pa

UNIT 1

1) Primitive and Non-Primitive Data Types in Java

O Primitive Data Types

Primitive data types are the **basic built-in data types** in Java. There are **8 primitive types**, and they are **not objects**.

→Byte(8),Short(16),Int(32),Long(64),Float(32),Double(64),Char(16),Boolean(1) .

Syntax : - int c = 5000; , char g = 'A'; , boolean h = true;

Example (Code Snippet) :-

```
public class PrimitiveDemo {

   public static void main(String[] args) {

      int age = 20;

      char grade = 'A';

      System.out.println("Age: " + age);

      System.out.println("Height: " + height);

      System.out.println("Grade: " + grade);

      System.out.println("Passed: " + isPassed);

   }

}
```

O Non-Primitive Data Types

Non-primitive types (also called **reference types**) are **objects and arrays**. They are created by the **programmer** and **can store multiple values or methods**.

Examples (Code Snippet):

1. String
2. Arrays
3. Classes
4. Interfaces

Code Snippet (Non-Primitive Types)

```
public class NonPrimitiveDemo {

   public static void main(String[] args) {

      String name = "Java";

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


      System.out.println("Name: " + name);

      System.out.println("First number: " + numbers[0]);

   }

}
```

2.) Define bytecode with brief description and list its advantages OR describe "platform independent" feature in java. OR describe "portability" feature in java. (4/6)

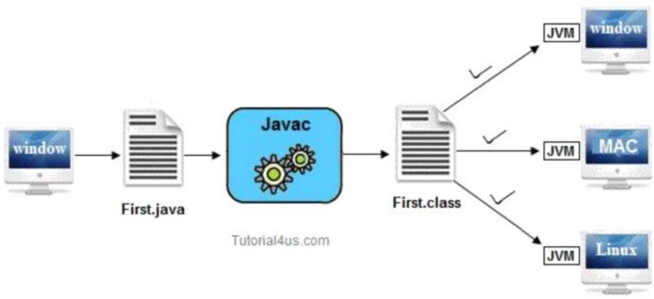
Java Byte Code:-

Java programs are **not** compiled directly into machine code (like C/C++). Instead, they are compiled into an intermediate form called **Java Byte Code**.

What is Java Byte Code?

Java Byte Code is a platform-independent, low-level code generated by the Java compiler (javac) from .java source files.

Platform Independent:-



This diagram illustrates the **platform independence of Java**:

1. A Java program (`File1.java`) is written on a Windows system.
2. It is compiled using the **Java Compiler** (`javac`) into bytecode (`File1.class`).
3. This **bytecode** is **platform-independent** and can be run on any system (Windows, Mac, Linux) that has a **Java Virtual Machine (JVM)**.
4. The **JVM on each OS** interprets the bytecode and executes it, enabling **"Write Once, Run Anywhere"** capability.

Portability:-

- Portability refers to the ability to run a program on different machines.
- The Java program is getting compiled into bytecode which is platform independent.
- For **example**, the same applet must can be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet.
- There is no need to keep different versions of the applet for different computers. The same code must run on all computers.

3.) Describe the following features in java /design goals of Java / Java characteristics: (4/6)

1. Robust

→Java is a strongly typed language with compile-time and run-time error checking, automatic memory management, and exception handling.

→Example: Java's garbage collector automatically frees unused memory, unlike C/C++ where the programmer must manually release it.

2. Secure

→Java provides a secure execution environment, restricts direct memory access, and prevents unauthorized file access.

→Example: Java applets run in a sandbox, preventing them from accessing local system files.

3. Portable

→Portability is the ability to run the same program on different platforms without modification.

→Example: A compiled Java bytecode file can run on Windows, Linux, and Mac using JVM.

4. Multithreaded

→Multithreading allows multiple parts of a program to run concurrently for better performance and responsiveness.

→Example: In a browser, separate threads handle page loading, rendering, and user input simultaneously.

5. Architecture-neutral

→Java programs are independent of the operating system and hardware architecture.

→Example: A Java program compiled on Windows will run on a Mac without changes.

6. Dynamic

→Java supports dynamic loading of classes and can integrate code from other languages when required.

→Example: A Java program can load a plugin at runtime without restarting the application.

7. Distributed

→Java is designed for networked environments and supports protocols like TCP/IP and features like RMI.

→Example: A Java application can call methods from a program running on another computer using RMI.

8. Interpreted & High Performance

→Java uses the JVM with a JIT compiler to convert bytecode into machine code at runtime for faster execution.

→Example: Java applications run faster with JIT compared to pure interpretation of bytecode.

9. Simple

→Java is easy to learn with a clean syntax and without complex features like pointers and manual memory management.

→Example: Java handles memory cleanup automatically using garbage collection.

10. Object Oriented

→Java follows object-oriented principles like inheritance, polymorphism, abstraction, and encapsulation.

→Example: A Car class can be used to create multiple car objects with shared and unique properties.

11. Encapsulation

→Wrapping data and methods together into a single unit (class) and protecting data by restricting direct access using access modifiers.

→Example: A Student class with private age variable accessed only through a public method.

12. Abstraction

→Hiding internal details and showing only the essential features, implemented using abstract classes and interfaces.

→Example: An Animal abstract class with an abstract sound() method implemented differently in Dog and Cat classes.

13. Polymorphism

→One entity behaving in multiple ways, achieved through method overloading and method overriding.

→Example: add() method performing addition with different numbers of parameters (overloading) or sound() method behaving differently in parent and child classes (overriding).

14. Inheritance

→Allowing one class to acquire the properties and behaviors of another class using the extends keyword.

→Example: A Car class inheriting the start() method from a Vehicle class.

4.) Define type casting and explain two types of type casting with suitable example. (4)

A. Implicit Casting (Widening)

- Java **automatically converts** smaller data types into larger ones (no data loss).

```
int a = 10;
double b = a; // int is automatically converted to double
```

Example:-

```
public class WideningExample {
   public static void main(String[] args) {
       int a = 10;
       double b = a;  // int to double

       System.out.println("Original int: " + a);
       System.out.println("Converted to double: " + b);
   }
}
```

B. Explicit Casting (Narrowing):

- You have to **manually convert** larger data types into smaller ones (possible data loss).

```
double x = 9.78;
int y = (int) x; // double to int (Result: 9 – decimal part is lost)
```

Example:-

```
public class NarrowingExample {
   public static void main(String[] args) {
       double x = 10.75;
       int y = (int) x;  // double to int

       System.out.println("Original double: " + x);
       System.out.println("Converted to int: " + y);
   }
}
```

5.)Describe character/Boolean/floating point numbers/integer with example. (4)

1. Character (char)

1. The char data type is used to store a **single 16-bit Unicode character**.
2. It must be enclosed in **single quotes**, like 'A', '1', etc.

Example:

```
public class CharExample {
  public static void main(String[] args) {
    char letter = 'A';
    System.out.println("Character: " + letter);
  }
}
```

2. Boolean (boolean)

1. The boolean type holds **only two values**: true or false.
2. Commonly used in **conditional statements** like if, while.

Example:

```
public class BooleanExample {
  public static void main(String[] args) {
    boolean isJavaFun = true;
    System.out.println("Is Java fun? " + isJavaFun);
  }
}
```

3. Floating Point Numbers (float and double)

1. Used to store **decimal values**.
2. float is a 32-bit single-precision value, and double is 64-bit double-precision.

Example:

```
public class FloatExample {
  public static void main(String[] args) {
    float price = 99.99f;
    double pi = 3.1415926535;
    System.out.println("Price: " + price);
    System.out.println("Pi: " + pi);
  }
}
```

4. Integer (byte, short, int, long)

1. Used to store **whole numbers**.
2. int is most commonly used; other types vary in size.

Example:

```
public class IntegerExample {
   public static void main(String[] args) {
      int age = 20;
      long population = 1400000000L;
      System.out.println("Age: " + age);
      System.out.println("Population: " + population);
   }
}
```

6. Explain how to take user input using Scanner class. (4 marks)

To take user input in Java, we use the Scanner class from the java.util package. It reads input from the keyboard using System.in.

Steps:

1. Import the Scanner class.
2. Create a Scanner object.
3. Use appropriate methods like nextInt(), nextLine(), nextDouble(), etc., to read input.

Syntax:

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);

// To read different data types:

int a = sc.nextInt();        // for integer input
```

Example:

```
import java.util.Scanner;


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


      System.out.print("Enter your name: ");
      String name = sc.nextLine();


      System.out.print("Enter your age: ");
      int age = sc.nextInt();


      System.out.println("Name: " + name);
      System.out.println("Age: " + age);


      sc.close();
   }
}
```

This program reads a string and an integer from the user using Scanner.

7. Explain arithmetic, logical, and relational operators with example OR with example program. (4 marks)

1. Arithmetic Operators

Used to perform basic mathematical operations.

Operators: +, -, *, /, %

Example:

```
int a = 10, b = 3;
System.out.println("Sum = " + (a + b));       // 13
System.out.println("Product = " + (a * b));   // 30
```

2. Relational Operators

Used to compare two values. Result is true or false.

Operators: ==, !=, <, >, <=, >=

Example:

```
int x = 5, y = 8;
System.out.println(x < y);   // true
System.out.println(x == y);  // false
```

3. Logical Operators

Used to combine two or more boolean expressions.

Operators: && (AND), || (OR), ! (NOT)

Example:

```
boolean a = true, b = false;
System.out.println(a && b);  // false
System.out.println(a || b);  // true
System.out.println(!a);      // false
```

Example Program (All in One):

```
public class OperatorExample {
   public static void main(String[] args) {
      int a = 10, b = 5;
      System.out.println("Addition: " + (a + b));
      System.out.println("a > b: " + (a > b));
      System.out.println("(a > b) && (b > 0): " + ((a > b) && (b > 0)));
   } }
```

8. Explain bitwise operator with example. (4)

Bit wise Operator:

Bitwise operators work on **individual bits (0 or 1)** of integers. These are mainly used for low-level programming, performance, or bit manipulation.

📋 List of Bitwise Operators:

Operator	Name	Description	Example (`a=5, b=3`)
`&`	AND	1 if **both** bits are 1	`a & b = 1`
`	`	OR	1 if **any one** of the bits is 1
`^`	XOR	1 if bits are **different**	`a ^ b = 6`
`~`	NOT	Inverts each bit (1→0, 0→1)	`~a = -6`
`<<`	Left Shift	Shifts bits to the left, adds 0s on the right	`a << 1 = 10`
`>>`	Right Shift	Shifts bits to the right, keeps the sign bit	`a >> 1 = 2`
`>>>`	Unsigned Right Shift	Same as right shift, but fills 0s from the left	`a >>> 1 = 2`

```java
public class BitwiseExample {
    public static void main(String[] args) {
        int a = 5;  // Binary: 0101
        int b = 3;  // Binary: 0011

        System.out.println("a & b: " + (a & b));   // 1
        System.out.println("a | b: " + (a | b));   // 7
        System.out.println("a ^ b: " + (a ^ b));   // 6
        System.out.println("~a: " + (~a));          // -6
        System.out.println("a << 1: " + (a << 1)); // 10
        System.out.println("a >> 1: " + (a >> 1)); // 2
    }
}
```

9. Explain conditional operator with example. (4)

The **conditional operator** in Java is a shorthand for an `if-else` statement.

It has the form:

sql Copy Edit

```sql
condition ? expression1 : expression2
```

- **condition** → an expression that evaluates to `true` or `false`.
- If **condition** is `true` → `expression1` is executed.
- If **condition** is `false` → `expression2` is executed.

The conditional operator :

```java
int marks = 45;
String status = (marks >= 50) ? "Pass" : "Fail";
System.out.println("Result: " + status);
```

10. Explain special operator with example. (4)

Special Operators in Java

In Java, some operators are called **special operators** because they serve unique purposes that aren't just arithmetic or logical operations. Common examples include:

1. `instanceof` **Operator** – Checks whether an object is an instance of a specific class or subclass.
2. `?:` **(Ternary/Conditional) Operator** – Acts as a shorthand for `if-else` statements.

Example using `instanceof`:

java Copy Edit

```java
public class SpecialOperatorExample {
    public static void main(String[] args) {
        String name = "Java";

        // instanceof operator
        if (name instanceof String) {
            System.out.println("name is a String object");
        }
    }
}
```

(conditional is ternary so 9 can be considered as special operator)

11. State the significance of Java Virtual Machine (JVM) in the Java programming environment. (4)

Significance of Java Virtual Machine (JVM)

The **Java Virtual Machine (JVM)** is an important part of the Java programming environment because it allows Java programs to be **platform-independent** and secure.

Key Points:

1. **Platform Independence** – JVM enables Java's "Write Once, Run Anywhere" feature by executing **Java bytecode** on any operating system.
2. **Bytecode Execution** – It converts Java bytecode (from `.class` files) into **machine code** for the host system.
3. **Memory Management** – JVM handles **automatic garbage collection** to free unused memory.
4. **Security & Sandbox** – It provides a secure runtime environment by running code inside a controlled sandbox, preventing unauthorized access.

12. Explain the concept of platform independence in Java and discuss how it is achieved. Give example to illustrate the concept. (4)

Platform Independence in Java

Concept:
Platform independence means that a Java program can run on **any operating system** without changing the source code. This is possible because Java programs are compiled into **bytecode**, which can be executed by the **Java Virtual Machine (JVM)** on any platform.

How it is Achieved:

1. **Compilation to Bytecode** – The Java compiler (`javac`) converts `.java` files into `.class` files containing **bytecode**.
2. **JVM Execution** – The JVM interprets or compiles bytecode into **machine code** specific to the underlying OS/CPU.
3. Since each OS has its own JVM implementation, the same bytecode runs everywhere.

Example:

java Copy Edit

```java
public class PlatformIndependent {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

- Compile once: `javac PlatformIndependent.java` → produces `PlatformIndependent.class` (bytecode).
- Run on Windows: `java PlatformIndependent` → Output: `Hello, Java!`
- Copy same `.class` file to Linux/Mac → Output remains the same.

13. Differentiate between method overloading and method overriding. (4)

Aspect	Method Overloading	Method Overriding
Definition	Defining multiple methods in the same class with **same name but different parameter lists.**	Redefining a method in the subclass that already exists in the parent class with the **same signature.**
Parameters	Must be **different** (number or type).	Must be **same** as the parent method.
Return Type	Can be different (if not ambiguous).	Must be same or covariant (subtype).
Inheritance	Not required.	Requires **inheritance.**
Compile/Run Time	Resolved at **compile-time** (Compile-time polymorphism).	Resolved at **run-time** (Run-time polymorphism).
Access Modifiers	Can have any access modifier.	Access level **cannot be more restrictive** than in parent method.

14. Explain the concept of argument passing and the usage of 'this' keyword in Java give example to illustrate their usage and benefits. (4/6)

Argument Passing in Java:

Java uses **Call by Value** for passing arguments.

- For **primitive types**, a copy of the value is passed.
- For **objects**, a copy of the reference is passed, so changes affect the original object's data.

`this` Keyword:

`this` refers to the **current object.**

Uses:

1. Differentiate instance variables from method parameters.
2. Pass current object as an argument.
3. Call another constructor in the same class.

Example:

java

```java
class Student {
    String name;
    Student(String name) { this.name = name; }
    void display(Student s) { System.out.println(s.name); }
    void show() { display(this); }
}
```

Output:

nginx

```nginx
John
```

UNIT 2

1. Define the following terms with example. (4 marks)

Class:

A class is a blueprint or template that defines variables (fields) and methods for creating objects.

Example:

```
class Car {
   String color;
   void display() {
      System.out.println("Car color: " + color);
   }
}
```

Object:

An object is an instance of a class. It represents a real-world entity and has state (variables) and behavior (methods).

Example:

```
Car myCar = new Car(); // myCar is an object
myCar.color = "Red";
myCar.display();
```

Reference:

A reference is a variable that holds the memory address of an object, not the object itself. Multiple references can point to the same object.

Example:

```
Car car1 = new Car();
Car car2 = car1; // car2 references the same object as car1
```

2. Describe the working of new operator in Java with appropriate example. (4 marks)

The new operator is used to create objects in Java.

→ It allocates memory for the object on the heap.

→ Calls the class constructor to initialize the object.

→ Returns a reference to the newly created object.

Syntax:

ClassName reference = new ClassName(arguments);

Basic Example:

```
class Car {
   String color;
}

public class Main {
   public static void main(String[] args) {
      Car myCar = new Car(); // creating object using new
      myCar.color = "Blue";
      System.out.println(myCar.color);
   }
}
```

3.) Describe the process of creation of objects in Java with suitable example. (4 marks)

In Java, an object is created from a class using the new operator. The process involves:

1. Declaring a reference variable of the class type.
2. Using the new operator to allocate memory for the object in the heap.
3. Calling the class constructor to initialize the object.
4. Assigning the reference of the object to the reference variable.

Syntax:

ClassName reference = new ClassName(arguments);

Basic Example:

```
class Student {
   String name;
}

public class Main {
   public static void main(String[] args) {
      Student s = new Student(); // object creation
      s.name = "John";
      System.out.println(s.name);
   }
}
```

4.) Define constructors and explain default and parameterized constructor with example. (4/6 marks)

- In Java, **constructors** are special methods that are called when an object is created.
- They initialize the object with default or specific values.
- A **constructor** is a block of code that gets executed when an instance (object) of a class is created.
- It has the same name as the class and does not have a return type (not even void).

Types of Constructors:

1. Default Constructor (No-Argument Constructor)
2. Parameterized Constructor

1. Default Constructor (No-Argument Constructor):

- A **default constructor** is a constructor that takes no parameters.
- If no constructor is explicitly defined, Java provides a **default constructor** automatically.
- This constructor initializes the object with default values (such as null for objects, 0 for integers, false for Booleans, etc.).

2. Parameterized Constructor:

- A **parameterized constructor** is a constructor that takes parameters (arguments) to initialize the object with specific values at the time of object creation.

Syntax:

```
class ClassName {
   ClassName() { }             // default constructor
   ClassName(parameters) { }     // parameterized constructor
}
```

Example:-

```
class Student {
   String name;
       Student() {  // default constructor
             name = "Unknown";
        }
        Student(String n) { //parameterized constuctors
              name = n;
         }
}
public class Main {
   public static void main(String[] args) {
      Student s1 = new Student();            // calls default constructor
      Student s2 = new Student("John", 20);    // calls parameterized constructor


      System.out.println(s1.name + " " + s1.age);
      System.out.println(s2.name + " " + s2.age);
   }
}
```

5. Explain constructor overloading with example. (4 marks)

Constructor overloading in Java means defining **multiple constructors** in the same class with **different parameter lists** (different number or type of parameters).

1. It allows creating objects in different ways.
2. The compiler decides which constructor to call based on the arguments passed.

Syntax:

```
class ClassName {
   ClassName() { }           // constructor 1
   ClassName(parameters) { }    // constructor 2
}
```

Example:

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

   Student() { // no-arg constructor
      name = "Unknown";
      age = 0;
   }

   Student(String n) { // single parameter
      name = n;
      age = 0;
   }

   Student(String n, int a) { // two parameters
      name = n;
      age = a;
   }
}
```

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("John");
        Student s3 = new Student("Alice", 22);

        System.out.println(s1.name + " " + s1.age);
        System.out.println(s2.name + " " + s2.age);
        System.out.println(s3.name + " " + s3.age);
    }
}
```

6.) Explain passing and returning objects to and from the method (function) with example. (4 marks)

In Java, objects can be:

1. **Passed as arguments** to a method – The reference of the object is passed, so changes made in the method affect the original object.
2. **Returned from a method** – A method can create an object and return its reference.

Syntax:

```
void methodName(ClassName obj) { }       // passing object
ClassName methodName() { return object; } // returning object
```

Example:

```
class Student {
   String name;
   Student(String n) {
      name = n;
   } }
public class Main {
   // method to display student (passing object)
   static void display(Student s) {
      System.out.println("Name: " + s.name);
   }
   // method to return student object
   static Student createStudent(String n) {
      return new Student(n);
   }
   public static void main(String[] args) {
      Student s1 = new Student("John");
      display(s1); // passing object
      Student s2 = createStudent("Alice"); // returning object
      display(s2);
   } }
```

7.) Explain static data and static member method with example program. (6 marks)

1. **Static Data Members (Static Variables)**
    1. Declared using the static keyword.
    2. Shared among all objects of the class (only one copy exists).
    3. Accessed using the class name or an object.
2. **Static Member Methods**
    1. Declared using the static keyword.
    2. Belong to the class rather than any specific object.
    3. Can be called without creating an object.
    4. Can only directly access other static members.

Syntax:

```
class ClassName {
   static dataType variableName;
   static returnType methodName() { }
}
```

Example:

```
class Student {
   String name;
   static String college = "ABC College"; // static variable
   Student(String n) {
      name = n;
   }
   static void changeCollege(String newCollege) { // static method
      college = newCollege;
   }
   void display() {
      System.out.println(name + " - " + college);
   }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("John");
        Student s2 = new Student("Alice");

        s1.display();
        s2.display();

        Student.changeCollege("XYZ College"); // calling static method

        s1.display();
        s2.display();
    }
}
```

This program shows how **static data** is shared among all objects and how **static methods** can be called without creating an object.

8. Describe call-by-value with suitable example program. (4 marks)

Explanation:

1. In **call-by-value**, when a method is called, the actual value of the argument is **copied** into the method's parameter.
2. Any changes made inside the method **do not affect** the original value outside the method.
3. Java always uses **call-by-value** for passing arguments (even for objects, it passes the reference value).

Syntax:

```
returnType methodName(dataType parameter) {
   // code
}
```

Basic Example:

```
class Example {
   void modifyValue(int num) {
      num = num + 10; // changes local copy only
      System.out.println("Inside method: " + num);
   }
}

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

      System.out.println("Before method call: " + a);
      obj.modifyValue(a);
      System.out.println("After method call: " + a);
   }
}
```

Output:

Before method call: 5

Inside method: 15

After method call: 5

Here, the original value of a remains unchanged because **only a copy** of a is passed to the method.

9. Describe the working of “this” keyword with an example. (4)

- In Java, the this keyword is used to refer to **the current instance of the class**. It is used within a class to reference the **current object** that the method or constructor is being called on.
- The **this** keyword is mainly used to:

1. **Refer to the instance variables (fields)** of the current class.
2. **Invoke the current class's constructor**.
3. **Pass the current object** as a parameter to other methods.

```
class Student {
    String name;  // Instance variable

    // Constructor with a parameter
    Student(String name) {
        this.name = name;  // 'this.name' refers to the instance variable, 'name' is the paramete
    }

    void display() {
        System.out.println("Name: " + this.name);  // Refers to the instance variable 'name'
    }
```

10. Explain constructor chaining with example program(4)

Constructor chaining in Java is the process of calling one constructor from another within the same class or from the parent class.

- **Within the same class:** Use `this()` to call another constructor.
- **From parent class:** Use `super()` to call the parent constructor.

Syntax:

java Copy Edit

```java
className() { this(args); }  // same class
className() { super(args); } // parent class
```

Example:

java Copy Edit

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

class Child extends Parent {
    Child() {
        this(5); // calls parameterized constructor of Child
        System.out.println("Child Default Constructor");
    }
    Child(int x) {
        super(); // calls Parent constructor
        System.out.println("Child Parameterized Constructor: " + x);
    }
}

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

Output:

rust Copy Edit

```rust
Parent Constructor
Child Parameterized Constructor: 5
Child Default Constructor
```

11. Explain Static blocks with example. (4)

A **static block** in Java is used to initialize static variables or execute code once when the class is loaded into memory.

- Runs **automatically** before any object creation or the `main()` method.
- Useful for setting up **static resources**.

Syntax:

java Copy Edit

```java
static {
    // code that runs once when the class is loaded
}
```

Example:

java Copy Edit

```java
class Demo {
    static int num;
    static {
        num = 5;
        System.out.println("Static block runs first");
    }
    public static void main(String[] args) {
        System.out.println("Main method");
        System.out.println("num = " + num);
    }
}
```

Output:

sql Copy Edit

```sql
Static block runs first
Main method
num = 5
```

12. Difference between class and object. (4)

Class	Object
Blueprint or template for creating objects.	Instance of a class.
Defines properties (fields) and behaviors (methods).	Holds actual data and can use methods.
Does not occupy memory until objects are created.	Occupies memory when created.
Example: `class Car {}`	Example: `Car c = new Car();`

13. Difference between continue and break. (4)

13. Difference between continue and break

continue	break
Skips the current iteration and moves to the next loop iteration.	Exits the loop entirely.
Used inside loops only.	Used inside loops or switch statements.
Control goes back to loop condition check.	Control moves outside the loop or switch.

Example:

java Copy Edit

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

Output: `1 2 4 5` | Example:

java Copy Edit

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

Output: `1 2` |

14. Explain Method Overloading with sample program. (4)

Explanation:

Method Overloading in Java means having multiple methods in the same class with the same name but different **parameter lists** (different number or type of parameters).

- It increases code readability.
- Return type may be same or different, but parameters must differ.

Syntax:

java

```java
returnType methodName(param1);
returnType methodName(param1, param2);
```

Example:

java

```java
class Demo {
    void show(int a) {
        System.out.println("Number: " + a);
    }
    void show(String s) {
        System.out.println("Text: " + s);
    }
    public static void main(String[] args) {
        Demo d = new Demo();
        d.show(10);
        d.show("Hello");
    }
}
```

Output:

makefile

```makefile
Number: 10
Text: Hello
```

15.Explain String class and its methods. (4/6/8)

The **String** class in Java represents a sequence of characters and is **immutable** (once created, it cannot be changed).

- Defined in `java.lang` package.
- Can be created using string literals (`"Hello"`) or with `new String("Hello")`.
- String class provides many useful methods for string operations.

Common String Methods:

Method	Description	Example
`length()`	Returns the length of the string.	`"Java".length()` → `4`
`charAt(int index)`	Returns character at given index.	`"Java".charAt(1)` → `a`
`concat(String s)`	Concatenates two strings.	`"Hello".concat(" World")` → `"Hello World"`
`equals(String s)`	Checks if strings are equal.	`"a".equals("a")` → `true`
`equalsIgnoreCase(String s)`	Compares ignoring case.	`"A".equalsIgnoreCase("a")` → `true`
`toUpperCase()`	Converts to uppercase.	`"java".toUpperCase()` → `"JAVA"`
`toLowerCase()`	Converts to lowercase.	`"JAVA".toLowerCase()` → `"java"`
`substring(int begin, int end)`	Extracts substring.	`"abcdef".substring(1,4)` → `"bcd"`
`replace(char old, char new)`	Replaces characters.	`"Java".replace('a','o')` → `"Jovo"`
`trim()`	Removes leading and trailing spaces.	`" Java ".trim()` → `"Java"`

Example:

java

Copy Edit

```java
class StringExamples {
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = " Hello World ";
        String s3 = "java";

        System.out.println("Length: " + s1.length()); // 4
        System.out.println("CharAt: " + s1.charAt(1)); // a
        System.out.println("Concat: " + s1.concat(" Programming")); // Java Programming
        System.out.println("Equals: " + s1.equals("Java")); // true
        System.out.println("EqualsIgnoreCase: " + s1.equalsIgnoreCase(s3)); // true
        System.out.println("ToUpperCase: " + s1.toUpperCase()); // JAVA
        System.out.println("ToLowerCase: " + s1.toLowerCase()); // java
        System.out.println("Substring: " + s1.substring(1, 3)); // av
        System.out.println("Replace: " + s1.replace('a', 'o')); // Jovo
        System.out.println("Trim: " + s2.trim()); // Hello World
    }
}
```

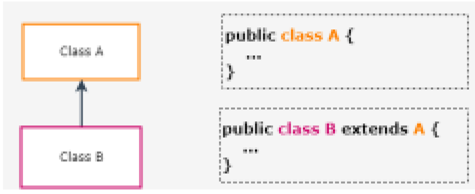
UNIT 3

1.) Define Inheritance. Enlist different types of inheritance and explain all with example program(4/6)

3.1 Inheritance Basics:

What is Inheritance?

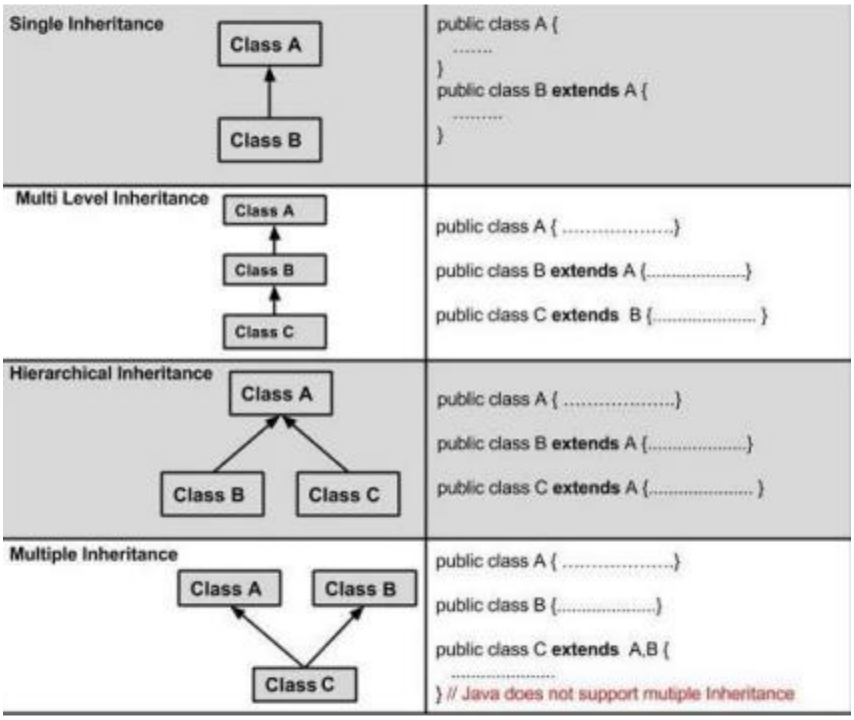
Inheritance is one of the core concepts of **Object-Oriented Programming (OOP)** that allows a new class to acquire (inherit) properties and behaviors (fields and methods) from an existing class.



- The existing class is called the **parent class** or **superclass**.
- The new class is called the **child class** or **subclass**.

Why use Inheritance?

- **Code reuse**: Reuse existing code without rewriting it.
- **Extensibility**: Add new features or modify existing behavior.
- **Method overriding**: Modify or customize behavior of inherited methods.
- **Hierarchy**: Models real-world relationships.



2.) Explain final Keyword with example. (4)

The keyword **final** in Java is a non-access modifier that restricts the modification of a variable, method, or class. It's used to declare an entity as constant, preventing it from being changed or overridden.

Examples:- Final variable , Final method , Final Class .

Ex:-

```java
// A final class can't be extended.
final class User {

    // A final variable can't be changed.
    final String USERNAME = "admin";

    // A final method can't be overridden.
    final void displayUsername() {
        System.out.println("Username is: " + USERNAME);
    }
}
```

3.) Explain use of Super Keyword with example(4)

The **super** keyword in Java is a reference variable used to access members of the parent class from within a child class. It's primarily used in three ways: to refer to a parent class's instance variables, to call its methods, or to invoke its constructor

```java
// Parent class
class Animal {
    String type = "mammal";

    void eat() {
        System.out.println("Animal eats food.");
    }
}

// Child class
class Dog extends Animal {
    String type = "dog";

    Dog() {
        // Calls the parent class constructor (Animal)
        super();
        System.out.println("A new dog is here.");
    }

    void displayInfo() {
        // Calls the parent class's eat() method
        super.eat();

        // Accesses the parent class's 'type' variable
        System.out.println("This is a " + super.type);
    }
}
```

4.Explain Method Overriding/ Dynamic method dispatch with example program. (4/6)

Method Overriding

- **Method Overriding** occurs when a subclass provides its own implementation of a method that is already defined in its superclass.
- It allows runtime polymorphism — the subclass's version of the method is called instead of the superclass's.
- The method signature (name, parameters) must be the same in both superclass and subclass.
- Access level cannot be more restrictive in the subclass.

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

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

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Superclass reference, subclass object
        myAnimal.sound();  // Calls Dog's sound method at runtime
    }
}
```

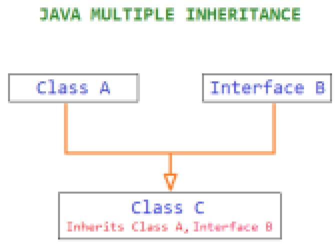
Output:

```
Dog barks
```

5.) Define Interface. Describe implementation of interface with appropriate example. (4)

Defining an Interface

- An **interface** in Java is a reference type, similar to a class, that can contain **only abstract methods** (until Java 7) and **constants** (static final variables).
- From Java 8 onwards, interfaces can also have **default methods** (methods with implementation) and **static methods**.
- Interfaces define a **contract** that implementing classes must follow.
- They provide a way to achieve **multiple inheritance** in Java (since Java doesn't support multiple inheritance with classes).



Implementing Interfaces

A class **implements** an interface using the `implements` keyword.

The class must provide concrete implementations of all abstract methods declared in the interface (unless the class is abstract).

A class can implement **multiple interfaces** separated by commas.

```
// Interface definition
interface Sports {
    int MAX_SCORE = 100;  // public static final by default

    void setScore(int score);
    int getScore();

    default void showMaxScore() {
        System.out.println("Max score in sports: " + MAX_SCORE);
    }
}
```

```
// Class implementing the interface
class Football implements Sports {
    private int score;

    public void setScore(int score) {
        if (score <= MAX_SCORE)
            this.score = score;
        else
            this.score = MAX_SCORE;
    }

    public int getScore() {
        return score;
    }
}
```

```
// Main class to test
public class InterfaceExample {
    public static void main(String[] args) {
        Football fb = new Football();

        fb.setScore(95);
        System.out.println("Football Score: " + fb.getScore());

        fb.showMaxScore();  // calling default method in interface
    }
}
```

6.) Explain access specifier OR Explain any four visibility controls in Java. with example and compare. (4)

1. Private 🔒

The `private` access specifier is the most restrictive. Members declared as `private` are only visible within the same class. They cannot be accessed from any other class, even a subclass.

2. Default (No Keyword) 📦

When no access specifier is explicitly declared, it is considered **default**. Members with `default` visibility are accessible only within the same package. They cannot be accessed from outside the package.

3. Protected 👫

The `protected` access specifier allows members to be accessed within the same package and by all subclasses, even if the subclasses are in a different package.

4. Public 🌐

The `public` access specifier is the least restrictive. Members declared as `public` are accessible from anywhere in the program, including from different packages and classes.

Access Specifier	Same Class	Same Package	Subclass (Different Package)	Anywhere
`private`	✅	❌	❌	❌
`default`	✅	✅	❌	❌
`protected`	✅	✅	✅	❌
`public`	✅	✅	✅	✅

7.) Explain how to create a package and import it with suitable example. (4)

Defining a Package

- You define a package at the **very beginning** of a Java source file using the `package` keyword.
- Syntax:

```
package package_name;
```

- Example:

```
package com.mycompany.project;

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

- The directory structure should mirror the package structure.

  For example:

```
com/mycompany/project/MyClass.java
```

8. Explain the concept of packages in Java and their significance in software development. Write an example to illustrate the usage and benefits of using packages. (4/6/8)

Package in Java?

- A **package** in Java is a namespace that organizes classes and interfaces.
- It helps to **avoid name conflicts** and **control access**.
- Packages also make searching/locating and usage of classes easier.
- Think of packages as **folders/directories** in your file system that contain related `.java` files.

Significance of Packages in Software Development

1. **Code Organization** – Groups related classes logically.
2. **Name Conflict Avoidance** – Different packages can have classes with the same name without clashing.
3. **Reusability** – Once written, packaged classes can be reused in other projects.
4. **Access Control** – Can restrict access to classes using access modifiers.
5. **Ease of Maintenance** – Easier to locate and modify code when organized.

Defining a Package

- You define a package at the **very beginning** of a Java source file using the `package` keyword.
- Syntax:

```
package package_name;
```

- Example:

```
package com.mycompany.project;

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

- The directory structure should mirror the package structure.

  For example:

```
com/mycompany/project/MyClass.java
```

9. Differentiate between class and interfaces. (4)

Feature	Class	Interface
Type	A blueprint for objects.	A contract for classes.
Instantiation	Can be instantiated (`new MyClass()`).	Cannot be instantiated directly.
Inheritance	Can extend only one class.	A class can implement multiple interfaces.
Constructors	Can have constructors.	Cannot have constructors.
Variables	Can have various access modifiers.	Variables are implicitly `public static final` .
Methods	Can have both concrete and abstract methods.	All abstract methods must be implemented by the class that implements it.

10. How to create user defined package in Java. Explain with an suitable example. (4)

To create a user-defined package in Java, you declare it with the `package` keyword at the top of your Java file. The file's location on your computer must match the package name.

Example

1. **Create a folder:** Make a folder named `myutils` .
2. **Write the class:** Inside that folder, create a file named `Calculator.java` .

Java

```java
// File: myutils/Calculator.java
package myutils;

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

3. **Use the package:** In a separate file (e.g., `Main.java`), you `import` the class to use it.

Java

```java
// File: Main.java
import myutils.Calculator;

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));
    }
}
```

11.) List out different ways to access package from another package. (4)

In Java, to use classes from another package, you must make them accessible using one of the following ways:

1. **Using fully qualified name** – Access the class with its complete package path.
2. **Using `import` statement** – Import specific class from a package.
3. **Using `import package.*`** – Import all classes from a package.
4. **Static import** – Import static members of a class directly.

Syntax & Example:

java Copy Edit

```java
// 1. Fully Qualified Name
mypack.MyClass obj1 = new mypack.MyClass();
obj1.show(); // Output from MyClass

// 2. Import Specific Class
import mypack.MyClass;
MyClass obj2 = new MyClass();
obj2.show(); // Output from MyClass

// 3. Import All Classes
import mypack.*;
MyClass obj3 = new MyClass();
obj3.show(); // Output from MyClass

// 4. Static Import
import static java.lang.Math.*;
System.out.println(sqrt(16)); // 4.0
```

12.) Enlist and explain any four inbuilt packages in Java. (4)

Java provides many pre-defined packages containing ready-made classes and interfaces to perform common tasks. Four commonly used inbuilt packages are:

1. `java.lang` – Contains fundamental classes like `Object`, `Math`, `String`, `Integer`. Automatically imported.
2. `java.util` – Provides utility classes like collections (`ArrayList`, `HashMap`), date/time, and more.
3. `java.io` – Contains classes for input and output operations, like `File`, `BufferedReader`, `PrintWriter`.
4. `java.sql` – Provides classes for database connectivity, like `Connection`, `Statement`, `ResultSet`.

Example:

java Copy Edit

```java
import java.util.ArrayList;
import java.io.File;
import java.sql.Date;

class PackageExample {
    public static void main(String[] args) {
        String str = "Java"; // java.lang
        System.out.println(str.toUpperCase()); // JAVA

        ArrayList<Integer> list = new ArrayList<>(); // java.util
        list.add(10);
        System.out.println(list); // [10]

        File file = new File("test.txt"); // java.io
        System.out.println(file.getName()); // test.txt

        Date date = new Date(System.currentTimeMillis()); // java.sql
        System.out.println(date); // current date
    }
}
```