CBCS SCHEME

USN | | | | | | | | | |

BCS306A

Third Semester B.E./B.Tech. Degree Examination, Dec.2024/Jan.2025

Object Oriented Programming with JAVA

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	List and explain any three features of object oriented programming.	6	L1	CO1
	b.	What do you mean by type conversion and type casting? Give examples.	8	L2	CO1
	c.	How to declare and initialize 1-D and 2-D arrays in Java. Give examples.	6	L2	CO1
		OR			
Q.2	a.	List the short circuit operators and show the concept using few examples.	4	L2	CO1
	b.	With a java program, illustrate the use of ternary operator to find the greatest of three numbers.	6	L3	CO1
	c.	Develop a Java program to demonstrate the working of for each version of for loop. Initialize the 2D array with values and print them using for each.	10	L2	CO1
		Module – 2			
Q.3	a.	Develop a program in Java to implement a stack of integers.	12	L3	CO2
	b.	What are constructors? Give the types and explain the properties of constructors. Support with appropriate examples.	8	L2	CO2
		OR			
Q.4	a.	Illustrate with an example program to pass objects as arguments.	10	L2	CO2
	b.	Explain different access specifies in Java with example program.	10	L2	CO2
		Module – 3			
Q.5	a.	Define inheritance. List and explain different types of inheritance in Java with code snippets.	10	L2	CO3
	b.	Compare and contrast between overloading and overriding in Java with example program for each.	10	L2	CO3
		OR			
Q.6	a.	Analyze an interface in Java and list out the speed of an interface. Illustrate with the help of a program the importance of an interface.	10	L2	CO3
	b.	List the different uses of final and demonstrate each with the of code snippets.	10	L2	CO3

Module – 4					
Q.7	**a.**	Define a package. Explain how to create user defined package with example.	**7**	**L2**	**CO4**
	b.	Discuss about exception handling in Java. Give the framework of the exception handling block. List the types of exception.	**8**	**L2**	**CO4**
	c.	Develop a Java program to raise a custom exception for division by zero using try, catch, throw and finally.	**5**	**L3**	**CO4**
OR					
Q.8	**a.**	Compare throw and throws keyword by providing suitable example program.	**10**	**L2**	**CO4**
	b.	Explain about the need for finally block.	**5**	**L2**	**CO4**
	c.	Discuss about chained exceptions.	**5**	**L2**	**CO4**
Module – 5					
Q.9	**a.**	Define thread. Demonstrate creation of multiple threads with a program.	**10**	**L2**	**CO5**
	b.	Explain the two ways in which Java threads can be instantiated. Support your explanation with a sample program.	**10**	**L2**	**CO5**
OR					
Q.10	**a.**	What is enumeration? Explain the methods values( ) and valueof( ).	**10**	**L2**	**CO5**
	b.	Explain about type wrappers and auto boxing.	**10**	**L2**	**CO5**

* * * * *

1 a) List and explain any three features of the Object Oriented Programming.

The three OOP features are –
1. Encapsulation
2. Inheritance
3. Polymorphism

1. Encapsulation:

● Encapsulation is the mechanism that binds together code and data it manipulates, and keeps both safe from outside interference and misuse.

● Encapsulation is the wrapping of data and function or method into a single unit.

● Encapsulation is a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

In Java, the basis of encapsulation is the class. A class defines the structure and behavior (data and code) that a set of objects will share. Objects are referred to as instances of a class.
Thus, a class is a logical construct; an object has physical reality. (Class is like a blueprint of a building and object is the real building). The code and data that constitute a class is collectively called members of the class. The data are referred to as member variables or instance variables. The code that operates on the data is referred to as member methods or just methods. Methods define how the member variables can be used. That is, the behavior and interface of a class are defined by the methods that operate on its instance data.

There are mechanisms for hiding the complexity of the implementation inside the class because the purpose of the class is to encapsulate complexity. Each member or variable in a class can be marked private or public. The public interface of a class represents everything that external users of the class need to know. The private methods and data can only be accessed by code that is a member of the class. Any other code that is not a member of the class cannot access a private method or variable.

2. Inheritance:

1. Inheritance is the process by which one object acquires the properties of another object.
2. Inheritance supports the concept of hierarchical classification. For example, a Golden Retriever belongs to the class - dog, a dog, in turn, is part of the class mammal, and mammals are under the larger class animal. Mammals are called the subclass of animals and animals are called the mammal's superclass.
3. Without inheritance, each object has to define all of its characteristics explicitly. But, by use of inheritance, an object needs to define only those qualities that make it unique within its class. It inherits its general attributes from its parent. Therefore, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

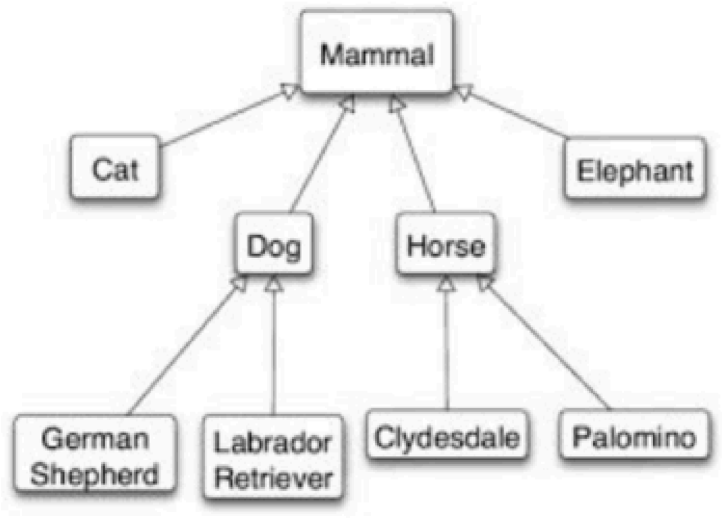


Fig: Animal Kingdom – Example for Inheritance

4. Inheritance interacts with encapsulation. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization. It is this key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new sub-class inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

3. Polymorphism:
Polymorphism in Greek means "many forms". It is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. For example, consider a program to implement three types of stacks, say, one for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, but the data stored is different. In a process-oriented model, we have to create three different stack routines each with different names. However, because of polymorphism, in Java we can create a general set of stack routines, all having the same name.

The concept of polymorphism is expressed by the phrase "one interface, multiple methods." It means that it is possible to design a generic interface to a group of related activities. Polymorphism helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action (or interface or method) as it applies to each situation. The programmer need not select the method manually.

1 b) What do you mean by type conversion and type casting? Give examples.

Java's Automatic Conversions: When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

1. The two types are compatible.

2. The destination type is larger than the source type

When these two conditions are met, a **widening conversion** takes place. For widening conversions, the numeric types, including integer and floating-point types are compatible with each other. There are no **automatic conversions from the numeric types to char or boolean**. Also char or boolean are not compatible with each other. Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types: If we want to **assign an int value to a byte variable**, conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is called **narrowing conversion** since we are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between the two incompatible types, we must use a cast. A cast is simply an explicit type conversion.

The general form of cast is -
(target-type) value

target-type specifies the desired type to convert the specified value to.
For example to cast an int to a byte

```
int a=20;
byte b;
 b= (byte) a;
```

If the **integer value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.** A different type of conversion called **truncation** will occur when a floating-point value is assigned to an integer type. Integers do not have fractional components. Hence **when a floating point value is assigned to an integer type, the fractional component is lost.** For example, if the value 1.23 is assigned to an integer, the resulting value will be 1. The 0.23 will be truncated. If the size of the whole number component is too large to fit into the target integer type, then the value will be reduced modulo the target type's range.

1 c) How do you declare and initialize 1-D and 2-D arrays in Java? Give Example

One-Dimensional Arrays:

A one-dimensional array is a list of similar data type variables. The general form of the one-dimensional array is

type var-name[];

The type determines the data type of each element of the array.

Example:

int month_days[ ];

Even though the above declaration tells that **month_days** is an array variable, no array exists. The value of **month_days** is set to null, that is, it represents an array with no value. To link **month_days** with an actual, physical array of integers, we must allocate memory using new and assign it to **month_days.**

new is a special operator that allocates memory. The syntax to allocate memory using the new operator is

array-var = new type [size];

Example:

month_days = new int [12];

Now, the **month_days** will refer to an array of 12 integers. **At the same time all the elements in the array will be initialized to zero.**

The following program demonstrates one dimensional array // Program to demonstrate one-dimensional array

```
class Array {

    public static void main(String args[]) {
            int month_days[];
            month_days = new int[12];
            month_days[0] = 31;
            month_days[1] = 28;
            month_days[2] = 31;
            month_days[3] = 30;
            month_days[4] = 31;
            month_days[5] = 30;
            month_days[6] = 31;
            month_days[7] = 31;
            month_days[8] = 30;
            month_days[9] = 31;
            month_days[10] = 30;
            month_days[11] = 31;
            System.out.println("April has " + month_days[3] + " days. ");
    }
}
```

One step process to define a array:

It is possible to combine the declaration of the array variable with the allocation of the array as shown below : Syntax: type array-var [ ] = new type [size];

Example: int month_days[] = new int [12];

Initialization of one dimensional array: Arrays can be initialized when they are declared. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will be automatically created large enough to hold the number of elements you specify in the array initializer. **There is no need to use new.**

Example:

```
class AutoArray {
    public static void main(String args[]) {
              int month_days [] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
              System.out.println("April has " + month_days[3] + " days. ");
    }
}
```

TWO DIMENSIONAL ARRAYS:

In Java, multidimensional arrays are array of arrays. To declare a multidimensional array variable, specify each element index using another set of square brackets.

Syntax:

type var-name[ ] [ ];

var-name = new type [size] [size];

or

type var-name [ ] [ ] = new type [size] [size];

Example int twoD [ ] [ ];

twoD = new int [4][5];

or

int twoD[ ] [ ] = new int [4][5];

The following program demonstrates a two-dimensional array

```
class TwoDArray {
    public static void main(String args[]) {
         int twoD [ ] [ ] = new int [4] [5];
         int i, j, k=0;
         for(i=0;i<4;i++)
            for(j=0;j<5;j++) {
             twoD[i][j] = k;
             k++;
             }
```

```
        for(i=0;i<4;i++) {
         for(j=0;j<5;j++)
                System.out.print (twoD[i][j] + " ");
                System.out.println();
                }
        }
}
```

The above program numbers each element in the array from the left to right, top to bottom, and then displays those values.

The output of the program is -

```
0 1 2 3 4 5 6 7 8 9
10  11  12  13  14  15
16 17 18 19
```

When you allocate memory for a multidimensional array, we need to only specify the memory for the first (leftmost) dimension. Then we can allocate the remaining dimensions separately.

Example

```
int twoD[] [] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
 twoD[3] = new int[5];
```

The advantage of allocating the second dimension array separately is we need not allocate the same number of elements for each dimension.

The following program creates a two-dimensional array in which the size of the second dimension is unequal.

```
// Program to demonstrate differing sizes in the second dimension
class TwoDAgain {
   public static void main(String args[]) {
      int twoD[][] = new int[4][];
      twoD[0] = new int[1];
      twoD[1] = new int[2];
      twoD[2] = new int[3];
      twoD[3] = new int[4];

      int i, j, k = 0;
```

```
        // Initialize the array with values
        for (i = 0; i < 4; i++) {
            for (j = 0; j < i + 1; j++) {
                twoD[i][j] = k;
                k++;
            }
        }

        // Display the array
        for (i = 0; i < 4; i++) {
            for (j = 0; j < i + 1; j++) {
                System.out.print(twoD[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

INITIALIZATION OF TWO-DIMENSIONAL ARRAYS:

The following example demonstrates how to initialize a multi-dimensional array

```
// Program to initialize two dimensional array class Matrix {
    public static void main(String args[]) {
            int array [ ] [ ] = {
                                    { 1, 2, 3, 4 },
                                    { 2, 3, 4, 5 },
                                    { 3, 4, 5, 6 },
                                    { 4, 5, 6, 7 }
                            };
                int i, j;
                for(i=0;i<4;i++) {
                            for(j=0;j<i+4;j++)
                               System.out.print(twoD[i][j] + " ");
                           System.out.println();
                }
```

The output of this program is

```
                1 2 3 4
```

```
2 3 4 5
3 4 5 6
 4 5 6 7
```

2 a) List the short circuit operators and show the concept using few examples.

Short-Circuit Operators:

Java provides two Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators and are known as short-circuit operators.

The OR operator results in true when the left-hand operand is true, no matter what the right-hand operand is. Similarly, the AND operator results in false when the left-hand operand is false, no matter what the right-hand operand is.

When we use || and && forms, rather than | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

Example

if (denom !=0 && num /denom > 10)

As the short circuit form of && is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero.

2 b) With a Java program illustrate the use of the ternary operator to find the greatest of three numbers.

The?: Operator:

Java includes a ternary (three-way) operator that can replace certain types of if-then-else-statements.

The operator is **?:**

General form:

expression1 ? Expression2 : expression3

expression1 can be any expression that evaluates to a boolean value.
If expression1 is true then expression2 is evaluated, else, expression3 is evaluated.

Example:
ratio = denom == 0 ? 0 : num /denom;
if denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? Expression.
If denom is not equal to zero, then the expression after the colon is evaluated and used for the

value of the entire ? Expression.
The result is then assigned to the ratio.

Program to find the greatest of three numbers using ternary operator -

```
import java.util.Scanner;

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

        // Taking input from user
        System.out.print("Enter first number: ");
        int a = sc.nextInt();
        System.out.print("Enter second number: ");
        int b = sc.nextInt();
        System.out.print("Enter third number: ");
        int c = sc.nextInt();

        // Using nested ternary operator to find the greatest number
        int max = (a > b) ? (a > c ? a : c) : (b > c ? b : c);

        // Printing the result
        System.out.println("The greatest number is: " + max);
        sc.close();
    }
}
```

2 c) Develop a Java program to demonstrate the working of for each version of for loop. Initialize the 2-D array with values and print them using for each loop.

```
public class ForEach2DArray {
    public static void main(String[] args) {
        // Initializing a 2D array
        int[][] numbers = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Using for-each loop to iterate through the 2D array
        System.out.println("Elements of the 2D array:");
        for (int[] row : numbers) {  // Iterating through each row
            for (int num : row) {    // Iterating through each element in the row
                System.out.print(num + " ");
            }
            System.out.println();  // Move to the next line after printing a row
        }
    }
}
```

3a) Develop a Java program to implement a stack of integers.

```
import java.util.Scanner;

class Stack {
    private int top;       // Index of the top element
    private int[] stack;   // Array to store stack elements
    private int maxSize;   // Maximum capacity of the stack

    // Constructor to initialize the stack
    public Stack(int size) {
        maxSize = size;
        stack = new int[maxSize];
        top = -1; // Stack is initially empty
    }

    // Push operation: Add element to stack
    public void push(int value) {
        if (top == maxSize - 1) {
            System.out.println("Stack Overflow! Cannot push " + value);
        } else {
            stack[++top] = value;
            System.out.println(value + " pushed into the stack.");
        }
    }

    // Pop operation: Remove and return top element
    public int pop() {
        if (top == -1) {
            System.out.println("Stack Underflow! Cannot pop.");
            return -1;
        } else {
            return stack[top--];
        }
    }

    // Peek operation: View the top element
    public int peek() {
        if (top == -1) {
            System.out.println("Stack is empty!");
            return -1;
        } else {
            return stack[top];
        }
    }

    // Check if stack is empty
    public boolean isEmpty() {
        return top == -1;
    }

    // Display stack elements
    public void display() {
        if (top == -1) {
            System.out.println("Stack is empty!");
        } else {
            System.out.print("Stack elements: ");
            for (int i = top; i >= 0; i--) {
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }
}

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

        // Initialize stack of size 5
        Stack stack = new Stack(5);

        // Menu-driven approach
        while (true) {
            System.out.println("\nStack Operations:");
            System.out.println("1. Push");
            System.out.println("2. Pop");
            System.out.println("3. Peek");
            System.out.println("4. Check if Empty");
            System.out.println("5. Display Stack");
            System.out.println("6. Exit");
            System.out.print("Enter your choice: ");
            int choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter value to push: ");
                    int value = sc.nextInt();
                    stack.push(value);
                    break;
                case 2:
                    int popped = stack.pop();
                    if (popped != -1) {
                        System.out.println("Popped element: " + popped);
                    }
                    break;
                case 3:
                    int topElement = stack.peek();
                    if (topElement != -1) {
                        System.out.println("Top element: " + topElement);
                    }
                    break;
                case 4:
                    System.out.println(stack.isEmpty() ? "Stack is empty." : "Stack is not empty.");
                    break;
                case 5:
                    stack.display();
                    break;
                case 6:
                    System.out.println("Exiting...");
                    sc.close();
                    System.exit(0);
                    break;
                default:
                    System.out.println("Invalid choice! Try again.");
            }
        }
    }
}
```

Output:

```
Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
```

```
5. Display Stack
6. Exit
Enter your choice: 1
Enter value to push: 10
10 pushed into the stack.

Enter your choice: 1
Enter value to push: 20
20 pushed into the stack.

Enter your choice: 3
Top element: 20

Enter your choice: 2
Popped element: 20

Enter your choice: 5
Stack elements: 10

Enter your choice: 6
Exiting...
```

3 b) What are constructors? Give the types and explain the properties of constructors, support with appropriate examples.

A **constructor** in Java is a **special method** used to initialize objects. It is called automatically when an object is created with the **same name as the class**.

Properties of Constructors:

1. **Name Same as Class**
   - The constructor must have the same name as the class.
2. **No Return Type**
   - Constructors do not have a return type, not even `void`.
3. **Automatic Invocation**
   - Constructors are automatically invoked when an object is created using the `new` keyword.
4. **Overloading**
   - Constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists.

Types of Constructors

1. Default constructor
2. Explicit Parameterless Constructor
3. Parameterized constructor

4. Copy constructor

1. **Default Constructor**
    - A constructor with no parameters.
    - If no constructor is explicitly defined in a class, the Java compiler automatically provides a default constructor.
    - It initializes the object with default values (e.g., `null` for objects, `0` for integers, `false` for booleans).

Example program:

```
class MyClass {
   int num;
   String str;

   // Default constructor (provided by the compiler if not explicitly defined)
   MyClass() {
      num = 0;
      str = null;
   }
}

public class Main {
   public static void main(String[] args) {
      MyClass obj = new MyClass(); // Default constructor is called
      System.out.println("num: " + obj.num); // Output: num: 0
      System.out.println("str: " + obj.str); // Output: str: null
   }
}
```

2. Explicit Parameterless Constructor

A user can define their parameterless constructor to initialize an object with default values.

Example Program

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

   // Parameterless constructor
   public Student() {
      name = "Unknown";
      age = 18;
   }
```

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(); // Calls the parameterless constructor
        s1.display(); // Output: Name: Unknown, Age: 18
    }
}
```

3. Parameterized Constructor

- A constructor that takes one or more parameters.
- Used to initialize objects with specific values passed as arguments.

Example Program:

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

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

public class Main {
    public static void main(String[] args) {
        Student student = new Student("Alice", 20); // Parameterized constructor is called
        System.out.println("Name: " + student.name); // Output: Name: Alice
        System.out.println("Age: " + student.age);   // Output: Age: 20
    }
}
```

4. **Copy Constructor**

- A constructor that takes an object of the same class as a parameter and copies its values to the new object.
- Used to create a copy of an existing object.

Example Program:

```
class Point {
    int x, y;

    // Parameterized constructor
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Copy constructor
    Point(Point p) {
        this.x = p.x;
        this.y = p.y;
    }
}

public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(10, 20); // Parameterized constructor
        Point p2 = new Point(p1);     // Copy constructor
        System.out.println("p2.x: " + p2.x); // Output: p2.x: 10
        System.out.println("p2.y: " + p2.y); // Output: p2.y: 20
    }
}
```

4a) Illustrate with an example program to pass objects as arguments.

In Java, objects can be passed as arguments to methods just like primitive data types. When an object is passed, its reference (memory address) is passed, allowing the method to access and modify the original object's properties.

```
// Class representing a simple Rectangle
class Rectangle {
    int length, breadth;

    // Constructor to initialize the rectangle
    Rectangle(int l, int b) {
        length = l;
        breadth = b;
    }

    // Method that takes another Rectangle object as an argument
    void compareArea(Rectangle r) {
        int area1 = this.length * this.breadth;
        int area2 = r.length * r.breadth;

        if (area1 > area2) {
            System.out.println("Current rectangle is larger.");
        } else if (area1 < area2) {
            System.out.println("Passed rectangle is larger.");
        } else {
            System.out.println("Both rectangles have the same area.");
        }
    }
}

// Main class to demonstrate passing objects
public class ObjectAsArgument {
    public static void main(String[] args) {
        Rectangle rect1 = new Rectangle(5, 10);  // Create first object
        Rectangle rect2 = new Rectangle(6, 8);   // Create second object

        // Compare areas of two rectangle objects
        rect1.compareArea(rect2);
    }
}
```

Class Rectangle has two instance variables length and breadth.
A constructor initializes these values.
The method compareArea(Rectangle r) accepts another Rectangle object as an argument and compares its area with the current object.
The main method creates two Rectangle objects and calls compareArea to compare their areas.

4b) Explain different access specifies in Java with example program.

Access Specifiers in Java

Access specifiers (or access modifiers) in Java define the scope and visibility of classes, methods, and variables.
Java provides four types of access specifiers:

1. Public
2. Private
3. Protected
4. Default (No Modifier)

1. Public Access Specifier

A public member is accessible from anywhere in the program. It can be accessed from different classes and packages.

```
package demo1;  // Package demo1

// Class with a public method
public class PublicExample {
    public void display() {
        System.out.println("Public method is accessible anywhere.");
    }
}

package demo2;  // Different package

import demo1.PublicExample;  // Importing the class

public class TestPublic {
    public static void main(String[] args) {
        PublicExample obj = new PublicExample(); // Allowed
        obj.display();  // Accessible
    }
}
```

Output:

```
Public method is accessible anywhere.
```

2. Private Access Specifier

A private member is accessible only within the same class. It cannot be accessed by other classes, even in the same package.

```
class PrivateExample {
    private int data = 50;

    private void show() {
        System.out.println("This is a private method.");
    }

    public void accessPrivate() {
        System.out.println("Accessing private variable: " + data);
        show();  // Private method can be accessed within the same class
    }
}
```

```
public class TestPrivate {
    public static void main(String[] args) {
        PrivateExample obj = new PrivateExample();
        obj.accessPrivate(); // Allowed
        // obj.show();  // Not allowed (Compile-time error)
        // System.out.println(obj.data);  // Not allowed (Compile-time error)
    }
}
```

Output:

```
Accessing private variable: 50
This is a private method.
Error if we directly access obj.data or obj.show().
```

3. Protected Access Specifier

A protected member is accessible: Within the same package. In subclasses (even in different packages).

```
package demo1;
public class ProtectedExample {
    protected void show() {
        System.out.println("Protected method can be accessed in subclasses.");
    }
}

package demo2;
import demo1.ProtectedExample;

// Subclass in a different package
class SubClass extends ProtectedExample {
    public void callShow() {
        show();  // Allowed in subclass
    }
}

public class TestProtected {
    public static void main(String[] args) {
        SubClass obj = new SubClass();
        obj.callShow();
    }
}
```

Output:
Protected method can be accessed in subclasses.

4. Default (No Modifier) Access Specifier

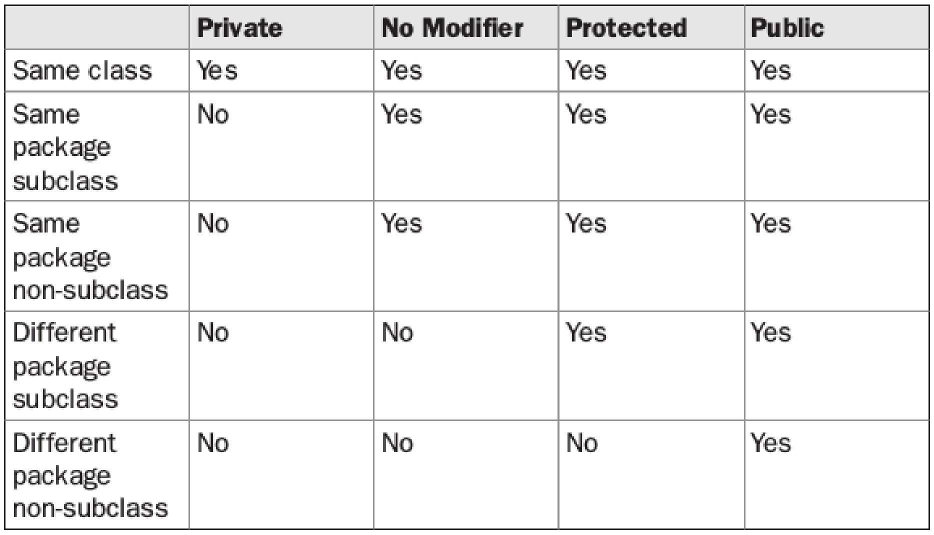
When no modifier is specified, it is called default access. The member is accessible only within the same package. It cannot be accessed outside the package.

```
package demo1;
class DefaultExample {
    void display() {
        System.out.println("Default method is accessible within the same package.");
    }
}

public class TestDefault {
    public static void main(String[] args) {
        DefaultExample obj = new DefaultExample();
        obj.display();  // Allowed
    }
}
```

If you try to access DefaultExample from another package, you'll get a compile-time error.

TABLE 9-1
Class Member Access



	Private	**No Modifier**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

5a) Define inheritance. List and explain different types of inheritance in Java with code snippets

Inheritance is a fundamental concept of Object-Oriented Programming (OOP) where one class (child class) derives properties and behaviors from another class (parent class). It enables code reusability and hierarchical classification.

In Java, inheritance is implemented using the extends keyword.

Types of Inheritance in Java:
Java supports the following types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance using Interfaces**

Note: Java does not support multiple inheritance with classes to avoid ambiguity but allows it through interfaces.

1. Single Inheritance

One class inherits from a single parent class.
The child class gets access to all non-private methods and variables of the parent class.

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

// Child class inheriting Animal
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks.");
   }
}

public class SingleInheritanceDemo {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.sound();  // Inherited method
      d.bark();
   }
}
```

Output:

```
Animals make sound.
Dog barks.
```

2. Multilevel Inheritance

A chain of inheritance where a class inherits from another class, which in turn inherits from another class.

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

// Intermediate class inheriting Animal
class Mammal extends Animal {
   void type() {
      System.out.println("Mammals give birth to young ones.");
   }
}

// Derived class inheriting Mammal
class Dog extends Mammal {
   void bark() {
      System.out.println("Dog barks.");
   }
}

public class MultilevelInheritanceDemo {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.sound();  // From Animal
      d.type();   // From Mammal
      d.bark();   // From Dog
   }
}
```

Output:

```
Animals make sound.
Mammals give birth to young ones.
Dog barks.
```

3. Hierarchical Inheritance

Multiple child classes inherit from a single parent class.

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

// Child class 1
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks.");
   }
}

// Child class 2
class Cat extends Animal {
   void meow() {
      System.out.println("Cat meows.");
   }
}

public class HierarchicalInheritanceDemo {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.sound();
      d.bark();

      Cat c = new Cat();
      c.sound();
      c.meow();
   }
}
```

Output:

```
Animals make sound.
Dog barks.
Animals make sound.
Cat meows.
```

4. Multiple Inheritance (Using Interfaces)

Java does not support multiple inheritance with classes to prevent ambiguity (diamond problem).

However, multiple inheritance is possible using interfaces.

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

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

// Class implementing both interfaces
class MultipleInheritanceDemo implements Printable, Showable {
    public void print() {
        System.out.println("Printing...");
    }

    public void show() {
        System.out.println("Showing...");
    }

    public static void main(String[] args) {
        MultipleInheritanceDemo obj = new MultipleInheritanceDemo();
        obj.print();
        obj.show();
    }
}
```

Output:

```
Printing...
Showing…
```

5b) Compare and contrast between overloading and overriding in Java with example program for each.

Feature	Method Overloading	Method Overriding
Definition	Defining multiple methods with the same name but different parameters within the same class.	Redefining a method from the parent class in the child class with the same signature.
Where does it occurs?	Within the **same class**.	Between **parent and child class** (inheritance required).
Parameters	Must be **different** (either in number, type, or both).	Must be **same** as in the parent class.
Return Type	Can be **different**.	Must be **same** or a **subtype** of the return type in the parent class.
Access Modifiers	No restrictions.	Cannot reduce the visibility (e.g., a public method cannot be overridden as private).
Static Methods	Can be overloaded.	Cannot be overridden (but can be hidden).
Final Methods	Can be overloaded.	**Cannot** be overridden.
Performance	Slightly faster as it is **resolved at compile time**.	Slightly slower due to **runtime polymorphism** (dynamic method dispatch).
Keyword Used	No specific keyword needed.	Uses @override annotation (optional but recommended).

1. Method Overloading

Multiple methods with same name but different parameters in the same class.

```
class MathOperations {
    // Method 1: Adding two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method 2: Adding three integers (Overloaded method)
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Adding two double values (Overloaded method)
    double add(double a, double b) {
        return a + b;
    }
}

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

        System.out.println("Sum (int, int): " + obj.add(5, 10));
        System.out.println("Sum (int, int, int): " + obj.add(5, 10, 15));
        System.out.println("Sum (double, double): " + obj.add(5.5, 10.5));
    }
}
```

Output:

```
Sum (int, int): 15
Sum (int, int, int): 30
Sum (double, double): 16.0
```

2. Method Overriding

A subclass provides a new implementation for a method already defined in the parent class.

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

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

public class OverridingDemo {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.sound();  // Calls parent class method

        Dog d = new Dog();
        d.sound();  // Calls overridden method in Dog class
    }
}
```

Output:

```
Animals make different sounds.
Dog barks.
```

6a) Analyze an interface in Java and list out the speed of an interface. Illustrate with the help of a program the importance of an interface.

An interface in Java is a blueprint of a class that contains only abstract methods and constants (before Java 8). It is used to achieve abstraction and multiple inheritance.

Declared using the interface keyword.

Methods inside an interface are implicitly public and abstract. Variables inside an interface are implicitly public, static, and final. A class implements an interface using the implements keyword. Since Java 8, interfaces can also have default and static methods.

Memory Usage: Interfaces do not store state (instance variables), so they are lightweight.
Execution Speed: Calling a method via an interface reference is slightly slower than a direct method call in an abstract class due to dynamic method dispatch.
Compilation Speed: Interfaces compile as fast as abstract classes.
Runtime Efficiency: If an interface is used extensively, JVM optimizations like inlining and just-in-time compilation (JIT) help reduce performance overhead.

Scenario: Implementing Multiple Inheritance Using Interfaces

Java does not support multiple inheritance with classes, but it allows multiple inheritance using interfaces.

```
// Interface 1: Animal
interface Animal {
    void eat();  // Abstract method
}

// Interface 2: Pet
interface Pet {
    void play();
}
```

```
// Class implementing multiple interfaces
class Dog implements Animal, Pet {
    // Implementing methods from interfaces
    public void eat() {
        System.out.println("Dog eats food.");
    }

    public void play() {
        System.out.println("Dog plays with a ball.");
    }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();  // Call method from Animal interface
        d.play(); // Call method from Pet interface
    }
}
```

Output:

```
Dog eats food.
Dog plays with a ball.
```

6b) List the different uses of final and demonstrate each with the of code snippets

The keyword final can be used in three situations in Java:

1. To create the equivalent of a named constant.
2. To prevent method overriding.
3. To prevent Inheritance

To create the equivalent of a named constant: A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared.

For example:

```
final int FILE_NEW = 1;
```

To prevent method overriding: Sometimes, we do not want a super class method to be overridden in the subclass. Instead, the same super class method definition has to be used by every subclass. In such situation, we can prefix a method with the keyword final as shown below –

```
class A
{
```

```
final void meth()
{
System.out.println("This is a final method.");
}
}
class B extends A
{
void meth() // ERROR! Can't override.
{
System.out.println("Illegal!");
}
}
```

To prevent Inheritance: As we have discussed earlier, the subclass is treated as a specialized class and super class is most generalized class. During multi-level inheritance, the bottom most class will be with all the features of real-time and hence it should not be inherited further. In such situations, we can prevent a particular class from inheriting further, using the keyword final.
For example –

```
final class A
{
// ...
}
class B extends A // ERROR! Can't subclass A
{
// ...
}
```

7a) Define a package. Explain how to create user defined package with example.

A package in Java is a collection of related classes and interfaces. It helps in organizing code, avoiding name conflicts, and enhancing reusability.

1. Create a package using the package keyword.
2. Compile the package using javac -d . Filename.java.
3. Import and use the package in another class using import package_name.*;.

Example: Creating and Using a User-Defined Package

Step 1: Create a Package (mypackage)

Save the following code as MyClass.java inside a directory.

```
// Step 1: Declare a package
package mypackage;
```

```
// Step 1.1: Create a class inside the package
public class MyClass {
    public void display() {
        System.out.println("This is a user-defined package example.");
    }
}
```

Step 2: Compile the Package
Run the following command in the terminal (inside the directory containing MyClass.java):

```
javac -d . MyClass.java
```
This will create a folder named mypackage containing the compiled .class file.

Step 3: Use the Package in Another Class
Create another Java file in the same directory and save it as TestPackage.java.

```
// Step 1: Import the package
import mypackage.MyClass;

public class TestPackage {
    public static void main(String[] args) {
        MyClass obj = new MyClass();  // Create object of class in package
        obj.display();  // Call method
    }
}
```

Step 4: Compile and Run the Program

Compile the main program:
```
javac TestPackage.java
```

Run the program:
```
java TestPackage
```

Output:

```
This is a user-defined package example.
```

7b) Discuss about exception handling in Java. Give the framework of the exception handling block. List the types of exception.

Exception handling in Java is a mechanism to handle runtime errors and ensure the smooth execution of a program. It prevents abnormal program termination by catching and handling errors gracefully.

An exception is an unexpected event that disrupts normal program flow.
Java provides a structured way to handle exceptions using the try-catch-finally blocks.
Framework of Exception Handling in Java

Java provides five key exception handling constructs:

1. try – Contains the code that may cause an exception.
2. catch – Catches and handles the exception.
3. finally – Block that executes always, whether an exception occurs or not.
4. throw – Used to explicitly throw an exception.
5. throws – Declares exceptions that a method can throw.

General Syntax:

```
try {
   // Code that may cause an exception
} catch (ExceptionType e) {
   // Handling the exception
} finally {
   // Code that executes always (optional)
}
```

Example of Exception Handling

```
public class ExceptionHandlingDemo {
   public static void main(String[] args) {
      try {
         int result = 10 / 0; // ArithmeticException occurs
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e);
      } finally {
         System.out.println("This block always executes.");
      }
   }
}
```

Output:

```
Exception caught: java.lang.ArithmeticException: / by zero
This block always executes.
```

Types of Exceptions in Java

1. Checked Exceptions (Compile-time Exceptions)

Exceptions that the compiler forces you to handle.
Occur at compile-time and must be handled using try-catch or throws.

Examples:
IOException (File not found)
SQLException (Database errors)
ClassNotFoundException (Class not found)

```
import java.io.*;

public class CheckedExceptionDemo {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("test.txt");  // FileNotFoundException
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e);
        }
    }
}
```

2. Unchecked Exceptions (Runtime Exceptions)

Exceptions that occur at runtime and are not checked at compile-time.
Usually caused by logical errors in the program.

Examples:
ArithmeticException (Divide by zero)
NullPointerException (Accessing an object with null reference)
ArrayIndexOutOfBoundsException (Accessing an array out of bounds)

```
public class UncheckedExceptionDemo {
    public static void main(String[] args) {
        int[] arr = new int[5];
        System.out.println(arr[10]);  // ArrayIndexOutOfBoundsException
    }
}
```

3. Errors

Severe issues that the application cannot recover from.
Usually related to system-level problems.
Examples:

StackOverflowError (Infinite recursion)
OutOfMemoryError (Heap memory exhausted)

```
public class ErrorDemo {
    public static void recursiveMethod() {
        recursiveMethod(); // Infinite recursion causes StackOverflowError
    }

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

7c) Develop a Java program to raise a custom exception for division by zero using try, catch, throw and finally.

```
// Step 1: Create a custom exception class
class DivideByZeroException extends Exception {
    // Constructor
    public DivideByZeroException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    // Step 2: Method to perform division
    public static int divide(int a, int b) throws DivideByZeroException {
        if (b == 0) {
            throw new DivideByZeroException("Error: Division by zero is not allowed!");
        }
        return a / b;
    }

    public static void main(String[] args) {
        try {
            // Step 3: Try to divide numbers
            int result = divide(10, 0);  // This will cause an exception
            System.out.println("Result: " + result);
        } catch (DivideByZeroException e) {
            // Step 4: Catch the custom exception
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            // Step 5: Finally block executes always
            System.out.println("Execution completed.");
        }
    }
}
```

Output:

```
Exception caught: Error: Division by zero is not allowed!
Execution completed.
```

Custom Exception (DivideByZeroException)
Inherits from Exception class.
Used to display a custom message.
Throwing the Exception (throw)
The divide method checks if b == 0, and if so, it throws DivideByZeroException.
Catching the Exception (try-catch)
The main method calls divide(10, 0), causing the exception.
The catch block catches the exception and prints a message.
Finally Block (finally)
Ensures that "Execution completed." is printed whether an exception occurs or not.

8a) Compare throw and throws keyword by providing suitable example program.

Both throw and throws are used for exception handling in Java, but they serve different purposes.

Feature	throw	throws
Definition	Used to explicitly throw an exception from within a method or block of code.	Used to declare exceptions that a method might throw, passing the responsibility to the caller.
Usage	Can be used within a method to throw a specific exception.	Used in the method signature to indicate that a method may throw an exception.
Scope	Used inside the method body to trigger an exception.	Used in the method signature to specify exceptions that a method might throw.
Example	throw new Exception("Message")	public void myMethod() throws Exception
Types of Exceptions	Can throw both checked and unchecked exceptions.	Can declare only checked exceptions.
Multiple Exceptions	You can throw a single exception at a time.	You can declare multiple xceptions using a omma-separated list.

throw is used inside a method to explicitly throw an exception.
throws is used in the method signature to declare that a method might throw an exception.
The responsibility for handling exceptions declared by throws lies with the caller, while exceptions thrown using throw are immediately handled or passed further up.

Java Program: Using throw and throws

```
// Custom exception class for Division by Zero
class DivideByZeroException extends Exception {
    public DivideByZeroException(String message) {
        super(message);
    }
}

public class ThrowVsThrowsDemo {

    // Method that throws an exception explicitly (using 'throw')
    public static int divide(int a, int b) throws DivideByZeroException {
        if (b == 0) {
            // Using 'throw' to raise the custom exception
            throw new DivideByZeroException("Error: Division by zero is not allowed.");
        }
        return a / b;
    }

    // Method that declares the exception (using 'throws')
    public static void performDivision(int a, int b) throws DivideByZeroException {
        // This method may throw an exception, so we declare it using 'throws'
        int result = divide(a, b);
        System.out.println("Result: " + result);
    }

    public static void main(String[] args) {
        try {
            // Call performDivision which throws a DivideByZeroException
            performDivision(10, 0);  // This will throw the exception
        } catch (DivideByZeroException e) {
            // Handle the exception
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

throw Keyword:

Inside the divide method, we use throw to explicitly throw a DivideByZeroException if b == 0. The exception is thrown manually when division by zero is detected.
throws Keyword:

In the performDivision method, we use throws to declare that this method may throw a DivideByZeroException. The responsibility of handling the exception is passed to the method's caller (which is the main method in this case).
Exception Handling in main:

The main method calls performDivision(10, 0). Since performDivision declares that it throws DivideByZeroException, we handle the exception in the catch block.

Output:
```
Exception caught: Error: Division by zero is not allowed.
```

8b) Explain about the need for finally block.

The finally block in Java is an important part of exception handling. It provides a way to ensure that certain cleanup actions are performed, regardless of whether an exception occurs or not. This ensures that resources are properly released, and any necessary cleanup tasks are executed after the try-catch block.

Syntax of the finally Block:

```
try {
   // Code that may cause an exception
} catch (ExceptionType e) {
   // Handling the exception
} finally {
   // Code that will execute regardless of exception (cleanup code)
}
```

finally will execute after the try block, regardless of whether an exception is thrown or caught.
The finally block is optional, but it is recommended to use when resources need to be cleaned up or finalized.

```
import java.io.*;

public class FinallyBlockDemo {
   public static void main(String[] args) {
      FileReader file = null;

      try {
         // Open the file
         file = new FileReader("testfile.txt");
```

```
            // Simulating an exception (e.g., file not found)
            int data = file.read();
            System.out.println("File Data: " + data);
        } catch (IOException e) {
            // Handling the exception
            System.out.println("Error reading the file: " + e);
        } finally {
            // Cleanup code (close the file stream)
            try {
                if (file != null) {
                    file.close();
                    System.out.println("File closed successfully.");
                }
            } catch (IOException e) {
                System.out.println("Error closing the file: " + e);
            }
        }
    }
}
```

Output:

```
Error reading the file: java.io.FileNotFoundException: testfile.txt (No such file or directory)
File closed successfully.
```

In the try block, we attempt to read from a file. If the file is not found, an IOException will be thrown.
The catch block handles the exception and prints the error message.
Regardless of whether an exception is thrown or not, the finally block ensures that the file is closed, preventing any potential resource leak.

Guaranteed Execution: The finally block always executes, even if:
An exception is thrown and not caught.
A return statement is executed in the try or catch block.
A System.exit() is called.
When finally Doesn't Execute:

If the JVM crashes or the program terminates abnormally, the finally block may not execute.
If the thread executing the try-catch-finally is interrupted or killed during execution, the finally block might not run.

8c) Discuss about chained exceptions.

Chained exceptions in Java refer to the mechanism where one exception is caused by another exception. It allows the propagation of exceptions in a chain, helping to track the original cause of the error. This concept is useful when handling complex issues, as it allows you to link related exceptions together, providing a more comprehensive explanation of what went wrong.

Chained exceptions help us understand the sequence of errors that led to the failure, making it easier to trace the root cause of the problem.
Chained exceptions provide more contextual information about the exception, helping developers debug the program more efficiently.
In scenarios where multiple errors happen, chaining allows the original error to be passed along, while each new exception provides additional details.
Chaining exceptions improves error reporting by preserving the original exception's context and including it in the new exception.

Java provides a mechanism for chaining exceptions using the Throwable constructor.

Specifically:
Throwable(Throwable cause): Allows you to associate an exception with another exception.
Throwable(String message, Throwable cause): Provides both a message and the underlying cause.

Syntax of Chained Exceptions:

```
try {
   // Code that may throw an exception
} catch (Exception e) {
   throw new SomeException("A new exception occurred", e);  // Chaining the original exception
}
```

The second parameter (e) in the constructor represents the cause of the new exception, i.e., the original exception that was thrown.

```
// Custom exception for demonstrating chaining
class InvalidInputException extends Exception {
   public InvalidInputException(String message) {
      super(message);
   }
}

class DataProcessingException extends Exception {
   public DataProcessingException(String message, Throwable cause) {
      super(message, cause);  // Chaining the cause (another exception)
   }
}

public class ChainedExceptionDemo {
   public static void main(String[] args) {
      try {
         processData("123abc");  // Invalid input will cause an exception
      } catch (DataProcessingException e) {
            System.out.println("Exception: " + e.getMessage());
            System.out.println("Caused by: " + e.getCause());
        }
    }

    // Method that throws an exception based on input
    public static void processData(String input) throws DataProcessingException {
        try {
            if (!input.matches("\\d+")) {  // Check if input is not a valid number
                throw new InvalidInputException("Input is not a valid number");
            }
            // Simulating further processing logic
            System.out.println("Processing data: " + input);
        } catch (InvalidInputException e) {
            // Catch the original exception and chain it into a new exception
            throw new DataProcessingException("Error processing data", e);
        }
    }
}
```

Output:

```
Exception: Error processing data
Caused by: InvalidInputException: Input is not a valid number
```

9a) Define thread. Demonstrate creation of multiple threads with a program.

A thread in Java is a lightweight process that allows multiple tasks to run concurrently within a program. Threads are the smallest unit of execution and share the resources of the parent process, such as memory and file handles. Java supports multithreading, which is the ability to run multiple threads concurrently, improving the performance of the program.

Java provides two ways to create a thread:

1. By Extending the Thread class
2. By Implementing the Runnable interface

1. Creating Multiple Threads by Extending the Thread Class

```
// Extending the Thread class
class MyThread extends Thread {
    @Override
    public void run() {
        // Code that will be executed by this thread
        System.out.println(Thread.currentThread().getId() + " is executing the thread.");
    }
}

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

        // Starting the threads
        t1.start();  // Thread 1 starts
        t2.start();  // Thread 2 starts
        t3.start();  // Thread 3 starts
    }
}
```

We create a class MyThread that extends the Thread class.
The run() method defines the code to be executed by the thread. This method is overridden.
We create three instances of the MyThread class and start them using the start() method.
Each thread runs concurrently, and the output will show different thread IDs executing.

Output:

```
1 is executing the thread.
2 is executing the thread.
3 is executing the thread.
```

2. Creating Multiple Threads by Implementing the Runnable Interface

```
// Implementing the Runnable interface
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code that will be executed by this thread
        System.out.println(Thread.currentThread().getId() + " is executing the Runnable.");
    }
}

public class RunnableThreadDemo {
    public static void main(String[] args) {
        // Creating Runnable instances
        MyRunnable myRunnable = new MyRunnable();

        // Creating Thread instances and passing the Runnable
        Thread t1 = new Thread(myRunnable);
        Thread t2 = new Thread(myRunnable);
        Thread t3 = new Thread(myRunnable);
        // Starting the threads
        t1.start();  // Thread 1 starts
        t2.start();  // Thread 2 starts
        t3.start();  // Thread 3 starts
    }
}
```

We define a class MyRunnable that implements the Runnable interface.
The run() method is overridden to define the task for the thread.
We create three Thread objects, passing the Runnable object as an argument to the Thread constructor.
Each thread is started using the start() method.

Output:

```
1 is executing the Runnable.
2 is executing the Runnable.
3 is executing the Runnable.
```

9b) Explain the two ways in which Java threads can be instantiated. Support your explanation with a sample program.

Two Ways to Instantiate Threads in Java

1. **By Extending the Thread class**
2. **By Implementing the Runnable interface**

Both methods allow you to define a task that a thread will execute, but the approach for each is slightly different. Let's explore both methods and see how they can be used with an example program.

1. Instantiating a Thread by Extending the Thread Class

In this approach, we extend the Thread class and override its run() method, which defines the code that the thread will execute.

Steps:
Create a class that extends the Thread class.
Override the run() method to define the task.
Create an object of the extended Thread class and call start() to begin execution.

```
// Step 1: Create a class that extends Thread
class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed by the thread
        System.out.println(Thread.currentThread().getId() + " is executing the thread by extending
Thread class.");
    }
}

public class ThreadExample1 {
    public static void main(String[] args) {
        // Step 2: Create instances of MyThread class
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        // Step 3: Start the threads
        t1.start();  // Start thread t1
        t2.start();  // Start thread t2
    }
}
```

MyThread extends the Thread class and overrides the run() method.
We create two instances of MyThread and call start() to initiate the threads.
start() internally calls run(), which contains the task the thread will execute.

Output:
```
1 is executing the thread by extending Thread class.
2 is executing the thread by extending Thread class.
```

2. Instantiating a Thread by Implementing the Runnable Interface

In this approach, we implement the Runnable interface, which requires us to override the run() method. This method contains the code to be executed by the thread.

Steps:
Create a class that implements the Runnable interface.
Override the run() method to define the task.
Create a Thread object and pass an instance of the Runnable class to it.
Call start() to begin execution.

```
// Step 1: Create a class that implements Runnable interface
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code to be executed by the thread
            System.out.println(Thread.currentThread().getId() + " is executing the thread by
implementing Runnable interface.");
    }
}
```

```
public class ThreadExample2 {
   public static void main(String[] args) {
       // Step 2: Create an instance of MyRunnable
       MyRunnable myRunnable = new MyRunnable();

       // Step 3: Create Thread objects and pass the Runnable instance
       Thread t1 = new Thread(myRunnable);
       Thread t2 = new Thread(myRunnable);

       // Step 4: Start the threads
       t1.start();  // Start thread t1
       t2.start();  // Start thread t2
   }
}
```

MyRunnable implements the Runnable interface and overrides the run() method.
A Thread object is created by passing the Runnable instance (myRunnable) to the Thread constructor.
Calling start() will invoke the run() method and execute the task.

Output:

```
1 is executing the thread by implementing Runnable interface.
2 is executing the thread by implementing Runnable interface.
```

10a)What is enumeration? Explain the methods values() and valueof().

An enumeration (enum) in Java is a special data type that defines a collection of constants. It is used to represent a fixed set of related constants, such as days of the week, months of the year, directions, etc. Enums provide a type-safe way of handling these constants, ensuring that only valid values are used.

Enums are implicitly final and cannot be subclassed.
Enums can have fields, methods, and constructors.
They are type-safe, meaning only predefined values (constants) can be assigned to an enum type.

Syntax for Enum:

```
enum EnumName {
   CONSTANT1, CONSTANT2, CONSTANT3; // Enum constants
}
```

Methods of Enum: values() and valueOf()

1. values(): A method automatically provided by the compiler that returns an array of all the constants in the enum, which can be iterated over.

2. valueOf(): A method automatically provided by the compiler that converts a string to its corresponding enum constant, throwing an IllegalArgumentException if the string does not match a constant.

1. values() Method:

The values() method is a static method that is automatically added to every enum type by the Java compiler.
It returns an array of all enum constants defined in the enum, in the order they are declared.

You can use the values() method to iterate over the constants of the enum.

Example:

```
enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}

public class EnumExample {
   public static void main(String[] args) {
      // Using values() to get all enum constants
      Day[] days = Day.values();

      for (Day day : days) {
         System.out.println(day);
      }
   }
}
```

Output:

```
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
```

The values() method returns an array of the enum constants.
We iterate through this array and print each constant (day of the week).

2. valueOf() Method:

The valueOf() method is a static method that is automatically provided for each enum.
It converts a string to the corresponding enum constant.

If the string passed to valueOf() doesn't match any of the enum constants, it throws an IllegalArgumentException.

You can use the valueOf() method to get an enum constant based on its name (string value).

Example:

```
enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}

public class EnumExample {
   public static void main(String[] args) {
      // Using valueOf() to get enum constant from string
      String dayString = "MONDAY";
      Day day = Day.valueOf(dayString);  // Convert string to enum constant

      System.out.println("Enum constant for " + dayString + ": " + day);
   }
}
```

Output:

```
Enum constant for MONDAY: MONDAY
```

The valueOf() method takes the string "MONDAY" and converts it into the corresponding Day.MONDAY enum constant.
If the string passed does not match an enum constant (e.g., "FUNDAY"), it will throw an IllegalArgumentException.

```
enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}

public class EnumExample {
   public static void main(String[] args) {
      try {
         // Trying an invalid string value
         Day day = Day.valueOf("FUNDAY"); // This will throw an exception
      } catch (IllegalArgumentException e) {
         System.out.println("Exception: " + e);
      }
   }
}
```

Output:

```
Exception: java.lang.IllegalArgumentException: No enum constant Day.FUNDAY
```

10b) Explain about type wrappers and auto boxing.

In Java, primitive types (like int, double, char, etc.) are not objects, but sometimes we need to treat them as objects for various reasons (e.g., collections, generic types). Wrapper classes are used to wrap these primitive types into corresponding objects. These wrapper classes are part of the java.lang package and they provide useful methods to convert between primitive types and objects.

For each primitive data type, there is a corresponding wrapper class:

byte → Byte
short → Short
int → Integer
long → Long
float → Float
double → Double
char → Character
boolean → Boolean

Example:

```
int x = 10;
Integer integerObject = new Integer(x);  // Boxing
System.out.println(integerObject);       // Unboxing
```

Auto-boxing and Unboxing:

1. Auto-boxing:

Auto-boxing is the automatic conversion of primitive types into their corresponding wrapper objects by the Java compiler.

For example, assigning an int to an Integer object automatically performs the conversion.

Example of Auto-boxing:

```
public class AutoBoxingExample {
   public static void main(String[] args) {
      int x = 10;

      // Auto-boxing: The primitive int x is automatically converted to Integer object
      Integer y = x;
```

```
        System.out.println("Auto-boxed value: " + y);
    }
}
```

Output:

```
Auto-boxed value: 10
```

In this example, the int variable x is automatically converted into an Integer object y without explicitly calling the new Integer(x) constructor. This is auto-boxing.

2. Unboxing:

Unboxing is the reverse process of auto-boxing, where the wrapper object is automatically converted back to its corresponding primitive type.

Example of Unboxing:

```
public class UnBoxingExample {
    public static void main(String[] args) {
        Integer y = new Integer(10);  // Boxing

        // Unboxing: The Integer object is automatically converted to primitive int
        int x = y;

        System.out.println("Unboxed value: " + x);
    }
}
```

Output:

```
Unboxed value: 10
```

In this case, the Integer object y is automatically converted back to the primitive int when it is assigned to the primitive variable x. This is unboxing.