Unit :- 1

Write programs using Java built-in functions using all data types.

Aim:

To demonstrate built-in functions with byte.

Theory:

Byte wrapper class in Java provides methods like Byte.MAX_VALUE, Byte.MIN_VALUE, and Byte.parseByte().

Program:

```
public class ByteDemo {
   public static void main(String[] args) {
      byte b = 10;
      System.out.println("Value of b: " + b);
      System.out.println("Max Value of byte: " + Byte.MAX_VALUE);
      System.out.println("Min Value of byte: " + Byte.MIN_VALUE);
      String s = "25";
      byte parsed = Byte.parseByte(s);
      System.out.println("Parsed Byte value: " + parsed);
   }
}
```

Output:

```
Value of b: 10
Max Value of byte: 127
Min Value of byte: -128
Parsed Byte value: 25
```

Program using short and Built-in Methods

Aim:

To demonstrate built-in functions with short.

Program:

```
public class ShortDemo {
   public static void main(String[] args) {
      short s = 500;
      System.out.println("Value of s: " + s);
      System.out.println("Max Value of short: " + Short.MAX_VALUE);
      System.out.println("Min Value of short: " + Short.MIN_VALUE);
      String str = "1234";
      short parsed = Short.parseShort(str);
      System.out.println("Parsed Short value: " + parsed);
   }
}
```

Output:

```
Value of s: 500
Max Value of short: 32767
Min Value of short: -32768
Parsed Short value: 1234
```

Program using int and Built-in Methods

```
public class IntDemo {
   public static void main(String[] args) {
      int num = 1000;
      System.out.println("Value of num: " + num);
      System.out.println("Max Value of int: " + Integer.MAX_VALUE);
      System.out.println("Min Value of int: " + Integer.MIN_VALUE);
      String s = "2025";
```

```
        int parsed = Integer.parseInt(s);
        System.out.println("Parsed Int value: " + parsed);
    }
}
```

Program using long and Built-in Methods

```
public class LongDemo {
    public static void main(String[] args) {
        long l = 9876543210L;
        System.out.println("Value of l: " + l);
        System.out.println("Max Value of long: " + Long.MAX_VALUE);
        System.out.println("Min Value of long: " + Long.MIN_VALUE);
        String s = "123456789";
        long parsed = Long.parseLong(s);
        System.out.println("Parsed Long value: " + parsed);
    }
}
```

Program using float and Built-in Methods

```
public class FloatDemo {
    public static void main(String[] args) {
        float f = 12.34f;
        System.out.println("Value of f: " + f);
        System.out.println("Max Value of float: " + Float.MAX_VALUE);
        System.out.println("Min Value of float: " + Float.MIN_VALUE);
        String s = "45.67";
        float parsed = Float.parseFloat(s);
        System.out.println("Parsed Float value: " + parsed);
    }
}
```

Program using double and Built-in Methods

```
public class DoubleDemo {
   public static void main(String[] args) {
      double d = 12345.6789;
      System.out.println("Value of d: " + d);
      System.out.println("Max Value of double: " + Double.MAX_VALUE);
      System.out.println("Min Value of double: " + Double.MIN_VALUE);
      String s = "98765.4321";
      double parsed = Double.parseDouble(s);
      System.out.println("Parsed Double value: " + parsed);
   }
}
```

Program using char and Built-in Methods

```
public class CharDemo {
   public static void main(String[] args) {
      char c = 'A';
      System.out.println("Character: " + c);
      System.out.println("Is Digit? " + Character.isDigit(c));
      System.out.println("Is Letter? " + Character.isLetter(c));
      System.out.println("Lowercase: " + Character.toLowerCase(c));
      System.out.println("Uppercase: " + Character.toUpperCase('z'));
   }
}
```

Program using boolean and Built-in Methods

```
public class BooleanDemo {
   public static void main(String[] args) {
      boolean flag = true;
      System.out.println("Value of flag: " + flag);
      String s = "true";
      boolean parsed = Boolean.parseBoolean(s);
      System.out.println("Parsed Boolean: " + parsed);
      System.out.println("Boolean TRUE constant: " + Boolean.TRUE);
      System.out.println("Boolean FALSE constant: " + Boolean.FALSE);
   }
}
```

Program using String and Built-in Methods

```
public class StringDemo {
   public static void main(String[] args) {
      String str = "Hello World";
      System.out.println("Original String: " + str);
      System.out.println("Length: " + str.length());
      System.out.println("Uppercase: " + str.toUpperCase());
      System.out.println("Lowercase: " + str.toLowerCase());
      System.out.println("Substring(0,5): " + str.substring(0,5));
      System.out.println("Character at 4: " + str.charAt(4));
      System.out.println("Replace: " + str.replace("World","Java"));
   }
}
```

Unit :- 2

Write programs using conditional statements and loop statements.

Aim:

To check whether a given number is positive, negative, or zero using if-else.

Program:

```
import java.util.Scanner;
public class PositiveNegative {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = sc.nextInt();

        if (num > 0) {
            System.out.println("Number is Positive");
        } else if (num < 0) {
            System.out.println("Number is Negative");
        } else {
            System.out.println("Number is Zero");
        }
    }
}
```

Output:

```
Enter a number: -8
Number is Negative
```

Find the Largest of Three Numbers using Nested If

Program:

```
import java.util.Scanner;
public class LargestOfThree {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter three numbers: ");
      int a = sc.nextInt(), b = sc.nextInt(), c = sc.nextInt();

      if (a >= b && a >= c) {
         System.out.println(a + " is the largest");
      } else if (b >= a && b >= c) {
         System.out.println(b + " is the largest");
      } else {
         System.out.println(c + " is the largest");
      }
   }
}
```

Calculator using Switch Case

Program:

```
import java.util.Scanner;
public class Calculator {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter two numbers: ");
      int a = sc.nextInt(), b = sc.nextInt();
      System.out.print("Enter operator (+,-,*,/): ");
      char op = sc.next().charAt(0);

      switch (op) {
         case '+': System.out.println("Sum = " + (a+b)); break;
         case '-': System.out.println("Difference = " + (a-b)); break;
         case '*': System.out.println("Product = " + (a*b)); break;
         case '/':
```

```
            if (b != 0) System.out.println("Quotient = " + (a/b));
            else System.out.println("Division by zero not allowed");
            break;
        default: System.out.println("Invalid operator");
    }
  }
}
```

Part B: Loop Statements

Print Numbers 1 to 10 using For Loop

Program:

```
public class ForLoopDemo {
   public static void main(String[] args) {
      for (int i = 1; i <= 10; i++) {
         System.out.println(i);
      }
   }
}
```

Print Sum of First N Natural Numbers using While Loop

Program:

```
import java.util.Scanner;
public class WhileLoopSum {
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter N: ");
        int n = sc.nextInt();
        int sum = 0, i = 1;

        while (i <= n) {
            sum += i;
            i++;
        }
        System.out.println("Sum of first " + n + " numbers = " + sum);
    }
}
```

Print Multiplication Table using Do-While Loop

Program:

```
import java.util.Scanner;
public class DoWhileTable {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = sc.nextInt();
        int i = 1;

        do {
            System.out.println(num + " x " + i + " = " + (num*i));
            i++;
        } while (i <= 10);
    }
```

Print Star Pattern using Nested Loops

Program:

```
public class StarPattern {
    public static void main(String[] args) {
        int rows = 5;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Find Factorial using For Loop

Program:

```
import java.util.Scanner;
public class Factorial {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = sc.nextInt();
        long fact = 1;

        for (int i = 1; i <= n; i++) {
```

```
            fact *= i;
        }
        System.out.println("Factorial of " + n + " = " + fact);
    }
}
```

For-Each Loop Example

Program:

```
public class ForEachDemo {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        System.out.println("Array elements:");
        for (int num : arr) {
            System.out.println(num);
        }
    }
}
```

Unit :- 3

Write a program to read data from keyboard.

Aim:

To write a Java program to read input from the keyboard and display it on the screen.

Theory:

- Java provides multiple ways to take input from the user:
    1. **Scanner class** (commonly used, introduced in Java 1.5).
    2. **BufferedReader class** (older approach, requires exception handling).
- Scanner is present in java.util package.
    - Common methods:
        - nextInt() → reads integer
        - nextFloat() → reads float
        - nextLine() → reads string
        - next() → reads word

Program 1: Using Scanner Class

```
import java.util.Scanner;

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

      // Reading String input
      System.out.print("Enter your name: ");
      String name = sc.nextLine();

      // Reading Integer input
      System.out.print("Enter your age: ");
```

```
        int age = sc.nextInt();

        // Reading Double input
        System.out.print("Enter your marks: ");
        double marks = sc.nextDouble();

        // Displaying output
        System.out.println("\n--- User Details ---");
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Marks: " + marks);

        sc.close();
    }
}
```

Sample Output:

```
Enter your name: Ramesh
Enter your age: 20
Enter your marks: 89.5

--- User Details ---
Name: Ramesh
Age: 20
Marks: 89.5
```

Program 2: Using BufferedReader

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class KeyboardInputBuffered {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

        System.out.print("Enter your city: ");
        String city = br.readLine();
```

```
        System.out.print("Enter your pincode: ");
        int pin = Integer.parseInt(br.readLine());

        System.out.println("\n--- Address Details ---");
        System.out.println("City: " + city);
        System.out.println("Pincode: " + pin);
    }
}
```

Sample Output:

```
Enter your city: Delhi
Enter your pincode: 110001

--- Address Details ---
City: Delhi
Pincode: 110001
```

Unit :- 4

Write a program to create class and objects.

Aim:

To write a Java program to create a class and objects, and demonstrate accessing class members.

Theory:

- **Class**: A class is a blueprint for creating objects. It contains **data members (fields/variables)** and **methods (functions)**.
- **Object**: An object is an instance of a class. Using objects, we can access the variables and methods of a class.

Syntax of Class and Object:

```
class ClassName {
   // data members
   // methods
}

public class Main {
   public static void main(String[] args) {
      ClassName obj = new ClassName(); // object creation
   }
}
```

Algorithm:

1. Define a class (e.g., Student) with data members and methods.
2. Create objects of the class inside the main method.
3. Assign values to data members.
4. Call methods using the created objects.
5. Display the results.

Program: Student Class Example

```
class Student {
    // Data members
    String name;
    int age;
    double marks;

    // Method to display student details
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Marks: " + marks);
    }
}

public class ClassObjectDemo {
    public static void main(String[] args) {
        // Creating first object
        Student s1 = new Student();
        s1.name = "Amit";
        s1.age = 20;
        s1.marks = 85.5;

        // Creating second object
        Student s2 = new Student();
        s2.name = "Neha";
        s2.age = 19;
        s2.marks = 92.0;

        // Display details
        System.out.println("Student 1 Details:");
        s1.display();

        System.out.println("\nStudent 2 Details:");
        s2.display();
    }
}
```

Sample Output:

Student 1 Details:
Name: Amit
Age: 20
Marks: 85.5

Student 2 Details:
Name: Neha
Age: 19
Marks: 92.0

Unit:-5

Write programs using constructors.

Aim:

To write Java programs using constructors (default, parameterized, and overloaded constructors).

Theory:

- A **constructor** in Java is a special method that is automatically called when an object is created.
- Characteristics:
  - Name is same as the class.
  - No return type (not even void).
  - Used to initialize objects.
- **Types of Constructors:**
  1. **Default Constructor** → No arguments, initializes default values.
  2. **Parameterized Constructor** → Accepts arguments to set values.
  3. **Constructor Overloading** → Multiple constructors in a class with different parameter lists.

Algorithm:

1. Define a class with data members.
2. Create a **default constructor** to initialize with fixed values.
3. Create a **parameterized constructor** to initialize with user-provided values.
4. Create **objects** using both constructors.
5. Display object details.

Program 1: Default Constructor

```
class Student {
   String name;
   int age;
```

```
    // Default Constructor
    Student() {
        name = "Unknown";
        age = 18;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class DefaultConstructorDemo {
    public static void main(String[] args) {
        Student s1 = new Student(); // calls default constructor
        s1.display();
    }
}
```

Sample Output:

```
Name: Unknown, Age: 18
```

Program 2: Parameterized Constructor

```
class Student {
    String name;
    int age;

    // Parameterized Constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

```
public class ParameterizedConstructorDemo {
    public static void main(String[] args) {
        Student s1 = new Student("Amit", 20);
        Student s2 = new Student("Neha", 19);

        s1.display();
        s2.display();
    }
}
```

Sample Output:

```
Name: Amit, Age: 20
Name: Neha, Age: 19
```

Program 3: Constructor Overloading

```
class Student {
    String name;
    int age;

    // Default Constructor
    Student() {
        name = "Unknown";
        age = 18;
    }

    // Parameterized Constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
```

```
}

public class ConstructorOverloadingDemo {
    public static void main(String[] args) {
        Student s1 = new Student();              // Calls default constructor
        Student s2 = new Student("Ravi", 21);   // Calls parameterized constructor

        s1.display();
        s2.display();
    }
}
```

Sample Output:

```
Name: Unknown, Age: 18
Name: Ravi, Age: 21
```

Unit:- 6

Write a program to illustrate usage of command line arguments.

Aim:

To write a Java program that accepts values from the **command line** and displays them.

Theory:

- In Java, command line arguments are passed to the main() method as an array of String objects:
- public static void main(String[] args)
- When executing the program from terminal/command prompt:
- java ProgramName arg1 arg2 arg3
  - arg1, arg2, arg3 are stored in the args[] array.
- All arguments are treated as **strings**. If needed, they can be converted to other data types using wrapper class methods like:
  - Integer.parseInt(args[i])
  - Double.parseDouble(args[i])

Algorithm:

1. Define a class with main(String[] args).
2. Accept arguments from args[] array.
3. Print the arguments using a loop.
4. Convert arguments (if needed) into integer/float/double.
5. Perform operations using these arguments.

Program 1: Display Command Line Arguments

```
public class CommandLineDemo {
   public static void main(String[] args) {
      System.out.println("Number of arguments: " + args.length);
      System.out.println("Arguments are:");
      for (int i = 0; i < args.length; i++) {
```

```
            System.out.println("args[" + i + "] = " + args[i]);
        }
    }
}
```

Execution Example:

```
> javac CommandLineDemo.java
> java CommandLineDemo Hello 123 Java
```

Output:

```
Number of arguments: 3
Arguments are:
args[0] = Hello
args[1] = 123
args[2] = Java
```

Program 2: Sum of Two Numbers using Command Line Arguments

```
public class SumCommandLine {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Please provide two numbers as arguments.");
            return;
        }
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        int sum = num1 + num2;

        System.out.println("First Number: " + num1);
        System.out.println("Second Number: " + num2);
        System.out.println("Sum = " + sum);
    }
}
```

Execution Example:

```
> javac SumCommandLine.java
> java SumCommandLine 15 25
```

Output:

```
First Number: 15
Second Number: 25
Sum = 40
```

Unit:- 7

Write programs using concept of overloading methods.

Aim:

To write Java programs that demonstrate the concept of **method overloading**.

Theory:

- **Method Overloading** in Java means defining **multiple methods with the same name but different parameter lists** (different number of parameters or different data types).
- The return type may or may not be different, but **parameter list must be different**.
- It is an example of **compile-time polymorphism**.

Rules of Method Overloading:

1. Methods must have the **same name**.
2. They must differ in:
    - Number of parameters, or
    - Type of parameters, or
    - Sequence of parameters.
3. Return type alone cannot differentiate overloaded methods.

Algorithm:

1. Define a class with a method name (e.g., add).
2. Overload the method by changing its parameter list.
3. Create an object in the main() method.
4. Call the overloaded methods with different arguments.
5. Observe which version of the method executes.

Program 1: Overloading with Different Number of Parameters

```
class Calculator {
    // Method with 2 parameters
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded Method with 3 parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class OverloadingDemo1 {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of 2 numbers: " + calc.add(10, 20));
        System.out.println("Sum of 3 numbers: " + calc.add(10, 20, 30));
    }
}
```

Sample Output:

```
Sum of 2 numbers: 30
Sum of 3 numbers: 60
```


Program 2: Overloading with Different Data Types

```
class Calculator {
    // Method with integer parameters
    int multiply(int a, int b) {
        return a * b;
    }

    // Overloaded Method with double parameters
    double multiply(double a, double b) {
        return a * b;
    }
```

```
}

public class OverloadingDemo2 {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Multiplication of integers: " + calc.multiply(5, 4));
        System.out.println("Multiplication of doubles: " + calc.multiply(2.5, 3.5));
    }
}
```

Sample Output:

```
Multiplication of integers: 20
Multiplication of doubles: 8.75
```

Program 3: Overloading with Different Sequence of Parameters

```
class Display {
    void show(String name, int age) {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    void show(int age, String name) {
        System.out.println("Age: " + age + ", Name: " + name);
    }
}

public class OverloadingDemo3 {
    public static void main(String[] args) {
        Display d = new Display();
        d.show("Amit", 20);  // Calls first method
        d.show(22, "Neha");  // Calls second method
    }
}
```

Sample Output:

Name: Amit, Age: 20
Age: 22, Name: Neha

Unit:-8

Exercise on inheritance.

Aim:

To write a program in Java to implement the concept of **inheritance**.

Theory:

- **Inheritance** allows one class (child/subclass) to acquire the properties and behaviors of another class (parent/superclass).
- It supports **code reusability** and **method overriding**.
- Types of inheritance in Java:
    1. Single Inheritance
    2. Multilevel Inheritance
    3. Hierarchical Inheritance

    (Note: Java does not support multiple inheritance using classes, but it is possible with interfaces.)

Algorithm (for Single Inheritance):

1. Create a base class Person with data members and methods.
2. Create a derived class Student that extends Person.
3. Add new properties/methods in Student.
4. Create an object of Student and access both parent and child methods.

Program (Single Inheritance):

```
// Base class
class Person {
    String name;
    int age;

    void displayPerson() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
```

```
}

// Derived class
class Student extends Person {
    int rollNo;
    double marks;

    void displayStudent() {
        displayPerson();  // calling parent method
        System.out.println("Roll No: " + rollNo + ", Marks: " + marks);
    }
}

// Main class
public class InheritanceDemo {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "Rahul";
        s1.age = 21;
        s1.rollNo = 101;
        s1.marks = 88.5;

        s1.displayStudent();
    }
}
```

Sample Output:

```
Name: Rahul, Age: 21
Roll No: 101, Marks: 88.5
```

Unit:- 9

Write a program using the concept of method overriding.

Aim:

To write a Java program to demonstrate the concept of **method overriding**.

Theory:

- **Method Overriding** occurs when a **subclass** provides its own implementation of a method already defined in its **superclass**.
- Rules of Method Overriding:
    1. The method in the child class must have the **same name, parameters, and return type** as in the parent class.
    2. Access modifier cannot be more restrictive than the parent method.
    3. @Override annotation is used for clarity (optional but recommended).
- It is mainly used for **runtime polymorphism**.

Algorithm:

1. Define a base class Animal with a method sound().
2. Create subclasses Dog and Cat that override the sound() method.
3. Create objects of each subclass and call the sound() method.
4. Observe how the overridden methods execute depending on the object.

Program:

```
// Base class
class Animal {
   void sound() {
      System.out.println("Animals make sounds");
```

```
    }
}

// Derived class Dog
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Derived class Cat
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

// Main class
public class MethodOverridingDemo {
    public static void main(String[] args) {
        Animal a1 = new Dog(); // upcasting
        Animal a2 = new Cat(); // upcasting

        a1.sound();  // Calls Dog's version
        a2.sound();  // Calls Cat's version
    }
}
```


Sample Output:

```
Dog barks
Cat meows
```

Unit:- 10

Exercise on importing packages.

Aim:

To write a Java program that demonstrates the concept of **importing packages**.

Theory:

- A **package** in Java is a collection of classes and interfaces.
- It helps in:
    1. **Organizing classes** logically.
    2. **Avoiding name conflicts**.
    3. **Reusing code**.
- Java has two types of packages:
    1. **Built-in packages** (e.g., java.util, java.io, java.lang, java.sql).
    2. **User-defined packages**.
- Packages are imported using the import keyword:
- import packageName.className;
- import packageName.*;

Algorithm:

1. Import a built-in package (like java.util).
2. Use a class (like Scanner) from that package.
3. Compile and execute the program.
4. Observe the result.

Program 1: Using Built-in Package (java.util.Scanner)

```
import java.util.Scanner;  // importing Scanner class from java.util package

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

```
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = sc.nextLine();

        System.out.print("Enter your age: ");
        int age = sc.nextInt();

        System.out.println("Hello " + name + "! You are " + age + " years old.");

        sc.close();
    }
}
```

Sample Output:

```
Enter your name: Neha
Enter your age: 20
Hello Neha! You are 20 years old.
```

Program 2: Using User-Defined Package

Step 1: Create a package file (mypack/Message.java)

```
package mypack;  // package name

public class Message {
    public void display() {
        System.out.println("Hello from User-Defined Package!");
    }
}
```

Step 2: Create the main file (TestPackage.java)

```
import mypack.Message;  // importing user-defined package

public class TestPackage {
    public static void main(String[] args) {
        Message m = new Message();
```

```
        m.display();
    }
}
```

Compilation & Execution:

```
> javac mypack/Message.java
> javac TestPackage.java
> java TestPackage
```

Output:

```
Hello from User-Defined Package!
```

Unit:-11

Exercise on interfaces.

Aim:

To write Java programs to demonstrate the concept of **interfaces**.

Theory:

- An **interface** in Java is a collection of **abstract methods** (methods without body) and constants.
- It is used to achieve **abstraction** and **multiple inheritance**.
- A class implements an interface using the implements keyword.
- From Java 8 onwards, interfaces can also have **default** and **static methods**.
- Key Points:
    1. All methods in an interface are implicitly public and abstract.
    2. A class can **implement multiple interfaces**.
    3. Interfaces support **runtime polymorphism**.

Algorithm:

1. Define an interface with one or more abstract methods.
2. Create a class that implements the interface.
3. Provide implementation for all abstract methods.
4. Create objects of the implementing class and call the methods.

Program 1: Simple Interface Example

```
// Interface
interface Animal {
   void sound();  // abstract method
}

// Implementing class
class Dog implements Animal {
```

```
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat implements Animal {
    public void sound() {
        System.out.println("Cat meows");
    }
}

// Main class
public class InterfaceDemo {
    public static void main(String[] args) {
        Animal a1 = new Dog();
        Animal a2 = new Cat();

        a1.sound();
        a2.sound();
    }
}
```

Sample Output:

```
Dog barks
Cat meows
```

Program 2: Multiple Interfaces Example

```
// First interface
interface Printable {
    void print();
}

// Second interface
interface Showable {
    void show();
}
```

```
// Implementing both interfaces
class Display implements Printable, Showable {
    public void print() {
        System.out.println("Printing from Printable Interface");
    }
    public void show() {
        System.out.println("Showing from Showable Interface");
    }
}

public class MultipleInterfaceDemo {
    public static void main(String[] args) {
        Display d = new Display();
        d.print();
        d.show();
    }
}
```

Sample Output:

```
Printing from Printable Interface
Showing from Showable Interface
```

Unit:-12

Exercise on exception handling.

Aim:

To write Java programs that demonstrate the concept of **exception handling** using try, catch, finally, throw, and throws.

Theory:

- **Exception**: An unwanted event that occurs during program execution and disrupts the flow.
- **Exception Handling**: Mechanism to handle runtime errors, ensuring the program continues execution.
- Keywords used:
    1. try – block of code to test for exceptions.
    2. catch – block of code to handle exceptions.
    3. finally – block that always executes (cleanup code).
    4. throw – used to explicitly throw an exception.
    5. throws – used in method signature to declare exceptions.

Algorithm:

1. Write code that may cause an exception inside a try block.
2. Handle the exception using catch.
3. Use finally for cleanup code (optional).
4. Test with different inputs to see exception handling in action.

Program 1: Handling ArithmeticException

```
public class ExceptionDemo1 {
   public static void main(String[] args) {
      try {
         int a = 10, b = 0;
```

```
            int result = a / b; // risky code
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is not allowed.");
        } finally {
            System.out.println("Program execution completed.");
        }
    }
}
```

Sample Output:

```
Error: Division by zero is not allowed.
Program execution completed.
```

Program 2: ArrayIndexOutOfBoundsException

```
public class ExceptionDemo2 {
    public static void main(String[] args) {
        try {
            int[] arr = {10, 20, 30};
            System.out.println(arr[5]); // invalid index
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Array index out of range.");
        }
    }
}
```

Sample Output:

```
Error: Array index out of range.
```

Program 3: Using throw and throws

```
class InvalidAgeException extends Exception {
   public InvalidAgeException(String msg) {
      super(msg);
   }
}

class Voting {
   void checkAge(int age) throws InvalidAgeException {
      if (age < 18) {
         throw new InvalidAgeException("Not eligible for voting");
      } else {
         System.out.println("Eligible for voting");
      }
   }
}

public class ExceptionDemo3 {
   public static void main(String[] args) {
      Voting v = new Voting();
      try {
         v.checkAge(16); // risky input
      } catch (InvalidAgeException e) {
         System.out.println("Exception Caught: " + e.getMessage());
      }
   }
}
```


Sample Output:

```
Exception Caught: Not eligible for voting
```

Unit:-13

Exercise on multithreading and thread priorities.

Aim:

To write Java programs demonstrating **multithreading** and the concept of **thread priorities**.

Theory:

- **Multithreading**: The process of executing multiple threads concurrently in a single program.
- **Thread**: A lightweight process that executes independently.
- Java provides two ways to create a thread:
    1. Extending the Thread class.
    2. Implementing the Runnable interface.
- **Thread Priorities**:
    - Each thread has a priority (1 to 10).
    - Constants: Thread.MIN_PRIORITY (1), Thread.NORM_PRIORITY (5), Thread.MAX_PRIORITY (10).
    - Scheduler generally uses priorities to decide execution order (not guaranteed, depends on JVM/OS).

Algorithm:

1. Create a class extending Thread (or implementing Runnable).
2. Override the run() method with the task to execute.
3. Create multiple thread objects.
4. Set priorities using setPriority().
5. Start threads using start().
6. Observe execution order (may vary due to scheduler).

Program 1: Creating Threads by Extending Thread Class

```
class MyThread extends Thread {
   public void run() {
      for (int i = 1; i <= 3; i++) {
         System.out.println(Thread.currentThread().getName() + " - Count: " +
i);
         try {
            Thread.sleep(500); // pause for 0.5 sec
         } catch (InterruptedException e) {
            System.out.println(e);
         }
      }
   }
}

public class ThreadDemo {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();
      MyThread t3 = new MyThread();

      t1.setName("Thread-1");
      t2.setName("Thread-2");
      t3.setName("Thread-3");

      t1.start();
      t2.start();
      t3.start();
   }
}
```


Sample Output (may vary):

```
Thread-1 - Count: 1
Thread-2 - Count: 1
Thread-3 - Count: 1
Thread-1 - Count: 2
Thread-2 - Count: 2
Thread-3 - Count: 2
Thread-1 - Count: 3
```

```
Thread-2 - Count: 3
Thread-3 - Count: 3
```

Program 2: Thread Priorities Example

```
class PriorityThread extends Thread {
   public void run() {
      System.out.println(Thread.currentThread().getName() + " with priority "
                  + Thread.currentThread().getPriority());
   }
}

public class ThreadPriorityDemo {
   public static void main(String[] args) {
      PriorityThread t1 = new PriorityThread();
      PriorityThread t2 = new PriorityThread();
      PriorityThread t3 = new PriorityThread();

      t1.setName("Low Priority Thread");
      t2.setName("Normal Priority Thread");
      t3.setName("High Priority Thread");

      t1.setPriority(Thread.MIN_PRIORITY); // 1
      t2.setPriority(Thread.NORM_PRIORITY); // 5
      t3.setPriority(Thread.MAX_PRIORITY); // 10

      t1.start();
      t2.start();
      t3.start();
   }
}
```

Sample Output (may vary due to scheduler):

```
Low Priority Thread with priority 1
Normal Priority Thread with priority 5
High Priority Thread with priority 10
```

Unit:- 14

Exercise on database connectivity using JDBC.

Aim:

To write a Java program that demonstrates **database connectivity** using **JDBC (Java Database Connectivity)**.

Theory:

- **JDBC** is a Java API that allows Java programs to interact with relational databases (e.g., MySQL, Oracle, PostgreSQL).
- Steps in JDBC:
    1. **Import the JDBC package** (java.sql.*).
    2. **Load the Driver** (Class.forName("com.mysql.cj.jdbc.Driver")).
    3. **Establish a Connection** (DriverManager.getConnection(...)).
    4. **Create a Statement/PreparedStatement**.
    5. **Execute Query/Update**.
    6. **Process the ResultSet** (for SELECT queries).
    7. **Close the Connection**.

Algorithm:

1. Import required JDBC classes.
2. Load and register the database driver.
3. Establish connection using DriverManager.
4. Create a Statement or PreparedStatement.
5. Execute SQL queries (Insert, Update, Delete, Select).
6. Display results (if any).
7. Close connection.

Program: JDBC Connectivity with MySQL

(Assuming MySQL is installed, a database college is created, and a table students(id INT, name VARCHAR(50), age INT) exists.)

```
import java.sql.*;
```

```
public class JdbcDemo {
   public static void main(String[] args) {
      try {
         // 1. Load JDBC Driver
         Class.forName("com.mysql.cj.jdbc.Driver");

         // 2. Establish Connection
         Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/college", "root", "password");

         // 3. Create Statement
         Statement stmt = con.createStatement();

         // 4. Insert Data
         String insertQuery = "INSERT INTO students VALUES (1, 'Amit',
20)";
         stmt.executeUpdate(insertQuery);
         System.out.println("Record inserted successfully!");

         // 5. Retrieve Data
         ResultSet rs = stmt.executeQuery("SELECT * FROM students");

         System.out.println("ID\tName\tAge");
         W
hile (rs.next()) {
            System.out.println(rs.getInt(1) + "\t" + rs.getString(2) + "\t" +
rs.getInt(3));
         }

         // 6. Close Connection
         con.close();

      } catch (Exception e) {
         System.out.println("Error: " + e);
      }
   }
}
```

Sample Output (after inserting one student):

```
Record inserted successfully!
ID   Name   Age
```

1 Amit 20