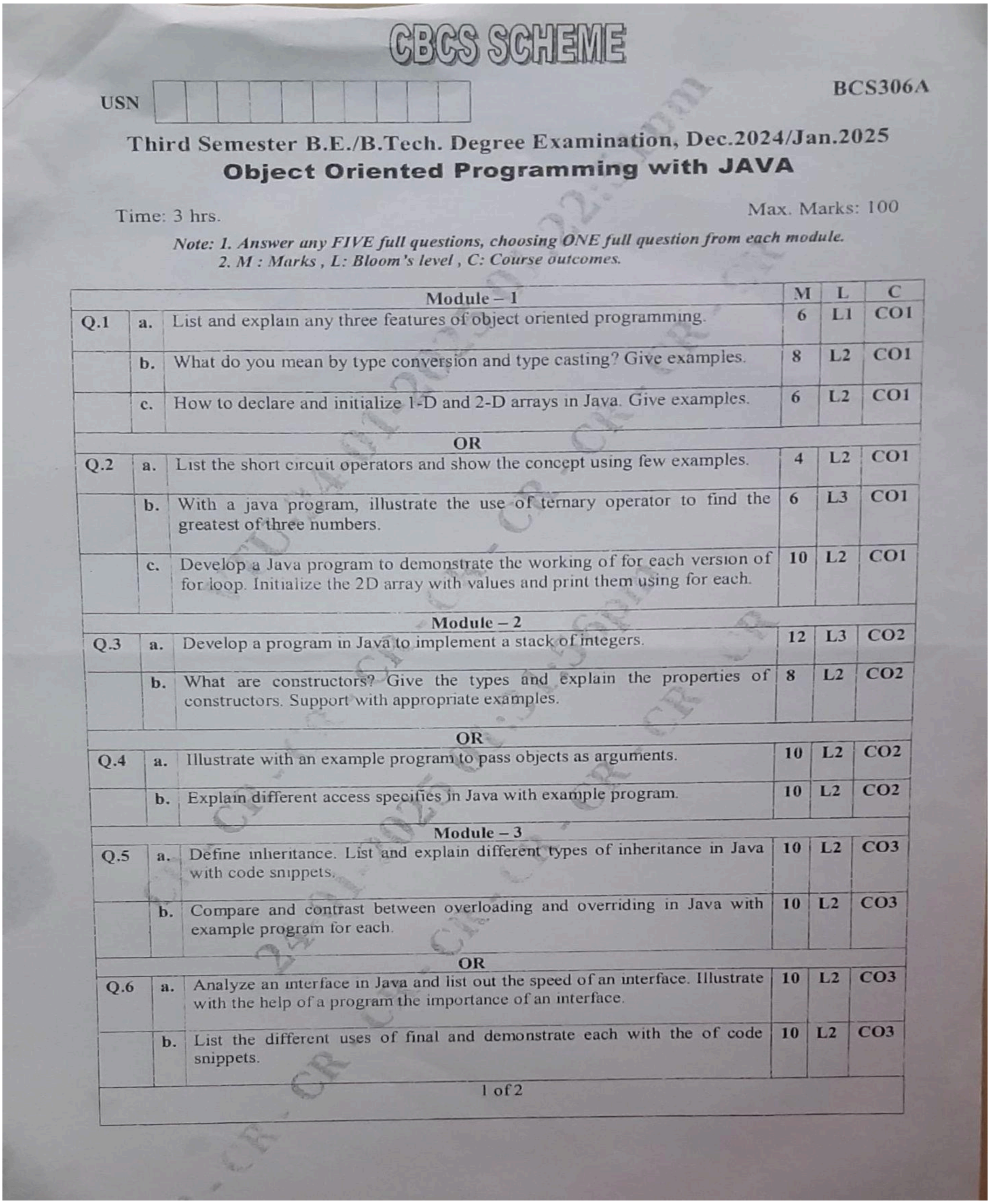
VTU EXAMINATION- DEC. 2024



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.

			M	L	C
		Module – 1			
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

* * * * *

SCHEME & SOLUTION

Sub:	Object Oriented Programming with Java	Sub Code:	BCS306A	Branch:	ISE

Answer any FIVE FULL questions		MARKS	CO	RBT
	MODULE1			
1	**A) List and explain any three features of object oriented programming.** **Detail Explaination of** **i.Encapsulation** **ii. Abstraction** **iii. Polymorphism or Inheritence**	2+2+2 [6]	CO 1	L1

	B) What do you mean by type conversion and type casting? Give examples.
	Detailed Explaination of
	i. Implicit Conversion with example
	
	`public class TypeConversionExample {`
	`public static void main(String[] args) {`
	`int num = 10;`
	`double val = num; // Implicit conversion from int to double`
	`System.out.println(val); // Output: 10.0`
	`}`
	`}`
	
	Detailed Explaination of
	ii. Explicit Conversion with example
	`public class TypeCastingExample {`
	`public static void main(String[] args) {`
	`double pi = 3.14159;`
	`int approxPi = (int) pi; // Explicit type casting from double to int`
	`System.out.println(approxPi); // Output: 3`
	`}`
	`}`
	
	Along with written key comparison
		4+4 [8]	CO 1	L2

	C) How to declare and initialize 1-D and 2-D arrays in Java. Give examples.
	
	Declaring 1d Array
	`// Declaration`
	`int[] arr;`
	
	`// Memory allocation`
	`arr = new int[5]; // Array of size 5`
		3+3 [6]	CO 1	L2

```
// Declaration + Memory allocation + Initialization
int[] arr2 = {10, 20, 30, 40, 50}; // Direct initialization
```

Declaring 2d Array

```
// Declaration
int[][] matrix;

// Memory allocation
matrix = new int[3][3]; // 3x3 matrix

// Declaration + Memory allocation + Initialization
int[][] matrix2 = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
}; // Direct initialization
```

Example of 2D Array

```
public class TwoDArrayExample {
    public static void main(String[] args) {
        // Declare and initialize a 2-D array
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Access and print 2-D array elements
        for (int i = 0; i < matrix.length; i++) { // Rows
            for (int j = 0; j < matrix[i].length; j++) { // Columns
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println(); // New line after each row
        }
    }
}
```

OR

2	**A) List the short circuit operators and show the concept using a few examples.** **Short-circuit operators are logical operators (&& and ||) that stop evaluating the remaining conditions once the result is determined. This improves efficiency and prevents unnecessary operations.** **Explaining both the operators with example.**	1+1+2 [4]	CO 1	L2
	B) With a java program, illustrate the use of ternary operator to find the greatest of three numbers.	6 [6]	CO 1	L3

```
import java.util.Scanner;

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

      // Taking three numbers as input
      System.out.print("Enter first number: ");
      int num1 = scanner.nextInt();

      System.out.print("Enter second number: ");
      int num2 = scanner.nextInt();

      System.out.print("Enter third number: ");
      int num3 = scanner.nextInt();

      // Using ternary operator to find the greatest number
      int greatest = (num1 > num2)
                  ? (num1 > num3 ? num1 : num3)
                  : (num2 > num3 ? num2 : num3);

      // Display the result
      System.out.println("The greatest number is: " + greatest);

      scanner.close();
   }
}
```

	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	CO 1	L2

```
public class ForEach2DArray {
   public static void main(String[] args) {
      // Initialize a 2D array with values
```

```
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        System.out.println("Printing 2D array using for-each loop:");

        // Using for-each loop to iterate over 2D array
        for (int[] row : matrix) {  // Iterates over rows
            for (int num : row) {   // Iterates over elements in a row
                System.out.print(num + " ");
            }
            System.out.println(); // New line after each row
        }
    }
}
```

[10]

MODULE 2

3	**A) Develop a program in Java to implement a stack of integers.**	12	CO 2	L3

```
import java.util.Scanner;

class Stack {
   private int[] stackArray;
   private int top;
   private int capacity;

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

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

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

   // Peek operation: Returns the top element without removing it
   public int peek() {
      if (top == -1) {
         System.out.println("Stack is empty!");
         return -1;
      } else {
         return stackArray[top];
      }
   }

   // Display operation: Prints the stack elements
   public void display() {
      if (top == -1) {
         System.out.println("Stack is empty!");
      } else {
         System.out.print("Stack elements: ");
```

```
            for (int i = 0; i <= top; i++) {
                System.out.print(stackArray[i] + " ");
            }
            System.out.println();
        }
    }
}

// Main class to test the Stack implementation
public class StackImplementation {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Create a stack of size 5
        Stack stack = new Stack(5);

        // Menu-driven program
        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. Display");
            System.out.println("5. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter value to push: ");
                    int value = scanner.nextInt();
                    stack.push(value);
                    break;
                case 2:
                    int popped = stack.pop();
                    if (popped != -1)
                        System.out.println("Popped: " + popped);
                    break;
                case 3:
                    int top = stack.peek();
                    if (top != -1)
                        System.out.println("Top element: " + top);
                    break;
                case 4:
                    stack.display();
                    break;
                case 5:
                    System.out.println("Exiting...");
                    scanner.close();
                    System.exit(0);
                default:
```

	System.out.println("Invalid choice! Try again."); } } } } **B) What are constructors? Give the types and explain the properties of constructors. Support with appropriate examples.** **A constructor is a special method in Java used to initialize objects. It is called automatically when an object of a class is created. Constructors have the same name as the class and do not have a return type (not even void).** **Types of Constructors in Java** **Java has three types of constructors:** **1. Default Constructor (No-Argument Constructor)** **2. Parameterized Constructor** **3. Copy Constructor (Manually implemented)** **Explanation each with Example**	1+1+6 [8]	CO 2	L2

4	**OR**
	
	A) Illustrate with an example program to pass objects as arguments.
	
	**class Rectangle {
	int length, width;
	
	// Constructor to initialize rectangle dimensions
	public Rectangle(int l, int w) {
	length = l;
	width = w;
	}
	
	// Method that takes a Rectangle object as an argument
	public void compareArea(Rectangle r) {
	int area1 = this.length * this.width;
	int area2 = r.length * r.width;
	
	System.out.println("Area of First Rectangle: " + area1);
	System.out.println("Area of Second Rectangle: " + area2);
	
	if (area1 > area2)
	System.out.println("First rectangle is larger.");
	else if (area1 < area2)
	System.out.println("Second rectangle is larger.");
	else
	System.out.println("Both rectangles have the same area.");
	}**
		10
		[10]
			CO 2	L2

```
}

public class ObjectAsArgumentExample {
    public static void main(String[] args) {
        // Creating two Rectangle objects
        Rectangle rect1 = new Rectangle(10, 5);
        Rectangle rect2 = new Rectangle(7, 8);

        // Passing one object as an argument to the method of another object
        rect1.compareArea(rect2);
    }
}
```

B) Explain different access specifies in Java with example program.

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

Access Specifier	Accessible Within the Same Class?	Accessible Within the Same Package?	Accessible in Subclasses?	Accessible from Other Packages?
private	✅ Yes	❌ No	❌ No	❌ No
default *(no modifier)*	✅ Yes	✅ Yes	❌ No	❌ No
protected	✅ Yes	✅ Yes	✅ Yes (only in subclasses)	❌ No
public	✅ Yes	✅ Yes	✅ Yes	✅ Yes

With Examples for every specifiers is mandatory

2+2+2+2+2 [10] CO 2 L2

MODULE 3

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

Inheritance is one of the core concepts of Object-Oriented Programming (OOP). It allows one class to acquire properties and behaviors (methods and fields) of another class.

- **The class that is inherited from is called the parent (superclass/base class).**

5	● **The class that inherits is called the child (subclass/derived class).** **Types of Inheritance in Java** **Java supports the following types of inheritance:** 1. **Single Inheritance** 2. **Multilevel Inheritance** 3. **Hierarchical Inheritance** 4. **Multiple Inheritance (via Interfaces)** **Java does NOT support multiple inheritance with classes (to avoid ambiguity). Instead, it is achieved using interfaces. Explain of each inheritance with example**	1+3+ 6 [10]	CO 3	L2
	B) Compare and contrast between overloading and overriding in Java with example program for each. ● **Overloading happens in the same class, while overriding occurs between parent and child classes.** ● **Overloading methods have different parameter lists, whereas overriding methods must have the same signature.** ● **Overloading enables compile-time polymorphism, while overriding enables runtime polymorphism.** ● **Overriding cannot be applied to static or final methods.** **Use overloading for flexibility in method calls and overriding to modify inherited behavior dynamically.** **Example of Method Overloading:** **class MathOperations {** **int add(int a, int b) { return a + b; }** **double add(double a, double b) { return a + b; }** **}** **public class OverloadingExample {** **public static void main(String[] args) {** **MathOperations obj = new MathOperations();**	5+5 [10]	CO 3	L2

```
        System.out.println(obj.add(5, 10));      // Calls int version

        System.out.println(obj.add(5.5, 2.5));  // Calls double version

    }

}
```

Example for Overriding

```
class Animal { void sound() { System.out.println("Animal makes sound"); } }

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



public class OverridingExample {

    public static void main(String[] args) {

        Animal a = new Dog();

        a.sound();  // Calls overridden method in Dog class (runtime
polymorphism)

    }

}
```

OR

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** **Understanding an Interface in Java** **An interface in Java is a blueprint for a class that contains abstract methods (methods without a body) and constants. It is used to achieve abstraction and multiple inheritance in Java.** **Characteristics of an Interface:**	10	CO 3	L2

	1. Defines behavior but does not implement it – Implementation is provided by classes that implement the interface. **2. Supports multiple inheritance – A class can implement multiple interfaces.** **3. Contains only abstract methods (before Java 8) – Since Java 8, it can have default and static methods with implementation.** **4. All methods are implicitly public and abstract, and all fields are public, static, and final.** **5. Achieves loose coupling – Helps in designing flexible and scalable systems.** --- **Speed of an Interface** ● **Interfaces have a slight overhead compared to direct class calls because method calls are resolved at runtime via dynamic method dispatch (virtual table mechanism).** ● **However, modern JVM optimizations (like Just-In-Time Compilation) minimize the performance difference between interface methods and regular class methods.** ● **Performance is not a major concern unless an application makes millions of interface calls in performance-critical code.** Example of Importance of an Interface should be given **B) List the different uses of final and demonstrate each with the of code snippet.** **Uses of final Keyword in Java** **The final keyword in Java is used to restrict modification. It can be applied to variables, methods, and classes to enforce immutability or prevent overriding and inheritance.** **Explaination of each scenario with example for variables,methods and classes**	1+3+3+3 [10]	CO 3	L2
7	**MODULE 4** **A) Define a package. Explain how to create user defined package with example.** **A package in Java is a collection of related classes and interfaces that are grouped together to organize code efficiently. It helps in avoiding name**	1+3+3 [7]	CO 4	L2

	conflicts, improves modularity, and provides better access control. **Types of Packages in Java:** 1. **Built-in Packages – Predefined packages in Java (java.util, java.io, etc.).** 2. **User-defined Packages – Custom packages created by the user.** **Explain procedure with neat example**	2+2+ 2+2 [8]	CO 4	L2
	B) Discuss about exception handling in Java. Give the framework of the exception handling block. List the types of exception. **Definition:Exception handling in Java is a mechanism that handles runtime errors and prevents program crashes. It ensures smooth program execution by catching and resolving exceptions.** **Java provides a structured way to handle exceptions using try, catch, finally, and throw blocks.** **Explanation of each handling scenarios with example** **Types of Exception** 1) **Checked Exception** 2) **Unchecked Exception**	5	CO 4	L3
	C) Develop a Java program to raise a custom exception for division by zero using try, catch, throw and finally.			

```
// Custom exception class for Division by Zero
class DivisionByZeroException extends Exception {
   public DivisionByZeroException(String message) {
      super(message);
   }
}
public class CustomExceptionDemo {
   // Method to perform division
   public static int divide(int a, int b) throws DivisionByZeroException {
      if (b == 0) {
         throw new DivisionByZeroException("Error: Division by Zero is 
not allowed!");
      }
      return a / b;
   }
   public static void main(String[] args) {
      try {
         int result = divide(10, 0); // Trying to divide by zero
         System.out.println("Result: " + result);
         } catch (DivisionByZeroException e) {
            System.out.println(e.getMessage()); // Handling the custom
exception
        } finally {
            System.out.println("Execution completed."); // Always executes
        }
    }
}
```

8

3+7 [10] | CO 4 | L2

OR

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

The throw and throws keywords in Java are used for exception handling but serve different purposes.

The throw keyword is used to explicitly throw an exception inside a method or block of code. It is followed by an instance of an exception. The thrown exception must be either caught using a try-catch block or declared in the method signature using throws.

The throws keyword is used to declare exceptions that a method might throw. It is placed in the method signature to inform the caller that the method could potentially generate an exception. Unlike throw, throws does not handle the exception itself; it just indicates that an exception might occur.

```
import java.io.*;

// Class to demonstrate throw and throws

class ExceptionExample {

    // Method using 'throws' to declare a possible exception
    public static void checkFile() throws IOException {
        // Using 'throw' to explicitly generate an exception
        throw new IOException("File not found!");
    }
    public static void main(String[] args) {
        try {
            checkFile(); // Calls method that throws an exception
        } catch (IOException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }   } }
```

[5] | CO 4 | L2

B) Explain about the need for finally block.

The finally block in Java is used to execute important cleanup code such as closing files, releasing resources, or terminating connections, regardless of whether an exception occurs or not.

Even if an exception is caught or not, the finally block always executes before the method exits, ensuring resource management and avoiding memory leaks.

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

[5] CO 4 L3

C) Discuss about chained exception.

Chained exceptions in Java allow one exception to be linked to another, helping to trace the root cause of an error. This mechanism is useful when one exception causes another exception, and we want to preserve both for debugging.

Why Use Chained Exceptions?

1. **Preserve the Root Cause – Helps track the original exception that led to another exception.**
2. **Better Debugging – Provides a clear sequence of errors.**
3. **Improved Exception Handling – Helps propagate errors effectively in a program.**

How to Implement Chained Exceptions?

Java provides special constructors in the Throwable class to support exception chaining:

1. **Throwable(Throwable cause) – Passes the original exception.**
2. **Throwable(String message, Throwable cause) – Provides a message along with the original exception.**

9		1+1+ 4+4 [10]	CO 5	L2
	MODULE 5 **A) Define thread. Demonstrate creation of multiple threads with a program.** **A thread in Java is the smallest unit of execution within a process. Java supports multithreading, allowing multiple threads to run concurrently, improving performance in applications.** **Creating Multiple Threads in Java** **In Java, there are two ways to create threads:** **1. Extending Thread class** **2. Implementing Runnable interface** **Example for each** **B) Explain the two ways in which Java threads can be instantiated. Support your explanation with a sample program.** **Creating a thread by extending the Thread class**	5+5 [10]	CO 5	L2

```
class MyThread extends Thread {
   public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println("Thread using Thread class: " + i);
         try {
            Thread.sleep(500); // Pauses execution for 500ms
         } catch (InterruptedException e) {
            System.out.println("Thread interrupted.");
         }
      }
   }

}



public class ThreadExample1 {
   public static void main(String[] args) {
      MyThread t1 = new MyThread(); // Creating a thread object
      t1.start(); // Starting the thread
   }
```

10

}

Creating a Thread by Implementing the Runnable Interface

```
// Creating a thread by implementing Runnable interface

class MyRunnable implements Runnable {
   public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println("Thread using Runnable interface: " + i);
         try {
            Thread.sleep(500);
         } catch (InterruptedException e) {
            System.out.println("Thread interrupted.");
         }
      }
   }
}

public class ThreadExample2 {
   public static void main(String[] args) {
      MyRunnable myRunnable = new MyRunnable(); // Creating an
instance
      Thread t1 = new Thread(myRunnable); // Passing it to Thread
constructor
      t1.start(); // Starting the thread
   }
}
```

OR

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

An enumeration (enum) in Java is a special data type that defines a set of constant values. It is used when we need a fixed set of values, such as days of the week, months, or directions.

In Java, enum is a class type that can have:
✔ Constant values
✔ Methods
✔ Constructors

Example for value and valueof()

1+4+4 [10]

CO 5

L2

```
// Define an enumeration

enum Day {

   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY;

}



public class EnumValuesExample {
   public static void main(String[] args) {
      // Loop through all values in the enum
      for (Day d : Day.values()) {
         System.out.println(d);
      }
   }
}



//Example for valueOf
public class EnumValueOfExample {
   public static void main(String[] args) {
      Day day = Day.valueOf("MONDAY");
      System.out.println("Selected day: " + day);
   }
}
```

B) Explain about type wrappers and auto boxing.

Java provides wrapper classes to convert primitive data types (like int, char, double) into objects. These wrapper classes are in the java.lang package.

Wrapper classes are

Byte, Short, Integer, Long, Float, Double, Character & Boolean

```
public class WrapperExample {
   public static void main(String[] args) {
      int num = 10; // Primitive type
      Integer obj = Integer.valueOf(num); // Convert int to Integer (Boxing)
      System.out.println("Integer object: " + obj);

      double d = 5.5;
      Double objD = Double.valueOf(d); // Convert double to Double
      System.out.println("Double object: " + objD);
   }
}
```

	1+3+3+3 [10]	CO 5	L2

Autoboxing

- **Converting a primitive type to its corresponding wrapper class object automatically.**
- **Happens implicitly when assigning a primitive to a wrapper object.**

```
public class AutoBoxingExample {
   public static void main(String[] args) {
      int num = 100;
      Integer obj = num; // Autoboxing (Automatically converts int to
Integer)
      System.out.println("Autoboxed Integer: " + obj);
   }
}
```

Unboxing

- **Converting a wrapper class object back to its primitive type.**
- **Happens implicitly when using an object in a primitive operation.**

```
public class UnboxingExample {
   public static void main(String[] args) {
      Integer obj = 200; // Autoboxing
      int num = obj; // Unboxing (Automatically converts Integer to int)
      System.out.println("Unboxed int: " + num);
   }
}
```
