CBCS SCHEME

USN | | | | | | | | | |

BCS306A

Third Semester B.E./B.Tech. Degree Examination, Dec.2023/Jan.2024
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.	Discuss the different data types supported by Java along with the default values and literals.	8	L2	CO1
	b.	Develop a Java program to convert Celsius temperature to Fahrenheit.	6	L3	CO2
	c.	Justify the statement "Compile once and run anywhere" in Java.	6	L2	CO1
		OR			
Q.2	a.	List the various operators supported by Java. Illustrate the working of >> and >>> operators with an example.	8	L2	CO1
	b.	Develop a Java program to add two matrices using command line argument.	10	L3	CO2
	c.	Explain the syntax of declaration of 2D arrays in Java.	2	L2	CO1
		Module – 2			
Q.3	a.	Examine Java Garbage collection mechanism by classifying the 3 generations of Java heap.	6	L2	CO1
	b.	Develop a Java program to find area of rectangle, area of circle and area of triangle using method overloading concept. Call these methods from main method with suitable inputs.	10	L3	CO2
	c.	Interpret the general form of a class with example.	4	L2	CO2
		OR			
Q.4	a.	Outline the following keywords with an example : (i) this (ii) static	6	L2	CO2
	b.	Develop a Java program to create a class called 'Employee' which contains 'name', 'designation', 'empid' and 'basic salary' as instance variables and read ( ) and write ( ) as methods. Using this class, read and write five employee information from main ( ) method.	10	L3	CO2
	c.	Interpret with an example, types of constructions.	4	L2	CO2
		Module – 3			
Q.5	a.	Illustrate the usage of super keyword in Java with suitable example. Also explain the dynamic method dispatch.	10	L2	CO3
	b.	Build a Java program to create an interface Resizable with method resize (int radius) that allow an object to be resized. Create a class circle that implements resizable interface and implements the resize method.	10	L3	CO3
		OR			
Q.6	a.	Compare and contrast method overloading and method overriding with suitable example.	8	L2	CO2

	b.	Define inheritance and list the different types of inheritance in Java.	4	L2	CO3
	c.	Build a Java program to create a class named 'Shape'. Create 3 sub classes namely circle, triangle and square ; each class has 2 methods named draw ( ) and erase ( ). Demonstrate polymorphism concepts by developing suitable methods and main program.	8	L3	CO3
		Module – 4			
Q.7	a.	Examine the various levels of access protections available for packages and their implications with suitable examples.	10	L2	CO4
	b.	Build a Java program for a banking application to throw an exception, where a person tries to withdraw the amount even though he/she has lesser than minimum balance (Create a custom exception)	10	L3	CO4
		OR			
Q.8	a.	Define Exception. Explain Exception handling mechanism provided in Java along with syntax and example.	10	L2	CO4
	b.	Build a Java program to create a package "balance" containing Account Class with displayBalance ( ) method and import this package in another program to access method of Account Class.	10	L3	CO4
		Module – 5			
Q.9	a.	Define a thread. Also discuss the different ways of creating a thread.	6	L2	CO5
	b.	How synchronization can be achieved between threads in Java? Explain with an example.	6	L2	CO5
	c.	Develop a Java program for automatic conversion of wrapper class type into corresponding primitive type that demonstrates unboxing.	8	L3	CO5
		OR			
Q.10	a.	Summarize the type wrappers supported in Java.	6	L2	CO5
	b.	Explain Autoboxing/Unboxing that occurs in expressions and operators.	6	L2	CO5
	c.	Develop a Java program to create a class myThread. Call the base class constructor in this class's constructor using super and start the thread. The run method of the class starts after this. It can be observed that both main thread and created child thread are executed concurrently.	8	L3	CO5

* * * * *

USN | | | | | | | | | | |

VTU Final Exam – December 2023/Jan2024

Sub:	OOPS WITH JAVA					Sub Code:	BCS306A	Branch:	CSE
Date:	12/4/2024	Duration:	180 mins	Max Marks:	100	Sem / Sec:	III(A, B & C)		OBE

Answer any FIVE FULL Questions,choosing one full question from each module.

	MARKS	CO	RBT

Module -1

1	**a) Discuss the different data types supported by java along with the default values and literals.** **Solution:** **1. Primitive Data Types** Primitive data types are the most basic data types provided by Java. They are predefined by the language and represent simple values. Java supports the following primitive data types: ● boolean: Represents a true or false value. ○ Default Value: `false` ○ Literals: `true`, `false` ● byte: Represents an 8-bit signed integer. ○ Default Value: `0` ○ Literals: Example: `byte b = 10;` ● short: Represents a 16-bit signed integer. ○ Default Value: `0` ○ Literals: Example: `short s = 1000;` ● int: Represents a 32-bit signed integer. ○ Default Value: `0` ○ Literals: Example: `int i = 12345;` ● long: Represents a 64-bit signed integer. ○ Default Value: `0L` ○ Literals: Example: `long l = 1234567890L;` (suffix `L` or `l`) ● float: Represents a 32-bit floating point number. ○ Default Value: `0.0f` ○ Literals: Example: `float f = 3.14f;` (suffix `f` or `F`) ● double: Represents a 64-bit floating point number. ○ Default Value: `0.0d` ○ Literals: Example: `double d = 3.14159;` (suffix `d` or `D`, but not required) ● char: Represents a single 16-bit Unicode character. ○ Default Value: `'\u0000'` (null character) ○ Literals: Example: `char c = 'A';` 2. Reference Types (Non-Primitive Data Types) Reference types, also known as non-primitive data types, include:	[8]	CO1	L2

1 (b)	**b) Develop a java program to convert celsius to Fahrenheit.**
	
	Solution:
	
	import java.util.Scanner;
	public class CelsiusToFahrenheit {
	public static void main(String[] args) {
	Scanner scanner = new Scanner(System.in);
	// Prompt user to enter Celsius temperature
	System.out.print("Enter temperature in Celsius: ");
	double celsius = scanner.nextDouble();
	// Convert Celsius to Fahrenheit
	double fahrenheit = (celsius * 9/5) + 32;
	// Display the result
	System.out.println(celsius + " Celsius is equal to " + fahrenheit + " Fahrenheit.");
	scanner.close();
	}
	}
		[6]	CO2	L3

1(c)	Justify the statement' Compile once and run anywhere' in java. **Solution:** "Compile once, run anywhere" epitomizes Java's architecture and design philosophy, facilitating its platform independence. Java programs are compiled into bytecode, an intermediate format understood by the JVM rather than specific to any operating system or hardware. This bytecode can execute on any device or system with a compatible JVM, ensuring consistency in behavior across diverse platforms. This approach streamlines software development and deployment processes, as developers need not rewrite or recompile code for different environments. This versatility has made Java a cornerstone in enterprise applications, web development, and mobile applications, where reliability and cross-platform compatibility are paramount. Java's ability to execute on different platforms stems from its bytecode execution model. Once compiled, Java applications can seamlessly run on Windows, macOS, Linux, and other operating systems with minimal adaptation. This flexibility extends to embedded systems and devices like smartphones, ensuring Java's relevance across a broad spectrum of computing domains. By adhering to the "write once, run anywhere" principle, Java has empowered developers to focus more on application logic and less on platform-specific intricacies, thereby enhancing productivity and accelerating software deployment cycles in today's interconnected world.	[6]	CO1	L2
	OR			

2 (a)	List the various operators supported by Java . Illustrate the working of >> and >>> operator with an example.
	
	Solution:
	
	ava supports various types of operators, each serving different purposes in programming. Here is a list of operators supported by Java:
	
	1. Arithmetic Operators
	
	● **+**: Addition
	● **-**: Subtraction
	● ***:** Multiplication
	● **/**: Division
	● **%**: Modulus (remainder)
	
	2. Relational Operators
	
	● **==**: Equal to
	● **!=**: Not equal to
	● **>**: Greater than
	● **<**: Less than
	● **>=**: Greater than or equal to
	● **<=**: Less than or equal to
	
	3. Logical Operators
	
	● **&&**: Logical AND
	● **||**: Logical OR
	● **!**: Logical NOT
	
	4. Bitwise Operators
	
	● **&**: Bitwise AND
	● **|**: Bitwise OR
	● **^**: Bitwise XOR (exclusive OR)
	● **~**: Bitwise NOT (complement)
	● **<<**: Left shift
	● **>>**: Signed right shift
	● **>>>**: Unsigned right shift
	
	Example of >> and >>> Operators:
	
	>> Operator (Signed Right Shift)
	
	The >> operator shifts the bits of a number to the right by a specified number of positions. It preserves the sign bit (0 for positive numbers, 1 for negative numbers).
	
	int num1 = 16; // Binary: 00000000 00000000 00000000 00010000
	int result1 = num1 >> 2; // Right shift by 2 positions
	
	// Resulting binary: 00000000 00000000 00000000 00000004
	// Decimal result: 4
		[8]	CO1	L2

2(b)	Develop a java program to add two matrices using command line arguments.
	Solution:
	public class MatrixAddition {
	public static void main(String[] args) {
	// Check if two matrices are provided as command line arguments
	if (args.length != 9) {
	System.out.println("Please provide two 3x3 matrices as command line arguments.");
	return;
	}
	
	// Parse the command line arguments into two 3x3 matrices
	int[][] matrix1 = parseMatrix(args, 0);
	int[][] matrix2 = parseMatrix(args, 9);
	
	// Check if parsing was successful
	if (matrix1 == null || matrix2 == null) {
	System.out.println("Invalid matrix format provided.");
	return;
	}
	
	// Add the matrices
	int[][] resultMatrix = addMatrices(matrix1, matrix2);
	
	// Display the result matrix
	System.out.println("Matrix 1:");
	printMatrix(matrix1);
	System.out.println("\nMatrix 2:");
	printMatrix(matrix2);
	System.out.println("\nResultant Matrix (Matrix1 + Matrix2):");
	printMatrix(resultMatrix);
	}
	
	// Method to parse a 3x3 matrix from command line arguments starting from a given index
	private static int[][] parseMatrix(String[] args, int startIndex) {
	int[][] matrix = new int[3][3];
	try {
	for (int i = 0; i < 3; i++) {
	for (int j = 0; j < 3; j++) {
	matrix[i][j] = Integer.parseInt(args[startIndex + i * 3 + j]);
	}
	}
	} catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
	return null; // Return null if parsing fails
	}
	return matrix;
	}
	
	// Method to add two 3x3 matrices
	private static int[][] addMatrices(int[][] matrix1, int[][] matrix2) {
	int[][] result = new int[3][3];
	for (int i = 0; i < 3; i++) {
	for (int j = 0; j < 3; j++) {
	result[i][j] = matrix1[i][j] + matrix2[i][j];
	}
	}
		[10]	CO2	L3

2(c)	Explain the syntax of declaration of 2D Arrays in Java.
	Solution:
	Syntax of Declaration:
	// Syntax 1: Declare a 2D array variable
	dataType[][] arrayName;
	
	// Syntax 2: Declare and allocate memory for a 2D array
	dataType[][] arrayName = new dataType[rows][columns];
	
	// Syntax 3: Declare, allocate memory, and initialize elements of a 2D array
	dataType[][] arrayName = { {val1, val2, ...}, {val3, val4, ...}, ... };
		[2]	CO1	L2

Module - 2				

3(a).	Examine java Garbage collection mechanism by classifying the 3 generations of java heap. Solution:	[6]	CO1	L2

Java's garbage collection (GC) mechanism manages memory automatically by reclaiming memory occupied by objects that are no longer referenced. The Java heap is divided into three generations based on the age of objects and their likelihood of surviving GC cycles. These generations are:

1. Young Generation

- **Purpose**: Newly created objects are allocated in the young generation.
- **Characteristics**:
  - **Eden Space**: Initially, all new objects are allocated in the Eden space.
  - **Survivor Spaces (S0 and S1)**: Objects that survive one GC cycle in the young generation are moved to one of the survivor spaces.
  - **Minor GC**: Collection of short-lived objects (young generation) is known as minor GC.
- **Objective**: Most objects die young, so efficient collection of short-lived objects minimizes the overhead of GC.

2. Old Generation (Tenured Generation)

- **Purpose**: Objects that survive multiple minor GC cycles are eventually promoted to the old generation.
- **Characteristics**:
  - **Tenured Space**: Large and long-lived objects reside here.
  - **Promotion**: Objects that survive several minor GC cycles in the young generation are promoted to the old generation.
  - **Major GC**: Collection of long-lived objects (old generation) is known as major GC or full GC.
- **Objective**: Collection in the old generation is less frequent but involves more objects and consumes more time.

3. Permanent Generation (Deprecated in Java 8 and removed in Java 9)

- **Purpose**: Stores metadata related to classes and methods.
- **Characteristics**:
  - **Method Area**: Previously included PermGen, it stored class metadata, interned strings, and static final variables.
  - **Removed**: In Java 8, PermGen was removed and replaced with the

3(b)	Develop a java program to find area of rectangle, area of circle and area of triangle using method overloading concept , call these methods from main method with suitable inputs. Solution:	[10]	CO2	L3

```
public class AreaCalculator {

   // Method to calculate area of a rectangle
   public static double calculateArea(double length, double width) {
      return length * width;
   }
   // Method to calculate area of a circle
   public static double calculateArea(double radius) {
      return Math.PI * radius * radius;
   }
   // Method to calculate area of a triangle
   public static double calculateArea(double base, double height) {
      return 0.5 * base * height;
   }
   public static void main(String[] args) {
      // Test the methods with sample inputs
      double length = 5.0;
      double width = 3.0;
      double radius = 4.0;
      double base = 6.0;
      double height = 8.0;
      // Calculate and display area of rectangle
      double areaRectangle = calculateArea(length, width);
      System.out.println("Area of Rectangle: " + areaRectangle);
      // Calculate and display area of circle
      double areaCircle = calculateArea(radius);
      System.out.println("Area of Circle: " + areaCircle);
      // Calculate and display area of triangle
      double areaTriangle = calculateArea(base, height);
      System.out.println("Area of Triangle: " + areaTriangle);
   }
}
```

3(c)	Interpret the general form of a class with example.
	Solution:
	In Java, a class serves as a blueprint or template for creating objects. It encapsulates data (fields) and behaviors (methods) that define the characteristics and operations of objects instantiated from it. Here's an interpretation of the general form of a class in Java, along with an example:
	public class ClassName {
	// Fields (variables)
	dataType fieldName1;
	dataType fieldName2;
	// ...
	
	// Constructors
	ClassName(parameters) {
	// Initialization code
	}
	
	// Methods
	returnType methodName1(parameters) {
	// Method body
	}
	
	returnType methodName2(parameters) {
	// Method body
	}
	
	// Other class elements: more fields, constructors, methods, etc.
	}
		[4]	CO2	L2

	OR			

4(a)	Outline the following keywords with the example: (i) this (ii)static Solution:	[6]	CO2	L2

(i) `this` Keyword

In Java, `this` is a reference variable that refers to the current object. It can be used inside any method or constructor to refer to the current instance of the class. Here's how `this` is typically used:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;  // 'this' refers to the instance variable 'name'
        this.age = age;    // 'this' refers to the instance variable 'age'
    }

    public void displayInfo() {
        System.out.println("Name: " + this.name);  // 'this' used to access instance variable 'name'
        System.out.println("Age: " + this.age);    // 'this' used to access instance variable 'age'
    }
}
```

(ii) `static` Keyword

4(b)	Develop a java program to create a class called Employee which contains name,degignation, empid,basic salary as an instance variable and read() and write() as methods . Using this class read and write five employee information from main() method. Solution:	[10]	CO2	L3

```
import java.util.Scanner;

public class Employee {
    private String name;
    private String designation;
    private int empid;
    private double basicSalary;

    // Method to read employee information
    public void read() {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter name: ");
        this.name = scanner.nextLine();
        System.out.print("Enter designation: ");
        this.designation = scanner.nextLine();
        System.out.print("Enter employee ID: ");
        this.empid = scanner.nextInt();
        System.out.print("Enter basic salary: ");
        this.basicSalary = scanner.nextDouble();
    }

    // Method to display employee information
    public void write() {
        System.out.println("Name: " + this.name);
        System.out.println("Designation: " + this.designation);
        System.out.println("Employee ID: " + this.empid);
        System.out.println("Basic Salary: " + this.basicSalary);
        System.out.println(); // Empty line for separation
    }

    public static void main(String[] args) {
        // Create an array to store multiple employees
        Employee[] employees = new Employee[5];

        // Read employee information using read() method
        for (int i = 0; i < employees.length; i++) {
            System.out.println("Enter details for Employee " + (i + 1) + ":");
            employees[i] = new Employee();
            employees[i].read();
        }

        // Display employee information using write() method
        System.out.println("Employee Information:");
        for (Employee emp : employees) {
            emp.write();
        }
    }
}
```

4(c)	Interpret with an example , types of constructions. Solution: types of constructors in Java: 1. **Default Constructor**: Automatically provided by Java if no other constructors are defined. Initializes object with default values. 2. **Parameterized Constructor**: Accepts parameters to initialize object with specific values. 3. **Constructor Overloading**: Multiple constructors in a class with different parameter lists, providing flexibility in object initialization. 4. **Private Constructor**: Prevents instantiation of a class by other classes. Often used in utility classes. 5. **Copy Constructor (emulated)**: Creates a new object as a copy of an existing object of the same class. Achieved by defining a constructor that accepts an object of the same class.	[4]	CO2	L2
Module-3				

5 (a)	Illustrate the usage of super keyword in java with suitable examples. Also explain dynamic method dispatch. Solution: In Java, the `super` keyword is used to refer to the superclass (parent class) of the current object. It can be used to access superclass methods, constructors, and variables. Here are the main uses of `super`: 1. **Accessing Superclass Variables and Methods**: ○ You can use `super` to access superclass variables and methods that are hidden by the subclass. 2. **Invoking Superclass Constructor**: ● `super()` is used to invoke the superclass constructor. It must be the first statement in the subclass constructor. **Dynamic Method Dispatch** Dynamic method dispatch is a mechanism in Java where the method to be executed is determined at runtime rather than compile-time. It is also known as runtime polymorphism or late binding. It allows a subclass to provide a specific implementation of a method that is already provided by its superclass.	[10]	CO3	L2

5 (b)	Build a java program to create an interface Resizable with method resize(int radius) that allow an object should be resized. Create a class circle that implements resizable interface and implement the resize method. Solution:	[10]	CO3	L3

```
// Resizable interface
interface Resizable {
    void resize(int radius);
}

// Circle class implementing Resizable interface
class Circle implements Resizable {
    private int radius;

    // Constructor
    public Circle(int radius) {
        this.radius = radius;
    }

    // Method to resize the circle by setting a new radius
    @Override
    public void resize(int radius) {
        this.radius = radius;
        System.out.println("Circle resized to radius: " + radius);
    }

    // Getter method for radius
    public int getRadius() {
        return radius;
    }
}

// Main class to test Resizable interface and Circle class
public class Main {
    public static void main(String[] args) {
        // Create a Circle object
        Circle circle = new Circle(5);

        // Display original radius
        System.out.println("Original Circle Radius: " + circle.getRadius());

        // Resize the circle
        circle.resize(10);

        // Display resized radius
        System.out.println("Resized Circle Radius: " + circle.getRadius());
    }
}
```

	OR			
6(a)	Compare and contrast method overloading and method overriding with suitable examples. Solution: **Method Overloading** allows a class to have multiple methods with the same name but different parameter lists. This is achieved by changing the number or types of parameters. Java determines which method to call based on the number and types of arguments passed at compile-time. For example, a class can have multiple `add` methods that accept different numbers or types of parameters, providing flexibility in method usage without needing different method names. **Method Overriding**, on the other hand, occurs in a subclass that provides a specific implementation of a method that is already defined in its superclass. The overriding method must have the same name, parameter list, and return type as the method in the superclass. This allows a subclass to provide its own implementation of inherited methods, promoting code customization and enabling polymorphic behavior. Method overriding is resolved dynamically at runtime based on the actual object type, facilitating runtime polymorphism and supporting the "is-a" relationship in object-oriented programming.	[8]	CO2	L2

6(b)	Define Inheritance and list the different types of inheritance in java. Solution: **Inheritance** in Java is a mechanism by which one class (subclass or derived class) acquires the properties (fields and methods) and behaviors of another class (superclass or base class). It promotes code reusability and allows the creation of hierarchical relationships between classes. Inheritance enables a subclass to extend and specialize the functionality of its superclass, thereby supporting the "is-a" relationship **Types of Inheritance in Java:** 1. **Single Inheritance**: ○ In single inheritance, a subclass inherits from only one superclass. Java supports single inheritance where a class can extend only one other class. 2. **Multilevel Inheritance**: ● Multilevel inheritance involves a chain of inheritance where one class extends another subclass. This creates a hierarchical relationship. 3. **Hierarchical Inheritance**: ● Hierarchical inheritance involves one superclass being extended by multiple subclasses. Each subclass inherits from the same superclass. 4. **Multiple Inheritance (through Interfaces)**: ● Java does not support multiple inheritance of classes (i.e., a class cannot directly extend more than one class). However, it supports multiple inheritance through interfaces, where a class can implement multiple interfaces. 5. **Hybrid Inheritance**: ● Hybrid inheritance is a combination of multiple types of inheritance. It typically involves multiple inheritance (through interfaces) and single or multilevel inheritance.	[4]	CO3	L2

6(c)	Build a java program to create a class named Shape , Create 3 subclasses namely circle, triangle and square. each class has 2 methods named draw() and erase(). Demonstrate polymorphism concepts by developing suitable methods and main program.	[8]	CO3	L3

Solution:

```
// Shape superclass

class Shape {

    // Draw method (to be overridden)

    void draw() {

        System.out.println("Drawing Shape");

    }


    // Erase method (to be overridden)

    void erase() {

        System.out.println("Erasing Shape");

    }

}


// Circle subclass

class Circle extends Shape {

    // Override draw method for Circle

    @Override

    void draw() {

        System.out.println("Drawing Circle");

    }


    // Override erase method for Circle

    @Override

    void erase() {

        System.out.println("Erasing Circle");

    }

}
```

Module - 4

7 (a)	Examine the various levels of access protections available for packages and their implications with suitable examples.
	
	Solution:
	
	In Java, there are four levels of access protection for classes, methods, and variables: `public`, `protected`, `default` (package-private), and `private`. Here's a brief overview with examples:
	
	1. Public
	
	Access: Accessible from any other class. **Implications**: Least restrictive, suitable for API methods or constants. **Example**:
	
	public class PublicExample {
	
	public int value = 10;
	
	}
	
	public class TestPublic {
	
	public static void main(String[] args) {
	
	PublicExample example = new PublicExample();
	
	System.out.println(example.value); // Accessible from any class
	
	}
	
	}
	
	2. Protected
	
	Access: Accessible within the same package and subclasses. **Implications**: More restricted, useful for inheritance while maintaining some encapsulation. **Example**:
	
	public class ProtectedExample {
	
	protected int value = 10;
	
	}
	
	class SubclassExample extends ProtectedExample {
	
	public void show() {
	
	System.out.println(value); // Accessible in subclass
	
	}
	
	}
	
	3. Default (Package-Private)
	
	Access: Accessible only within the same package. **Implications**: Package-level
		10	CO4	L2

7(b)	Build a java program for a Banking application to throw an exception , where the person tries to withdraw the amount even though he/she has lesser than minimum balance (Create a custom exception).	10	CO4	L3

Solution:

Step-by-Step Breakdown:

1. **Define a Custom Exception**: Create a custom exception class `InsufficientFundsException`.
2. **BankAccount Class**: Define the bank account with methods for deposit and withdraw.
3. **Main Class**: Use the `BankAccount` class and handle the custom exception.

Custom Exception

```
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

BankAccount Class

```
class BankAccount {
    private double balance;
    private static final double MINIMUM_BALANCE = 100.0;

    public BankAccount(double initialBalance) {
        if (initialBalance >= MINIMUM_BALANCE) {
            this.balance = initialBalance;
        } else {
            this.balance = MINIMUM_BALANCE;
        }
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
```

OR

8(a)	Define Exception. Explain exception handling mechanism provided in java along with syntax and examples.
	
	Solution:
	
	An exception is an event that disrupts the normal flow of a program's execution. In Java, exceptions are objects that represent these errors and can be caught and handled to maintain the program's normal flow.
	
	Exception Handling Mechanism
	
	Java provides several mechanisms to handle exceptions:
	
	1. **try**: Block of code that might throw an exception.
	2. **catch**: Block of code that handles the exception.
	3. **finally**: Block of code that executes regardless of whether an exception was caught or not.
	4. **throw**: Used to explicitly throw an exception.
	5. **throws**: Indicates that a method can throw one or more exceptions.
	
	Syntax
	
	try-catch-finally:
	
	try {
	// Code that might throw an exception
	} catch (ExceptionType1 e1) {
	// Handle exception of type ExceptionType1
	} catch (ExceptionType2 e2) {
	// Handle exception of type ExceptionType2
	} finally {
	// Code to be executed regardless of an exception
	}
	throw:
	if (condition) {
	throw new ExceptionType("Error message");
	}
	throws:
	public void methodName() throws ExceptionType {
	// Method code
	}
		10	CO4	L2

8(b)	Build a java program to create a package 'Balance' containing Account class with displayBalance() method and import this p-ackage in another program to access the method of Account class.
	
	Solution:
	
	Step 1: Create the `Balance` Package
	
	First, you need to create a directory named `Balance` and then create the `Account` class inside this directory.
	
	Balance/Account.java
	
	package Balance;
	
	public class Account {
	private double balance;
	
	public Account(double balance) {
	this.balance = balance;
	}
	
	public void displayBalance() {
	System.out.println("Current balance: $" + balance);
	}
	}
	
	Step 2: Import and Use the `Balance` Package in Another Program
	
	Next, you need to create another Java program that imports the `Balance` package and uses the `Account` class.
	
	MainProgram.java
	
	import Balance.Account;
	
	public class MainProgram {
	public static void main(String[] args) {
	Account myAccount = new Account(1500.00);
	myAccount.displayBalance();
		10	CO4	L3

Module -5

9(a)	Define a thread. Also discuss the different ways of creating a Thread.
	
	Solution:
	
	Thread in Java
	
	Definition: A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution within a program, allowing concurrent execution of tasks. Threads are used to perform multiple tasks simultaneously within a single program.
	
	Different Ways of Creating a Thread in Java
	
	Java provides two main ways to create a thread:
	
	1. **By Extending the `Thread` Class**
	2. **By Implementing the `Runnable` Interface**
	
	1. By Extending the `Thread` Class
	
	When you extend the `Thread` class, you create a new class that inherits from `Thread` and override its `run()` method. The `run()` method is where you define the code that constitutes the new thread's task.
	
	Example:
	
	class MyThread extends Thread {
	public void run() {
	for (int i = 0; i < 5; i++) {
	System.out.println(Thread.currentThread().getId() + " - Value: " + i);
	try {
	Thread.sleep(500); // Pause for 500 milliseconds
	} catch (InterruptedException e) {
	System.out.println(e);
	}
	}
	}
	}
	
	public class ThreadExample1 {
	public static void main(String[] args) {
	MyThread t1 = new MyThread();
		6	CO5	L2

9(b)	How synchronization can be achieved between thread in java? Explain with an example.
	
	Solution:
	
	Synchronization in Java is a mechanism to control the access of multiple threads to shared resources. It helps to prevent thread interference and consistency problems, ensuring that only one thread can access the resource at a time.
	
	Ways to Achieve Synchronization
	
	1. **Synchronized Method**
	2. **Synchronized Block**
	3. **Static Synchronization**
	
	1. Synchronized Method
	
	A synchronized method ensures that only one thread can execute it at a time for the same object.
	
	Example:
	
	class Table {
	synchronized void printTable(int n) { // synchronized method
	for (int i = 1; i <= 5; i++) {
	System.out.println(n * i);
	try {
	Thread.sleep(400);
	} catch (InterruptedException e) {
	System.out.println(e);
	}
	}
	}
	}
	
	class MyThread1 extends Thread {
	Table t;
	
	MyThread1(Table t) {
	this.t = t;
	}
		6	CO5	L2

9(c)	Develop a java program for automatic conversion of Wrapper class type into corresponding primitive type that demonstrates unboxing.
	
	Solution:
	
	public class UnboxingExample {
	  public static void main(String[] args) {
	  // Example of unboxing
	  Integer wrappedInt = new Integer(50); // Creating an Integer wrapper object
	  int primitiveInt = wrappedInt; // Unboxing: converting Integer to int
	
	  System.out.println("Wrapped Integer: " + wrappedInt);
	  System.out.println("Unboxed Primitive Integer: " + primitiveInt);
	
	  // Another example with Double
	  Double wrappedDouble = 25.5; // Autoboxing: double to Double
	  double primitiveDouble = wrappedDouble; // Unboxing: Double to double
	
	  System.out.println("\nWrapped Double: " + wrappedDouble);
	  System.out.println("Unboxed Primitive Double: " + primitiveDouble);
	  }
	}
		8	CO5	L3

OR				

10(a)	Summarize the type Wrapper supported in Java.
	
	Solution:
	
	In Java, wrapper classes are used to encapsulate primitive data types into objects. They provide a way to treat primitive data types as objects. Here's a summary of the wrapper classes supported in Java:
	
	1. **Integer Types**:
	○ `Byte`: Represents a byte value.
	○ `Short`: Represents a short value.
	○ `Integer`: Represents an int value.
	○ `Long`: Represents a long value.
	2. **Floating-Point Types**:
	○ `Float`: Represents a float value.
	○ `Double`: Represents a double value.
	3. **Boolean Type**:
	○ `Boolean`: Represents a boolean value (`true` or `false`).
	4. **Character Type**:
	○ `Character`: Represents a char value.
	
	Example:
	
	// Example of using Integer wrapper class
	
	Integer num1 = new Integer(10); // Explicit creation
	
	Integer num2 = 20; // Autoboxing
	
	int sum = num1 + num2; // Unboxing implicitly
	
	System.out.println("Sum: " + sum);
	
	// Example of using Boolean wrapper class
	
	Boolean flag1 = new Boolean(true); // Explicit creation
	
	Boolean flag2 = false; // Autoboxing
	
	if (flag1.equals(flag2)) {
	System.out.println("Flags are equal.");
	} else {
	System.out.println("Flags are not equal.");
	}
		6	CO5	L2

10(b)	Explain Autoboxing/Unboxing that occurs in expressions and operators.
	
	Solution:
	
	Autoboxing and unboxing in Java are automatic conversions performed by the Java compiler to seamlessly convert between primitive types and their corresponding wrapper classes (objects) when necessary. This feature was introduced to simplify coding and make it more intuitive when working with primitive types and their object representations.
	
	Autoboxing
	
	Autoboxing is the automatic conversion of a primitive type to its corresponding wrapper class object. This conversion happens when:
	
	● Assigning a primitive value to a wrapper class reference.
	● Passing a primitive value as a parameter to a method that expects the corresponding wrapper class object.
	● Using a primitive value in expressions that require a wrapper class object.
	
	Example of Autoboxing:
	
	// Assigning primitive value to wrapper class reference
	
	Integer num1 = 10; // Autoboxing: int to Integer
	
	// Passing primitive value to a method expecting Integer
	
	void processInteger(Integer number) {
	
	// Method implementation
	
	}
	
	processInteger(20); // Autoboxing: int to Integer
	
	// Using primitive value in an expression requiring Integer
	
	List<Integer> numbers = new ArrayList<>();
	
	numbers.add(30); // Autoboxing: int to Integer
	
	Unboxing
	
	Unboxing is the automatic conversion of a wrapper class object to its corresponding primitive type. This conversion happens when:
	
	● Assigning a wrapper class object to a primitive type variable.
	● Passing a wrapper class object as a parameter to a method that expects the corresponding primitive type.
	● Using a wrapper class object in expressions that require a primitive type.
		6	CO5	L2

10(c)	Develop a java program to create a class myThread. Call the base class constructor in this class's constructor using super and start the thread . The run method of the class starts after this . It can be observed that both the main thread and created child thread are executed concurrently.
	
	Solution:
	
	// MyThread.java
	
	class MyThread extends Thread {
	  MyThread(String name) {
	  super(name); // Calling superclass Thread's constructor
	  start(); // Start the thread
	  }
	
	  public void run() {
	  // Define the behavior of the thread here
	  for (int i = 1; i <= 5; i++) {
	  System.out.println("Child Thread: " + getName() + " - Count: " + i);
	  try {
	  Thread.sleep(1000); // Pause for 1 second
	  } catch (InterruptedException e) {
	  System.out.println(e);
	  }
	  }
	  System.out.println("Child Thread " + getName() + " exiting.");
	  }
	}
	
	// MainThread.java
	public class MainThread {
	  public static void main(String[] args) {
	  MyThread thread1 = new MyThread("Thread 1");
	
	  // Main thread continues to execute concurrently
		8	CO5	L3