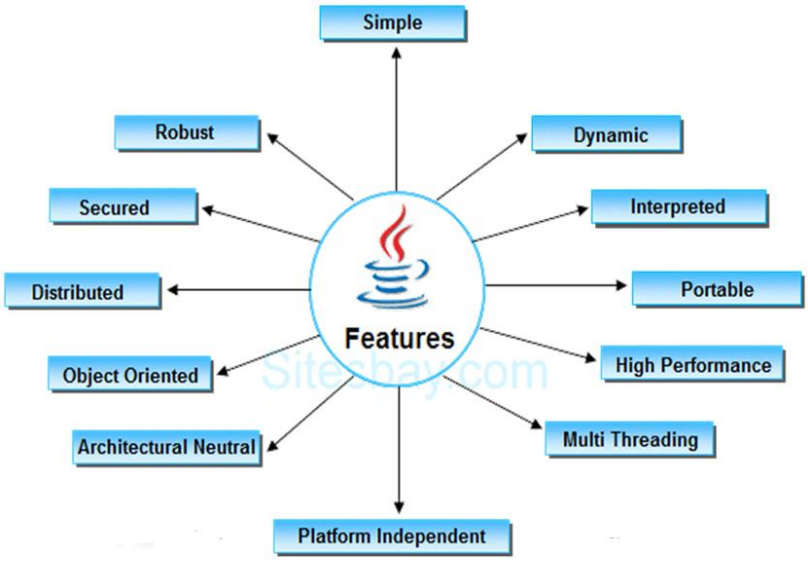
Solution with scheme-Model Answer

Dr. D. Adhimuga Sivasakthi									
University Examination – February 2026									
Sub	**Object Oriented Programming with Java**					Sub code	**BCS306A**	Branch	**CSE**
Date	**16.02.2025**	Duration	**3 Hrs**	Max Marks	**100**	Sem /Sec	**III Sem (A, B, C)**		**OBE**

	Answer any FIVE FULL Questions	**MARKS**	**CO**	**RBT**
	Module-1			
1	**a) Explain features of Java**	**[7]**	**CO1**	**L2**
	b) Define array. Write a java program to calculate the average among the elements [8,6,2,7]	**[7]**	**CO1**	**L3**
	c) List and explain operators in Java wish example	**[6]**	**CO1**	**L2**
	OR			
2	**a) Explain OOPS features in java**	**[7]**	**CO1**	**L2**
	b) Write a java program to sort the elements using a for loop	**[7]**	**CO1**	**L3**
	c) With example, explain different types of if statement in Java	**[6]**	**CO1**	**L2**
	Module-2			
3	**a) Define Constructor. Explain two types of constructors with an example**	**[7]**	**CO2**	**L3**
	b) Define Recursion. Write a recursive program to find factorial of a number	**[7]**	**CO2**	**L3**
	c) Explain garbage collection with an example, explain final and finalize() method	**[6]**	**CO2**	**L2**
	OR			
4	**a) Define class. Explain call by value and call by reference with an example program**	**[7]**	**CO2**	**L3**
	b) Using proper class and methods, write a program to perform stack operations	**[7]**	**CO2**	**L3**
	c) Explain the use of this keyword in java with an example	**[6]**	**CO2**	**L2**
	Module-3			
5	**a) Write a java program to implement multilevel inheritance with 3 levels of hierarchy**	**[7]**	**CO3**	**L3**
	b) Define interface. With suitable program explain nested interface in java	**[7]**	**CO3**	**L3**
	c) Explain dynamic method dispatch with a suitable example	**[6]**	**CO3**	**L2**
	OR			
6	**a) Explain inheritance. Write a java program to implement single level inheritance**	**[7]**	**CO3**	**L3**
	b) Explain the importance of the super keyword in inheritance, illustrate with a suitable example	**[7]**	**CO3**	**L3**
	c) Define method overloading and overriding with example	**[6]**	**CO3**	**L2**
	Module-4			
7	**a) Define package, with an example, explain the steps are involved in creating a user-defined package.**	**[7]**	**CO4**	**L2**
	b) With sample code, explain chained exception	**[7]**	**CO4**	**L3**
	c) Define an exception, with syntax explain all five keywords used in exception handling	**[6]**	**CO4**	**L2**
	OR			
8	**a) Explain the concept of package importing in java with example**	**[7]**	**CO4**	**L2**
	b) How do you create your own exception class. Explain with a program	**[7]**	**CO4**	**L2**
	c) With an example, explain working of a nested try block within an exception	**[6]**	**CO4**	**L3**
	Module-5			
9	**a) Define Thread. With diagram explain the java thread model**	**[7]**	**CO5**	**L2**
	b) Define Synchronization with an example, how synchronization is implemented in java	**[7]**	**CO5**	**L3**
	c) With suitable example, explain values() and valueOf() method in enumeration	**[6]**	**CO5**	**L2**
	OR			
10	**a) Define Multithreading, write a program to create multiple threads in java**	**[7]**	**CO5**	**L2**
	b) Demonstrate the usage of compareTo() and equals() method with enumeration constants	**[7]**	**CO5**	**L3**
	c) Explain autoboxing / unboxing in expressions	**[6]**	**CO5**	**L2**

1	**a) Explain features of Java** **List the features (2)** **Explanation (5)**  **Simple** Java is easy to learn and understand. Removed complex features like pointers Automatic memory management **Object-Oriented** Everything is based on objects. **Main OOP concepts:** **Encapsulation** Encapsulation is the process of **wrapping data (variables) and methods (functions) together into a single unit (class)** and restricting direct access to the data to protect it. **Inheritance** Inheritance is a mechanism where **one class acquires the properties and behaviors of another class**, allowing code reusability. **Polymorphism** Polymorphism means **one name having many forms**, where the same method behaves differently in different situations (method overloading or overriding). **Abstraction** Abstraction is the process of **hiding implementation details and showing only essential features** of an object. **Platform Independent** Java follows “Write Once, Run Anywhere” (WORA). Java code is compiled into bytecode Bytecode runs on the Java Virtual Machine (JVM)	[7]

JVM makes it platform independent

Secure

Java provides strong security features:

No explicit pointers

Bytecode verifier

Security manager

Runtime checking

Robust

Java is strong and reliable.

Exception handling

Garbage collection

Strong memory management

Multithreaded

Java supports multiple threads (tasks) running simultaneously.

Distributed

Java supports distributed applications.

Networking support

Remote Method Invocation (RMI)

High Performance

Though slower than C/C++, Java is faster than many interpreted languages because:

Uses Just-In-Time (JIT) compiler

Portable

Bytecode can run on any system with JVM.

Dynamic

Java supports dynamic loading of classes at runtime.

b) **Define array. Write a java program to calculate the average among the elements [8,6,2,7]**

Definition (1)
Program (6)
An **array** is a collection of elements of the same data type stored in **contiguous memory locations** and accessed using a single variable name with an index.

Program:

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

int[] numbers = {8, 6, 2, 7};
int sum = 0;

// Calculate sum
for(int i = 0; i < numbers.length; i++) {
sum += numbers[i];
```

```
}

// Calculate average
double average = (double) sum / numbers.length;

System.out.println("Average = " + average);
}
}
```

Output:

Sum = 8 + 6 + 2 + 7 = 23
Average = 23 / 4 = **5.75**

c) **List and explain operators in Java wish example**

List of operators 1)
Explanation (4)
Example (1)

	Java Operator	Examples
1	Arithmetic	+ , – , / , * , %
2	Unary	++ , – – , !
3	Assignment	= , += , -= , *= , /= , %= , ^=
4	Relational	==, != , < , >, <= , >=
5	Logical	&& , ||
6	Ternary	(Condition) ? (Statement1) : (Statement2)
7	Bitwise	& , | , ^ , ~
8	Shift	<< , >> , >>>

Explanation

Operators in Java are special symbols that perform operations on variables and values (operands) to produce a result.

Arithmetic operators are used to perform **mathematical calculations** like addition, subtraction, multiplication, division, and modulus.

Relational operators are used to **compare two values** and return a boolean result (`true` or `false`).

Logical operators are used to **combine multiple boolean conditions**.

Assignment operators are used to **assign values to variables**.

Unary operators operate on **only one operand**.

Bitwise operators perform operations on **individual bits** of integer data.

The ternary operator is a **conditional operator** that works as a short form/ replacement of `if-else`.

Shift operators are bitwise operators used to **shift the bits of a number to the left or right**. They work on integer data types (`byte`, `short`, `int`, `long`). The left shift operator shifts all bits to the **left** by a specified number of positions. Zeroes (`0`) are filled in from the right. Each left shift by 1 position multiplies the number by 2.

The right shift operator shifts bits to the **right**. For positive numbers, zeroes are filled from the left. For negative numbers, the sign bit is preserved. Each right shift by 1 position divides the number by 2.

2. a) **Explain OOPS features in java**

List the OOPS features (2)

Explanation of major 4 features (4)

Example (1)

Object-Oriented Programming (OOP) is a programming paradigm based on objects and classes. Java follows OOP principles to make programs secure, reusable, and easy to maintain.

OOPS Features:

Encapsulation

Encapsulation is the process of **wrapping data (variables) and methods (functions) together into a single unit (class)** and restricting direct access to data.

It is achieved using:

- `private` data members
- `public` getter and setter methods

Advantages:

- Data hiding
- Security
- Better control over data
- Improves maintainability

Example idea: A `Student` class with private marks and public methods to access them.

```
class Student {
private int marks; // data hidden

public void setMarks(int m) { // setter
marks = m;
}

public int getMarks() { // getter
return marks;
}
}
```

Inheritance

Inheritance is a mechanism where **one class acquires the properties and behaviors of another class**.

It is achieved using the `extends` keyword.

Types of inheritance in Java:

- Single
- Multilevel
- Hierarchical
  (Multiple inheritance is achieved using interfaces.)

Advantages:

- Code reusability
- Reduces redundancy
- Supports method overriding

Example: `Dog` class inheriting from `Animal` class.

```
class Animal {
void eat() {
System.out.println("Animal is eating");
}
}

class Dog extends Animal {
void bark() {
System.out.println("Dog is barking");
}
}
```

Polymorphism

Polymorphism means **one name having many forms**. It allows methods to perform different tasks based on context.

Types:

- **Compile-time Polymorphism** (Method Overloading)
- **Runtime Polymorphism** (Method Overriding)

Advantages:

- Flexibility
- Improves readability
- Supports dynamic method dispatch

Example: Overloading as same method `add()` with different parameters and Overriding as same method sound() with same parameters.

Overloading :-

```
class MathOperation {
int add(int a, int b) {
return a + b;
}

double add(double a, double b) {
return a + b;
}
}
```

```
class Main
{ public static void main(String arg[])
{
MathOperation c=new MathOperation ();
c.add(1.5,2.6);
c.add(10,3);
}
}
```

Overriding :-

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

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

class Main
{ public static void main(String arg[])
{
Dog c=new Dog ();
c.sound();
}
}
```

Abstraction

Abstraction is the process of **hiding implementation details and showing only essential features**.

It is achieved using:

- Abstract classes
- Interfaces

Advantages:

- Reduces complexity
- Improves security
- Focuses on what an object does instead of how it does it

Example: A `Shape` class with abstract method `draw()`.

```
abstract class Shape {
abstract void draw(); // no implementation
}

class Circle extends Shape {
void draw() {
System.out.println("Drawing Circle");
```

```
}
}

class Main
{ public static void main(String arg[])
{
Circle c=new Circle();
c.draw();
}
}
```

2 b) Write a java program to sort the elements using a for loop

Reading runtime element (1)

Program to Sort elements in ascending order (6)

```
import java.util.Scanner;

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

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of elements: ");
int n = sc.nextInt();

int[] arr = new int[n];

// Getting elements at runtime
System.out.println("Enter elements:");
for(int i = 0; i < n; i++) {
arr[i] = sc.nextInt();
}

// Sorting using simple for loop
int temp;
for(int i = 0; i < n; i++) {
for(int j = i + 1; j < n; j++) {
if(arr[i] > arr[j]) {
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
}
}

System.out.println("Sorted elements:");
for(int i = 0; i < n; i++) {
System.out.print(arr[i] + " ");
}

sc.close();
}
}
```

Input:

Enter number of elements: 4
Enter elements:
8
6
2
7

Output:- 2 6 7 8

2 c) **With example, explain different types of if statement in Java**

Explanation (4)
Example (2)

In Java, **if statements** are used to make decisions based on conditions.They control the flow of execution depending on whether a condition is true or false.

There are **four main types** of if statements in Java:

1.Simple `if` Statement

It executes a block of code **only if the condition is true**.

Syntax:

```
if(condition) {
    // statements
}
```

Example:

```
int age = 20;

if(age >= 18) {
    System.out.println("Eligible to vote");
}
```

2.`if-else` Statement

It executes one block if the condition is true, and another block if it is false.

Syntax:

```
if(condition) {
    // true block
} else {
    // false block
}
```

Example:

```
int number = 5;

if(number % 2 == 0) {
    System.out.println("Even number");
} else {
    System.out.println("Odd number");
}
```

3.`else-if` Ladder

Used when there are **multiple conditions** to check.

Syntax:

```
if(condition1) {
    // block1
} else if(condition2) {
```

```
    // block2
} else {
    // default block
}
```

Example:

```
int marks = 75;

if(marks >= 90) {
    System.out.println("Grade A");
}
else if(marks >= 60) {
    System.out.println("Grade B");
}
else {
    System.out.println("Grade C");
}
```

4. Nested `if` Statement

An `if` statement inside another `if` statement.

Syntax:

```
if(condition1) {
    if(condition2) {
        // statements
    }
}
```

Example:

```
int age = 25;
boolean hasLicense = true;

if(age >= 18) {
    if(hasLicense) {
        System.out.println("You can drive");
    }
}
```

3 a) **Define Constructor. Explain two types of constructors with an example**

Definition (2)

List of constructors (1)

Explanation for 2 types of constructors (2)

Example (2)

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

There are mainly **two types** of constructors:

- Default Constructor
- Parameterized Constructor

A **default constructor** is a constructor that does not take any parameters. If we do not write any constructor, Java automatically provides a default constructor.

A **parameterized constructor** accepts parameters to initialize variables with user-defined values.

```
class Student {
int rollNo;

// Default Constructor
Student() {
rollNo = 101;
}

// Parameterized Constructor
Student(int r) {
rollNo = r;
}

void display() {
System.out.println("Roll No: " + rollNo);
}

public static void main(String[] args) {
Student s = new Student(); // Constructor called
s.display();

Student s = new Student(203); // Passing value
s.display();
}
}
```

Output: -

```
Roll No: 101
Roll No: 203
```

3 b) **Define Recursion. Write a recursive program to find factorial of a number**

Definition (2)
Program (5)

Recursion is a process in which a method **calls itself** repeatedly to solve a problem, until a stopping condition (base case) is reached.

A recursive function must have:

- **Base case** – stops recursion
- **Recursive call** – function calls itself

```
import java.util.Scanner;

public class fact {

// Recursive method
static int factorial(int n) {
if (n == 0 || n == 1) { // Base case
return 1;
} else {
return n * factorial(n - 1); // Recursive call
}
}

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

```
System.out.print("Enter a number: ");
int num = sc.nextInt();

int result = factorial(num);
System.out.println("Factorial = " + result);

sc.close();
}
}
```

Output:-

```
Enter a number: 5
Factorial = 120
```

3 c) **Explain garbage collection with an example, explain final and finalize() method**

Definition of garbage collection (2)

Example of garbage collection (1)

Explanation of final and finalize() (4)

Garbage Collection (GC) is the process by which the Java Virtual Machine (JVM) automatically removes objects from memory that are no longer being used.

- It helps in **automatic memory management**
- It removes **unreferenced objects** from heap memory

An object becomes eligible for garbage collection when:

- Its reference is set to `null`
- It is reassigned to another object
- It goes out of scope
- Anonymous object has no reference

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

Test obj1 = new Test(); // Object 1 created
Test obj2 = new Test(); // Object 2 created

obj1 = null; // Object 1 eligible for GC

obj2 = new Test(); // Old Object 2 eligible for GC

System.gc(); // Request JVM to run GC
}
}
```

The **`final` keyword** is used to restrict modification.

It can be used with:

- Variables
- Methods

- Classes

Final Variable

Once assigned, its value **cannot be changed**.

```
final int x = 10;
x = 20;   // Error
```

Final Method

A final method **cannot be overridden**.

```
class A {
    final void display() {
        System.out.println("Final method");
    }
}
```

Final Class

A final class **cannot be inherited**.

Example: `String` class in Java is final.

finalize():-

The `finalize()` method is a method defined in the `Object` class.

- It is called by the Garbage Collector **before destroying an object**
- Used to perform clean-up activities

class Test {

protected void finalize() {
System.out.println("Object is garbage collected");
}

public static void main(String[] args) {
Test obj = new Test();
obj = null;
System.gc();
}
}

4 a) **Define class. Explain call by value and call by reference with an example program**

Definition (1)

Explanation of call by value and reference (4)

Example (2)

A **class** in Java is a blueprint or template used to create objects.
It defines the properties (variables) and behaviors (methods) that the objects created from it will have.

A class contains:

- Data members (variables)
- Methods (functions)
- Constructors

Java supports **call by value only**, but it may appear like call by reference when objects are passed.

In call by value, a **copy of the variable's value** is passed to the method. Changes made inside the method **do not affect** the original variable. Since, the parameters are local to the function, updates in values cannot be reflected in the main program. ie., lifetime/scope of parameters that are passed to variables are local to the function.

```
 class Main {

void change(int num) {
num = num + 10;
}

public static void main(String[] args) {
Main obj = new Main();
int x = 5;

obj.change(x); // The value of x remains unchanged because only a copy was passed

System.out.println("Value of x: " + x);
}
}
```

Output:-

```
Value of x: 5
```

In Java, when an object is passed, the **reference value (address) is passed by value**. So changes made to object data inside the method will affect the original object.

```
class Main {

int x;

}

class reference {

void change(Main s) {
s.x=10;
}

public static void main(String[] args) {
Main obj = new Main();
 obj.x = 5;
```

```
reference obj1=new reference();

obj1.change(obj);
System.out.println("Value of x: " + obj.x);
}
}
```

Output:-

Value of x: 10

4 b) **Using proper class and methods, write a program to perform stack operations**

Program (7)

```
import java.util.Scanner;
// Stack class to hold a maximum of 10 integers
class IntStack {
    private int[] stack = new int[10];
    private int top = -1;
    // Push an element onto the stack
    public void push(int value) {
        if (top == stack.length - 1) {
            System.out.println("Stack Overflow! Cannot push
" + value);
        } else {
            stack[++top] = value;
            System.out.println(value + " pushed to stack.");
        }
    }
    // Pop an element from the stack
    public int pop() {
        if (top == -1) {
            System.out.println("Stack Underflow! Cannot
pop.");
            return -1; // or throw exception
        } else {
            int value = stack[top--];
            System.out.println(value + " popped from stack.");
            return value;
        }
    }
    // Display all elements
    public void display() {
        if (top == -1) {
            System.out.println("Stack is empty.");
        } else {
            System.out.print("Stack elements (top to bottom):
");
    }
}
            for (int i = top; i >= 0; i--) {
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
// Main class to illustrate stack operations
public class IntStackDemo {
```

```
    public static void main(String[] args) {
        IntStack s = new IntStack();
        Scanner sc = new Scanner(System.in);
        int choice;
        do {
            System.out.println("\n--- Stack Menu ---");
            System.out.println("1. Push");
            System.out.println("2. Pop");
            System.out.println("3. Display");
            System.out.println("4. Exit");
            System.out.print("Enter choice: ");
            choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter integer to push: ");
                    int val = sc.nextInt();
                    s.push(val);
                    break;
                case 2:
                    s.pop();
                    break;
                case 3:
                    s.display();
                    break;
                case 4:
                    System.out.println("Exiting...");
                    break;
                default:
                    System.out.println("Invalid choice!");
            }
        } while (choice != 4);

        sc.close();
         new Scanner(System.in).nextLine();
    }
}
```

4 c) **Explain the use of this keyword in java with an example**

Definition of this keyword (2)

Uses of this keyword (2)

Example (2)

The **`this` keyword** in Java is a reference variable that refers to the **current object** of a class. It is mainly used to differentiate between **instance variables** and **local variables**, and to invoke constructors or methods of the current class.

Uses:

- To Refer Current Class Instance Variable
- To Invoke Current Class Method
- To Invoke Current Class Constructor
- To Pass Current Object as Argument

- To Refer Current Class Instance Variable

When local variable and instance variable have the same name, `this` is used to distinguish them.

```
class Student {
int rollNo;

Student(int rollNo) {
this.rollNo = rollNo; // instance variable = local variable
}

void display() {
System.out.println("Roll No: " + rollNo);
}

public static void main(String[] args) {
Student s1 = new Student(101);
s1.display();
}
}
```

- To Invoke Current Class Method

```
class Test {

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

    void display() {
        this.show();   // calling current class method
    }

    public static void main(String[] args) {
        Test obj = new Test();
        obj.display();
    }
}
```

- To Invoke Current Class Constructor

Used in constructor chaining.

```
class Test {

    Test() {
        this(10);   // calling parameterized constructor
        System.out.println("Default Constructor");
    }

    Test(int x) {
        System.out.println("Parameterized Constructor: " + x);
    }

    public static void main(String[] args) {
        Test obj = new Test();
    }
}
```

- To Pass Current Object as Argument

```
class Test {

    void display(Test obj) {
        System.out.println("Method called");
    }

    void show() {
        display(this);   // passing current object
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show();
    }
}
```

5 a) **Write a java program to implement multilevel inheritance with 3 levels of hierarchy**

Definition of Multilevel inheritance (1)

Program (6)

In multilevel inheritance, a class inherits from a class, and another class inherits from that derived class.

Example hierarchy:
Grandparent → Parent → Child

```
// Level 1 (Grandparent)
class Grandparent {
void house() {
System.out.println("Grandparent's House");
}
}

// Level 2 (Parent inherits Grandparent)
class Parent extends Grandparent {
void car() {
System.out.println("Parent's Car");
}
}

// Level 3 (Child inherits Parent)
class Child extends Parent {
void bike() {
System.out.println("Child's Bike");
}

public static void main(String[] args) {
Child obj = new Child();

obj.house(); // Method from Grandparent
obj.car(); // Method from Parent
obj.bike(); // Method from Child
}
}
```

Output:

Grandparent's House
Parent's Car
Child's Bike

5 b) **Define interface. With suitable program explain nested interface in java**

Definition of interface (2)

Explanation of nested interface (1)

Program for nested interface (4)

Definition of interface:

- An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- It's a contract between interface and classes
- The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It **cannot have a method body**.
- Java Interface also represents the IS-A relationship.
- It cannot be instantiated just like the abstract class.

Nested Interface in Java

A **nested interface** is an interface declared **inside another class or interface**.

It is also called a **member interface**.

- It is useful for logically grouping interfaces that are only used in one place.

```
class Outer
{
 void display()
{
System.out.println("This is the display() method of the outer class.");
}
class Inner
{
void display()
{
System.out.println("This is the display() method of the inner class.");
}
}
}
 class OuterInner
{
public static void main(String[] args)
{
Outer outer = new Outer();
//Calling the display() method of the outer class
outer.display();
//Creating an instance of the inner class and calling its display() method
Outer.Inner inner = outer.new Inner();
inner.display();
}
}
```

5 c) Explain dynamic method dispatch with a suitable example

Explanation (2)

Example (4)

Dynamic Method Dispatch is a mechanism by which a call to an **overridden method** is resolved at **runtime**, rather than at compile time.

- It is a part of **runtime polymorphism**
- It occurs when a **superclass reference variable** refers to a **subclass object**
- Achieved using **method overriding**

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

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

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

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

Animal a; // Superclass reference

a = new Dog(); // Refers to Dog object
a.sound(); // Calls Dog's sound()

a = new Cat(); // Refers to Cat object
a.sound(); // Calls Cat's sound()
}
}
```

Output:

```
Dog barks
Cat meows
```

6 a) **Explain inheritance. Write a java program to implement single level inheritance**

Definition (2)

Single level inheritance (1)

Program (4)

Inheritance is a mechanism in Java by which one class acquires the properties (variables) and behaviors (methods) of another class.

- It promotes **code reusability**.
- It is achieved using the `extends` keyword.
- The existing class is called the **superclass (parent class)**.
- The new class is called the **subclass (child class)**.

In **single level inheritance**, one subclass inherits from one superclass.

Structure:

```
Parent  →  Child
```

```
class Animal {
void eat() {
System.out.println("Animal is eating");
}
}

// Child Class
class Dog extends Animal {
void bark() {
System.out.println("Dog is barking");
}

public static void main(String[] args) {
Dog obj = new Dog();

obj.eat(); // Inherited method
obj.bark(); // Own method
}
}
```

Output:

```
Animal is eating
Dog is barking
```

6 b) **Explain the importance of the super keyword in inheritance, illustrate with a suitable example**
Explanation (2)
Uses (1)
Example (4)

The **`super` keyword** in Java is used to refer to the **immediate parent class object**.

Uses:

- Avoids ambiguity between parent and child members
- Enables constructor chaining
- Allows reuse of parent class functionality

- Essential in method overriding

It is mainly used in inheritance to:

- Access parent class variables
- Call parent class methods
- Invoke parent class constructor

It helps avoid confusion when both parent and child classes have members with the same name.

Accessing Parent Class Variable

When parent and child have the same variable name, `super` is used to access the parent's variable.

Example:

```
class Parent {
    int x = 10;
}

class Child extends Parent {
    int x = 20;

    void display() {
        System.out.println("Child x: " + x);
        System.out.println("Parent x: " + super.x);
    }

    public static void main(String[] args) {
        Child obj = new Child();
        obj.display();
    }
}
```

Output:

```
Child x: 20
Parent x: 10
```

Calling Parent Class Method

When a method is overridden, `super` can call the parent's version.

```
class Parent {
    void show() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("Child method");
    }

    void display() {
        super.show();   // Calls parent method
        show();         // Calls child method
    }

    public static void main(String[] args) {
        Child obj = new Child();
        obj.display();
    }
}
```

Output:

```
Parent method
Child method
```

Calling Parent Class Constructor

`super()` is used to call the parent class constructor.

```
class Parent {
    Parent() {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent {
    Child() {
        super();   // Calls parent constructor
        System.out.println("Child Constructor");
    }

    public static void main(String[] args) {
        Child obj = new Child();
    }
}
```

Output:

```
Parent Constructor
Child Constructor
```

6 c) **Define method overloading and overriding with example**

Definition (2)

Example (4)

Method Overloading is a feature in Java where **two or more methods have the same name but different parameter lists** (different number, type, or order of parameters) within the same class.

It is an example of **compile-time polymorphism**.
Method selection is done at compile time.

Example of Method Overloading

```
class Calculator {

    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }

    double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator obj = new Calculator();

        System.out.println(obj.add(5, 3));
        System.out.println(obj.add(5, 3, 2));
        System.out.println(obj.add(2.5, 3.5));
    }
}
```

Output

```
8
10
6.0
```

Method Overriding occurs when a **subclass provides a specific implementation of a method that is already defined in its superclass**.

☞ It is an example of **runtime polymorphism**.
☞ Method call is decided at runtime.

Example of Method Overriding

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

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

    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.sound();
    }
}
```

Output

```
Dog barks
```

Difference Between Overloading and Overriding

Method Overloading	Method Overriding
Same method name, different parameters	Same method name, same parameters
Within same class	In parent and child classes
Compile-time polymorphism	Runtime polymorphism
No inheritance required	Requires inheritance

7 a) Define package, with an example, explain the steps are involved in creating a user-defined package.

Definition (1)

steps are involved in creating a user-defined package (2)

Example (4)

A **package** in Java is a namespace that organizes related classes and interfaces into a single unit.
It helps to:

- Avoid name conflicts
- Provide access protection
- Make code reusable and modular
- Improve maintainability

In simple words, a package is like a **folder** that stores related Java classes.

- Package name should be in lowercase.
- A class must be declared public to access it outside the package.
- -d option is mandatory to create directory structure properly.

Steps Involved in Creating a User-Defined Package

1. Declare the package using `package package_name;` (must be first line).
2. Save the file with class name.
3. Compile using `javac -d . filename.java`
4. Package folder is created automatically.
5. Import the package using `import package_name.classname;`
6. Use the class by creating an object.
7. Compile and run the main program.

Example of a User-Defined Package:
Let us create a package named mypack that contains a class Message.

Step 1: Create the Package Class

Create a file named **Message.java**

```
package mypack;   // Declaring package

public class Message {
    public void display() {
        System.out.println("Welcome to User Defined Package");
    }
}
```

Step 2: Compile the Package

Open command prompt and compile using:

```
javac -d . Message.java
```

- `-d .` tells the compiler to create a directory structure.
- A folder named `mypack` will be created automatically.
- `Message.class` will be placed inside that folder.

Step 3: Use the Package in Another Class

Create another file named **TestPackage.java**

```
import mypack.Message;   // Importing the package class

class TestPackage {
    public static void main(String[] args) {
        Message obj = new Message();
        obj.display();
    }
}
```

Step 4: Compile the Main Class

```
javac TestPackage.java
```

Step 5: Run the Program

```
java TestPackage
```

Output:

```
Welcome to User Defined Package
```

7 b) **With sample code, explain chained exception**

Explanation of chained exception (2)

Example (5)

Chained Exception in Java

A **chained exception** is when one exception is caused by another exception. It allows you to wrap one exception inside another so that the original cause is not lost. This is useful in layered applications (like DAO → Service → UI), where you want to throw a higher-level exception but still keep the root cause.

Java supports chained exceptions using:

- `Throwable(Throwable cause)` constructor
- `Throwable(String message, Throwable cause)` constructor
- `initCause()` method
- `getCause()` method

Suppose a method throws `ArithmeticException`, but you want to throw a custom exception instead. You wrap the original exception inside the new one.

Example Program

```
class ChainedExample {

    static void divide() throws Exception {
        try {
            int a = 10 / 0;   // This causes ArithmeticException
        } catch (ArithmeticException e) {
            // Wrapping original exception inside a new exception
            throw new Exception("Error in divide method", e);
        }
    }

    public static void main(String[] args) {
        try {
            divide();
        } catch (Exception e) {
            System.out.println("Caught in main: " + e);
            System.out.println("Original cause: " + e.getCause());
        }
    }
}
```

Output

```
Caught in main: java.lang.Exception: Error in divide method
Original cause: java.lang.ArithmeticException: / by zero
```

7 c) **Define an exception, with syntax explain all five keywords used in exception handling**

Definition of Exception (1)

List the five keywords of exception handling (1)

Explanation of five keywords (2)

Example (2)

An **exception** is an unwanted or abnormal event that occurs during program execution and disrupts the normal flow of instructions.

In Java, exceptions are objects that occur at runtime and can be handled using exception-handling mechanisms.

Java provides **five keywords** to handle exceptions:

1. `try`
2. `catch`
3. `finally`
4. `throw`
5. `throws`

1. try Keyword

The `try` block contains the code that may cause an exception.

Syntax:

```
try {
    // code that may cause exception
}
```

Example:

```
try {
    int a = 10 / 0;
}
```

2. catch Keyword

The `catch` block handles the exception thrown in the try block.

Syntax:

```
catch(ExceptionType referenceName) {
    // handling code
}
```

Example:

```
try {
    int a = 10 / 0;
}
catch(ArithmeticException e) {
    System.out.println("Cannot divide by zero");
}
```

3. finally Keyword

The `finally` block always executes whether an exception occurs or not.
It is mainly used for cleanup operations (closing files, releasing resources).

Syntax:

```
finally {
    // cleanup code
}
```

Example:

```
try {
    int a = 10 / 2;
}
catch(ArithmeticException e) {
    System.out.println("Error");
}
finally {
    System.out.println("This always executes");
}
```

4. throw Keyword

The `throw` keyword is used to explicitly throw an exception.

Syntax:

```
throw new ExceptionType("message");
```

Example:

```
int age = 15;
if(age < 18) {
    throw new ArithmeticException("Not eligible to vote");
}
```

5. throws Keyword

The `throws` keyword is used in method declaration to declare exceptions that may occur.

Syntax:

```
returnType methodName() throws ExceptionType {
    // code
}
```

Example:

```
class Test {
    static void check() throws ArithmeticException {
        int a = 10 / 0;
    }

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

8 a) Explain the concept of package importing in java with example

Explanation of Importing Package (2)

Importance (2)

Example (3)

Package importing in Java means accessing classes that are defined in another package using the `import` keyword.

Since Java organizes classes inside packages (like folders), we must import a package if we want to use its classes in another program.

Why Import is Needed?

- To reuse existing classes
- To avoid writing fully qualified class names every time
- To improve readability

For example, the class `Scanner` belongs to the package `java.util`.
To use it, we must import it.

Types of Package Importing

There are **three ways** to import packages in Java:

Importing a Single Class

Syntax:

```
import package_name.class_name;
```

Example:

```
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a number:");
        int n = sc.nextInt();
        System.out.println("Number is: " + n);
    }
}
```

Here:

- `java.util` → package name
- `Scanner` → class name

Importing Entire Package

Syntax:

```
import package_name.*;
```

Example:

```
import java.util.*;

class Example {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        System.out.println(list);
    }
}
```

`*` means all classes inside `java.util` package are available.

Using Fully Qualified Name (Without Import)

Instead of importing, we can directly use the full path.

Example:

```
class Demo {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.println("Enter value:");
        int x = sc.nextInt();
        System.out.println(x);
    }
}
```

Here, no `import` statement is used.

Importing User-Defined Package

Step 1: Create Package Class

```
package mypack;

public class Hello {
    public void show() {
        System.out.println("Hello from package");
    }
}
```

Compile:

```
javac -d . Hello.java
```

Step 2: Import in Another Class

```
import mypack.Hello;

class TestPackage {
    public static void main(String[] args) {
        Hello obj = new Hello();
        obj.show();
    }
}
```

8 b) How do you create your own exception class. Explain with a program

Explanation (2)
Program (5)

In Java, you can create your **own exception class** by extending:

- `Exception` → for **checked exception**
- `RuntimeException` → for **unchecked exception**

A user-defined exception is useful when built-in exceptions do not describe your problem clearly.

Steps to Create a Custom Exception

1. Create a class that extends `Exception`
2. Define a constructor
3. Use `throw` to generate the exception
4. Handle it using `try-catch`

Example Program (Checked Exception)

Step 1: Create Custom Exception Class

```
class InvalidAgeException extends Exception {

    public InvalidAgeException(String message) {
        super(message);   // Passing message to parent class
    }
}
```

Step 2: Use the Custom Exception

```
class TestCustomException {

    static void checkAge(int age) throws InvalidAgeException {
        if(age < 18) {
            throw new InvalidAgeException("Age must be 18 or above");
        }
        else {
            System.out.println("Eligible to vote");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        }
        catch(InvalidAgeException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output

```
Exception caught: Age must be 18 or above
```

Example (Unchecked Exception)

If we extend `RuntimeException`, no need to use `throws`.

```
class InvalidAmountException extends RuntimeException {
    public InvalidAmountException(String message) {
        super(message);
    }
}
```

8 c) With an example, explain working of a nested try block within an exception
Explanation (2)
Example (4)

A **nested try block** means placing one `try-catch` block inside another `try` block.

It is used when:

- A part of code inside a `try` block may throw a different type of exception.
- We want to handle some exceptions locally and others at a higher level.

Syntax

```
try {
    // Outer try block

    try {
        // Inner try block
    }
    catch(ExceptionType1 e) {
        // Inner catch
    }

}
catch(ExceptionType2 e) {
    // Outer catch
}
```

Example Program

```
class NestedTryExample {

    public static void main(String[] args) {

        try {   // Outer try block

            int arr[] = {10, 20, 30};

            try {   // Inner try block
                int result = 10 / 0;   // ArithmeticException
                System.out.println(result);
            }
            catch (ArithmeticException e) {
                System.out.println("Inner catch: Cannot divide by zero");
            }

            // This statement executes after inner catch
            System.out.println(arr[5]);   // ArrayIndexOutOfBoundsException
        }

        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: Array index is invalid");
        }
```

```
        catch (Exception e) {
            System.out.println("Outer catch: General exception");
        }

        System.out.println("Program continues...");
    }
}
```

Output

```
Inner catch: Cannot divide by zero
Outer catch: Array index is invalid
Program continues...
```

9 a) Define Thread. With diagram explain the java thread model

Definition of Thread (1)
Java Thread Model (2)
Explanation (3)

A **Thread** is a lightweight sub-process that allows a program to perform multiple tasks concurrently.
In Java, a thread is an independent path of execution within a program.

- A process may contain multiple threads.
- Threads share the same memory space of the process.
- Multithreading improves CPU utilization and performance.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

Java Thread Model

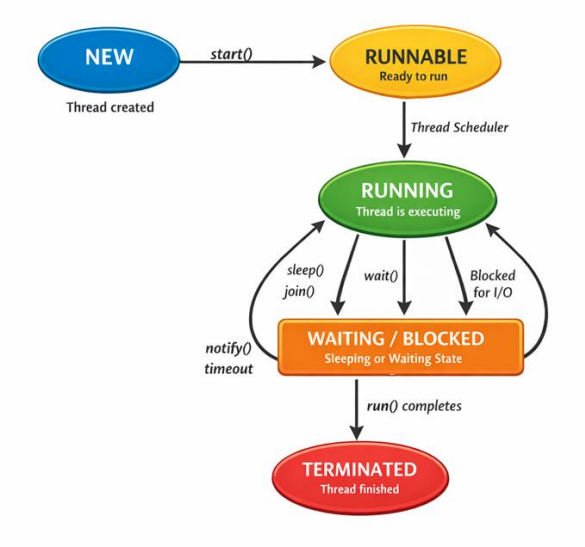
The **Java Thread Model** is based on multithreading and defines the life cycle and behavior of threads in a Java program.

Java supports:

- User threads
- Daemon threads
- Thread priorities
- Synchronization

Thread Life Cycle (Java Thread Model)

A thread moves through different states during execution.



Thread States in Java

1. **New**
2. **Runnable**
3. **Running**
4. **Blocked / Waiting**
5. **Terminated (Dead)**

New State

- Thread object is created.
- `start()` method not yet called.

```
Thread t = new Thread();
```

Runnable State

- After calling `start()`.
- Thread is ready to run.
- Waiting for CPU from thread scheduler.

```
t.start();
```

Running State

- Thread scheduler selects the thread.
- `run()` method is executing.

Blocked / Waiting State

Thread enters this state when:

- `sleep()` is called
- `wait()` is called
- `join()` is called
- Waiting for I/O
- Waiting for lock (synchronization)

After condition is satisfied, thread returns to **Runnable** state.

Terminated (Dead) State

- `run()` method completes.
- Thread execution ends.
- Cannot restart the thread.

9 b) Define Synchronization with an example, how synchronization is implemented in java

Definition of Synchronization (1)

Importance (1)

Ways to implement synchronization (1)

Example (4)

Synchronization is the process of controlling access of multiple threads to a shared resource so that only one thread can access it at a time.

It prevents:

- Data inconsistency
- Race condition
- Thread interference

In simple words, synchronization ensures **mutual exclusion** (only one thread enters critical section at a time).

Why Synchronization is Needed?

When multiple threads access the same data simultaneously, incorrect results may occur.

Example problem (without synchronization):

```
class Counter {
    int count = 0;

    void increment() {
        count++;
    }
}
```

If two threads execute `count++` at the same time, the final value may be incorrect.

How Synchronization is Implemented in Java

- Every object in Java has a monitor (lock).
- When a thread enters a synchronized method/block:
  - It acquires the object's lock.
- Other threads must wait until lock is released.

- Lock is released when:
  - Method/block completes
  - Exception occurs

Java provides synchronization using:

1. `synchronized` method
2. `synchronized` block
3. Static synchronization
4. Inter-thread communication (`wait()`, `notify()`, `notifyAll()`)

1. Synchronized Method

Entire method is locked.

Example:

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}

class TestSync {
    public static void main(String[] args) throws Exception {

        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for(int i = 0; i < 1000; i++)
                c.increment();
        });

        Thread t2 = new Thread(() -> {
            for(int i = 0; i < 1000; i++)
                c.increment();
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + c.count);
    }
}
```

Output:

```
Final Count: 2000
```

Here:

- Only one thread executes `increment()` at a time.
- Object-level lock is used.

2. Synchronized Block

Used to synchronize only a specific part of code.

Syntax:

```
synchronized(object_reference) {
    // critical section
}
```

Example:

```
class Counter {
    int count = 0;

    void increment() {
        synchronized(this) {
            count++;
        }
    }
}
```

More efficient than synchronizing entire method.

3. Static Synchronization

Locks the **class**, not object.

```
class Demo {
    static synchronized void display() {
        System.out.println("Static synchronized method");
    }
}
```

9 c) With suitable example, explain values() and valueOf() method in enumeration

Definition of Enumeration (1)

Explanation of values(), valueOf() (2)

Example (3)

An **enum** in Java is a special data type used to define a collection of constants.

Example:

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

Java automatically provides some built-in methods for every enum type, including:

- values()
- valueOf()

values() Method

Definition:

values() returns an array containing all the constants of the enum in the order they are declared.

Syntax:

```
EnumType[] arrayName = EnumType.values();
```

Example:

```
enum Day {
   MONDAY, TUESDAY, WEDNESDAY
}
```

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

      Day[] days = Day.values();

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

Output:
MONDAY
TUESDAY
WEDNESDAY

valueOf() Method

Definition:
valueOf(String name) returns the enum constant whose name matches the given string.
Syntax:
EnumType variable = EnumType.valueOf("CONSTANT_NAME");
Example:

```
enum Day {
   MONDAY, TUESDAY, WEDNESDAY
}

class TestEnum2 {
   public static void main(String[] args) {

      Day d = Day.valueOf("MONDAY");

      System.out.println(d);
   }
}
```

Output:
MONDAY

10 a) Define Multithreading, write a program to create multiple threads in java

Definition of Multithreading (2)

Ways to create (1)

Example (4)

Multithreading is a feature in Java that allows multiple threads to execute concurrently within a single program.

- A thread is a lightweight subprocess.
- Multithreading improves CPU utilization.
- Threads share the same memory space.

In simple words, multithreading allows a program to perform multiple tasks at the same time.

Ways to Create Multiple Threads in Java

Java provides two main ways:

1. By extending `Thread` class
2. By implementing `Runnable` interface

Example: Creating Multiple Threads (Using Thread Class)

```
class MyThread extends Thread {

    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + i);
        }
    }
}

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

        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.start();   // Starts Thread-0
        t2.start();   // Starts Thread-1
        t3.start();   // Starts Thread-2
    }
}
```

Possible Output (Order May Vary)

```
Thread-0 : 1
Thread-1 : 1
Thread-2 : 1
Thread-0 : 2
Thread-1 : 2
Thread-2 : 2
...
 ** output order may change because thread execution depends on the scheduler.
```

Example: Creating Multiple Threads (Using Runnable Interface)

```
class MyRunnable implements Runnable {

    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + i);
        }
    }
}

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

        MyRunnable obj = new MyRunnable();

        Thread t1 = new Thread(obj);
        Thread t2 = new Thread(obj);

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

10 b) Demonstrate the usage of compareTo() and equals() method with enumeration constants

Explanation (2)

Example (5)

In Java, every enum implicitly extends the Enum class.
So enum constants automatically get some built-in methods, including:

- compareTo()
- equals()

compareTo() Method

Definition:

compareTo() compares the **ordinal position** (index) of enum constants.

Syntax:

enumConstant1.compareTo(enumConstant2);

Return Value:

- 0 → if both constants are same
- Negative value → if first comes before second
- Positive value → if first comes after second

Example of compareTo()

```
enum Level {
   LOW, MEDIUM, HIGH
}

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

      Level l1 = Level.LOW;
      Level l2 = Level.HIGH;

      System.out.println("Compare LOW with HIGH: " + l1.compareTo(l2));
      System.out.println("Compare HIGH with LOW: " + l2.compareTo(l1));
      System.out.println("Compare MEDIUM with MEDIUM: " +
                  Level.MEDIUM.compareTo(Level.MEDIUM));
   }
}
```

Output:

Compare LOW with HIGH: -2
Compare HIGH with LOW: 2
Compare MEDIUM with MEDIUM: 0

Explanation:

Ordinal values:

- LOW → 0
- MEDIUM → 1
- HIGH → 2

compareTo() subtracts ordinal positions.

equals() Method

Definition:

equals() checks whether two enum constants refer to the **same constant**.

Syntax:

enumConstant1.equals(enumConstant2);

Return Value:

- true → if both are same constant
- false → otherwise

Example of equals()

```
enum Day {
   MONDAY, TUESDAY, WEDNESDAY
}

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

      Day d1 = Day.MONDAY;
      Day d2 = Day.MONDAY;
      Day d3 = Day.TUESDAY;

      System.out.println(d1.equals(d2));  // true
      System.out.println(d1.equals(d3));  // false
   }
}
```

Output:

true
false

10 c) Explain autoboxing / unboxing in expressions

Explanation (2)
Example (4)

Autoboxing and Unboxing in Java (in Expressions)

Java provides **wrapper classes** for primitive data types:

Primitive	Wrapper Class
int	Integer
double	Double
char	Character
boolean	Boolean
etc.	

Autoboxing

Definition:

Autoboxing is the automatic conversion of a primitive data type into its corresponding wrapper class object.

Example:

```
Integer obj = 10;   // int → Integer (Autoboxing)
```

Here:

- 10 is primitive int
- Automatically converted to Integer object

Internally:

```
Integer obj = Integer.valueOf(10);
```

Unboxing

Definition:

Unboxing is the automatic conversion of a wrapper class object into its corresponding primitive type.

Example:

```
Integer obj = 20;
int num = obj;   // Integer → int (Unboxing)
```

Internally:

int num = obj.intValue();

Autoboxing / Unboxing in Expressions

Autoboxing and unboxing happen automatically inside arithmetic expressions.

Example 1: Arithmetic Expression

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

      Integer a = 10;   // Autoboxing
      Integer b = 20;   // Autoboxing

      Integer c = a + b;  // Unboxing + Addition + Autoboxing

      System.out.println("Result: " + c);
   }
}
```

Explanation:

Step-by-step:

1. a and b are Integer objects.
2. For a + b:
    - a is unboxed → int
    - b is unboxed → int
3. Addition is performed.
4. Result (int) is autoboxed into Integer c.

Example 2: Comparison Expression

```
Integer x = 100;
int y = 100;

if(x == y) {
   System.out.println("Equal");
}
```

Here:

- x is unboxed to int
- Compared with y

Example 3: Using Collections

```
import java.util.ArrayList;

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

      ArrayList<Integer> list = new ArrayList<>();

      list.add(5);   // Autoboxing (int → Integer)

      int num = list.get(0);  // Unboxing (Integer → int)

      System.out.println(num);
   }
}
```

Collections store objects only, so autoboxing is very useful.