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

Internal Assessment Test 2 – January 2026

Sub:	Object Oriented Programming with JAVA					Sub Code:	BCS306A	Branch:	AIML/CSE AIML
Date:	**/01/26**	Duration:	90 min	Max Marks:	50	Sem/Sec:	III A, B & C		OBE

	Answer any FIVE FULL Questions	**MARKS**	**CO**	**RBT**
1a	Explain **method overriding and dynamic method dispatch** with example. **1. Method Overriding(3 marks)** Method overriding occurs when a **subclass provides a specific implementation of a method that is already defined in its superclass**. **Rules for Method Overriding:** • Method name and parameter list must be **exactly the same**. • The method must be **inherited** from the superclass. • Access level cannot be more restrictive than the superclass method. • Overriding happens only in **runtime polymorphism**. **2. Dynamic Method Dispatch(3 marks)** Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime, based on the object being referenced, not the reference type. In Java, this is achieved using a superclass reference pointing to a subclass object.	6	CO3	L2
1b	Explain **abstract classes and interfaces**. Compare both. **1. Abstract Class (2 marks)** An **abstract class** is a class that **cannot be instantiated** and may contain **abstract methods (without body)** as well as **concrete methods (with body)**. It is used when classes share **common behavior and state**. **Key Features** • Declared using the abstract keyword • Can have **abstract and non-abstract methods** • Can contain **instance variables** • Can have **constructors** • Supports **single inheritance** (extends one class) **2. Interface (2 marks)** An **interface** is a collection of **abstract methods and constants** that represents a **contract**. A class implementing an interface must provide implementations for all its methods. **Key Features** • Declared using the interface keyword • All methods are **public and abstract by default** • Variables are **public, static, and final** • No constructors • Supports **multiple inheritance**	4	CO3	L3

2a	Explain **multilevel inheritance** with a Java program. **Multilevel Inheritance in Java(3 marks)** **Multilevel inheritance** is a type of inheritance where a class is derived from another class, which itself is derived from another class. In simple terms: **Class A → Class B → Class C** Here, Class C inherits the properties and methods of **both** Class A and Class B. **Example(3 marks)**	6	CO3	L2
2b	Explain **final keyword** with respect to class, method, and variable. **1. final Variable(1)** A **final variable** is treated as a **constant**. Its value **cannot be changed once assigned**. **Key Points** • Must be initialized only once • Can be initialized at declaration, in constructor, or in initializer block • Prevents reassignment **2. final Method(1.5)** A **final method cannot be overridden** by subclasses.	4	CO3	L3

	It is used to prevent modification of method behavior in child classes. **Key Points** • Method implementation remains unchanged in subclasses • Supports method overloading but not overriding **3. final Class(1.5)** A **final class cannot be inherited**. It is used when a class should not be extended for security or design reasons. **Key Points** • Prevents inheritance • All methods are implicitly final			
3a	Explain **exception handling in Java** with suitable examples. **Exception Handling in Java(3 marks)** **Exception handling** in Java is a mechanism to **handle runtime errors** so that the normal flow of the program can be maintained. An **exception** is an event that occurs during program execution and disrupts normal execution. **1. Types of Exceptions** **a) Checked Exceptions** • Checked at **compile time** • Must be handled using try-catch or throws • Example: IOException, SQLException **b) Unchecked Exceptions** • Occur at **runtime** • Subclasses of RuntimeException • Example: ArithmeticException, NullPointerException **2. Exception Handling Keywords** **Keyword Purpose** try Contains code that may cause an exception catch Handles the exception finally Executes whether exception occurs or not throw Explicitly throws an exception throws Declares exceptions in method signature Example(2 marks)	5	CO4	L3

3b	Explain **user-defined exceptions** with a program. **User-Defined Exceptions in Java(3 marks)** A **user-defined (custom) exception** is an exception created by the programmer to handle **application-specific error conditions**. It is created by **extending the Exception class** (for checked exceptions) or **RuntimeException class** (for unchecked exceptions). **Steps to Create a User-Defined Exception** 1. Create a class that extends Exception or RuntimeException 2. Provide a constructor to pass the error message 3. Use throw to explicitly throw the exception 4. Handle it using try-catch Example (2 marks)	5	CO4	L3
4a	Explain **throw and throws** keywords with examples. **1. throw Keyword(2.5 marks)** The **throw keyword** is used to **explicitly throw a single exception** from a method or block of code. **Key Points** • Used inside a method or block • Can throw **one exception at a time** • Used with **exception objects** • Transfers control to the nearest catch block **2. throws Keyword(2.5 marks)** The **throws keyword** is used to **declare exceptions** that a method may pass to the calling method instead of handling them. **Key Points** • Used in **method signature** • Can declare **multiple exceptions** • Mainly used for **checked exceptions** • Responsibility of handling exception is passed to caller	5	CO4	L2

4b	Explain **chained exceptions** with example. A **chained exception** is an exception that is **caused by another exception**. Java provides this mechanism to **preserve the original cause** of an exception while throwing a new one. This helps in **better debugging and error tracing**. **Why Chained Exceptions Are Needed** • To wrap a low-level exception with a higher-level exception • To maintain the **root cause** of the error • Common in layered applications (DAO → Service → Controller) **How Chaining Works in Java** Java supports chained exceptions using: • **Constructor** that accepts a cause • **initCause()** method • **getCause()** method	5	CO4	L2

5a	What is thread? Explain the two ways to creating a thread in java with an example. A **thread** is the **smallest unit of execution** within a program. In Java, a thread represents a **separate path of execution**, allowing multiple tasks to run **concurrently** within a single program. Java supports **multithreading**, which improves: • Program responsiveness • CPU utilization • Performance in concurrent applications **Two Ways to Create a Thread in Java** Java provides **two main approaches** to create a thread: **1. By Extending the Thread Class** **Steps** 1. Create a class that extends Thread 2. Override the run() method 3. Create an object of the class 4. Call the start() method **2. By Implementing the Runnable Interface** **Steps** 1. Create a class that implements Runnable 2. Override the run() method 3. Pass the object to a Thread class constructor 4. Call the start() method	5	CO5	L3
5b	**Explain Thread Synchronization with suitable example.** **Thread Synchronization in Java** **Thread synchronization** is a mechanism used to **control access to shared resources** in a multithreaded environment. It ensures that **only one thread at a time** can access a critical section of code, thereby preventing **data inconsistency and race conditions**. **Why Synchronization Is Needed** When multiple threads access a **shared object** simultaneously: • Data may become inconsistent • Output may be unpredictable • Race conditions can occur **Types of Synchronization in Java** 1. **Synchronized Method** 2. **Synchronized Block** 3. **Static Synchronization** (class-level lock)	5	CO5	L3
6a	Explain **Enumerations in Java** including: values(), valueOf() **Enumerations (Enums) in Java** An **Enumeration (enum)** in Java is a **special data type** used to define a **fixed set of named constants**.	6	CO5	L3

	Enums improve **type safety, readability, and maintainability** compared to using constants like int or String.
	
	Defining an Enum
	enum Day {
	MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
	• Day is an enum
	• Each identifier is a **constant object** of type Day
	
	values() Method
	The **values() method** returns an **array of all enum constants** in the order they are declared.
	Example
	class ValuesExample {
	public static void main(String[] args) {
	for (Day d : Day.values()) {
	System.out.println(d);
	}
	}
	}
	valueOf() Method
	The **valueOf() method** returns the **enum constant** corresponding to the specified **string name**.
	Example
	class ValueOfExample {
	public static void main(String[] args) {
	Day d = Day.valueOf("FRIDAY");
	System.out.println(d);
	}
	}			

6b	Explain **Wrapper Classes and Autoboxing** with examples.
	1. Wrapper Classes
	Wrapper classes in Java provide a way to convert **primitive data types** into **objects**. This is required because Java collections and many APIs work only with **objects**, not primitives. class WrapperDemo {
	public static void main(String[] args) {
	int a = 10; // primitive
	Integer obj = Integer.valueOf(a); // wrapping
	
	System.out.println(obj);
	}
	}
	2. Autoboxing
	Autoboxing is the **automatic conversion** of a **primitive type into its corresponding wrapper object** by the Java compiler.
	class AutoboxingDemo {
	public static void main(String[] args) {
	int a = 20;
	Integer obj = a; // autoboxing
	
	System.out.println(obj);
	}
	}
		4	CO5	L3