BCS306A

Model Question Paper-I/II with effect from 2023-24 (CBCS Scheme)

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

Third Semester B.E. Degree Examination

Object Oriented Programming with JAVA

TIME: 03 Hours **Max. Marks: 100**

Note: 01. Answer any **FIVE** full questions, choosing at least **ONE** question from each **MODULE**.
02. Use a JAVA code snippet to illustrate a specific code design or a purpose.

		Module -1	*Bloom's Taxonomy Level	Marks
Q.01	a	Explain different lexical issues in JAVA.	L2	7
	b	Define Array. Write a Java program to implement the addition of two matrixes.	L3	7
	c	Explain the following operations with examples. (i)<< (ii)>> (iii)>>>	L2	6
		OR		
Q.02	a	Explain object-oriented principles.	L2	7
	b	Write a Java program to sort the elements using a for loop.	L3	7
	c	Explain different types of if statements in JAVA	L2	6
		Module-2		
Q. 03	a	What are constructors? Explain two types of constructors with an example program.	L3	7
	b	Define recursion. Write a recursive program to find nth Fibonacci number.	L3	7
	c	Explain the various access specifiers in Java.	L2	6
		OR		
Q.04	a	Explain call by value and call by reference with an example program	L3	7
	b	Write a program to perform Stack operations using proper class and Methods.	L3	7
	c	Explain the use of **this** in JAVA with an example.	L2	6
		Module-3		
Q. 05	a	Write a Java program to implement multilevel inheritance with 3 levels of hierarchy.	L3	7
	b	Explain how an interface is used to achieve multiple Inheritances in Java.	L3	7
	c	Explain the method overriding with a suitable example.	L2	6
		OR		
Q. 06	a	What is single-level inheritance? Write a Java program to implement single-level inheritance.	L3	7
	b	What is the importance of the **super** keyword in inheritance? Illustrate with a suitable example.	L3	7
	c	What is abstract class and abstract method? Explain with an example.	L2	6
		Module-4		
Q. 07	a	Define package. Explain the steps involved in creating a user-defined package with an example.	L2	7
	b	Write a program that contains one method that will throw an IllegalAccessException and use proper exception handles so that the exception should be printed.	L3	7
	c	Define an exception. What are the key terms used in exception handling? Explain.	L2	6
		OR		
Q. 08	a	Explain the concept of importing packages in Java and provide an example demonstrating the usage of the import statement.	L2	7

	b	How do you create your own exception class? Explain with a program.	L3	7
	c	Demonstrate the working of a nested try block with an example.	L2	6

BCS306A

Module-5				
Q. 09	a	What do you mean by a thread? Explain the different ways of creating threads.	L2	7
	b	What is the need of synchronization? Explain with an example how synchronization is implemented in JAVA.	L3	7
	c	Discuss values() and value Of() methods in Enumerations with suitable examples.	L2	6
OR				
Q. 10	a	What is multithreading? Write a program to create multiple threads in JAVA.	L2	7
	b	Explain with an example how inter-thread communication is implemented in JAVA.	L3	7
	c	Explain auto-boxing/unboxing in expressions.	L2	6

1. Explain different lexical issues in JAVA.

Solution:

block.

➢ Lexical Issues

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.

- **Whitespace:** In Java, whitespace is a space, tab, or newline.
- **Identifiers:** Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. Java is case-sensitive, so **VALUE** is a different identifier than **Value**. In Java, there are several points to remember about identifiers. They are as follows –
    1. All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
    2. After the first character, identifiers can have any combination of characters.
    3. A key word cannot be used as an identifier. Identifiers are case sensitive.
    4. Examples of legal identifiers: age, $salary, _value, l_value.
    5. Examples of illegal identifiers: 123abc, -salary]
    6. Some valid identifiers are:

AvgTemp	count a4	$test	this_is_ok

    7. Invalid identifiers are:

count	high-temp	Not/ok

- **Literals:** A constant value in Java is created by using a literal representation of it. For example, here are some literals: 100 98.6 'X' "This is a test". Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string.
- **Comments:** There are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This

Java solution

type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a * /.

- **Separators:** In Java, there are a few characters that are used as separators. The most used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

e Java Keywords

b	Define Array. Write a Java program to implement the addition of two matrixes.

Solution:

➢ **Arrays**

- An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

```
import java.util.Scanner;

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

      System.out.print("Enter number of rows for matrices: ");
      int rows = scanner.nextInt();

      System.out.print("Enter number of columns for matrices: ");
      int columns = scanner.nextInt();
      int[][] matrix1 = new int[rows][columns];
      int[][] matrix2 = new int[rows][columns];
      int[][] sum = new int[rows][columns];
      System.out.println("Enter elements for matrix 1:");
```

Java solution

```
        inputMatrix(matrix1, scanner);
        System.out.println("Enter elements for matrix 2:");
        inputMatrix(matrix2, scanner);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                sum[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        System.out.println("Resultant matrix after addition:");
        displayMatrix(sum);
        scanner.close();
    }
    public static void inputMatrix(int[][] matrix, Scanner scanner) {
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[0].length; j++) {
                System.out.print("Enter element at position [" + i + "][" + j + "]: ");
                matrix[i][j] = scanner.nextInt();
            }
        }
    }
    public static void displayMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int elem : row) {
                System.out.print(elem + " ");
            }
            System.out.println();
        }
    }
}
```


c. Explain the following operations with examples. (i)<< (ii)>> (iii)>>>

(i) `<<` (Left Shift Operator):

The left shift operator (`<<`) shifts the bits of a number to the left by a specified number of positions. This operation effectively multiplies the number by 2 raised to the power of the shift amount. Bits shifted off the left end are discarded, and zero bits are shifted in from the right end.

Example:

Java solution

```
int num = 5;
int result = num << 2;
System.out.println(result);
```

(ii) `>>` (Signed Right Shift Operator):
The signed right shift operator (`>>`) shifts the bits of a number to the right by a specified number of positions. This operation effectively divides the number by 2 raised to the power of the shift amount. Bits shifted off the right end are discarded, and the leftmost bit (the sign bit) is shifted in from the left end, preserving the sign of the number.

Example:

```
int num = -16;
int result = num >> 2;
System.out.println(result);
```

(iii) `>>>` (Unsigned Right Shift Operator):
The unsigned right shift operator (`>>>`) shifts the bits of a number to the right by a specified number of positions. Unlike the signed right shift operator, the unsigned right shift operator fills the leftmost positions with zero bits regardless of the sign of the number.

Example:

```
int num = -16; // Binary representation: 1111 0000
int result = num >>> 2;
System.out.println(result);
```

It's important to note that the behavior of these operators may vary based on the data type of the operand.

Q.02	a	Explain object-oriented principles.	L2	7

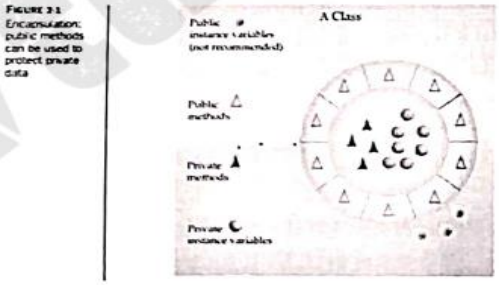
Java solution

➢ The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism.

1. Encapsulation

- *Encapsulation* is the mechanism that binds together code and the data it manipulates, and **keeps both safe from outside interference and misuse.** One way to think about encapsulation is as 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.
- Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked **private or public.**
- The *public* interface of a class represents everything that external users of the class need to know, or may know.
- The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable.



2. Inheritance

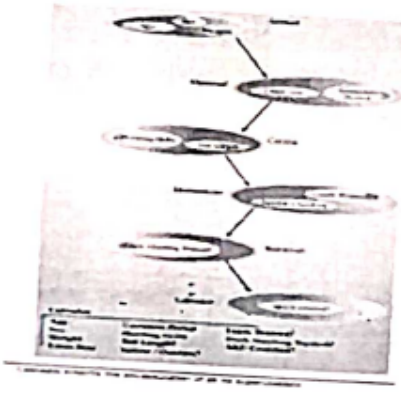
- *Inheritance* is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.

Solution:

- For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal.
- Without the use of hierarchies, each object would need to define all of its characteristics explicitly. By use of inheritance, an object need only define those qualities that make it unique within its class.
- It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

3. Polymorphism

- *Polymorphism* (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods."
- This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.*



Java solution

	b	Write a Java program to sort the elements using a for loop.	L3	7

Solution:

```
/*
  Demonstrate the for loop.

  Call this file "ForTest.java".
*/
class ForTest {
  public static void main(String args[]) {
    int x;

    for(x = 0; x<10; x = x+1)
      System.out.println("This is x: " + x);
  }
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

- In this example, x ~~is the loop control variable. It is initialized to zero in the initialization~~ portion of the for. At the start of each iteration (including the first one), the conditional test x < 10 is performed. If the outcome of this test is true, the println( ) statement is executed, and then the iteration portion of the loop is executed. This process continues until the conditional test is false.

Java solution

2	c	Explain different types of if statements in JAVA	L2	6

Selection Statements

The **if statement** – if statement executes a block of code only if the specified expression is true. If the value is false, if block is skipped & execution continues with the rest of the program.

Syntax

```
if (condition)
{                  T/F
   Statements
}
```

Eg-

```
if (a>b)
   S.o.p (a);
if (a<b)
   S.o.p (b);
```

if-else statement

If the statements in the if-statement fails, the statements in the else block are executed.

Syntax

```
if (condition)
{                  T/F
   Stmts
}
else
{
   Stmts
}
```

Eg-

```
if (a>b)
  { S.o.p (a);
  }
else
{ S.o.p(b); }
```

Java solution

Q. 03	a	What are constructors? Explain two types of constructors with an example program.	L3	7

Constructors

A constructor is a special method that is used to initialize a newly created object & is called just after the memory is allocated for an object. It is not mandatory for the coder to write a constructor for the class. Java provides a default constructor that initializes the object members to its default values. Constructor declaration is same as method declaration except the name that is same as classname and have no return type.

Syntax -

```
class classname
{
    classname(args)
    {
    }
}
```

→ Constructor

Java solution

Eg-
```
class student
{
    student(int a, String n, double d)
    {
        USN = a;
        name = n;
        score = d;
    }
}
```

Types of Constructors

There are two types of constructor:

1. Default constructors – The constructor with zero number of arguments is called default constructor.

Eg-
```
class c1
{
    c1()
    {=
    }
}
```

2. Parameterized constructors – The constructor with one or more number of arguments is called parameterized constructor.

Eg-1.
```
class c1
{
    c1(int a, String n)
    {=
    }
}
```

2.
```
class c1
{
    c1(double s)
    {=
    }
}
```

b	Define recursion. Write a recursive program to find nth Fibonacci number.	L3	7

➢ Recursion

- Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.
- The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N*. For example, factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

// A simple example of recursion.

```
public class Fibonacci {
   public static void main(String[] args) {
      int n = 10; // Example: Find the 10th Fibonacci number
      int fibonacciNumber = fibonacci(n);
      System.out.println("The " + n + "th Fibonacci number is: " + fibonacciNumber);
   }

   public static int fibonacci(int n) {
      if (n <= 1) {
         return n; // Base case: Fibonacci of 0 is 0, Fibonacci of 1 is 1
      } else {
         // Recursive case: Fibonacci of n is the sum of Fibonacci(n-1) and Fibonacci(n-2)
         return fibonacci(n - 1) + fibonacci(n - 2);
      }
   }
}
```

	c	Explain the various access specifiers in Java.	L2	6

Solution:

Access Specifiers in Java

Java allows to control access to classes, methods & fields via access-specifiers.

Java provide four different types of access specifiers :

1. **public** - Public classes, methods & fields can be accessed from everywhere.
2. **protected** - Protected methods & fields can only be accessed within the same class to which the methods & fields belong, also within its subclasses but not from anywhere else.
3. **private** - Private methods & fields can only be accessed within the same class to which the methods & fields belong. They are not inherited by subclasses. It is mostly used for encapsulation
4. **default** - If we do not set access to specific level, then such a class, method or field will be accessible from inside the same package to which the class, method or field belongs, but not from outside this package.

Q.04	a	Explain call by value and call by reference with an example program	L3	7

Solution: when you pass arguments to a method, there are two ways in which these arguments can be passed: "call by value" and "call by reference". These concepts refer to how the parameters are passed to the method and how changes made to the parameters affect the original variables.

Call by Value:In call by value, a copy of the actual parameter's value is passed to the method. Any changes made to the parameter inside the method do not affect the original variable.

```
public class CallByValueExample {
   public static void main(String[] args) {
      int num = 10;
      System.out.println("Before calling the method: " + num);
      changeValue(num);
      System.out.println("After calling the method: " + num);
   }
   public static void changeValue(int value) {
      value = 20;
      System.out.println("Inside the method: " + value);
   }
}
```

In the example above, even though the value of value is changed inside the changeValue method, it does not affect the original variable num because Java passes arguments by value.

Call by Reference:

In call by reference, instead of passing the actual value, a reference to the memory location of the actual parameter is passed to the method. This means any changes made to the parameter inside the method will affect the original variable.

```
public class CallByReferenceExample {
   public static void main(String[] args) {
      StringBuilder str = new StringBuilder("Hello");
      System.out.println("Before calling the method: " + str);
      changeValue(str);
      System.out.println("After calling the method: " + str);
   }
   public static void changeValue(StringBuilder value) {
      value.append(" World");
      System.out.println("Inside the method: " + value);
   }
}
```

In this example, the **changeValue** method modifies the **StringBuilder** object passed to it, and this modification affects the original variable **str** because Java passes arguments by reference for objects.

Java solution

b	Write a program to perform Stack operations using proper class and Methods.	L3	7

Solution:

```
public class Stack {
  private int maxSize;
  private int[] stackArray;
  private int top;
  public Stack(int size) {
    maxSize = size;
    stackArray = new int[maxSize];
    top = -1; // Stack is initially empty
  }
  public void push(int value) {
    if (top == maxSize - 1) {
      System.out.println("Stack overflow! Cannot push element " + value);
    } else {
      stackArray[++top] = value;
      System.out.println("Pushed element: " + value);
    }
  }
  public int pop() {
    if (top == -1) {
      System.out.println("Stack underflow! Cannot pop element");
      return -1;
    } else {
      int value = stackArray[top--];
      System.out.println("Popped element: " + value);
      return value;
    }
  }
  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();
    }
  }
  public static void main(String[] args) {
    Stack stack = new Stack(5); // Create a stack of size 5
    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.display();
    stack.pop();
    stack.display();
  }
}
```

Java solution

	c	Explain the use of **this** in JAVA with an example.	L2	6

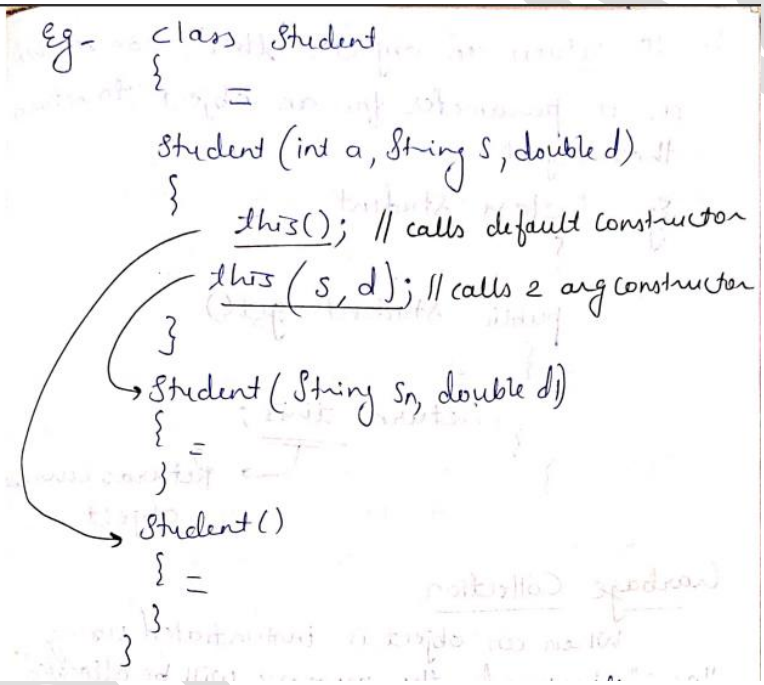
Solution:

"this" keyword

Java defines the "this" keyword to be used within any method to refer to the current object. Any member of the current object from within an instance method or a constructor can be referred by using "this".

There are many uses of "this" keyword.

1. To call a constructor within another constructor : this keyword can be used to call a constructor within another constructor of the same class. If used, it must be the first statement in the constructor



Java solution

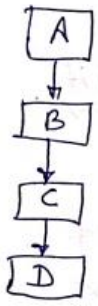
Q. 05	a	Write a Java program to implement multilevel inheritance with 3 levels of hierarchy.	L3	7

Solution:

4. Multilevel Inheritance:

Java does allows a class to extend another class which further extends another class & so on. This type is called as Multilevel Inheritance

The order of execution of constructors when an object of the class at the leaf level.



O/P

A's Constructor
B's constructor
C's Constructor
D's constructor

```
class A
{   A()
    { S.o.p("A's constructor");
    }
}
class B extends A
{   B()
    { S.o.p("B's constructor");
    }
}
class C extends B
{   C()
    { S.o.p("C's constructor");
    }
}
class D extends C
{   D()
    { S.o.p("D's constructor");
    }
}
public class Test.
{ public static void main(String args[])
  { D ob= new D();
  }
}
```

Java solution

```
// Parent class
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}
// Child class inheriting from Animal
class Dog extends Animal {
   void bark() {
      System.out.println("Dog is barking");
   }
}
// Grandchild class inheriting from Dog
class Labrador extends Dog {
   void color() {
      System.out.println("Labrador is brown in color");
   }
}
public class Main {
   public static void main(String[] args) {
      Labrador labrador = new Labrador();
      labrador.eat();   // Inherited from Animal
      labrador.bark();  // Inherited from Dog
      labrador.color(); // Own method of Labrador
   }
}
```

b	Explain how an interface is used to achieve multiple Inheritances in Java.

Solution:multiple inheritance refers to a scenario where a class inherits behaviors (methods) and properties (fields) from more than one superclass. Unlike some other programming languages like C++, Java does not support multiple inheritance with classes. However, Java provides a mechanism to achieve a form of multiple inheritance using interfaces.

An interface in Java is a reference type, similar to a class,that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot have instance fields or constructors. Classes can implement one or more interfaces, thus allowing them to inherit behavior from multiple sources.

Here's how an interface is used to achieve multiple inheritances in Java:

1. Define interfaces: Interfaces are declared using the `interface` keyword. They contain method signatures without any implementation details.

```
interface Interface1 {
   void method1();
}
interface Interface2 {
   void method2();
}
```

2. Implement interfaces: Classes implement interfaces using the `implements` keyword. A class can implement multiple interfaces.

```
class MyClass implements Interface1, Interface2 {
   // Implement method1() from Interface1
   public void method1() {
      // Implementation
   }
```

Java solution

```
// Implement method2() from Interface2
  public void method2() {
    // Implementation
  }
}
```

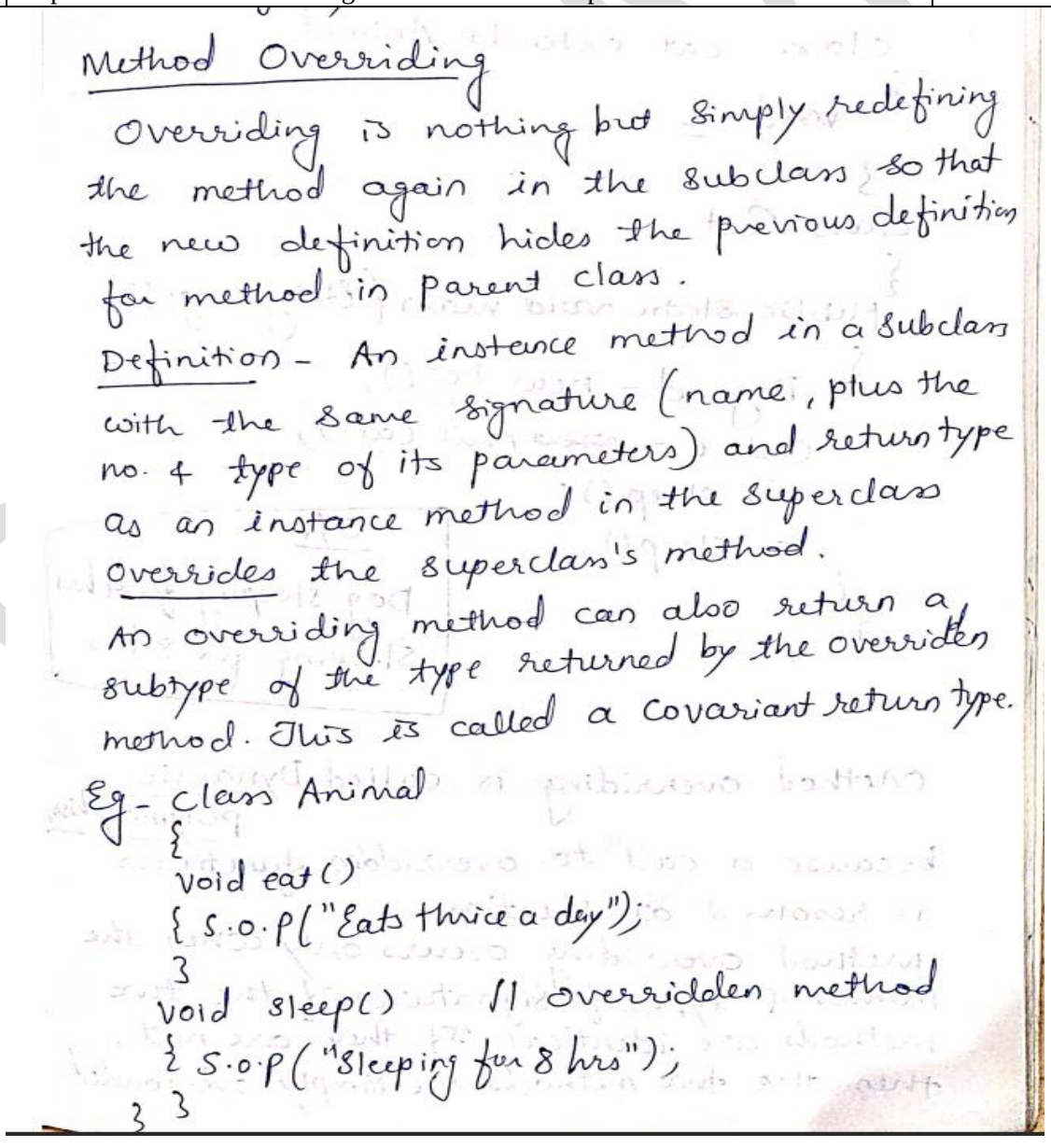
3. Override interface methods: The implementing class must provide concrete implementations for all the methods declared in the interfaces it implements.
4. Use the class: You can create an object of the implementing class and call its methods.

```
public class Main {
  public static void main(String[] args) {
    MyClass obj = new MyClass();
    obj.method1();
    obj.method2();
  }
}
```

By implementing multiple interfaces, a class can inherit and provide implementations for methods defined in those interfaces. This approach promotes code reusability and flexibility in design.

	c	Explain the method overriding with a suitable example.	L2	6

Solution:



Method Overriding

Overriding is nothing but simply redefining the method again in the subclass so that the new definition hides the previous definition for method in parent class.

Definition - An instance method in a subclass with the same signature (name, plus the no. & type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.

An overriding method can also return a subtype of the type returned by the overriden method. This is called a covariant return type.

Eg -
```
class Animal
 {
  void eat()
  { s.o.p("Eats thrice a day");
  }
 void sleep()   // overridden method
  { s.o.p("Sleeping for 8 hrs");
 } }
```

Java solution

Q. 06	a	What is single-level inheritance? Write a Java program to implement single-level inheritance.	L3	7

Solution:

Single Inheritance Example: When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal{
    void eat(){
        System.out.println("eating...");
    }
}
class Dog extends Animal{
    void bark(){
        System.out.println("barking...");
    }
}
class TestInheritance{
    public static void main(String args[]){
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}
```

Output:

```
barking...
eating...
```

b	What is the importance of the **super** keyword in inheritance? Illustrate with a suitable example.

Solution:

"Super" Keyword

Super keyword is used for:

1. To make a call to superclass constructor
2. To refer to superclass member when there is a name collision between superclass member & a derived/subclass member.

1. Super can be used to make a call to superclass constructor from the subclass constructor.

Eg

Syntax

```
super(arg-list);
```

super() must always be the first stmt executed inside a subclass' constructor.

eg-

```
class Box
{
  Box(double w, double h, double d)
  {
  }
}
class BoxWeight extends Box
{
  double weight;
  BoxWeight(double w, double h, double d,
            double m)
  {
    super(w, h, d); // superclass Box constructor is called.
    weight = m;
  }
}
```

Java solution

	c	What is abstract class and abstract method? Explain with an example.	L2	6

Solution:

Abstract Classes in Java

There are situations in which we want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

The situations may be encountered where we need to enforce a restriction that some functions in the superclass has to mandatorily overridden by the subclass. In that case, prefix its declaration with 'abstract' keyword.

Abstract method - It is a method that is declared without an implementation.

```
abstract void f1(int a);
```

Abstract class - It is a class that is declared abstract - it may or may not include abstract methods. These classes cannot be instantiated, but they can be subclassed. If a class includes abstract methods, the class itself must be declared abstract

```
abstract class classname
{  =
   abstract rettype methodname(args);
   =
}
```

Eg -

```
abstract class Animal
{ abstract void f1();
  =
}
```

```
class Dog extends Animal
{ void f1()
  {
  }
}
```

Animal a = new Animal(); // can't create obj of abstract

Java solution

7a	Define package. Explain the steps involved in creating a user-defined package with an example.	L2	7

Solution:

Packages & Interfaces

Packages. Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.

The package is both a naming & a visibility control mechanism.

Classes can be defined inside a package that are not accessible by code outside that package. Even class members can be defined that are ~~not~~ only exposed to other members of the same package.

Defining a Package

To create a package, simply include a package command as the first stmt in a Java source file. Any classes declared within that file will belong to the specified package.

The package stmt defines a name space in which classes are stored. If package stmt is omitted, class names are put into the default package, which has no name.

Syntax

```
package pkg;
```

pkg → name of the package.

Eg- package MyPackage;

Java solution

b	Write a program that contains one method that will throw an IllegalAccessException and use proper exception handles so that the exception should be printed.	L3	7

Solution:

```
public class Main {
public static void main(String[] args) {
    try {
        // Call a method that throws IllegalAccessException
        performOperation();
    } catch (IllegalAccessException e) {
        // Handle the exception
        System.out.println("Caught IllegalAccessException: " + e.getMessage());
    }
}
// Method that throws IllegalAccessException
public static void performOperation() throws IllegalAccessException {
    // Simulating a scenario where IllegalAccessException occurs
    throw new IllegalAccessException("Access denied");
}
}
```

c	Define an exception. What are the key terms used in exception handling? Explain.	L2	6

Solution:

In programming, an exception is an event or object that occurs during the execution of a program, which disrupts the normal flow of instructions. Exceptions are typically caused by errors in the program's logic, unexpected conditions, or external factors such as input/output failures. In Java and many other programming languages, exceptions are represented as objects that contain information about the error, such as its type, message, and possibly other relevant details.

Key terms used in exception handling include:

1. Exception: An object representing an exceptional condition that has occurred during the execution of a program. Exceptions can be thrown (generated) by the program or by the runtime environment.

2. Throw: The act of explicitly raising (throwing) an exception within a program. This is typically done using the `throw` keyword followed by an exception object.

3. Try: A block of code where exceptions may occur. It is followed by one or more catch blocks and/or a finally block.

4. Catch: A block of code that handles (catches) an exception thrown within a try block. It specifies the type of exception it can handle and provides code to handle the exception gracefully.

5. Finally: A block of code that is always executed, regardless of whether an exception occurs or not. It is typically used for cleanup tasks such as closing resources (e.g., files, network connections) or releasing locks.

Java solution

6. Try-with-resources: A Java feature introduced in Java 7 that allows automatic resource management. It ensures that resources (e.g., files, streams) are closed properly, even if an exception occurs, by declaring and initializing the resources within a try block. The resources are automatically closed when the try block exits, either normally or due to an exception.

7. Checked Exception: An exception that must be declared in a method's signature using the `throws` keyword or handled using a try-catch block. Examples include IOException and SQLException.

8. Unchecked Exception: An exception that does not need to be declared or handled explicitly. These are subclasses of RuntimeException or Error. Examples include NullPointerException and ArrayIndexOutOfBoundsException.

9. Exception Handling: The process of detecting, reacting to, and resolving exceptions in a program. It involves writing code to handle exceptions gracefully, prevent program crashes, and provide meaningful error messages to users.

10. Exception Propagation: The mechanism by which an exception is passed from one method to another or from one part of the program to another. If an exception is not caught and handled within a method, it is propagated up the call stack until it is caught or until it reaches the top-level of the program, resulting in termination of the progr

Java solution

Q. 08	a	Explain the concept of importing packages in Java and provide an example demonstrating the usage of the import statement.	L2	7

Solution:

Importing Packages

Java includes the import stmt to bring certain classes, or entire packages into visibility.

Once imported, a class can be referred to directly, using only its name.

The import stmt is a convenience to the programmer & is not technically needed to write a complete Java program. It saves lot of typing.

In a source file, import stmt occur immediately following the package stmt & before any class definitions.

Syntax:

```
import pkg1[.pkg2[.(classname|*)];
```

pkg1 → top level package; pkg2 → subordinate package

① Eg-

```
import java.io.*;
import java.util.Date;
import java.lang.*;
```

②

```
package MyPack;
class Balance
{
  ...
}
```

```
import MyPack.*;
class Cl
{
  P S V main()
  {
    Balance b = new Balance();
  }
}
```

Java solution

b	How do you create your own exception class? Explain with a program.	L3	7

Solution: In Java, you can create your own custom exception classes by extending the built-in Exception class or one of its subclasses like **RuntimeException**. This allows you to define specific types of exceptions that are relevant to your application's domain. to create a custom exception class:

```
class CustomException extends Exception {
   // Constructor to initialize the exception message
   public CustomException(String message) {
      super(message);
   }
}
class DataProcessor {
      public void processData(int data) throws CustomException {
      if (data < 0) {
         throw new CustomException("Invalid data: " + data);
      } else {
         System.out.println("Data processed successfully: " + data);
      }
   }
}
public class Main {
   public static void main(String[] args) {
      DataProcessor processor = new DataProcessor();
      try {
         processor.processData(10);
         processor.processData(-5);
      } catch (CustomException e) {
         System.out.println("CustomException occurred: " + e.getMessage());
      }
}
}
```

Java solution

c	Demonstrate the working of a nested try block with an example.	L2	6

Solution: Nested try blocks are used when one part of a block may raise multiple exceptions, and each type of exception requires different handling. By nesting try-catch blocks, you can handle exceptions at different levels of granularity. Here's an example demonstrating the working of nested try blocks:

```
public class Main {
   public static void main(String[] args) {
      try {
         try {
            int[] numbers = {1, 2, 3};
            System.out.println("Array element at index 3: " + numbers[3]); // This will throw
ArrayIndexOutOfBoundsException
         } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException caught: " + e.getMessage());
         }
         String str = null;
         System.out.println("Length of string: " + str.length()); // This will throw NullPointerException
      } catch (NullPointerException e) {
         System.out.println("NullPointerException caught: " + e.getMessage());
      } catch (Exception e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }
}
```

In this example:

- The outer try block contains an inner try block and another block of code.

- The inner try block tries to access an element at index 3 of an array, which results in an `ArrayIndexOutOfBoundsException`.

- The `catch` block inside the inner try block catches and handles the `ArrayIndexOutOfBoundsException`.

- After handling the exception, execution continues with the code in the outer try block.

- Inside the outer try block, another block of code tries to get the length of a null string, which results in a `NullPointerException`.

- The `catch` block outside the inner try block catches and handles the `NullPointerException`.

- If any other exception occurs that is not caught by the specific catch blocks, the catch block for `Exception` at the outermost level will catch and handle it.

This example demonstrates how nested try blocks allow for more granular exception handling, where specific exceptions can be caught and handled at different levels of the code.

Java solution

Q. 09	a	What do you mean by a thread? Explain the different ways of creating threads.	L2	7

Solution: In programming, a thread refers to the smallest unit of execution within a process. A thread is a lightweight process that can execute independently and concurrently with other threads. Threads share the same memory space and resources within a process, allowing them to interact with each other efficiently. Threads enable concurrent execution of tasks, which can improve performance and responsiveness in multi-tasking and multi-user environments.

There are several ways to create threads in Java:

1. Extending the Thread class: In this approach, you create a new class that extends the `Thread` class and overrides its `run()` method. You then instantiate objects of this class and call the `start()` method on them to start the execution of the thread.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}
public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

2. Implementing the Runnable interface: Instead of extending the `Thread` class, you can implement the `Runnable` interface, which defines a single method `run()`. This approach is preferred because Java allows multiple interfaces to be implemented by a class, but only single inheritance is allowed. You then create a `Thread` object, passing an instance of your class that implements `Runnable` as a parameter, and call the `start()` method on the thread object.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}
public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
    }
}
```

Java solution

3. Using lambda expressions: Since Java 8, you can use lambda expressions to define the `run()` method directly when creating a new `Thread` object.

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Thread is running");
        });
        thread.start();
    }
}
```

Each of these methods has its own advantages and use cases. However, using the `Runnable` interface is generally recommended because it separates the task from the thread's execution logic, promoting better code organization and reusability.

9b	What is the need of synchronization? Explain with an example how synchronization is implemented in JAVA.	L3	7

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- **To enter an object's monitor, just call a method that has been modified with the synchronized keyword.**
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

multiple threads. Synchronization in Java ensures that only one thread at a time can access a shared resource, thereby preventing data corruption or inconsistency due to concurrent access by multiple threads. This is particularly important in multithreaded environments where multiple threads may be accessing and modifying shared data simultaneously.

In Java, synchronization can be achieved using several mechanisms:

1.Synchronized methods: By declaring a method as `synchronized`, Java ensures that only one thread can execute that method at a time on a given instance of the class.

```
public synchronized void synchronizedMethod() {
```

2. Synchronized blocks: You can use synchronized blocks to lock on a specific object or class instance.

```
public void someMethod() {
    synchronized(this) {
    }
}
```

Java solution

3. Synchronization on a specific object: You can synchronize on a specific object to control access to critical sections of code.

```
Object lock = new Object();

public void someMethod() {
   synchronized(lock) {
   }
}
```

4. Using the `synchronized` keyword: You can also use the `synchronized` keyword to synchronize on class-level methods or blocks.

```
public class MyClass {
   public static synchronized void staticMethod() {
   }
   public void someMethod() {
      synchronized(MyClass.class) {
      }
   }
}
```

Synchronization ensures that only one thread can execute the synchronized code block at a time, preventing potential race conditions and ensuring data consistency when accessing shared resources. However, it's essential to use synchronization judiciously as it can impact performance, and improper synchronization can lead to deadlocks or other concurrency issues.

c	Discuss values() and value Of() methods in Enumerations with suitable examples.	L2	6

Solution:

The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: **values**( ) and **valueOf**( ). Their general forms are shown here:

public static *enum-type* [ ] values( )
public static *enum-type* valueOf(String *str*)

- **The values( ) method returns an array that contains a list of the enumeration constants.**
- **The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.**
- **In both cases, enum-type is the type of the enumeration.**

The following program demonstrates the **values**( ) and **valueOf**( ) methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Apple ap;

    System.out.println("Here are all Apple constants:");

    // use values()
    Apple allapples[] = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);

    System.out.println();

    // use valueOf()
    ap = Apple.valueOf("Winesap");
    System.out.println("ap contains " + ap);

  }
}
```

Java solution

10.a	What is multithreading? Write a program to create multiple threads in JAVA.	L2	7

Solution: Multithreading is a programming concept that allows multiple threads of execution to run concurrently within a single process. Each thread represents a separate flow of control within the program, enabling it to perform tasks independently and concurrently with other threads. Multithreading is commonly used in applications that require handling multiple tasks simultaneously, such as user interfaces, server applications, and parallel processing tasks.

Here's a simple Java program that demonstrates how to create multiple threads:

```
public class MultiThreadDemo {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable(), "Thread 1");
        Thread thread2 = new Thread(new MyRunnable(), "Thread 2");
        Thread thread3 = new Thread(new MyRunnable(), "Thread 3");
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
            try {
                Thread.sleep(1000); // Sleep for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

b	Explain with an example how inter-thread communication is implemented in JAVA.	L3	7

SOLUTION: Inter-thread communication in Java refers to the ability of threads to coordinate their activities by signaling each other. This coordination allows threads to synchronize their actions and exchange data effectively. Java provides mechanisms such as wait(), notify(), and notifyAll() methods, along with the synchronized keyword, to facilitate inter-thread communication.

Java solution

```
public class InterThreadCommunicationExample {
   public static void main(String[] args) {
      Message message = new Message();
      Thread producerThread = new Thread(new Producer(message), "Producer");
      Thread consumerThread = new Thread(new Consumer(message), "Consumer");
      producerThread.start();
      consumerThread.start();
   }
}
class Message {
   private String message;
   private boolean empty = true;
    public synchronized void produce(String message) {
       while (!empty) {
          try {
             wait();
          } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
          }
       }
       this.message = message;
       empty = false;
       notify();
   }
      public synchronized String consume() {
       while (empty) {
          try {
             wait();
          } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
          }
       }
```

Java solution

```
        empty = true;
        notify();
        return message;
    }
}
class Producer implements Runnable {
    private Message message;
    public Producer(Message message) {
        this.message = message;
    }
    @Override
    public void run() {
        String[] messages = {"Message 1", "Message 2", "Message 3", "Message 4", "Message 5"};
        for (String msg : messages) {
            message.produce(msg);
            System.out.println("Produced: " + msg);
  try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
class Consumer implements Runnable {
    private Message message;
   public Consumer(Message message) {
    this.message = message;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            String msg = message.consume();
            System.out.println("Consumed: " + msg);
```

Java solution

```
 try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
         }
      }
   }
}
```

c	Explain auto-boxing/unboxing in expressions.	L2	6

Solution:

Autoboxing/Unboxing Occurs in Expressions

- In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required.
- This applies to expressions.
- Within an expression, a numeric object is automatically unboxed.
- The outcome of the expression is reboxed, if necessary.
- For example, consider the following program:

Java solution

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);

    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);

    // The same expression is evaluated, but the
    // result is not reboxed.
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);

  }
}
```