println("A = " +a);
        System.out.println("B = " +b);
        System.out.println("Subt = " +sub);
    }
}
```

```

class SingleIn
{
    public static void main(String[] args) {
        Derived obj = new Derived();

        obj.Add(10, 20);
        obj.display();

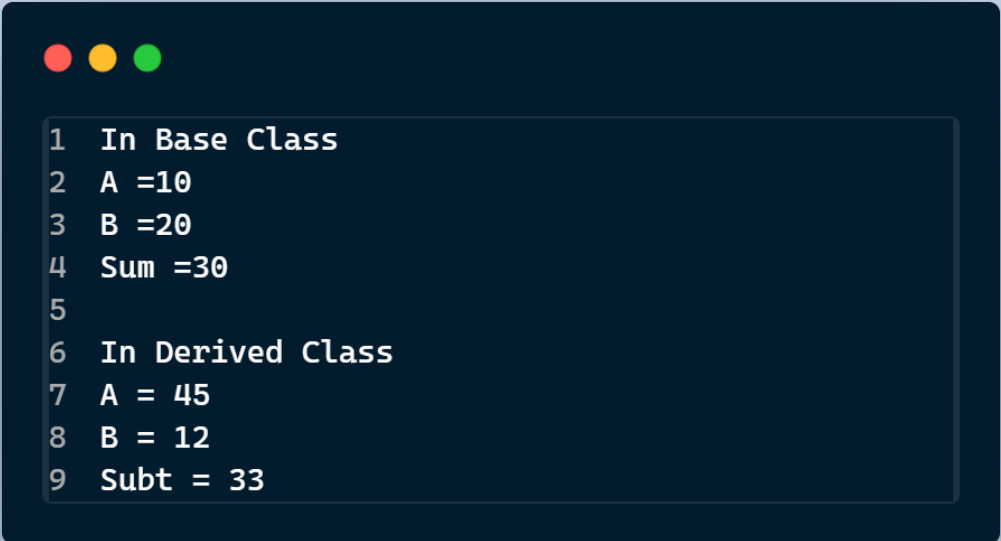
        obj.Subtract(45, 12);
        obj.show();
    }
}

```

Key Points of the Program

1. **Inheritance:**
 - This program demonstrates *single inheritance* in Java.
 - The `Derived` class extends the `Base` class, inheriting its properties and methods.
2. **Method Overriding:**
 - While no method is directly overridden in this example, the `Derived` class adds additional functionality on top of the `Base` class without altering its inherited methods.
3. **Method Invocation:**
 - Both `Add()` and `Subtract()` methods are invoked on an instance of the `Derived` class (`obj`), which shows that the derived class can access and use methods from the base class.
4. **Encapsulation of Functionality:**
 - The `Base` class encapsulates the addition operation, while the `Derived` class encapsulates the subtraction operation.
 - Each class has its own `display()` and `show()` methods for presenting output relevant to the respective operations.

Output :)



```
1 In Base Class
2 A =10
3 B =20
4 Sum =30
5
6 In Derived Class
7 A = 45
8 B = 12
9 Subt = 33
```

Multilevel Inheritance (2)

Objective: Demonstrates multilevel inheritance in Java, showing how a child class can inherit properties and methods from multiple levels of a class hierarchy. Each class performs its own arithmetic operation.

```
import java.util.Scanner;
class GrandParent {
    int num1, num2, sum;
    void Sum(int a, int b){
        num1 = a;
        num2 = b;
        sum = num1 + num2;
    }

    void Display1(){
        System.out.println("\n Grand Parent Class");
        System.out.println("Number-1: " + num1);
        System.out.println("Number-2: " + num2);
        System.out.println("Result After Addition: " + sum);
    }
}
class Parent extends GrandParent{
    int sub;
    void Subtract(int a,int b){
        num1 = a;
        num2 = b;
        sub = num1 - num2;
    }
    void Display2(){
        System.out.println("\n Parent Class");
        System.out.println("Number-1: " + num1);
        System.out.println("Number-2: " + num2);
        System.out.println("Result After Substraction: " + sub);
    }
}
class Child extends Parent{
    int mul;
```

```

    void Multiply(int a,int b){
        num1 = a;
        num2 = b;
        mul = num1 * num2;
    }
    void Display3(){
        System.out.println("\n Child Class");
        System.out.println("Number-1: " + num1);
        System.out.println("Number-2: " + num2);
        System.out.println("Result After Multiplication: " +
mul);
    }
}
class Multilevel{
    public static void main(String[] args) {
        int x,y;
        Scanner scan = new Scanner(System.in);
        try{
            System.out.println("Enter 1st Number ->");
            x = scan.nextInt();

            System.out.println("Enter 2nd Number ->");
            y = scan.nextInt();

            //create object of child class
            Child obj = new Child();

            //accessing methods of GrandParent class using Child
class obj
            obj.Sum(x,y);
            obj.Display1();

            //accessing methods of Parent class using Child class
obj
            obj.Subtract(x,y);

            //accessing methods of Class
            obj.Multiply(x,y);
            obj.Display3();

            System.out.println("\nYout Entered");
            System.out.println("First Number: "+ x );
            System.out.println("Second Number:" + y);

```

```

    }
    catch(Exception e){
        System.out.println("This Error part is Executed");
    }
    finally{
        scan.close();
    }
}
}

```

Key Points

1. Class Structure and Inheritance

- **GrandParent Class:**
 - **Purpose:** Performs addition.
 - **Methods:**
 - Sum(int a, int b): Calculates the sum of two numbers.
 - Display1(): Prints the result of the addition operation.
- **Parent Class** (inherits from GrandParent):
 - **Purpose:** Performs subtraction.
 - **Methods:**
 - Subtract(int a, int b): Calculates the difference between two numbers.
 - Display2(): Prints the result of the subtraction operation.
- **Child Class** (inherits from Parent):
 - **Purpose:** Performs multiplication.
 - **Methods:**

- `Multiply(int a, int b)`: Calculates the product of two numbers.
 - `Display3()`: Prints the result of the multiplication operation.
-

2. Method Access through Inheritance

- An instance of the Child class is created (`Child obj = new Child();`), allowing access to methods of all three classes (GrandParent, Parent, and Child).
 - This demonstrates multilevel inheritance where the Child class inherits methods from both Parent and GrandParent classes.
-

3. Key Program Functionality

- **User Input:**
 - The program prompts the user to input two integers, which are used as parameters for the arithmetic operations.
 - **Arithmetic Operations:**
 - The program performs addition, subtraction, and multiplication in the sequence:
 - **Addition** (`obj.Sum(x, y);`): Executed by the GrandParent class.
 - **Subtraction** (`obj.Subtract(x, y);`): Executed by the Parent class.
 - **Multiplication** (`obj.Multiply(x, y);`): Executed by the Child class.
 - Each result is displayed by the respective Display method in each class.
-

4. Exception Handling

- A try-catch-finally block handles any invalid input and ensures that the Scanner is closed:
 - **try**: Captures valid user input and performs calculations.
 - **catch**: Handles errors such as invalid input types.
 - **finally**: Closes the Scanner to free resources.

Output:)

```
Enter 1st Number ->
23
Enter 2nd Number ->
7

Grand Parent Class
Number-1: 23
Number-2: 7
Result After Addition: 30

Child Class
Number-1: 23
Number-2: 7
Result After Mulitiplication: 161

Yout Entered
First Number: 23
Second Number:7
```

Tree Inherit (3)

Objective

The program's primary objective is to demonstrate **inheritance in Java**, specifically:

- How subclasses (B and C) can inherit methods from a base class (A) while also having their own unique methods.
- Each subclass (B and C) extends the functionality of the base class without interfering with each other, which highlights the concept of a tree structure in inheritance.

```
class A {
    int sqr;

    void square(int p) {
        sqr = p * p;
        System.out.println("Square of " + p + " is: " + sqr);
    }
}

class B extends A {
    int cub;

    void cube(int p) {
        cub = p * p * p;
        System.out.println("Cube of " + p + " is: " + cub);
    }
}

class C extends A {
    double squareRoot;

    void squareRoot(int p) {
        squareRoot = Math.sqrt(p);
        System.out.println("Square Root of " + p + " is: " + squareRoot);
    }
}

class TreeInherit {
    public static void main(String[] args) {
        // creating object of class B
        B leftChild = new B();
        leftChild.cube(4);
        leftChild.square(4);
    }
}
```

```

// creating the object of class c
C rightChild = new C();
rightChild.squareRoot(25);
rightChild.square(25);

}
}

```

Key Points

1. Class Hierarchy and Structure:

- **Class A:**
 1. Base class with an integer field `sqr`.
 2. Defines a method `square(int p)` to calculate and print the square of an integer `p`.
- **Class B:**
 1. Subclass of A with an additional integer field `cub`.
 2. Defines a method `cube(int p)` to calculate and print the cube of an integer `p`.
- **Class C:**
 1. Subclass of A with an additional double field `squareRoot`.
 2. Defines a method `squareRoot(int p)` to calculate and print the square root of an integer `p`.

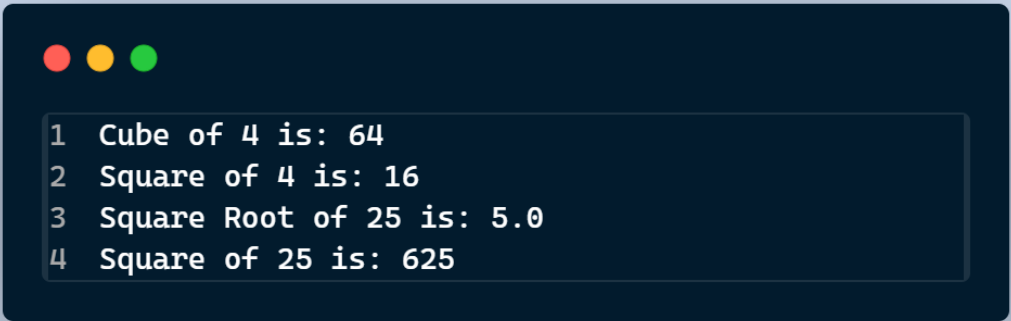
2. Inheritance Structure:

- **Tree Inheritance:**
 1. A is the parent (base) class.
 2. B and C both extend A, forming a tree structure of inheritance where B and C inherit from A but not from each other.

3. Main Method (**TreeInherit** class):

- Demonstrates the functionality of the classes:
 1. Creates an object `leftChild` of class B, then calls `cube(4)` and `square(4)` on it.
 2. Creates an object `rightChild` of class C, then calls `squareRoot(25)` and `square(25)` on it.

Output:)

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays four lines of output, each preceded by a line number. The text is in a monospaced font.

```
1 Cube of 4 is: 64
2 Square of 4 is: 16
3 Square Root of 25 is: 5.0
4 Square of 25 is: 625
```

Dynamic Method Dispatch (4)

Objective

The primary objective of this code is to demonstrate the concepts of polymorphism and dynamic dispatch in Java through method overriding. It illustrates how a superclass reference can invoke overridden methods in its subclass, allowing for dynamic behavior based on the actual object type.

```
class Base{
    public void show(){
        System.out.println("Base Class Version");
    }
}

class Derived extends Base{
    //applying the concept of polymorphism overriding
    public void show(){
        System.out.println("Derived Class Version");
    }
}

class DynamicDispatch {
    public static void main(String[] args) {
        //a method of dynamic(runtime) calling to a method

        Base reference;    //base class reference
        Base b = new Base();
        reference = b;
        reference.show();

        reference = new Derived();
        reference.show();
    }
}
```

Key Points

1. Class Definitions:

- A **Base Class** that contains a method `show()` which prints "Base Class Version".
- A **Derived Class** that extends `Base` and overrides the `show()` method to print "Derived Class Version".

2. Dynamic Dispatch Example:

- A reference of type `Base` is declared, which can point to objects of type `Base` or its subclasses.
- The code showcases how a base class reference can first point to a `Base` object, and then to a `Derived` object.

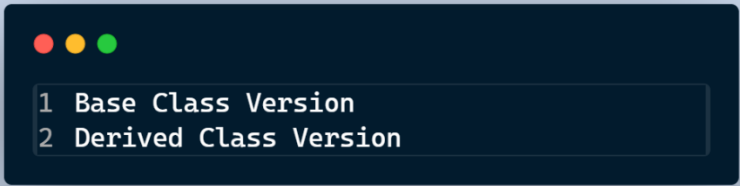
3. Calling `show()` Method:

- When the `show()` method is called while referencing a `Base` object, it invokes the `show()` method from the `Base` class.
- When the reference points to an instance of `Derived`, the overridden `show()` method in `Derived` is called, demonstrating dynamic dispatch.

4. Key Concepts Illustrated:

- **Polymorphism:** The ability to treat objects of different types (in this case, `Derived` as `Base`) and call overridden methods.
- **Method Overriding:** The `Derived` class provides its specific implementation of the `show()` method.
- **Dynamic Dispatch:** The method executed is determined at runtime based on the actual object that the reference points to, showcasing runtime polymorphism.

Output:)



```
1 Base Class Version
2 Derived Class Version
```

Super Keyword (5)

Objective

The objective of this program is to demonstrate **inheritance** in Java, where a subclass (*Instance*) inherits properties from a superclass (*Instance2*). It also showcases how the subclass can initialize its own variables as well as those of the superclass and use `super` to access the superclass's methods.

```
class Instance2 {  
    String fullname;  
  
    public Instance2(String firstName) {  
        this.fullname = firstName + "Singh";  
    }  
  
    void display() {  
        System.out.println("The value of fullname : " + fullname);  
    }  
}  
  
class Instance extends Instance2 {  
    String instanceVariable;  
  
    Instance(String value) {  
        super(value);  
        tellName(value);  
    }  
  
    void tellName(String name) {  
        this.instanceVariable = name;  
    }  
  
    void display() {  
        System.out.println("\n The value of instanceVariable : " +  
instanceVariable);  
        super.display();  
    }  
}
```



```

    public static void main(String[] args) {
        Instance ins;
        ins = new Instance("Taranpreet ");
        ins.display();
    }
}

```

Key Points

1. **Class Instance2 (Superclass):**
 - Contains an instance variable `fullname`.
 - Has a constructor that takes a `firstName` parameter, appends "Singh" to it, and assigns it to `fullname`.
 - Has a `display` method that prints the value of `fullname`.
2. **Class Instance (Subclass):**
 - Inherits from `Instance2`.
 - Contains an additional instance variable `instanceVariable`.
 - The constructor calls `super(value) ;`, which invokes the superclass's constructor to initialize `fullname`.
 - It also calls a method `tellName` to set `instanceVariable` based on the provided value.
3. **Method Overriding:**
 - The `display` method is overridden in `Instance` to display both `instanceVariable` and `fullname`.
 - The overridden `display` method in `Instance` calls `super.display()` to execute the superclass's version of `display` and print `fullname`.

Output:)



```

1 The value of instanceVariable : Taranpreet
2 The value of fullname : Taranpreet Singh

```

Abstract Class (6)

Objective

The objective of this program is to demonstrate the use of **abstract classes** and **method overriding** in Java. It shows how a subclass (Dog) can inherit an abstract class (Animal), implement abstract methods, and use concrete methods from the abstract class.

```
abstract class Animal {
    String name;

    // Constructor
    public Animal(String name) {
        this.name = name;
    }

    // Abstract method (no implementation)
    abstract void sound();

    // Concrete (non-abstract) method
    public void sleep() {
        System.out.println(name + " is sleeping.");
    }
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

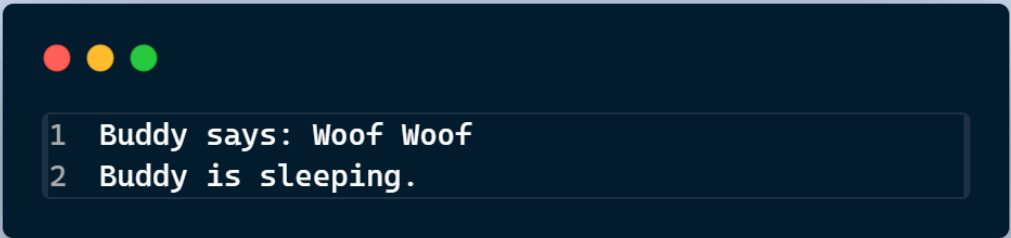
    // Implementing the abstract method
    @Override
    void sound() {
        System.out.println(name + " says: Woof Woof");
    }
}

class Abstract {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.sound();
        dog.sleep();
    }
}
```

Key Points

1. **Abstract Class `Animal`:**
 - Contains an instance variable `name`.
 - Has an **abstract method** `sound()`, which must be implemented by any subclass.
 - Has a **concrete method** `sleep()`, providing shared functionality that subclasses can use without modification.
2. **Constructor in Abstract Class:**
 - The constructor `Animal(String name)` initializes the `name` attribute. Though abstract classes cannot be instantiated, they can have constructors that are called when a subclass is instantiated.
3. **Subclass `Dog`:**
 - `Dog` extends `Animal` and implements the abstract `sound()` method to provide specific behavior for `Dog`.
 - Calls `super(name)` ; in its constructor to initialize the `name` attribute using the `Animal` constructor.
4. **Method Overriding:**
 - `Dog` overrides the abstract `sound()` method from `Animal`, providing a specific implementation for dogs.
 - The `sleep()` method is inherited as is from `Animal`, so `Dog` can use it directly.
5. **Main Class `Abstract`:**
 - Creates an instance of `Dog` with the name "Buddy".
 - Calls `dog.sound()` to print the specific behavior of the `Dog` class.
 - Calls `dog.sleep()` to invoke the inherited `sleep()` method from `Animal`.

Output:)

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. It displays two lines of output: "1 Buddy says: Woof Woof" and "2 Buddy is sleeping.".

```
1 Buddy says: Woof Woof
2 Buddy is sleeping.
```

Array (7)

Objective

The main objective of this program is to demonstrate the use of arrays in Java by creating arrays of different data types (integer, character, and string) and accessing elements within these arrays using their indices.

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

        // int array
        int marks[] = { 10, 20, 30, 40, 50 };

        // accessing element via index
        System.out.println("Marks-3 : \t" + marks[2]);

        // char array
        char name[] = { 'T', 'A', 'R', 'A', 'N' }; //
{"T", "A", "R", "A", "N"};
        System.out.println("Last Letter : \t" + name[name.length - 1]);

        // string array
        String names[] = { "Taranpreet Singh", "Hasan", "Joi", "Arjun" };
        System.out.println("The name at 3 inex: \t" + names[2]);

    }
}
```

Key Points

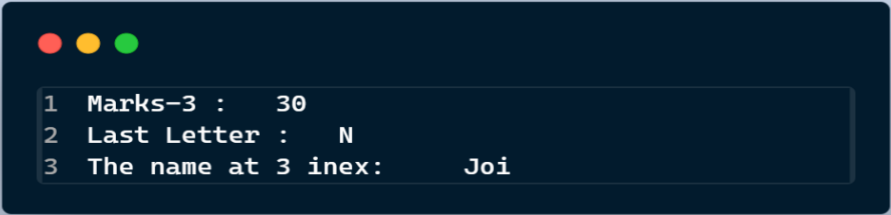
1. Array Declaration and Initialization:

- **Integer Array:** An integer array `marks` is created and initialized with values {10, 20, 30, 40, 50}.
- **Character Array:** A character array `name` is created and initialized with the letters {'T', 'A', 'R', 'A', 'N'}.
- **String Array:** A string array `names` is created and initialized with names such as "Taranpreet Singh", "Hasan", "Joi", and "Arjun".

2. Accessing Array Elements via Index:

- **Integer Array:** The program retrieves and prints the third element (index 2) from the `marks` array.
- **Character Array:** The program retrieves and prints the last element in the `name` array using `name.length - 1` as the index.
- **String Array:** The program retrieves and prints the third element (index 2) from the `names` array.

Output:)

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays three lines of output, each preceded by a line number in a light blue font.

```
1 Marks-3 : 30
2 Last Letter : N
3 The name at 3 index: Joi
```

Average Marks (8)

Objective

The main objective of this program is to calculate the average and total marks of a student for five subjects. The program allows user input for the marks, calculates the total and average, and handles any input errors using exception handling.

```
import java.util.Scanner;

class AverageMarks {
    public static void main(String[] args) {
        int marks[] = new int[5];
        int total = 0;
        double average = 0;
        int n = marks.length;
        Scanner sc = new Scanner(System.in);

        try{
            System.out.println("Enter Marks 5 Subjects\n");
            for(int index=0; index<marks.length; index++){

                marks[index] = sc.nextInt();
            }

            System.out.println("\n MARKS ARE \t");

            for(int index=0; index<marks.length; index++){
                System.out.println("Marks are: "+marks[index]);
                total = total + marks[index];
            }

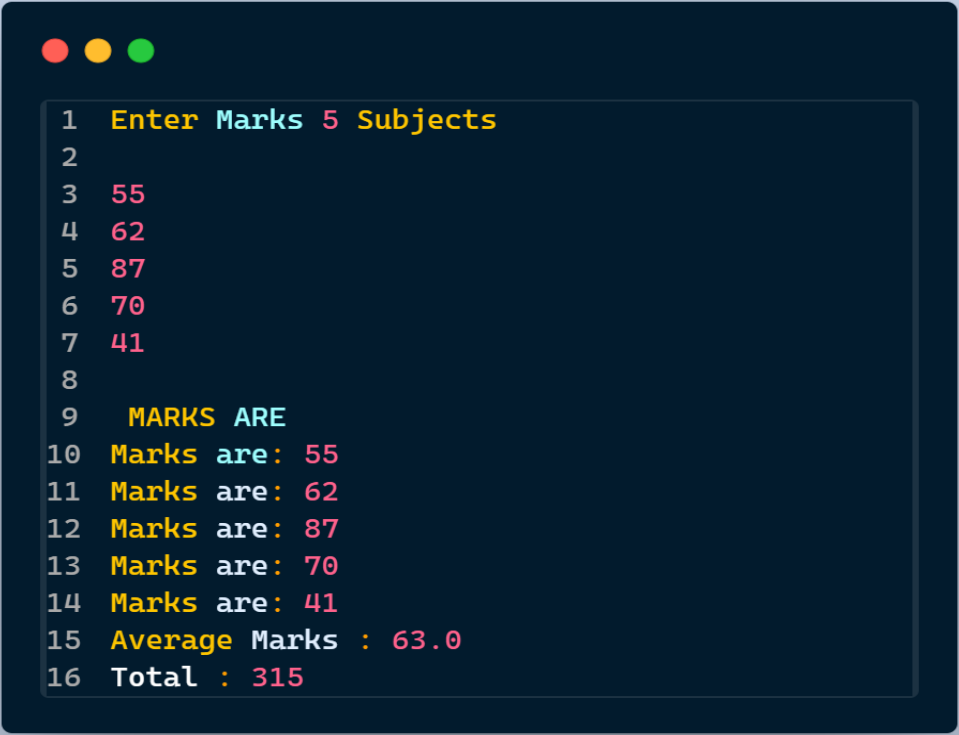
            average = total/n;
            System.out.println("Average Marks : "+average);
            System.out.println("Total : "+total);

        }
        catch(Exception e){
            System.out.println("You Got the error");
            if(!sc.hasNextInt()){
                System.out.println("You have not entered the Integer");
            }
        }
    }
}
```

Key Points

1. **Array Declaration:**
 - An integer array `marks` of size 5 is declared to store marks for five subjects.
2. **User Input:**
 - The program uses a `Scanner` object to prompt the user to enter marks for each subject. It reads inputs in a loop, filling the `marks` array with user-provided integers.
3. **Error Handling with Try-Catch:**
 - The `try` block captures any potential input errors.
 - The `catch` block is used to handle exceptions, printing an error message if the user enters a non-integer value.
 - Inside the `catch` block, `sc.hasNextInt()` is used to check if the input is not an integer, providing a specific error message.
4. **Calculation of Total and Average:**
 - After all marks are entered, the program iterates through the `marks` array to calculate the `total` sum of marks.
 - The `average` is calculated by dividing `total` by `n`, where `n` is the length of the array (5 in this case).

Ouput:)



```
1  Enter Marks 5 Subjects
2
3  55
4  62
5  87
6  70
7  41
8
9  MARKS ARE
10 Marks are: 55
11 Marks are: 62
12 Marks are: 87
13 Marks are: 70
14 Marks are: 41
15 Average Marks : 63.0
16 Total : 315
```

Matrix Addition (9)

Objective

The main objective of this program is to perform matrix addition by adding corresponding elements from two matrices of the same dimensions. It takes matrix dimensions and elements as input from the user and displays the resulting matrix after addition.

```
import java.util.Scanner;

public class MatrixAddition {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input the dimensions of the matrices
        System.out.print("Enter the number of rows: ");
        int rows = scanner.nextInt();
        System.out.print("Enter the number of columns: ");
        int cols = scanner.nextInt();

        int[][] matrix1 = new int[rows][cols];
        int[][] matrix2 = new int[rows][cols];
        int[][] result = new int[rows][cols];

        // Input elements for the first matrix
        System.out.println("Enter elements of first matrix:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print("Element [" + i + "][" + j + "]: ");
                matrix1[i][j] = scanner.nextInt();
            }
        }

        // Input elements for the second matrix
        System.out.println("Enter elements of second matrix:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print("Element [" + i + "][" + j + "]: ");
                matrix2[i][j] = scanner.nextInt();
            }
        }

        // Adding the matrices
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }
    }
}
```



```

    }

    // Displaying the result
    System.out.println("Result of matrix addition:");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            System.out.print(result[i][j] + " ");
        }
        System.out.println();
    }

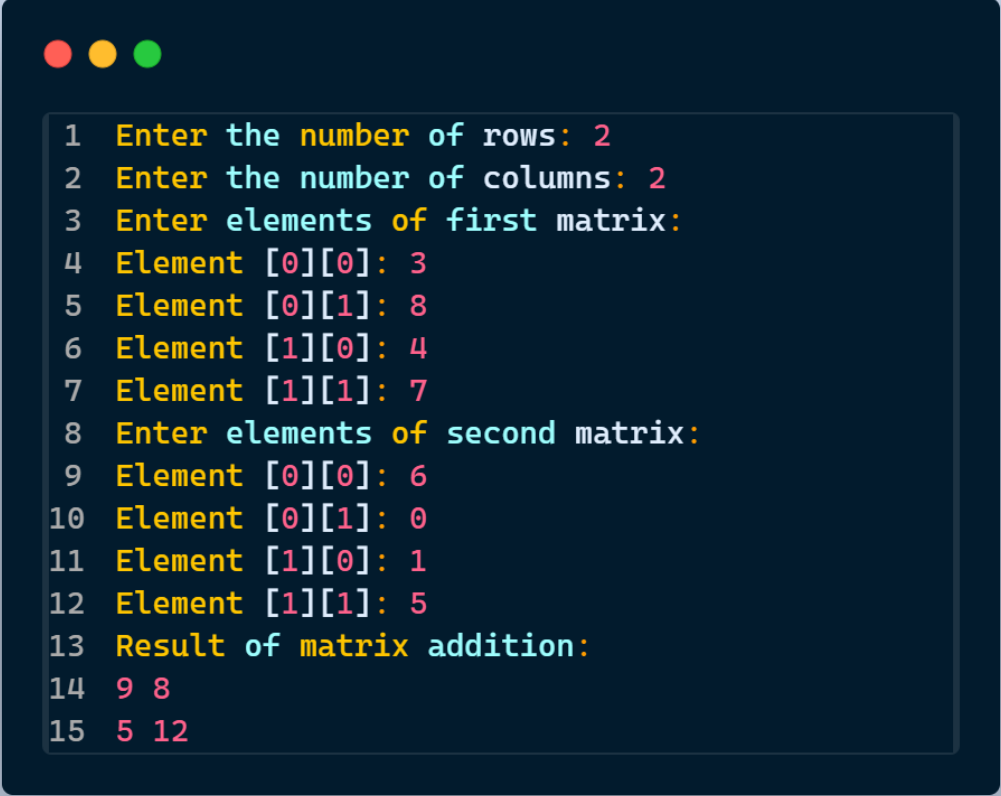
    scanner.close();
}
}

```

Key Points

1. **Input Dimensions:**
 - The program begins by prompting the user to enter the number of rows and columns for the matrices. Both matrices must have the same dimensions for addition.
2. **Matrix Initialization:**
 - Three 2D integer arrays (`matrix1`, `matrix2`, and `result`) are created with the specified dimensions.
 - `matrix1` and `matrix2` will hold the user-input values, and `result` will store the sum of the corresponding elements from `matrix1` and `matrix2`.
3. **Input Matrix Elements:**
 - The program prompts the user to enter each element of `matrix1` and `matrix2` individually, iterating through rows and columns to populate the matrices.
4. **Matrix Addition:**
 - Each element from `matrix1` is added to the corresponding element in `matrix2`, and the sum is stored in the `result` matrix at the same index.
5. **Display Result:**
 - The program iterates through the `result` matrix and prints each element in a formatted grid, showing the final matrix after addition.
6. **Closing Resources:**
 - `scanner.close()` is used to close the `Scanner` object and release any resources it may be holding.

Output:)



```
1 Enter the number of rows: 2
2 Enter the number of columns: 2
3 Enter elements of first matrix:
4 Element [0][0]: 3
5 Element [0][1]: 8
6 Element [1][0]: 4
7 Element [1][1]: 7
8 Enter elements of second matrix:
9 Element [0][0]: 6
10 Element [0][1]: 0
11 Element [1][0]: 1
12 Element [1][1]: 5
13 Result of matrix addition:
14 9 8
15 5 12
```

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays the output of a Python program for matrix addition. The program prompts the user to enter the number of rows (2) and columns (2). It then prompts for the elements of the first matrix, which are 3, 8, 4, and 7. Next, it prompts for the elements of the second matrix, which are 6, 0, 1, and 5. Finally, it displays the result of the matrix addition as two rows: [9, 8] and [5, 12].

String Function (10)

Objective

The main objective of this program is to demonstrate various string operations in Java, including concatenation, length calculation, character access, substring extraction, case conversion, substring search, and word replacement. The program uses the string "Hello, Taranpreet Singh!" for these operations.

```
public class DemoString {  
  
    public static void main(String[] args) {  
        // Initializing a string  
        String str = "Hello, Taranpreet Singh!";  
  
        // Display the string  
        System.out.println("Original String: " + str);  
  
        // Get the length of the string  
        int length = str.length();  
        System.out.println("Length of the string: " + length);  
  
        // Concatenate two strings  
        String str2 = " Welcome to Java!";  
        String concatenatedStr = str + str2;  
        System.out.println("Concatenated String: " + concatenatedStr);  
  
        // Character at a specific index  
        char charAt5 = str.charAt(5);  
        System.out.println("Character at index 5: " + charAt5);  
  
        // Extracting a substring  
        String substring = str.substring(7, 20);  
        System.out.println("Substring from index 7 to 20: " + substring);  
  
        // Convert to uppercase  
        String upperStr = str.toUpperCase();  
        System.out.println("Uppercase String: " + upperStr);  
  
        // Convert to lowercase  
        String lowerStr = str.toLowerCase();  
        System.out.println("Lowercase String: " + lowerStr);  
    }  
}
```

```

        // Check if the string contains a certain word
        boolean containsWord = str.contains("Taranpreet");
        System.out.println("Does the string contain 'Taranpreet'? " +
containsWord);

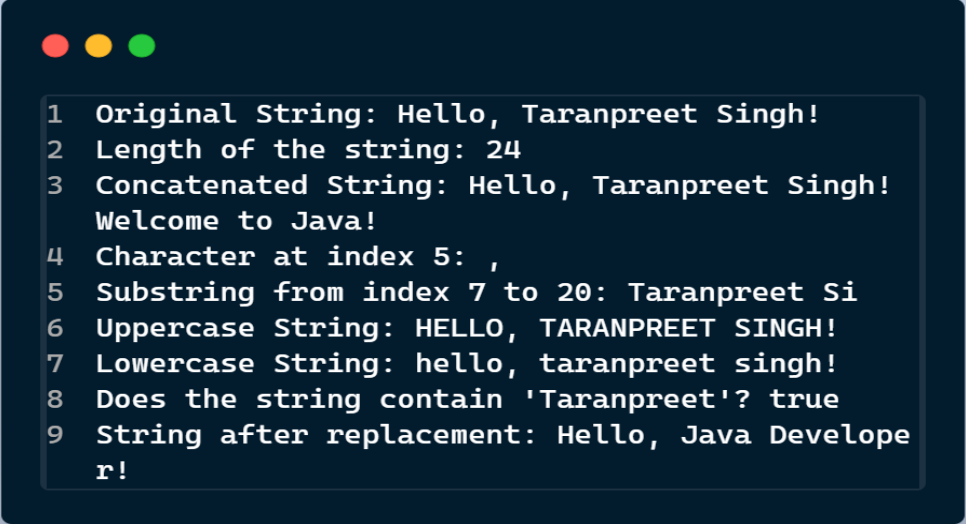
        // Replace a word in the string
        String replacedStr = str.replace("Taranpreet Singh", "Java
Developer");
        System.out.println("String after replacement: " + replacedStr);
    }
}

```

Key Points

1. **String Initialization:**
 - The program initializes a string `str` with the value "Hello, Taranpreet Singh!".
2. **String Length:**
 - `str.length()` is used to determine the length of the string, which is then printed.
3. **String Concatenation:**
 - Another string, " Welcome to Java!", is concatenated with `str` using the `+` operator, creating `concatenatedStr`, which is then printed.
4. **Character Access:**
 - `str.charAt(5)` retrieves the character at index 5 (the sixth character) from `str`.
5. **Substring Extraction:**
 - `str.substring(7, 20)` extracts a substring from index 7 to 19 (not including 20), resulting in "Taranpreet Singh".
6. **Case Conversion:**
 - `str.toUpperCase()` and `str.toLowerCase()` convert the string to uppercase and lowercase, respectively.
7. **String Contains Check:**
 - `str.contains("Taranpreet")` checks if `str` includes the substring "Taranpreet" and returns a boolean result.
8. **String Replacement:**
 - `str.replace("Taranpreet Singh", "Java Developer")` replaces "Taranpreet Singh" with "Java Developer" in `str`, creating `replacedStr`.

Output:)



```
1 Original String: Hello, Taranpreet Singh!
2 Length of the string: 24
3 Concatenated String: Hello, Taranpreet Singh!
  Welcome to Java!
4 Character at index 5: ,
5 Substring from index 7 to 20: Taranpreet Si
6 Uppercase String: HELLO, TARANPREET SINGH!
7 Lowercase String: hello, taranpreet singh!
8 Does the string contain 'Taranpreet'? true
9 String after replacement: Hello, Java Developer!
```

String Buffer (11)

Objective

The objective of this program is to demonstrate various operations using the `StringBuffer` class in Java, including appending, inserting, replacing, deleting, reversing, and retrieving the length of the string buffer. `StringBuffer` is a mutable class, which makes it efficient for performing multiple modifications on strings.

```
public class DemoStringBuffer {  
  
    public static void main(String[] args) {  
        // Initializing a StringBuffer  
        StringBuffer buffer = new StringBuffer("Hello, Taranpreet  
Singh!");  
  
        // Display the original buffer  
        System.out.println("Original StringBuffer: " + buffer);  
  
        // Append a string  
        buffer.append(" Welcome to Java!");  
        System.out.println("After append: " + buffer);  
  
        // Insert a string at a specific index  
        buffer.insert(7, "Mr. ");  
        System.out.println("After insert: " + buffer);  
  
        // Replace a part of the string  
        buffer.replace(7, 11, "Dr."); // Replaces "Mr." with "Dr."  
        System.out.println("After replace: " + buffer);  
  
        // Delete a portion of the string  
        buffer.delete(7, 11); // Deletes "Dr. "  
        System.out.println("After delete: " + buffer);  
  
        // Reverse the entire buffer  
        buffer.reverse();  
        System.out.println("After reverse: " + buffer);  
  
        // Get the length of the buffer  
        int length = buffer.length();  
        System.out.println("Length of StringBuffer: " + length);  
  
        // Set the buffer back to original by reversing it again  
        buffer.reverse();  
    }  
}
```

```

        System.out.println("Reset to original form (after second
reverse): " + buffer);
    }
}

```

Key Points

1. **StringBuffer Initialization:**
 - o The program initializes a `StringBuffer` with the value "Hello, Taranpreet Singh!".
2. **Append Operation:**
 - o `buffer.append(" Welcome to Java!")` adds the text " Welcome to Java!" to the end of the original `StringBuffer`.
3. **Insert Operation:**
 - o `buffer.insert(7, "Mr. ")` inserts "Mr. " at index 7, adding a title to the name.
4. **Replace Operation:**
 - o `buffer.replace(7, 11, "Dr.")` replaces the text from index 7 to 10 (i.e., "Mr.") with "Dr.", updating the title.
5. **Delete Operation:**
 - o `buffer.delete(7, 11)` removes the substring "Dr. " from the buffer, starting from index 7 to 10.
6. **Reverse Operation:**
 - o `buffer.reverse()` reverses the entire content of the `StringBuffer`.
7. **Retrieve Length:**
 - o `buffer.length()` returns the current length of the `StringBuffer`.
8. **Reset to Original:**
 - o Reversing the buffer a second time returns it to its original form.

Output:)

```

1 Original StringBuffer: Hello, Taranpreet Singh!
2 After append: Hello, Taranpreet Singh! Welcome to Java!
3 After insert: Hello, Mr. Taranpreet Singh! Welcome to Java!
4 After replace: Hello, Dr.Taranpreet Singh! Welcome to Java!
5 After delete: Hello, aranpreet Singh! Welcome to Java!
6 After reverse: !avaJ ot emocleW !hgniS teerpna ,olleH
7 Length of StringBuffer: 40
8 Reset to original form (after second reverse): Hello, aranpreet Singh! Welcome to Java!

```

Packages (12)

Objective

The objective of this program is to demonstrate how to create and use a Java package from a separate directory. By defining a `graphics` package containing a `Rectangle` class, we explore how to organize classes in a structured way, import them from other files, and compile and run the program from outside the package directory.

Src/graphics/Rectangle.java

```
package graphics;

public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}
```

Src/MainProgram.java

```
import graphics.Rectangle;

public class MainProgram {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10, 20);
        System.out.println("Area of the rectangle: " + rect.getArea());
    }
}
```


Output:)



Interfaces (13)

Objective

The objective of this program is to demonstrate various concepts in Java related to interfaces and abstract classes. Specifically, it shows how to define interfaces, implement multiple interfaces to achieve multiple inheritance, use variables in interfaces, and differentiate between interfaces and abstract classes. The example includes a `Shape` interface, a `ThreeDimensional` interface, a concrete class implementing multiple interfaces (`Cube`), and an abstract class (`ColoredShape`) that provides additional functionality.

```
// Defining an Interface for Shape
interface Shape {
    // Variables in interfaces are implicitly public, static, and final
    String TYPE = "Geometric Shape"; // Constant variable in an interface

    // Abstract method - by default, public and abstract
    double calculateArea();
}

// Another Interface for 3D Shapes
interface ThreeDimensional {
    double calculateVolume(); // Abstract method to calculate volume
}

// Implementing Interfaces and Achieving Multiple Inheritance
class Cube implements Shape, ThreeDimensional {
    private double side;

    public Cube(double side) {
        this.side = side;
    }

    // Implementing the calculateArea method from Shape
    @Override
    public double calculateArea() {
        return 6 * side * side;
    }

    // Implementing the calculateVolume method from ThreeDimensional
    @Override
    public double calculateVolume() {
        return side * side * side;
    }
}
```

```

}

// Abstract Class for Shapes with Color Property
abstract class ColoredShape {
    protected String color;

    public ColoredShape(String color) {
        this.color = color;
    }

    // Abstract method
    abstract void displayColor();

    // Non-abstract method
    public void setColor(String color) {
        this.color = color;
    }
}

// Class that extends the abstract class and implements the Shape
interface
class ColoredSquare extends ColoredShape implements Shape {
    private double side;

    public ColoredSquare(double side, String color) {
        super(color);
        this.side = side;
    }

    // Implementing calculateArea method from Shape interface
    @Override
    public double calculateArea() {
        return side * side;
    }

    // Implementing the abstract method from ColoredShape
    @Override
    void displayColor() {
        System.out.println("Color of the square: " + color);
    }
}

// Main Class to Test the Program
public class InterfaceDemo {
    public static void main(String[] args) {
        // Interface variables are constants and can be accessed directly
        using the interface name
        System.out.println("Shape type: " + Shape.TYPE);
    }
}

```

```

// Creating a Cube instance
Cube cube = new Cube(3);
System.out.println("Cube area: " + cube.calculateArea());
System.out.println("Cube volume: " + cube.calculateVolume());

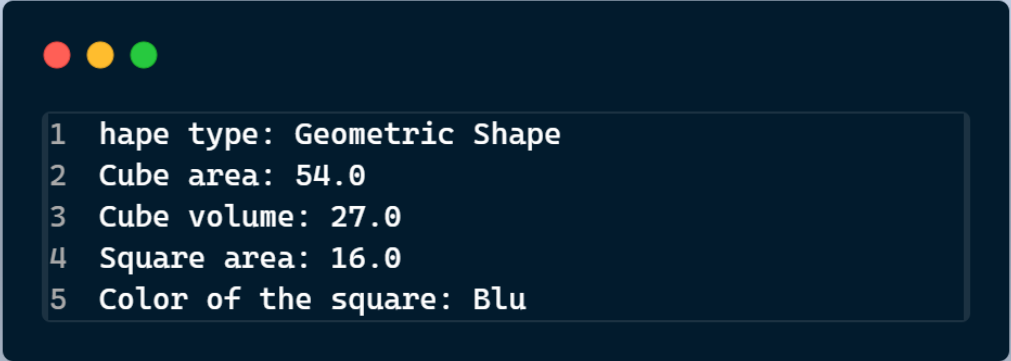
// Creating a ColoredSquare instance
ColoredSquare square = new ColoredSquare(4, "Blue");
System.out.println("Square area: " + square.calculateArea());
square.displayColor(); // Display color from abstract class
method
    }
}

```

Key Points

1. **Defining an Interface:**
 - Interfaces in Java define a contract for classes that implement them. They contain abstract methods that must be implemented by any class that adheres to the interface.
 - In this example, `Shape` and `ThreeDimensional` are interfaces with abstract methods `calculateArea` and `calculateVolume`, respectively.
2. **Variables in Interfaces:**
 - Variables in interfaces are implicitly `public`, `static`, and `final`, making them constants. They can be accessed directly using the interface name.
 - Here, `TYPE` in the `Shape` interface is a constant and is accessed in the `main` method as `Shape.TYPE`.
3. **Implementing Interfaces and Achieving Multiple Inheritance:**
 - Java allows multiple inheritance through interfaces. A class can implement multiple interfaces, fulfilling the requirements of each.
 - `Cube` demonstrates this by implementing both `Shape` and `ThreeDimensional`, thus achieving multiple inheritance. It provides implementations for both `calculateArea` and `calculateVolume`.
4. **Abstract Classes vs. Interfaces:**
 - Abstract classes in Java can contain both abstract methods (which subclasses must implement) and concrete methods (which subclasses inherit directly).
 - The `ColoredShape` abstract class defines a `color` property and methods `displayColor` (abstract) and `setColor` (concrete).
 - `ColoredSquare` extends `ColoredShape` and implements the `Shape` interface, showcasing that abstract classes can be combined with interfaces to create flexible and reusable designs.
5. **Practical Application in the `main` Method:**
 - The program demonstrates creating and using instances of classes that implement interfaces and extend abstract classes.
 - It prints the area and volume of a `Cube` instance and the area and color of a `ColoredSquare` instance, illustrating both interface and abstract class usage.

Output:)

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top-left corner. The output is displayed in a light blue monospaced font, with each line preceded by a line number from 1 to 5.

```
1 hape type: Geometric Shape
2 Cube area: 54.0
3 Cube volume: 27.0
4 Square area: 16.0
5 Color of the square: Blu
```

Exception Handling (14)

Objective

The objective of this program is to demonstrate Java's basic exception handling mechanisms, including the use of `try` and `catch` blocks to handle exceptions, as well as the `finally` block. This example shows how to catch specific exceptions and handle them gracefully, and illustrates the role of the `finally` block to ensure certain code runs regardless of whether an exception occurs.

```
public class BasicExceptionHandling {
    public static void main(String[] args) {
        int[] numbers = {10, 0};
        String input = null;

        try {
            // Attempting division and accessing an array
            System.out.println("Result: " + (numbers[0] /
numbers[1]));
            System.out.println("Array element: " + numbers[2]);
            System.out.println("Input length: " + input.length());

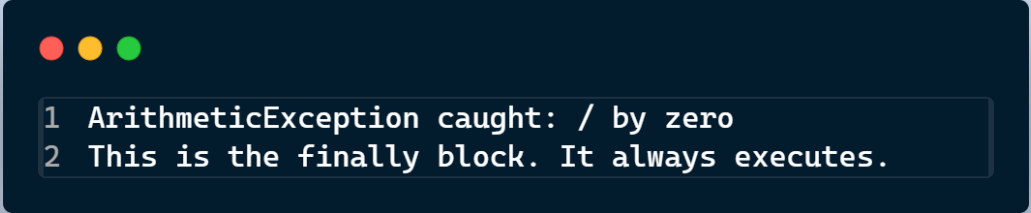
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " +
e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException
caught: " + e.getMessage());
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " +
e.getMessage());
        } finally {
            System.out.println("This is the finally block. It
always executes.");
        }
    }
}
```

Key Points

1. Exception Handling Basics with Try and Catch:

- The program uses a `try` block to attempt operations that might throw exceptions.
 - `catch` blocks handle specific exceptions, allowing the program to handle errors gracefully instead of terminating abruptly.
2. **Multiple Catch Clauses:**
- Multiple `catch` blocks are used to handle different types of exceptions that may arise in the `try` block.
 - Here, three exceptions are anticipated and handled:
 - `ArithmeticException` for division by zero.
 - `ArrayIndexOutOfBoundsException` for accessing an array index that doesn't exist.
 - `NullPointerException` for calling a method on a `null` object reference.
3. **Finally Block:**
- The `finally` block is used to execute code that should run regardless of whether an exception is thrown.
 - This block is particularly useful for resource cleanup (like closing files or releasing connections) in real-world applications.

Output:)



```
1 ArithmeticException caught: / by zero
2 This is the finally block. It always executes.
```

Custom Exception (15)

Objective

The objective of this program is to demonstrate advanced exception handling techniques in Java, focusing on **nested try-catch statements** and **custom exceptions**. This example shows how to create a custom exception to handle specific error conditions and use nested `try` blocks to handle exceptions at different levels within a program.

```
// Custom Exception Class
class InvalidValueException extends Exception {
    public InvalidValueException(String message) {
        super(message);
    }
}

public class NestedTryAndCustomException {
    public static void main(String[] args) {
        int[] numbers = {10, 5, -1}; // -1 is an invalid value for
        this example

        try {
            System.out.println("Outer try block");

            try {
                if (numbers[2] < 0) {
                    throw new InvalidValueException("Negative
                    values are not allowed: " + numbers[2]);
                }
                System.out.println("Number: " + numbers[2]);
            } catch (InvalidValueException e) {
                System.out.println("Caught custom exception: " +
                e.getMessage());
            }

            // Simulate another exception
            int result = numbers[0] / 0;
            System.out.println("Result: " + result);

        } catch (ArithmeticException e) {
```



```

        System.out.println("Caught ArithmeticException in
outer try: " + e.getMessage());
    } finally {
        System.out.println("Outer finally block executed.");
    }
}
}

```

Key Points

1. Custom Exception:

- A custom exception (`InvalidValueException`) is defined by extending the `Exception` class.
- Custom exceptions are useful for enforcing specific business logic or validation requirements.
- In this example, `InvalidValueException` is thrown if an invalid (negative) number is found in the `numbers` array.

2. Nested Try-Catch Statements:

- The program demonstrates nested `try` blocks, which allow handling specific errors at different levels of a code block.
- The inner `try` block checks for a specific validation error, while the outer `try` block handles other general exceptions, such as division by zero.
- This approach is useful when different types of exceptions need to be managed in different ways or when handling one error can lead to potential downstream issues that need further handling.

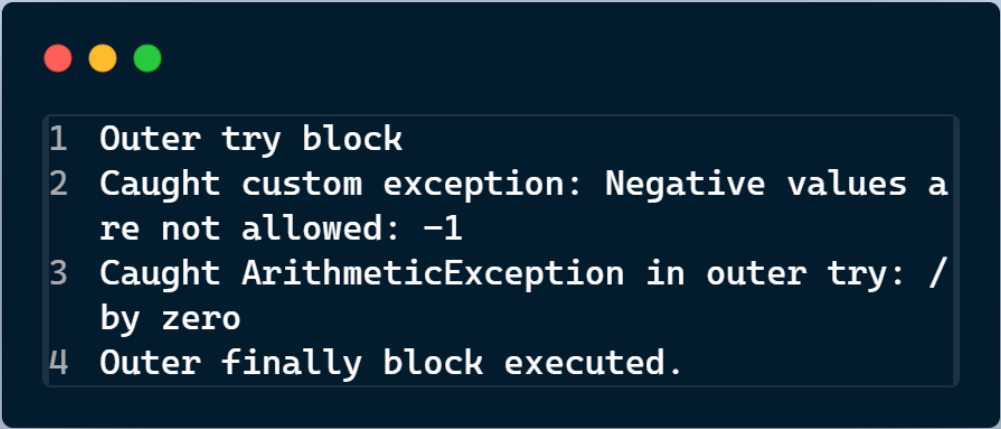
3. Finally Block in Outer Try-Catch:

- The `finally` block in the outer `try-catch` structure is used to ensure that certain actions (like resource cleanup) occur regardless of the success or failure of the code in the `try` blocks.
- This block runs even if an exception is thrown, making it reliable for final operations.

4. Demonstration of Specific Exception Handling:

- `InvalidValueException` is caught in the inner `catch` block, allowing specific handling for custom errors.
- `ArithmeticException` is caught in the outer `catch` block to handle division by zero errors separately.

Output:)

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays four lines of text, each preceded by a line number.

```
1 Outer try block
2 Caught custom exception: Negative values a
  re not allowed: -1
3 Caught ArithmeticException in outer try: /
  by zero
4 Outer finally block executed.
```

Banking Transaction (16)

Objective

The objective of this program is to demonstrate a comprehensive example of Java exception handling in a real-world scenario involving file operations and bank transactions. This program uses **nested try-catch statements**, **custom exceptions**, and a **finally block** to manage potential issues like file errors and insufficient funds for withdrawal, while ensuring all resources are properly managed.

```
import java.io.*;

// Custom Exception for Banking Transactions
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}

public class BankTransaction {
    public static void main(String[] args) {
        double balance = 500.0;
        double withdrawalAmount = 600.0;

        System.out.println("Bank Transaction Program by Taranpreet Singh");

        try {
            // File operation simulated
            BufferedReader reader = new BufferedReader(new
            FileReader("account.txt"));
            try {
                System.out.println("Reading file...");
                String line = reader.readLine();
                System.out.println("File Content: " + line);
            } catch (IOException e) {
                System.out.println("IOException while reading
            file: " + e.getMessage());
            } finally {
                reader.close();
                System.out.println("File closed.");
            }
        }

        // Nested try for bank transaction
```

```

        try {
            if (withdrawalAmount > balance) {
                throw new
InsufficientFundsException("Insufficient balance for withdrawal");
            }
            balance -= withdrawalAmount;
            System.out.println("Withdrawal successful!
Remaining balance: $" + balance);
        } catch (InsufficientFundsException e) {
            System.out.println("Transaction failed: " +
e.getMessage());
        }

        } catch (FileNotFoundException e) {
            System.out.println("FileNotFoundException: " +
e.getMessage());
        } catch (IOException e) {
            System.out.println("IOException: " + e.getMessage());
        } finally {
            System.out.println("End of transaction – Program by
Taranpreet Singh.");
        }
    }
}

```

Key Points

1. **Custom Exception for Specific Business Logic:**
 - The program defines a custom exception, `InsufficientFundsException`, to handle cases where a withdrawal amount exceeds the balance.
 - Custom exceptions provide clear, domain-specific error messages that make it easier to understand the cause of specific errors in the program.
2. **File Handling and IOException Management:**
 - The program simulates a file read operation using `BufferedReader`.
 - It handles `FileNotFoundException` if the file doesn't exist and `IOException` for general I/O errors.
 - These exceptions are caught and reported separately to provide meaningful error messages.
3. **Nested Try-Catch Statements:**
 - Nested `try` blocks allow handling specific parts of the code independently.
 - In this program:
 - The first inner `try` block handles file reading, while ensuring the file is closed properly in a `finally` block.
 - The second inner `try` block handles bank transaction logic, checking if there is enough balance for withdrawal.

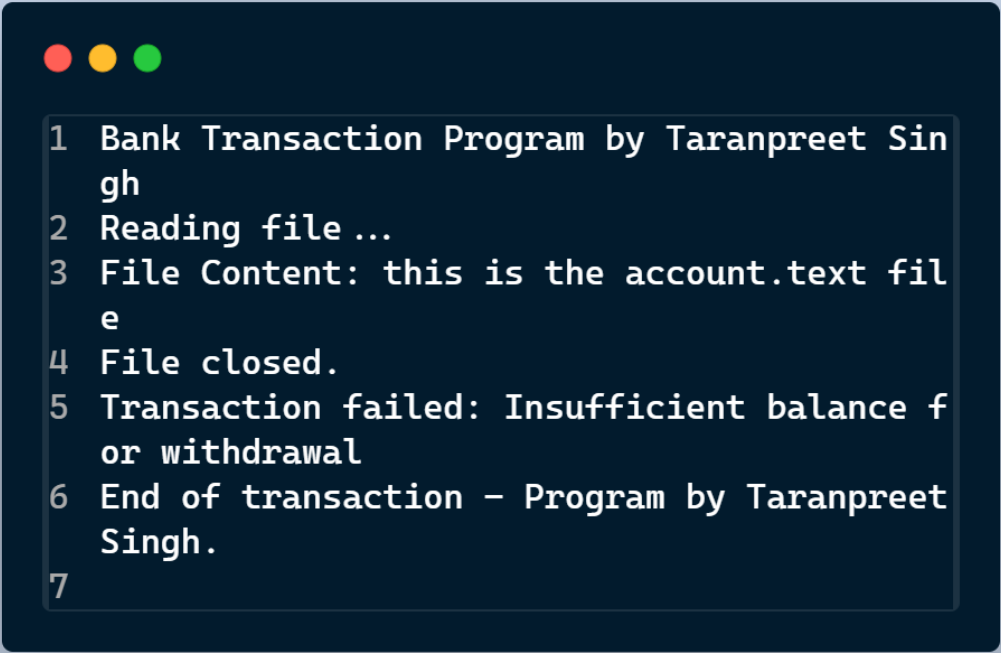
4. **Finally Block:**

- The `finally` block in both the file handling and the outermost try-catch ensures that resources are always closed and final actions are taken.
- The outer `finally` block displays a message indicating the end of the transaction, ensuring clarity of program flow.

5. **Structured Error Handling:**

- Multiple catch blocks allow the program to differentiate between `FileNotFoundException`, `IOException`, and custom exceptions like `InsufficientFundsException`.
- This approach helps in creating clear, targeted responses to various types of errors.

Output:)



```
1 Bank Transaction Program by Taranpreet Singh
2 Reading file ...
3 File Content: this is the account.text file
4 File closed.
5 Transaction failed: Insufficient balance for withdrawal
6 End of transaction – Program by Taranpreet Singh.
7
```

Multithreading (17)

Objective

The objective of this program is to demonstrate the basic structure of multithreading in Java. It shows how to create a thread by implementing the `Runnable` interface and by extending the `Thread` class, and then executes two threads in parallel.

```
// Thread creation using Runnable Interface
class RunnableTask implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable Task - Count: " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.println("Runnable Task Interrupted.");
            }
        }
    }
}

// Thread creation by extending the Thread class
class ThreadTask extends Thread {
    @Override
    public void run() {
        for (int i = 5; i > 0; i--) {
            System.out.println("Thread Task - Count: " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.println("Thread Task Interrupted.");
            }
        }
    }
}

public class BasicThreadCreation {
    public static void main(String[] args) {
        System.out.println("Starting Threads using Runnable and Thread");

        // Creating and starting the Runnable task
        Thread runnableThread = new Thread(new RunnableTask());
        runnableThread.start();

        // Creating and starting the Thread task
        ThreadTask threadTask = new ThreadTask();
```

```
        threadTask.start();
    }
}
```

Key Points

1. Runnable Interface vs. Thread Class:

- The `RunnableTask` class demonstrates thread creation by implementing `Runnable`, offering flexibility since `Runnable` can be implemented by classes that already extend another class.
- The `ThreadTask` class extends `Thread`, offering a direct way to create threads but limiting it to a single inheritance model.

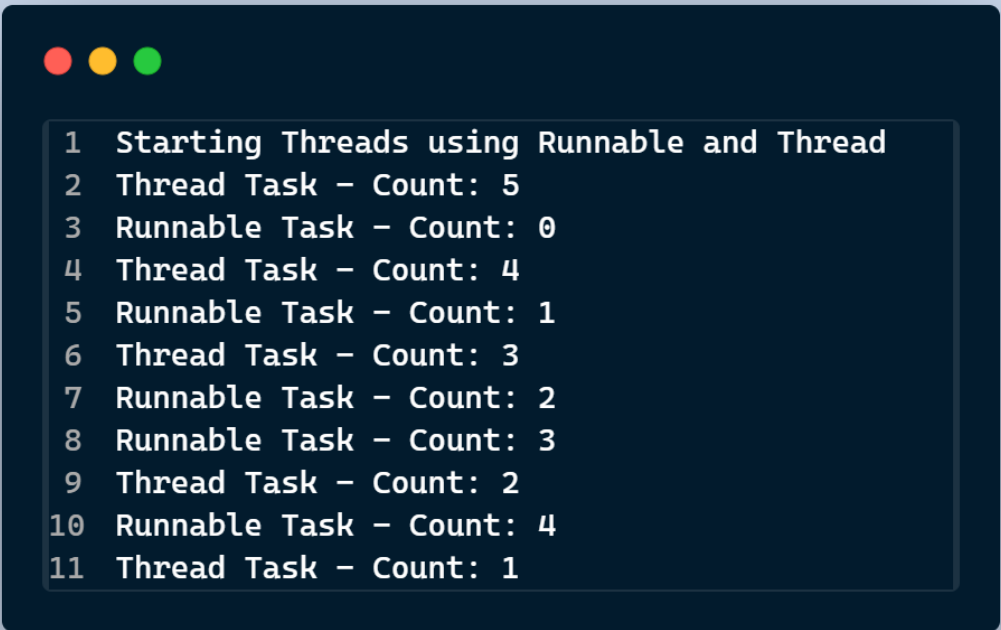
2. Thread Execution:

- Each thread executes its `run` method in parallel, demonstrating concurrent execution.

3. Sleep and InterruptedException:

- The `Thread.sleep` method pauses the execution, and `InterruptedException` is handled to manage possible interruptions.

Output:)



```
1 Starting Threads using Runnable and Thread
2 Thread Task - Count: 5
3 Runnable Task - Count: 0
4 Thread Task - Count: 4
5 Runnable Task - Count: 1
6 Thread Task - Count: 3
7 Runnable Task - Count: 2
8 Runnable Task - Count: 3
9 Thread Task - Count: 2
10 Runnable Task - Count: 4
11 Thread Task - Count: 1
```

Multiple Threads (18)

Objective

The objective of this program is to demonstrate thread synchronization, setting thread priorities, and creating multiple threads to work on a shared resource safely.

```
class SharedCounter {
    private int counter = 0;

    // Synchronized method to increment the counter
    public synchronized void increment() {
        counter++;
        System.out.println(Thread.currentThread().getName() + " -
Counter: " + counter);
    }
}

class CounterThread extends Thread {
    private SharedCounter sharedCounter;

    public CounterThread(SharedCounter sharedCounter, String name) {
        super(name);
        this.sharedCounter = sharedCounter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            sharedCounter.increment();
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                System.out.println(getName() + " interrupted.");
            }
        }
    }
}

public class SynchronizationAndPriorities {
    public static void main(String[] args) {
        SharedCounter counter = new SharedCounter();
```



```

// Creating multiple threads with different priorities
CounterThread thread1 = new CounterThread(counter, "High Priority
Thread");
thread1.setPriority(Thread.MAX_PRIORITY);

CounterThread thread2 = new CounterThread(counter, "Low Priority
Thread");
thread2.setPriority(Thread.MIN_PRIORITY);

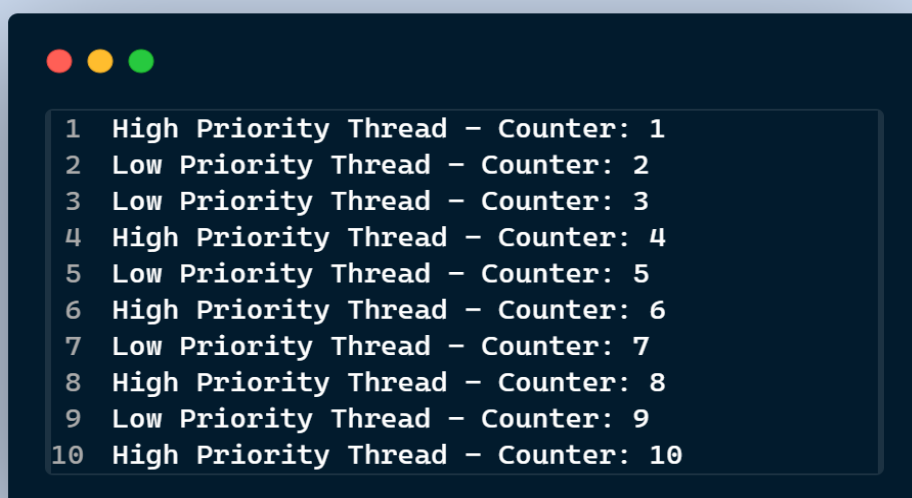
thread1.start();
thread2.start();
}
}

```

Key Points

1. **Synchronization:**
 - The `increment` method in `SharedCounter` is synchronized, ensuring only one thread modifies `counter` at a time.
2. **Multiple Threads:**
 - Two threads (`thread1` and `thread2`) work on the same `SharedCounter` instance, showing how synchronized access prevents race conditions.
3. **Thread Priorities:**
 - Threads are assigned different priorities, showing how thread scheduling can be influenced by priority values.

Output:)



```

1 High Priority Thread - Counter: 1
2 Low Priority Thread - Counter: 2
3 Low Priority Thread - Counter: 3
4 High Priority Thread - Counter: 4
5 Low Priority Thread - Counter: 5
6 High Priority Thread - Counter: 6
7 Low Priority Thread - Counter: 7
8 High Priority Thread - Counter: 8
9 Low Priority Thread - Counter: 9
10 High Priority Thread - Counter: 10

```

Inter-Thread Communication (19)

Objective

The objective of this program is to demonstrate inter-thread communication using `wait` and `notify` and illustrate a simple deadlock scenario, along with ways to avoid it.

```
class BankAccount {
    private int balance = 1000;

    // Method for withdrawal with inter-thread communication
    public synchronized void withdraw(int amount) {
        System.out.println("Attempting to withdraw: " + amount);
        while (balance < amount) {
            System.out.println("Insufficient funds, waiting for
deposit...");
            try {
                wait(); // Waits until balance is updated
            } catch (InterruptedException e) {
                System.out.println("Withdrawal interrupted.");
            }
        }
        balance -= amount;
        System.out.println("Withdrawal successful! New balance: " +
balance);
    }

    // Method for deposit with notify
    public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println("Deposited: " + amount + ", New balance: " +
balance);
        notify(); // Notifies any waiting thread
    }
}

class DeadlockDemo implements Runnable {
    private final Object lock1;
    private final Object lock2;

    public DeadlockDemo(Object lock1, Object lock2) {
        this.lock1 = lock1;
        this.lock2 = lock2;
    }

    @Override
```

```

        public void run() {
            synchronized (lock1) {
                System.out.println(Thread.currentThread().getName() + "
locked " + lock1);
                try { Thread.sleep(50); } catch (InterruptedException e) {}

                synchronized (lock2) {
                    System.out.println(Thread.currentThread().getName() + "
locked " + lock2);
                }
            }
        }
    }

    public class InterThreadCommunicationAndDeadlock {
        public static void main(String[] args) {
            BankAccount account = new BankAccount();

            // Thread for withdrawal
            new Thread(() -> account.withdraw(1200), "Withdrawal
Thread").start();
            // Thread for deposit
            new Thread(() -> account.deposit(500), "Deposit Thread").start();

            // Deadlock example
            Object lockA = new Object();
            Object lockB = new Object();

            Thread t1 = new Thread(new DeadlockDemo(lockA, lockB), "Thread
1");
            Thread t2 = new Thread(new DeadlockDemo(lockB, lockA), "Thread
2");

            t1.start();
            t2.start();
        }
    }

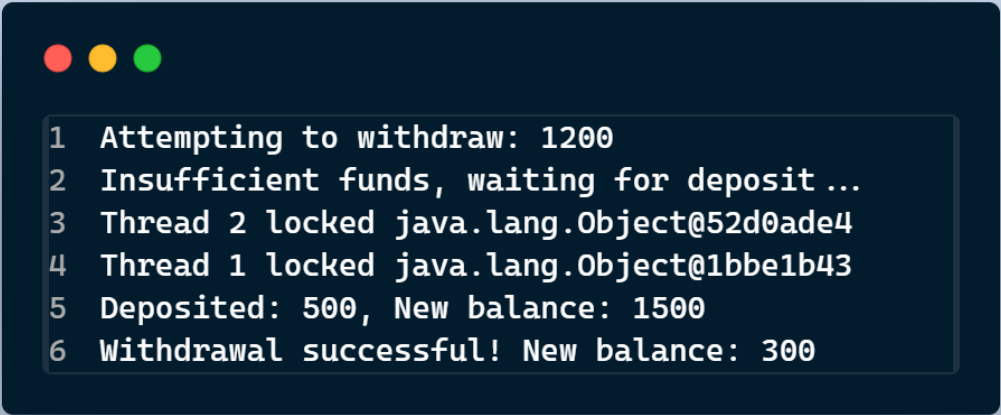
```

Key Points

1. **Inter-Thread Communication with wait/notify:**
 - o withdraw and deposit methods in BankAccount use wait and notify for managing funds, allowing withdraw to wait if balance is insufficient and deposit to notify waiting threads when balance is updated.
2. **Deadlock Demonstration:**

- The `DeadlockDemo` class simulates a deadlock by creating a situation where each thread waits for a resource locked by another thread, showing how cyclic dependencies can halt execution.
3. **Avoiding Deadlock:**
- To avoid deadlock, resource locking order should be consistent, and timeout mechanisms can be used.

Output:)



```
1 Attempting to withdraw: 1200
2 Insufficient funds, waiting for deposit ...
3 Thread 2 locked java.lang.Object@52d0ade4
4 Thread 1 locked java.lang.Object@1bbe1b43
5 Deposited: 500, New balance: 1500
6 Withdrawal successful! New balance: 300
```

Applet Structure (20)

Objective

This program demonstrates the basic structure of a Java Applet, focusing on its lifecycle methods (init, start, stop, and destroy). It also covers the two main types of applets: standalone applets and applets embedded in HTML.

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

// Basic Applet demonstrating Life Cycle Methods
public class LifeCycleApplet extends Applet {

    // Called once when the applet is initialized
    @Override
    public void init() {
        System.out.println("Applet Initialized");
    }

    // Called every time the applet starts or is brought back into focus
    @Override
    public void start() {
        System.out.println("Applet Started");
    }

    // Called every time the applet is stopped or minimized
    @Override
    public void stop() {
        System.out.println("Applet Stopped");
    }

    // Called once when the applet is destroyed
    @Override
    public void destroy() {
        System.out.println("Applet Destroyed");
    }

    // Called when applet window is redrawn
    @Override
    public void paint(Graphics g) {
```

```
        g.drawString("Welcome to the Applet Life Cycle!", 20, 20);  
    }  
}
```

Key Points

1. **Types of Applets:**
 - **Standalone Applets:** Run with the `appletviewer` tool.
 - **Embedded Applets:** Embedded in an HTML file and viewed in a browser.
2. **Applet Lifecycle Methods:**
 - `init`: Initializes the applet; called only once.
 - `start`: Starts or restarts the applet.
 - `stop`: Pauses the applet (e.g., when minimized).
 - `destroy`: Final cleanup before the applet is removed from memory.
3. **Using `applet` Tag in HTML:**
 - Save the above program in a file, `LifeCycleApplet.java`.
 - Compile it and embed it in an HTML file as follows:

```
<html>  
  <body>  
    <applet code="LifeCycleApplet.class" width="300"  
height="150"></applet>  
  </body>  
</html>
```

AWT Components (21)

Objective

This program demonstrates the creation of a simple AWT-based GUI application with basic components (label, text box, text area, check boxes, and radio buttons) and introduces layout managers (FlowLayout, BorderLayout, and GridLayout).

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

public class BasicAWTExample extends Frame {

    public BasicAWTExample() {
        setTitle("Basic AWT Components");
        setSize(400, 400);
        setLayout(new FlowLayout());

        // Label
        Label nameLabel = new Label("Name:");
        add(nameLabel);

        // TextBox
        TextField nameTextField = new TextField(20);
        add(nameTextField);

        // TextArea
        Label commentLabel = new Label("Comments:");
        add(commentLabel);
        TextArea commentTextArea = new TextArea(5, 30);
        add(commentTextArea);

        // CheckBox
        Label genderLabel = new Label("Gender:");
        add(genderLabel);
        Checkbox male = new Checkbox("Male");
        Checkbox female = new Checkbox("Female");
        add(male);
        add(female);
    }
}
```

```

// Radio Buttons (Checkbox in a Group)
Label ageGroupLabel = new Label("Age Group:");
add(ageGroupLabel);
CheckboxGroup ageGroup = new CheckboxGroup();
Checkbox below18 = new Checkbox("Below 18", ageGroup, false);
Checkbox above18 = new Checkbox("Above 18", ageGroup, true);
add(below18);
add(above18);

// Choice list
Label cityLabel = new Label("City:");
add(cityLabel);
Choice cityChoice = new Choice();
cityChoice.add("New York");
cityChoice.add("Los Angeles");
cityChoice.add("Chicago");
add(cityChoice);

// Closing the window
addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        dispose();
    }
});
}

public static void main(String[] args) {
    BasicAWTExample frame = new BasicAWTExample();
    frame.setVisible(true);
}
}

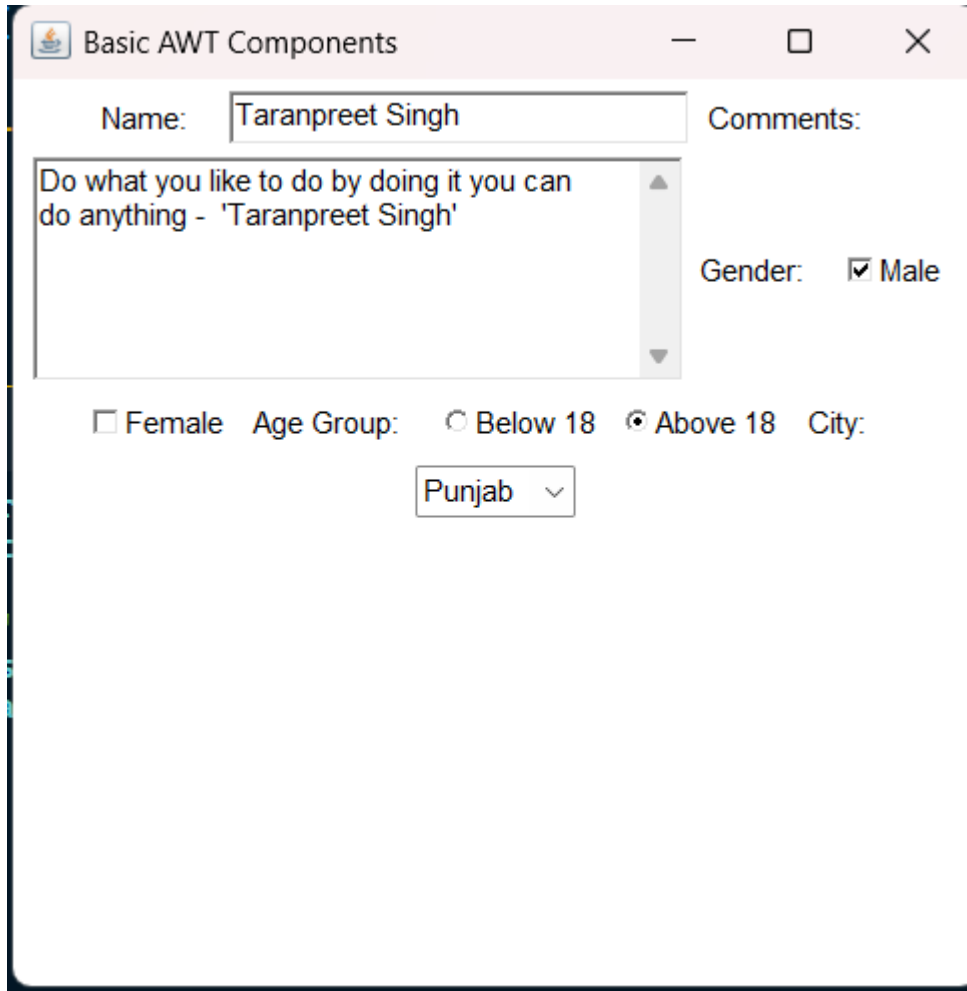
```

Key Points

1. **AWT Basics:** AWT provides a platform-independent windowing toolkit for Java.
2. **Components:**
 - **Label:** Used to display text.
 - **TextField:** Single-line input.
 - **TextArea:** Multi-line input area.
 - **Checkbox:** Used to select options; `CheckboxGroup` can create radio buttons.
 - **Choice:** Dropdown list for selecting one option from multiple.
3. **Layout Managers:**
 - `FlowLayout`: Places components left to right in a row.
 - `BorderLayout`, `GridLayout`, etc., allow further customization in positioning components.

4. **Window Closing Event:** Uses `WindowAdapter` to close the frame when the window is closed.

Output:)



The screenshot shows a Java AWT window titled "Basic AWT Components". The window contains a form with the following elements:

- Name:** A text field containing "Taranpreet Singh".
- Comments:** A text area containing the text "Do what you like to do by doing it you can do anything - 'Taranpreet Singh'".
- Gender:** A group box containing two radio buttons: "Male" (which is selected) and "Female".
- Age Group:** A group box containing two radio buttons: "Above 18" (which is selected) and "Below 18".
- City:** A dropdown menu currently showing "Punjab".

Advanced Layout Managers

&

Event Handling (22)

Objective

This program demonstrates the use of advanced layout managers (`BorderLayout`, `GridLayout`) and introduces event handling in AWT using the Delegation Event Model.

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

public class AWTEventHandlingExample extends Frame implements
    ActionListener {

    private TextField nameTextField;
    private Label displayLabel;

    public AWTEventHandlingExample() {
        setTitle("AWT Event Handling Example");
        setSize(400, 300);
        setLayout(new BorderLayout());

        // North Panel with GridLayout for TextField and Button
        Panel northPanel = new Panel();
        northPanel.setLayout(new GridLayout(2, 2));

        Label nameLabel = new Label("Enter your name:");
        northPanel.add(nameLabel);

        nameTextField = new TextField(20);
        northPanel.add(nameTextField);

        Button submitButton = new Button("Submit");
        submitButton.addActionListener(this);
        northPanel.add(new Label("")); // Empty cell for alignment
        northPanel.add(submitButton);

        add(northPanel, BorderLayout.NORTH);
    }
}
```

```

// Center Panel for Display Label
displayLabel = new Label("Your name will be displayed here.");
add(displayLabel, BorderLayout.CENTER);

// South Panel with FlowLayout for Radio Buttons and Check Boxes
Panel southPanel = new Panel(new FlowLayout());
Label preferenceLabel = new Label("Choose your preferences:");
southPanel.add(preferenceLabel);

Checkbox sportsCheckBox = new Checkbox("Sports");
Checkbox musicCheckBox = new Checkbox("Music");
southPanel.add(sportsCheckBox);
southPanel.add(musicCheckBox);

CheckboxGroup languageGroup = new CheckboxGroup();
Checkbox english = new Checkbox("English", languageGroup, true);
Checkbox spanish = new Checkbox("Spanish", languageGroup, false);
southPanel.add(english);
southPanel.add(spanish);

add(southPanel, BorderLayout.SOUTH);

// Window closing event
addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        dispose();
    }
});
}

@Override
public void actionPerformed(ActionEvent e) {
    String name = nameTextField.getText();
    displayLabel.setText("Hello, " + name + "!");
}

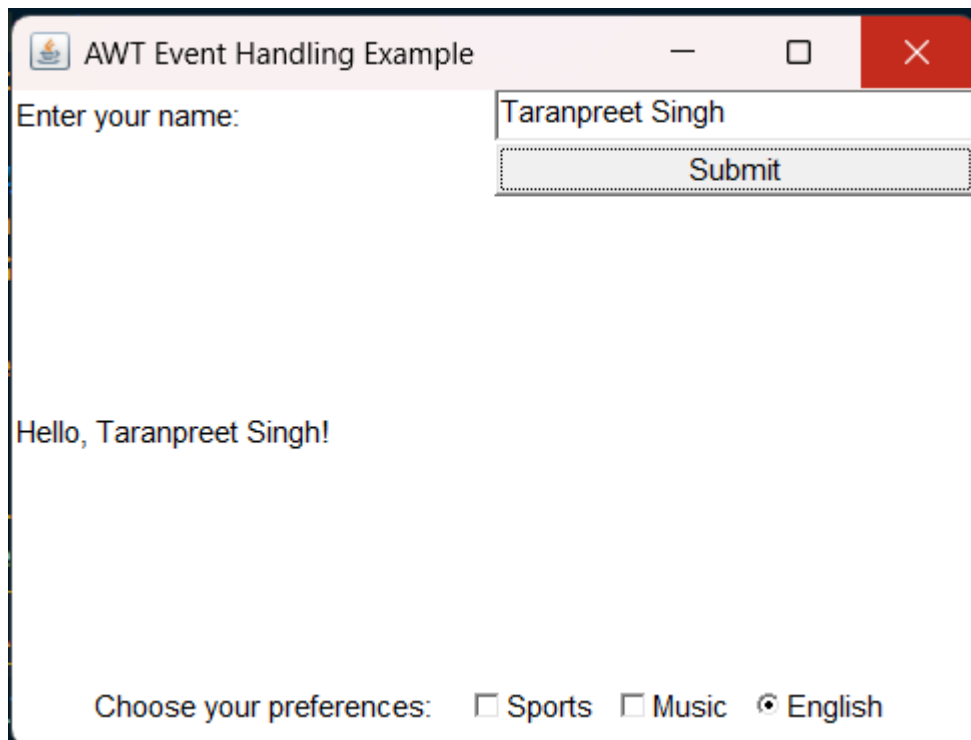
public static void main(String[] args) {
    AWTEventHandlingExample frame = new AWTEventHandlingExample();
    frame.setVisible(true);
}
}

```

Key Points

1. **Event Handling:** Uses the Delegation Event Model, where events are sent to listeners implementing specific interfaces.
 - **ActionListener:** Captures `ActionEvent` for button clicks.
 - `addActionListener(this)`: Registers the current class as a listener.
2. **Layout Managers:**
 - `BorderLayout`: Divides the frame into five regions (NORTH, SOUTH, EAST, WEST, CENTER).
 - `GridLayout`: Arranges components in a grid of rows and columns.
3. **Action Events:**
 - `ActionPerformed`: Updates the label based on user input from the text field.
4. **Window Closing Event:**
 - **WindowAdapter**: Simplifies handling of window events like closing the window.

Output:)



JDBC (23)

Objective

To demonstrate the basics of JDBC architecture, establish a connection to a database using JDBC, and explore the necessary JDBC drivers and setup.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnectionExample {
    public static void main(String[] args) {
        // Database URL, username, and password
        String url = "localhost:3306/taran"; //
        Change "mydatabase" to your database name
        String user = "root";
        String password = "password"; // Replace with
        your database password

        // Establishing the connection
        try (Connection connection =
DriverManager.getConnection(url, user, password)) {
            if (connection != null) {
                System.out.println("Connected to the
database successfully!");
            }
        } catch (SQLException e) {
            System.out.println("Failed to connect to
the database.");
            e.printStackTrace();
        }
    }
}
```

```
}
```

Key Points

1. **JDBC Drivers:** Different types of JDBC drivers (Type-1 to Type-4) provide different levels of efficiency and compatibility with databases. This example uses a Type-4 driver (Pure Java driver).
2. **Connection URL:** Specifies the database type (`jdbc:mysql`), host (`localhost`), and database name (`mydatabase`).
3. **DriverManager Class:** Used to manage a list of database drivers and establish a connection with the specified database.
4. **Exception Handling:** `SQLException` is caught to handle any issues connecting to the database.

Output:)



```
1 Connected to the database successfully!
```

Crud Operations (24)

Objective

To perform basic data operations—specifically `INSERT` and `SELECT`—to demonstrate how to add new records to a database and retrieve existing records.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DatabaseCRUDInsertSelect {
    public static void main(String[] args) {
        String url = "localhost:3306/taran";
        String user = "root";
        String password = "password";

        try (Connection connection =
DriverManager.getConnection(url, user, password)) {
            if (connection != null) {
                System.out.println("Connected to the
database.");

                // Insert Operation
                String insertSQL = "INSERT INTO employees
(name, position, salary) VALUES ('John Doe', 'Manager',
75000)";

                try (Statement statement =
connection.createStatement()) {
                    int rowsInserted =
statement.executeUpdate(insertSQL);
```

```

        System.out.println("Inserted " +
rowsInserted + " row(s).");

        // Select Operation
        String selectSQL = "SELECT * FROM
employees";

        ResultSet resultSet =
statement.executeQuery(selectSQL);

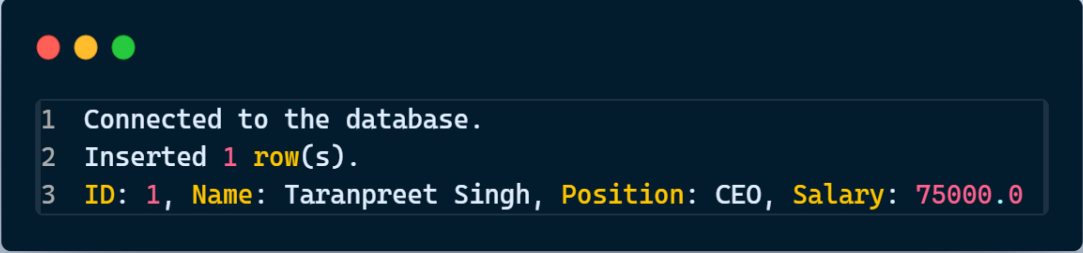
        while (resultSet.next()) {
            int id = resultSet.getInt("id");
            String name =
resultSet.getString("name");
            String position =
resultSet.getString("position");
            double salary =
resultSet.getDouble("salary");
            System.out.println("ID: " + id + ",
Name: " + name + ", Position: " + position + ", Salary: " +
salary);
        }
    }
} catch (SQLException e) {
    System.out.println("Database operation error.");
    e.printStackTrace();
}
}
}

```

Key Points

1. **Statement Interface:** Used to execute SQL queries like INSERT and SELECT.
2. **executeUpdate():** Executes SQL statements that modify the database (e.g., INSERT, UPDATE, DELETE), returning the count of affected rows.
3. **executeQuery():** Executes SQL statements that retrieve data, returning a `ResultSet` object containing the result set.
4. **ResultSet Interface:** Allows the retrieval of data from the database, row by row.

Output:)

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays three lines of output from a database operation.

```
1 Connected to the database.  
2 Inserted 1 row(s).  
3 ID: 1, Name: Taranpreet Singh, Position: CEO, Salary: 75000.0
```

Update & Delete (25)

Objective

To perform UPDATE and DELETE operations on the database and add exception handling for any issues encountered during database interaction.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class DatabaseCRUDUpdateDelete {
    public static void main(String[] args) {
        String url = "localhost:3306/taran";
        String user = "root";
        String password = "password";

        try (Connection connection = DriverManager.getConnection(url,
user, password)) {
            if (connection != null) {
                System.out.println("Connected to the database.");

                // Update Operation
                String updateSQL = "UPDATE employees SET salary = 80000
WHERE name = 'John Doe'";
                try (Statement statement = connection.createStatement())
                {
                    int rowsUpdated = statement.executeUpdate(updateSQL);
                    System.out.println("Updated " + rowsUpdated + "
row(s).");
                }

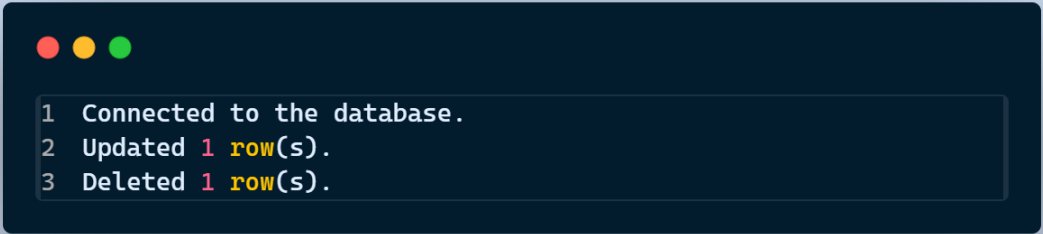
                // Delete Operation
                String deleteSQL = "DELETE FROM employees WHERE name =
'John Doe'";
                try (Statement statement = connection.createStatement())
                {
                    int rowsDeleted = statement.executeUpdate(deleteSQL);
                    System.out.println("Deleted " + rowsDeleted + "
row(s).");
                }
            }
        } catch (SQLException e) {
```

```
        System.out.println("Database operation error.");
        e.printStackTrace();
    }
}
```

Key Points

1. **Update Operation:** Modifies existing records in the database.
2. **Delete Operation:** Removes records from the database.
3. **Error Handling:** Handles SQL exceptions that might occur during database modifications.
4. **Resources Management:** Uses try-with-resources for automatic resource management, ensuring connections are closed properly.

Output:)

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays three lines of output: '1 Connected to the database.', '2 Updated 1 row(s).', and '3 Deleted 1 row(s).'.

```
1 Connected to the database.
2 Updated 1 row(s).
3 Deleted 1 row(s).
```