Module 6 – Core Java

1. Introduction to Java

Question: 1

History of Java

Answer:

- Java is a high-level programming language developed by James Gosling and his team at Sun Microsystems in 1995.
- Initially, Java was created for embedded systems, but later it became popular for web, desktop, and mobile applications.
- In 2010, Sun Microsystems was acquired by Oracle Corporation, and now Java is maintained by Oracle.

Question: 2

Features of Java

Answer:

Java has many important features:

1. **Platform Independent**

   Java programs can run on any operating system like Windows, Linux, or Mac because Java uses JVM.

2. **Object-Oriented**

   Java is based on OOP concepts such as Class, Object, Inheritance, Polymorphism, Encapsulation, and Abstraction.

3. **Simple**

   Java is easy to learn and understand compared to languages like C++.

4. **Secure**

   Java provides security through JVM and does not allow direct memory access.

5. **Robust**

   Java handles errors using exception handling and has strong memory management.

6. **Multithreaded**

   Java supports running multiple tasks at the same time.

7. **Portable**

   Java programs can be transferred from one system to another without changes.

Question: 3

Understanding JVM, JRE, and JDK

Answer:

1. **JVM (Java Virtual Machine)**

   JVM executes the bytecode and makes Java platform independent.

2. **JRE (Java Runtime Environment)**

   JRE = JVM + Libraries

   It is used to run Java programs.

3. **JDK (Java Development Kit)**

   JDK = JRE + Development Tools (javac, debugger, etc.)

   It is used to develop and run Java programs.

- **Relation:**

JDK ⊃ JRE ⊃ JVM

Question: 4

Setting up the Java Environment and IDE

Answer:

- **Steps to install Java:**

  1. Download JDK from Oracle website.

  2. Install JDK on the system.

  3. Set PATH and JAVA_HOME environment variables.

  4. Verify installation using java --version.

- **IDE (Integrated Development Environment):**

  An IDE helps to write, compile, and run Java programs easily.

Common Java IDEs:

- Eclipse
- IntelliJ IDEA
- NetBeans

Benefits of IDE:

- Auto code suggestion
- Error checking
- Easy debugging
- Fast development

Question: 5

Java Program Structure

Answer:

A java program has a fixed structure.

- **A basic Java program structure includes:**

  1. Package – Used to group related classes.

  2. Class – Main building block of Java.

  3. Method – Contains program logic.

  4. Main Method – Program execution starts from here.

- **Example Java Program:**

```
package mypackage;

class HelloJava {

   public static void main(String[] args) {

      System.out.println("Hello Java");

   }

}
```

- **Explanation:**

  package mypackage; → Defines package name

  class HelloJava → Class name

  main() → Entry point of program

  System.out.println() → Prints output

2. Data Types, Variables, and Operators (Core Java)

Question: 1

Primitive Data Types in Java

Answer:

➢ **Meaning** - Data type defines what type of data a variable can store.

Java has 8 primitive data types.

Data Type	Size	Description	Example
byte	1 byte	Small integer	byte b = 10;
short	2 bytes	Small integer	short s = 100;
int	4 bytes	Integer numbers	int a = 25;
long	8 bytes	Large integer	long l = 50000L;
float	4 bytes	Decimal (single precision)	float f = 10.5f;
double	8 bytes	Decimal (double precision)	double d = 99.99;
char	2 bytes	Single character	char c = 'A';
boolean	1 bit	True or False	boolean flag = true;

➢ **Points:**

- Primitive data types store single values
- They are faster and use less memory

Question: 2

Variable Declaration and Initialization

Answer:

- **Variable**

  A variable is a container used to store data.

- **Declaration**

  Telling Java the data type and variable name.

  Example - int age;

- **Initialization**

  Assigning value to the variable.

  Example - age = 20;

- **Declaration + Initialization together**

  Example - int age = 20;

- **Example:**

  int marks = 85;

  float percentage = 75.5f;

  char grade = 'A';

  boolean pass = true;

- **Rules for Variables:**
  - Variable name must start with a letter, _, or $
  - Cannot use Java keywords
  - Case-sensitive (Age and age are different)

Question: 3

Operators in Java

Answer:

Operators are used to perform operations on variables.

1. **Arithmetic Operators**

   Used for mathematical calculations.

Operator	**Meaning**	**Example**
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

Example:

int a = 10, b = 3;

System.out.println(a + b); // 13

2. **Relational Operators**

   Used to compare values and return true or false.

Operator	**Meaning**
==	Equal to
!=	Not equal
>	Greater than

<	Less than
>=	Greater than or equal
<=	Less than or equal

Example:

```
System.out.println(a > b);  // true
```

3. **Logical Operators**

Used with boolean values.

Operator	**Meaning**
&&	Logical AND
||	Logical OR
!	Logical NOT

Example:

```
boolean x = true, y = false;

System.out.println(x && y);  // false
```

4. **Assignment Operators**

Used to assign values.

Operator	**Example**
=	a = 5
+=	a += 2
-=	a -= 2
*=	a *= 2
/=	a /= 2

Example:

```
int a = 5;

a += 3;   // a = 8
```

5. **Unary Operators**

Used with single operand.

Operator	**Meaning**
+	Unary plus
-	Unary minus
++	Increment
--	Decrement
!	Logical NOT

Example:

```
int a = 10;

a++;

System.out.println(a);  // 11
```

6. **Bitwise Operators**

   Used to perform operations at bit level.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift

Example:

```
int a = 5;   // 0101

int b = 3;   // 0011

System.out.println(a & b); // 1
```

Question: 4

Type Conversion and Type Casting

Answer:

1. **Type Conversion (Implicit / Widening)**

- Automatic conversion from smaller to larger data type.
- **Example:**

```
int a = 10;

double d = a;
```

- No data loss

2. **Type Casting (Explicit / Narrowing)**

- Manual conversion from larger to smaller data type.
- **Example:**

```
double d = 10.5;

int a = (int)d;
```

- Data loss possible

- **Example:**

```
int x = 5;
double y = x;       // Implicit
double a = 9.7;
int b = (int)a;     // Explicit
```

3. Control Flow Statements

Question: 1

If-Else Statements

Answer:

- **If-Else Statements:**
  - If-Else statement is used to make decisions based on a condition.
  - If the condition is true, one block executes; if false, another block executes.

a) Simple If

```
int age = 20;
if (age >= 18) {
   System.out.println("Eligible for voting");
}
```

b) If-Else

```
int marks = 30;
if (marks >= 35) {
   System.out.println("Pass");
} else {
   System.out.println("Fail");
}
```

c) Else-If Ladder

Used when multiple conditions are checked.

```
int marks = 80;
if (marks >= 90) {
   System.out.println("Grade A");
} else if (marks >= 75) {
   System.out.println("Grade B");
} else {
   System.out.println("Grade C");
}
```

Question: 2

Switch Case Statements

Answer:

Switch case is used when multiple choices depend on one variable.

- **Syntax:**

```
switch (expression) {
   case value1:
      statements;
      break;
   case value2:
      statements;
      break;
   default:
      statements;
}
```

- **Example:**

```
int day = 2;
switch (day) {
   case 1:
      System.out.println(“Monday”);
      break;
   case 2:
      System.out.println(“Tuesday”);
      break;
   default:
      System.out.println(“Invalid day”);
}
```

- **Points:**
  - break stops execution
  - default runs if no case matches
  - Faster than multiple if-else

Question: 3

Loops(for, while, do-while)

Answer:

Loops are used to repeat statements.

a) **For Loop**

Used when the number of iterations is known.

Example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println(i);
}
```

b) **While Loop**

Used when the number of iterations is not known.

Example:

```
int i = 1;
while (i <= 5) {
   System.out.println(i);
   i++;
}
```

c) **Do-While Loop**

Executes at least once, even if condition is false.

Example:

```
int i = 1;
do {
   System.out.println(i);
   i++;
} while (i <= 5);
```

Question: 4

Break and Continue Keywords

Answer:

a) Break

- Used to stop the loop or switch immediately.
- **Example:**

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) {
      break;
   }
   System.out.println(i);
}
```

Output: 1 2

b) Continue

- Used to skip current iteration and continue with next loop cycle.
- **Example:**

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) {
      continue;
   }
   System.out.println(i);
}
```

Output: 1 2 4 5

----

4. Classes and Objects

Question: 1

Defining a Class and Object in Java.

Answer:

- **Class**
  - A class is a blueprint or template used to create objects.
  - It contains:

    Variables (data members)

    Methods (functions)

  - A class does not occupy memory until an object is created.
  - **Example:**

```
class Student {
    int id;
    String name;

    void display() {
        System.out.println(id + " " + name);
    }
}
```

- **Object**
  - An object is an instance of a class.
  - It represents real-world entities and occupies memory.
  - **Example:**

```
Student s1 = new Student();
```

Question: 2

Constructors and Overloading

Answer:

- **Constructor**

  A constructor is a special method used to initialize objects.

- **Characteristics:**
    - Same name as class
    - No return type
    - Automatically called when object is created

a) **Default Constructor**

```
class Student {
    int id;
    Student() {
        id = 0;
    }
}
```

b) **Parameterized Constructor**

```
class Student {
    int id;
    Student(int i) {
        id = i;
    }
}
```

- **Constructor Overloading**

  When a class has more than one constructor with different parameters, it is called constructor overloading.

- **Example:**

```
class Student {
    int id;
    String name;
    Student() {
        id = 0;
        name = "Unknown";
    }
    Student(int i, String n) {
        id = i;
        name = n;
    }
}
```

Question: 3

Object Creation and Accessing Members of the Class.

Answer:

- **Object Creation**

  Objects are created using the new keyword.

  Example:

  ```
  Student s1 = new Student();
  ```

- **Accessing Data Members and Methods**

  Members of a class are accessed using dot (.) operator.

  Example:

  ```
  s1.id = 1;
  s1.name = "Rahul";
  s1.display();
  ```

Question: 4

this Keyword.

Answer:

- **this Keyword**

  Meaning: The this keyword refers to the current object of the class.

- **Uses of this keyword:**

  1) **To differentiate instance variables and parameters**

     ```
     class Student {
     ```

```
    int id;
    Student(int id) {
        this.id = id;
    }
}
```

2) To call current class method

```
class Demo {
    void show() {
        System.out.println("Show method");
    }
    void display() {
        this.show();
    }
}
```

3) To call current class constructor

```
class Test {
    Test() {
        System.out.println("Default Constructor");
    }
    Test(int a) {
        this();
        System.out.println("Parameterized Constructor");
    }
}
```

- **Advantages of this Keyword**
    - Avoids confusion between variables
    - Improves code readability
    - Helps in constructor chaining

---

5. Methods in Java

Question: 1

Defining Methods in Java

Answer:

- **Method**

  A method is a block of code that performs a specific task.

  Methods help in:

  - Code reusability
  - Easy maintenance
  - Better readability

- **Syntax:**

```
returnType methodName(parameters) {

   // method body

}
```

- **Example:**

```
class Demo {

   void show() {

      System.out.println("Hello Java");

   }

}
```

Question: 2

Method Parameters and Return Types

Answer:

- **Method Parameters**

  Parameters are values passed to a method to perform an operation.

  - **Example:**

```
class Demo {

   void add(int a, int b) {

      System.out.println(a + b);
```

```
    }
}
```

Here a and b are parameters.

- **Return Type**

  The return type specifies what type of value a method returns.

  - **Example:**

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

  - **Types of Return:**

    1. void – returns nothing

    2. Primitive type – int, float, etc.

    3. Non-primitive type – object, array, string

  - **Method Call**

    ```
    Demo d = new Demo();
    int result = d.add(5, 3);
    ```

Question: 3

Method Overloading

Answer:

- **Method Overloading**

  Method overloading means having multiple methods with the same name but different parameters.

  - **Rules:**

  - Method name must be same
  - Parameters must be different (number, type, or order)
  - Return type alone cannot overload a method

- **Example:**

```
class MathOperation {

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

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

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

- **Advantages of Method Overloading**
  - Improves code readability
  - Saves memory
  - Same operation, different inputs

Question: 4

Static Methods and Variables

Answer:

- **Static Variable**

  A static variable is shared among all objects of a class.

  Memory is allocated only once.

  - **Example:**

    ```
    class Student {
      int id;
      static String college = "ABC College";
    }
    ```

- **Static Method**

  A static method belongs to the class, not to objects.

  - **Example:**

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

  - **Calling Static Method:**

    ```
    Demo.display();
    ```

- **Rules of Static Members**
  - Static methods can access only static variables
  - Cannot use this keyword
  - Called using class name

---

6. Object – Oriented Programming (OOPs) Concepts

Question: 1

Basics of OOP Concepts

Answer:

1) Encapsulation

- Encapsulation means wrapping data and function together into a single unit (class).
- It also means data hiding using access modifiers like private.
- **Example:**

```
class Student {
   private int marks;

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

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

- **Advantages:**
  - Protects data
  - Improves security
  - Better control over data

2) Inheritance

- Inheritance allows one class to acquire properties and methods of another class.
- It uses the extends keyword.
- **Example:**

```
class Animal {
   void eat() {
      System.out.println("Eating");
```

```
  }
}

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

- **Advantages:**
  - Code reusability
  - Method overriding
  - Reduces redundancy

3) Polymorphism

- Polymorphism means one name, many forms.
- Same method behaves differently in different situations.
- **Types of Polymorphism:**

  1. Compile-time (Method Overloading)
  2. Runtime (Method Overriding)

- **Example:**

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

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

4) Abstraction

- Abstraction means hiding implementation details and showing only functionality.
- **Achieved using:**
  - Abstract class
  - Interface
- **Example:**

```
abstract class Vehicle {
  abstract void start();
}


class Bike extends Vehicle {
  void start() {
    System.out.println("Bike starts");
  }
}
```

- **Advantages:**
  - Reduces complexity
  - Improves security
  - Enhances flexibility

Question: 2

Types of Inheritance

Answer:

1) Single Inheritance

- One child class inherits from one parent class.
- **Example:**

```
class A {
  void show() {}
}
```

```
class B extends A {
}
```

2) Multilevel Inheritance

- Inheritance chain of more than two classes.
- **Example:**

```
class A {

   void showA() {}
}


class B extends A {
   void showB() {}
}


class C extends B {
   void showC() {}
}
```

3) Hierarchical Inheritance

- Multiple child classes inherit from one parent class.
- **Example:**

```
class A {

   void display() {}
}


class B extends A {
}


class C extends A {
}
```

Question: 3

Method Overriding and Dynamic Method Dispatch

Answer:

- **Method Overriding**
  - Method Overriding occurs when a child class provides its own implementation of a parent class method.
  - **Rules:**
    - Same method name
    - Same parameters
    - IS-A relationship (inheritance required)
  - **Example:**

```
class Parent {

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


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

- **Dynamic Method Dispatch**
  - Dynamic Method Dispatch is a process in which method call is resolved at runtime, not at compile time.
  - It is achieved using method overriding and parent class reference.
  - **Example:**

```
class Parent {

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

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

public class Test  {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show();
    }
}
```

- **Output:** Child class
- **Explanation:**
  - Parent reference refers to Child object
  - JVM decides method at runtime

----

7. Constructors and Destructors

Question: 2

Copy Constructor (Emulated in Java)

Answer:

- **What is a Copy Constructor?**

  A copy constructor is a constructor that creates a new object by copying data from an existing object.

- **Important:**

  Java does not support copy constructor directly like C++, but we can emulate (create manually) it by passing an object as a parameter to a constructor.

- **Why Copy Constructor is Needed?**
  - To create a duplicate object
  - To copy object data safely
  - To avoid reference sharing
- **How Copy Constructor is Emulated in Java**
  - **Example:**

```
class Student {
    int id;
    String name;

    // Parameterized constructor
    Student(int i, String n) {
        id = i;
        name = n;
    }

    // Copy constructor (emulated)
    Student(Student s) {
        id = s.id;
        name = s.name;
    }
}
```

Object Creation:

```
Student s1 = new Student(1, "Amit");

Student s2 = new Student(s1);
```

- **Explanation:**
  - s1 is the original object
  - s2 is the copied object
  - Both objects have same data but different memory locations

---

Question: 4

Object Life Cycle and Garbage Collection

Answer:

❖ **Object Life Cycle in Java**

The object life cycle describes the stages through which an object passes in a Java program.

1) **Object Creation**
   - Object is created using the new keyword.
   - **Example:**

     ```
     Student s = new Student();
     ```

2) **Object Usage**
   - The object is used to access variables and methods.
   - **Example:**

     ```
     s.id = 10;
     ```

3) **Object Becomes Unreachable**

   An object becomes eligible for garbage collection when:

   - Reference is set to null
   - Reference is assigned to another object
   - Object goes out of scope

- **Example:**

  s = null;

4) Garbage Collection

Garbage Collection is an automatic process where JVM removes unused objects from memory.

- **Features:**
  - Automatic memory management
  - Controlled by JVM
  - Improves performance
  - **Example:**

    System.gc(); // Request JVM to run garbage collector

    (Only a request, not a guaranteed**)**

➢ **Advantages of Garbage Collection**

- Prevents memory leaks
- Automatic memory management
- No need to delete objects manually

----

8. Arrays and Strings

Question: 1

One-Dimensional and Multidimensional Arrays

Answer:

❖ **One-Dimensional Array**

A one-dimensional array stores multiple values of the same data type in a single variable.

- **Syntax:**

```
dataType[] arrayName = new dataType[size];
```

- **Example:**

```
int marks[] = new int[3];
marks[0] = 70;
marks[1] = 80;
marks[2] = 90;
```

- **Using for loop:**

```
for (int i = 0; i < marks.length; i++) {
   System.out.println(marks[i]);
}
```

❖ **Multidimensional Array**

A multidimensional array stores data in rows and columns (matrix form).

- **Syntax:**

```
dataType[][] arrayName = new dataType[rows][columns];
```

- **Example:**

```
int a[][] = {
   {1, 2, 3},
   {4, 5, 6}
};
```

- **Accessing elements:**

```
System.out.println(a[1][2]); // Output: 6
```

Question: 2

String Handling in Java

Answer:

Java provides three classes for string handling:

1. String
2. StringBuffer
3. StringBuilder

a) String Class

- String is immutable (cannot be changed)
- Stored in String Constant Pool
- **Example:**

  String s = "Java";

  s = s.concat(" Programming");

  ➞ A new object is created, original string remains unchanged.

b) StringBuffer

- Mutable (can be changed)
- Thread-safe (synchronized)
- Slower than StringBuilder
- **Example:**

  StringBuffer sb = new StringBuffer("Java");

  sb.append(" Programming");

c) StringBuilder

- Mutable
- Not thread-safe
- Faster than StringBuffer
- **Example:**

  StringBuilder sb = new StringBuilder("Java");

  sb.append(" Programming");

- **Difference Between String, StringBuffer, StringBuilder**

Feature	String	StringBuffer	StringBuilder
Mutable	No	Yes	Yes
Thread-safe	Yes	Yes	No
Performance	Slow	Medium	Fast

Question: 3

Array of Objects

Answer:

- **Array of Objects**

An array of objects stores multiple objects of the same class.

- **Example:**

```
class Student {
    int id;
    String name;

    Student(int i, String n) {
        id = i;
        name = n;
    }

    void display() {
        System.out.println(id + " " + name);
    }
}
```

- **Creating Array of Objects:**

```
Student s[] = new Student[2];

s[0] = new Student(1, "Amit");
s[1] = new Student(2, "Neha");
```

```
s[0].display();
s[1].display();
```

Question: 4

String Methods in Java

Answer:

- **Common String Methods:**

1. **length()**
   - Returns length of string.
   - **Example:**

     ```
     String s = "Java";
     System.out.println(s.length()); // 4
     ```

2. **charAt()**
   - Returns character at given index.
   - **Example:**

     ```
     System.out.println(s.charAt(1)); // a
     ```

3. **substring()**
   - Returns part of the string.
   - **Example:**

     ```
     System.out.println(s.substring(1, 3)); // av
     ```

4. **toUpperCase()**
   - Returns a new string with all characters converted to uppercase.
   - **Example:**

     ```
     System.out.println(s.toUpperCase()); // JAVA
     ```

5. **toLowerCase()**
   - Returns a new string with all characters converted to low ercase.
   - **Example:**

     ```
     System.out.println(s.toLowerCase()); // java
     ```

6. **equals()**
   - Compares content of strings.
   - **Example:**

```
String a = "Java";
String b = "Java";
System.out.println(a.equals(b)); // true
```

7. **compareTo()**
   - Compares two strings lexicographically.
   - **Example:**

```
System.out.println(a.compareTo(b)); // 0
```

----

9. Inheritance and Polymorphism

Question: 1

Inheritance Types and Benefits

Answer:

❖ **Inheritance**

- Inheritance is an OOP concept where one class (child/subclass) acquires properties and methods of another class (parent/superclass).
- It uses the extends keyword.
- **Example:**

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

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

➢ **Types of Inheritance in Java**

a) Single Inheritance

○ One child class inherits from one parent class.

○ **Example:**

```
class A {}
class B extends A {}
```

b) Multilevel Inheritance

○ A class inherits from another class, which itself inherits from another class.

○ **Example:**

```
class A {}
class B extends A {}
class C extends B {}
```

c) Hierarchical Inheritance

- Multiple child classes inherit from a single parent class.
- **Example:**

```
class A {}
class B extends A {}
class C extends A {}
```

> Java does not support multiple inheritance using classes (to avoid ambiguity).

- **Benefits of Inheritance**
  - Code reusability
  - Reduces redundancy
  - Easy maintenance
  - Supports method overriding

Question: 2

Method Overriding

Answer:

- **Method Overriding**

  Method overriding occurs when a subclass provides its own implementation of a method already defined in the parent class.

- **Rules:**
  - Same method name
  - Same parameters
  - Inheritance must exist
- **Example:**

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

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

Question: 3

Dynamic Binding (Run-Time Polymorphism)

Answer:

❖ **Dynamic Binding**

- Dynamic binding means the method call is resolved at runtime, not at compile time.
- It is achieved using method overriding and parent class reference.

➢ **Example:**

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

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

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.display();
    }
}
```

➢ **Output:**

```
Child display
```

- **Explanation:**
    - Reference type is Parent
    - Object type is Child
    - JVM decides method at runtime

Question: 4

Super Keyword and Method Hiding

Answer:

- **Super Keyword**

  The super keyword refers to the parent class object.

- **Uses of** super**:**

  a) Access parent class variable

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

class Child extends Parent {
  int a = 20;
  void show() {
    System.out.println(super.a);
  }
}
```

  b) Call parent class method

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

class Child extends Parent {
  void show() {
    super.show();
```

```
        System.out.println("Child method");
    }
}
```

c) **Call parent class constructor**

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

class Child extends Parent {
    Child() {
        super();
        System.out.println("Child constructor");
    }
}
```

- **Method Hiding**

Method hiding occurs when a static method in the child class has the same name and signature as a static method in the parent class.

- **Example:**

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

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

- **Important Points:**
    - Applies only to static methods
    - Resolved at compile time
    - Not runtime polymorphism

----

10. Interfaces and Abstract Classes

Question: 1

Abstract Classes and Methods

Answer:

❖ **Abstract Class**

- An abstract class is a class that cannot be instantiated (object cannot be created).
- It is used to hide implementation details and provide a base structure for subclasses.

➢ **Key Points:**

- Declared using abstract keyword
- Can have abstract and non-abstract methods
- Can have variables and constructors

➢ **Example:**

```
abstract class Vehicle {

   abstract void start();

   void fuel() {

      System.out.println("Petrol or Diesel");

   }

}
```

➢ **Abstract Method**

An abstract method has no body and must be implemented by the subclass.

Syntax:

```
abstract void start();
```

➢ **Example:**

```
class Bike extends Vehicle {

   void start() {

      System.out.println("Bike starts");

   }

}
```

Question: 2

Interfaces: Multiple Inheritance in Java

Answer:

- **Interface**
  - An interface is a collection of abstract methods (and constants).
  - It provides 100% abstraction (traditional Core Java concept).
- **Key Points:**
  - Declared using interface keyword
  - Methods are public and abstract by default
  - Variables are public, static, final
  - Object of interface cannot be created
- **Example:**

```
interface Animal {
   void sound();
}
```

- **Multiple Inheritance in Java**
  - Java does not support multiple inheritance using classes, but it supports multiple inheritance using interfaces.
  - A class can implement multiple interfaces.

Question: 3

Implementing Multiple Interfaces

Answer:

- **Example:**

```
interface Printable {
    void print();
}

interface Showable {
    void show();
}

class Demo implements Printable, Showable {
    public void print() {
        System.out.println("Printing");
    }

    public void show() {
        System.out.println("Showing");
    }
}
```

- **Usage:**

```
Demo d = new Demo();
d.print();
d.show();
```

- **Difference Between Abstract Class and Interface**

Abstract Class	**Interface**
Can have abstract & non-abstract methods	Only abstract methods
Uses extends	Uses implements
Supports single inheritance	Supports multiple inheritance
Can have constructor	No constructor

- **Advantages of Interfaces**
  - Supports multiple inheritance
  - Achieves abstraction
  - Loose coupling
  - Better design

----

11. Packages and Access Modifiers

Question: 1

Java Packages: Built – in and User – Defined Packages

Answer:

- **What is a Package?**

  A package is a collection of related classes and interfaces.

  It helps to:

  - Organize large programs
  - Avoid class name conflicts
  - Improve code reusability

- **Types of Packages in Java**

  a) Built-in Packages

  Built-in packages are provided by Java API.

  - **Examples:**
    - java.lang – basic classes (String, Math)
    - java.util – utility classes (Scanner, ArrayList)
    - java.io – input/output classes
    - java.sql – database connectivity
    - java.lang is imported automatically.

  b) User-Defined Packages

  Packages created by the programmer.

  - **Example:**

```
package mypackage;

public class Demo {

   public void show() {

      System.out.println("User-defined package");

   }

}
```

Question: 2

Access Modifiers: Private, Default, Protected, Public

Answer:

- **Access Modifiers:**
  - Access modifiers define where a class, method, or variable can be accessed.
  - Java has four access modifiers:

a) **Private**

- Accessible only within the same class
- Most restrictive
- **Example:**

```
class Demo {

   private int a = 10;

}
```

b) **Default (No keyword)**

- Accessible within the same package
- Also called package-private
- **Example:**

```
class Demo {

   int a = 10;

}
```

c) **Protected**

- Accessible within the same package
- Also accessible in subclasses outside the package
- **Example:**

```
class Demo {

   protected int a = 10;

}
```

d) **Public**

- Accessible from anywhere
- Least restrictive
- **Example:**

```
public class Demo {

   public int a = 10;

}
```

Question: 3

Importing Packages and ClassPath

Answer:

❖ **Importing Packages**

The import statement is used to access classes defined in other packages.

➢ **Example:**

```
import java.util.Scanner;
```

Or import all classes:

```
import java.util.*;
```

❖ **Classpath**

Classpath is the location where Java looks for .class files and packages.

- Set using environment variables
- Helps JVM find user-defined classes
- **Example** (concept):

  ```
  CLASSPATH = C:\myclasses;
  ```

---

12. Exception Handling

Question: 1

Types of Exceptions in Java

Answer:

Java exceptions are mainly divided into two types:

a) Checked Exceptions

- Checked at compile time
- Programmer must handle or declare them
- Occur due to external factors

➢ **Examples:**

- IOException
- SQLException
- ClassNotFoundException

➢ **Example:**

```
try {
   FileReader fr = new FileReader("abc.txt");
} catch (IOException e) {
   System.out.println(e);
}
```

b) Unchecked Exceptions

- Checked at runtime
- Caused by programming errors
- Not compulsory to handle

➢ **Examples:**

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException

➢ **Example:**

```
int a = 10 / 0;  // ArithmeticException
```

- **Difference Between Checked and Unchecked Exceptions**

Checked Exception	**Unchecked Exception**
Compile-time	Runtime
Must be handled	Not compulsory
External causes	Programming errors

Question: 2

Exception Handling Keywords

Answer:

a) try

- The try block contains code that may cause an exception.

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

b) catch

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

```
catch (ArithmeticException e) {
   System.out.println("Error occurred");
}
```

c) finally

- The finally block always executes, whether an exception occurs or not.
- **Used to close resources.**

```
finally {
   System.out.println("Always executed");
}
```

- **Example of try-catch-finally:**

```
     try {
        int a = 10 / 2;
     } catch (Exception e) {
        System.out.println("Exception handled");
} finally {
   System.out.println("Program ended");
}
```

d) throw

- The throw keyword is used to explicitly throw an exception.
- throw new ArithmeticException("Invalid operation");

e) throws

- The throws keyword is used to declare exceptions in method signature.

```
void readFile() throws IOException {
FileReader fr = new FileReader("abc.txt");
}
```

- **Difference Between throw and throws**

throw	throws
Used to throw exception	Used to declare exception
Inside method	In method signature
One exception	Multiple exceptions

Question: 3

Custom Exception Classes

Answer:

- **Custom Exception**

  A custom exception is a user-defined exception created by extending the Exception class.

- **Why Custom Exception?**
  - To handle application-specific errors
  - To improve readability
- **Example:**

```
class InvalidAgeException extends Exception {
   InvalidAgeException(String msg) {
      super(msg);
   }
}
```

- **Using Custom Exception:**

```
class Test {
    static void validate(int age) throws InvalidAgeException {
        if (age < 18)
            throw new InvalidAgeException("Age not valid");
    }

    public static void main(String[] args) {
        try {
            validate(16);
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

----

13. Multithreading

Question: 1

Introduction to Threads

Answer:

- **What is a Thread?**
    - A thread is a small unit of execution within a program.
    - Java allows multiple threads to run at the same time, which is called multithreading.
- **Benefits of Multithreading:**
    - Better CPU utilization
    - Faster program execution
    - Improves performance
    - Useful in games, animations, and server applications

Question: 2

Creating Threads by Extending Thread Class or Implementing Runnable Interface

Answer:

Java provides two ways to create a thread:

a) By Extending Thread Class

- **Steps:**

  1. Extend Thread class

  2. Override run() method

  3. Call start() method

- **Example:**

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

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

b) By Implementing Runnable Interface

- **Steps:**

  1. Implement Runnable interface

  2. Override run() method

  3. Pass object to Thread class

  4. Call start() method

- **Example:**

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running");
    }
}


class Test {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

  - Runnable is preferred because Java supports single inheritance.

Question: 3

Thread Life Cycle

Answer:

A thread passes through different states during execution.

➢ **Thread States:**

1. New – Thread is created
2. Runnable – Ready to run
3. Running – Thread is executing
4. Waiting / Blocked – Waiting for resource
5. Terminated (Dead) – Execution finished

➢ **Diagram (Conceptual):**

```
New → Runnable → Running → Dead
          ↓
       Waiting
```

Question: 4

Synchronization and Inter – thread Communication

Answer:

❖ **What is Synchronization?**

Synchronization is the process of controlling access to shared resources by multiple threads to avoid data inconsistency.

➢ **Why Synchronization?**

- Prevents race condition
- Maintains data integrity

➢ **Example:**

```
class Table {
   synchronized void print(int n) {
      for (int i = 1; i <= 5; i++) {
         System.out.println(n * i);
      }
   }
}
```

- **Inter-Thread Communication**

- **What is Inter-Thread Communication?**

  It allows threads to communicate and cooperate with each other.

- **Methods Used:**
  - **wait() –** Makes thread wait
  - **notify() –** Wakes one waiting thread
  - **notifyAll() –** Wakes all waiting threads
  - **Important Note –** These methods belong to **Object class**, not Thread class.

- **Example:**

```
class Data {
    synchronized void produce() throws InterruptedException {
        System.out.println("Producing");
        wait();
        System.out.println("Resumed");
    }

    synchronized void consume() {
        System.out.println("Consuming");
        notify();
    }
}
```

----

14. File Handling

Question: 1

Introduction to File I/O in Java (java.io package)

Answer:

File I/O (Input/Output) in Java allows a program to read data from a file and write data to a file.

Java provides the java.io package which contains classes for handling files, streams, and serialization.

- **File I/O is mainly used for:**
  - Storing data permanently
  - Reading configuration files
  - Logging data
  - Saving and loading objects
- **Java supports two types of streams:**
  - Byte streams (for binary data like images, audio)
  - Character streams (for text data)

Question: 2

FileReader and FileWriter Classes

Answer:

1) FileReader

- Used to read character data from a text file.
- Reads one character at a time.
- Suitable for reading plain text files.

➢ **Key points:**

- Works with characters (Unicode)
- Part of character stream
- Throws IOException

➢ **Example:**

```
FileReader fr = new FileReader("data.txt");

int ch;
```

```
while ((ch = fr.read()) != -1) {
   System.out.print((char) ch);
}
fr.close();
```

2) FileWriter

- Used to write character data into a text file.
- Creates the file if it does not exist.
- Can overwrite or append data.

➢ **Example:**

```
FileWriter fw = new FileWriter("data.txt");

fw.write("Hello Java");

fw.close();
```

➢ **Append mode:**

```
FileWriter fw = new FileWriter("data.txt", true);
```

Question: 3

BufferedReader and BufferedWriter

Answer:

These classes improve performance by using a buffer (temporary memory).

1) BufferedReader

- Reads text efficiently using buffering.
- Can read data line by line using readLine() method.
- Used with FileReader.

➢ **Advantages:**

- Faster than FileReader
- Easy to read large files

➢ **Example:**

```
BufferedReader br = new BufferedReader(new FileReader("data.txt"));
String line;
```

```
while ((line = br.readLine()) != null) {
   System.out.println(line);
}
br.close();
```

2) **BufferedWriter**

- Writes text efficiently using buffering.
- Used with FileWriter.
- Reduces disk access.

- **Example:**

```
BufferedWriter bw = new BufferedWriter(new FileWriter("data.txt"));
bw.write("Java File Handling");
bw.newLine();
bw.write("Buffered Writer Example");
bw.close();
```

Question: 4

Serialization and Deserialization

Answer:

1) **Serialization**

- Process of converting an object into a byte stream.
- Used to save object state into a file.
- The class must implement Serializable interface.

- **Uses:**
  - Saving objects
  - Sending objects over network
  - Storing object data permanently

- **Example:**

```
import java.io.*;
class Student implements Serializable {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
FileOutputStream fos = new FileOutputStream("student.txt");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(new Student(1, "Rahul"));
oos.close();
```

2) Deserialization

- Process of converting byte stream back into an object.
- Restores the original object.

➢ **Example:**

```
FileInputStream fis = new FileInputStream("student.txt");
ObjectInputStream ois = new ObjectInputStream(fis);
Student s = (Student) ois.readObject();
System.out.println(s.id + " " + s.name);
ois.close();
```

➢ **Important Notes**

- transient keyword is used to skip variables during serialization.
- Serializable is a marker interface (no methods).
- IOException and ClassNotFoundException must be handled.

---

15. Collections Framework

Question: 1

Introduction to Collections Framework

Answer:

The Collections Framework in Java is a set of classes and interfaces used to store, manipulate, and retrieve groups of objects efficiently.

Advantages:

- Dynamic size (grows and shrinks)
- Ready-made data structures
- Improves performance
- Reduces programming effort

Package:

java.util

Question: 2

Collection Interfaces: List, Set, Map, and Queue Interfaces

Answer:

a) List Interface

- Allows duplicate elements
- Maintains insertion order
- **Examples:** ArrayList, LinkedList

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("A");
```

b) Set Interface

- Does not allow duplicates
- Order depends on implementation
- **Examples:** HashSet, TreeSet

```
Set<Integer> set = new HashSet<>();
set.add(10);
set.add(10);  // ignored
```

c) Map Interface

- Stores data in key-value pairs
- Keys are unique
- **Examples:** HashMap, TreeMap

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Java");
```

d) Queue Interface

- Follows FIFO (First In First Out)
- Used in scheduling and task management
- **Example:** PriorityQueue

```
Queue<Integer> q = new PriorityQueue<>();
q.add(10);
```

Question: 3

Important Collection Classes: ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

Answer:

a) ArrayList

- Uses dynamic array
- Fast access, slow insertion/deletion
- **Example:**

```
ArrayList<String> al = new ArrayList<>();
```

b) LinkedList

- Uses doubly linked list
- Fast insertion/deletion, slower access
- **Example:**

```
LinkedList<Integer> ll = new LinkedList<>();
```

c) HashSet

- No duplicate elements
- No insertion order
- **Example:**

```
HashSet<Integer> hs = new HashSet<>();
```

d) TreeSet

- Stores elements in sorted order
- No duplicates
- **Example:**

  ```
  TreeSet<Integer> ts = new TreeSet<>();
  ```

e) HashMap

- Stores key-value pairs
- No ordering
- **Example:**

  ```
  HashMap<Integer, String> hm = new HashMap<>();
  ```

f) TreeMap

- Stores key-value pairs in sorted order (by key)
- **Example:**

  ```
  TreeMap<Integer, String> tm = new TreeMap<>();
  ```

Question: 4

Iterators and ListIterator

Answer:

a) Iterator

- Used to traverse elements forward only.
- **Example:**

  ```
  Iterator<String> it = list.iterator();
  while(it.hasNext()) {
     System.out.println(it.next());
  }
  ```

b) ListIterator

- Used to traverse forward and backward (only for List).
- **Example:**

  ```
  ListIterator<String> li = list.listIterator();
  ```

- **Features:**
  - Bidirectional
  - Can add, update, remove elements
- **Difference: Iterator vs ListIterator**

Iterator	**ListIterator**
Forward only	Forward & backward
Works for all collections	Only for List
Cannot add element	Can add element

----

16. Java Input/Output (I/O)

Question: 1

Streams in Java (InputStream and OutputStream)

Answer:

What is a Stream?

A stream is a flow of data between a program and an input/output source such as a file, keyboard, or network.

Java uses streams to read and write data.

a) InputStream

- InputStream is used to read data from a source.

➢ **Common InputStream classes:**

- FileInputStream
- BufferedInputStream
- DataInputStream
- **Example:**

  InputStream in = new FileInputStream("data.txt");

b) OutputStream

- OutputStream is used to write data to a destination.

➢ **Common OutputStream classes:**

- FileOutputStream
- BufferedOutputStream
- DataOutputStream
- **Example:**

  OutputStream out = new FileOutputStream("data.txt");

Question: 2

Reading and Writing Data Using Streams

Answer:

- **Writing Data to a File Example:**

```
FileOutputStream fos = new FileOutputStream("file.txt");
String msg = "Hello Java";
fos.write(msg.getBytes());
fos.close();
```

- **Reading Data from a File Example:**

```
FileInputStream fis = new FileInputStream("file.txt");
int i;
while ((i = fis.read()) != -1) {
   System.out.print((char)i);
}
fis.close();
```

Question: 3

Handling File I/O Operations

Answer:

- **Steps for File I/O:**

  1. Create stream object

  2. Read or write data

  3. Close the stream

- **File Class**
  - The File class is used to create and manage files and directories.
  - **Example:**

```
File f = new File("demo.txt");
System.out.println(f.exists());
```

- **Exception Handling in File I/O**
  - File operations may throw IOException, so try-catch is required.
  - **Example:**

```
try {
   FileInputStream fis = new FileInputStream("abc.txt");
} catch (IOException e) {
   System.out.println(e);
}
```

- **Advantages of Streams**
  - Efficient data handling
  - Supports large files
  - Platform independent
