DAY-4(21-07-2025)

SESSION-1

TOPICS DISCUSSED:

- Non-Access Specifiers

  (These are not used in constructor)

  - Static
  - Final
  - Abstract
  - Synchronized
  - Transient
  - Volatile
  - Native

- Strictfp

Common Non-Access Specifiers in Java:

Specifier	Used With	Purpose
static	Variables, Methods, Blocks	Belongs to the class, not instance. Shared among all objects.
final	Variables, Methods, Classes	Prevents change (value, overriding, inheritance).
abstract	Classes, Methods	Represents incomplete functionality. Must be extended/implemented.
synchronized	Methods, Blocks	Ensures thread safety by allowing only one thread at a time.
volatile	Variables	Ensures changes to a variable are visible to all threads.
transient	Variables	Prevents serialization of the variable.
native	Methods	Declares a method implemented in native code (like C/C++).
strictfp	Classes, Methods	Enforces consistent floatingpoint calculations across platforms.

Example for non-Access specifiers:

```
    class Example {

    static int count = 0;
final int ID = 100;
transient String password;

    synchronized void increment() {
        count++;
    }

    native void callNative();

    strictfp double compute() {
        return 10.0 / 3.0;
    }
}
```

JAVA METHODS:

A method in Java is a block of code that performs a specific task. You define a method once and reuse it whenever needed. It's like a function in other programming languages.

KEY POINTS IN JAVA METHODS:

To break big programs into smaller, reusable pieces.
To avoid repeating the same code.
To improve readability and maintainability.

Structure of a Java Method: modifier returnType methodName(parameters) { return value; // if returnType is not void
}

EXAMPLE 1:A Simple Method:

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

- public – access specifier (can be called from outside the class)
- int – return type (method returns an integer)
- add – method name
- (int a, int b) – parameters
- return a + b; – returns the result of the addition

Types of Methods:

PREDEFINED:

Already available in Java libraries

EXAMPLE: System.out.println(), Math.sqrt()

USER-DEFINED:

Created by the programmer

EXAMPLE: add(), multiply(), etc.

EXAMPLE 2: Calling a Method:

```
public class Calculator {

   public int multiply(int x, int y) {
      return x * y;
   }

   public static void main(String[] args) {
      Calculator calc = new Calculator();
      int result = calc.multiply(4, 5);
      System.out.println("Result: " + result);
   }
}
```

Method Components Explained

Part	Description
Modifier	public, private, static, etc. (access and behavior control)
Return Type	What type of value the method returns (e.g. int, void, String)
Method Name	Any valid identifier (should be meaningful)
Parameters	Input values the method accepts
Method Body	Code to execute
Return Statement	Used if the method returns a value

- In Java, methods can be **declared in four main ways**, depending on **return type**, **parameters**, and **whether the method returns a value or not**.

1. Method with No Return Type and No Parameters

```
void methodName() {
  // code
}
```

- Does **not return any value**.
- Takes **no input**.
- Just performs an action.

2. Method with No Return Type and With Parameters

```
void methodName(dataType param1, dataType param2) {
  // code
     }
```

- **Takes input (parameters).**
- **Does not return any value.**
- **Just does some task using the inputs.**

3. Method with Return Type and No Parameters

```
returnType methodName() {
  // code
```

```
    return value;
}
```

- Does not take inputs.
- Returns a value.

4. Method with Return Type and With Parameters

```
returnType methodName(dataType param1, dataType param2) {
    // code
    return value;
}
```

- Takes input.
- Returns a result after using the input.

Type	Example
1. No Return, No Parameters	void greet()
2. No Return, With Parameters	void displaySum(int a, int b)
3. Return, No Parameters	int getNumber()
4. Return, With Parameters	int multiply(int x, int y)

SESSION-2

INHERITANCE:

Inheritance is one of the core concepts of Object-Oriented Programming (OOP) in Java. It allows a class to inherit properties (fields) and behaviors (methods) from another class.

- **Inheritance is the process by which one class acquires the properties and behaviors of another class.**

KEY POINTS IN INHERITANCE:

- **To reuse code (no need to rewrite same methods)**
- **To support hierarchical classification**
- **To improve code readability and organization**
- **To support polymorphism (runtime method overriding)**
- **Inheritance lets one class reuse the fields and methods of another.**
- **Inheritance lets one class reuse the fields and methods of another.**
- **Encourages code reuse and modular design.**
- **Java supports single, multilevel, and hierarchical inheritance using classes, and multiple inheritance using interfaces.**

Term	**Meaning**
Super Class (Parent/Base)	**The class whose features are inherited**
Sub Class (Child/Derived)	**The class that inherits the features**
extends	**Keyword used to implement inheritance**

IS-A & HAS-A RELATIONSHIP:

IS-A and HAS-A are two important types of relationships used to design class structures.

IS-A Relationship (Inheritance):

- **The IS-A relationship represents inheritance. It means a subclass is a type of its superclass.**

- extends (for classes), implements (for interfaces)
- only inherits the properties

```
class Animal {

   void eat() {

      System.out.println("This animal eats food.");

   }

}


class Dog extends Animal {

   void bark() {

      System.out.println("The dog barks.");

   }

}
```

HAS-A Relationship (Composition / Aggregation):

- The HAS-A relationship represents composition or aggregation. It means one class contains another class as a member.
- creating an object inside another class
- containment

```
class Engine {

   void start() {

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

   }

}


class Car {

   Engine engine = new Engine();


   void startCar() {

      engine.start();

      System.out.println("Car is running");

   }
```

}

Types of Inheritance in Java:

Type	Description
Single Inheritance	One child inherits from one parent
Multilevel Inheritance	Child → Parent → Grandparent
Hierarchical Inheritance	Multiple children inherit from one parent
Multiple Inheritance (by classes)	One child inherits from multiple classes
Multiple Inheritance (using interfaces)	One class implements multiple interfaces

- **Java does not support multiple inheritance with classes to avoid the Diamond Problem, but it supports it via interfaces.**

Single Inheritance:

- **In single inheritance, one subclass (child) inherits from one superclass (parent).**

```
// Parent class (Super Class)
class Animal {
   void eat() {
      System.out.println("The animal eats food.");
   }
}

// Child class (Sub Class)
class Dog extends Animal {
   void bark() {
      System.out.println("The dog barks.");
```

```
    }
}

// Main class to test inheritance
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.bark();
    }
}
```

Explanation:

- Animal is the superclass.
- Dog is the subclass.
- Dog inherits the eat() method from Animal.
- Dog has its own method bark().

GENERALIZATION AND SPECIALIZATION:

1.Generalization in Java:

- Generalization is the process of extracting common features (fields and methods) from two or more classes and putting them into a general superclass.
- It helps reduce code duplication and makes your code more abstract and reusable.

```
class Animal { // Generalized class
    void eat() {
        System.out.println("This animal eats.");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
```

```
  }
}

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

Explanation:

- Animal is a generalized class — it represents common behavior (eat()).
- Dog and Cat are special classes with additional behavior.

2. Specialization in Java:

- Specialization is the process of creating a subclass from a superclass and adding more specific behavior or properties.
- This is the reverse of generalization.

```
class Vehicle {

  void start() {

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

  }

}


class Car extends Vehicle {

  void openSunroof() {

    System.out.println("Sunroof opened.");

  }

}
```

Explanation:

- Car is a specialized version of Vehicle with more specific features.

- **Vehicle gives general behavior, and Car adds specific behavior.**

Concept	**Meaning**	**Example**
Generalization	**Making a common parent class from multiple classes**	**Animal from Dog, Cat**
Specialization	**Creating a subclass with more specific behavior**	**Car extends Vehicle**