OBJECT ORIENTED PROGRAMMING THOUGH JAVA

B.Tech. - II Year, I Sem.

UNIT-II TOC (TABLE OF CONTENTS)

Introduction to CLASSES AND METHODS in Java

SYLALBUS:

General form of Class, Declaring Objects, Assigning Object Reference Variables, Methods and Constructors – all types, new operator, this keyword, garbage collection, finalize () method, Autoboxing and Unboxing, Wrapper, Scanner and Bufferedreader Classes.

Method Overloading, Objects and Parameters, Argument Passing, Return objects, Recursion, Access control, static, final, command line arguments, variable length arguments

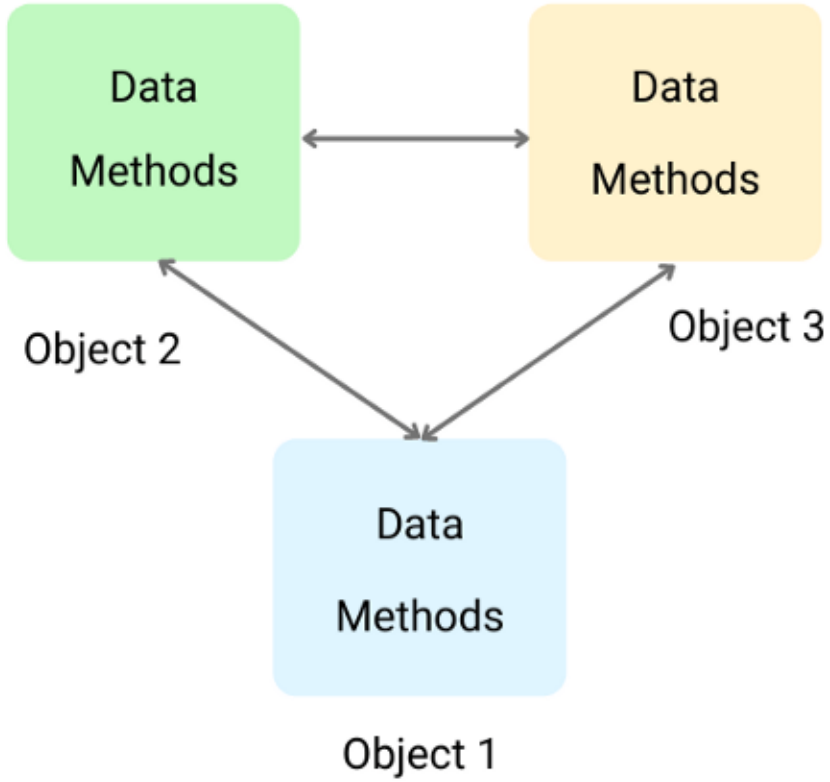
Object-Oriented Programming

The following are the OOP Concepts

- Object
- Class
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
- Dynamic Binding

Objects and Classes

- Object means a **real-world entity** such as a pen, chair, table, computer, watch, etc.
- OOP is a methodology or paradigm to design a program using **classes and objects.** It simplifies software development
- **Modularity and Reusability:**
  The OOP is an efficient method for design and development of software systems using **REUSABLE COMPONENTS** called as **OBJECTS.**
- In OOP, the program is divided into **set of objects.**
- In OOP **emphasis more on Data**.
- In general, an object is a **real-world entity** which contains **data** and **behavior**.
- An object in programming can be defined by using a **Class**.
- i.e.. An object is an **instance** of a class.
- A class is **Template** to create an object.
- Supports **Encapsulation, Inheritance , Polymorphism**



General Form of a Class

- A class is like a **blueprint**.
- The **blueprint** defines the **structure**
- A **class** defines the **structure** contains **variables (state)** and **methods (behavior).**
  - **int speed;** — A variable (property of the car).
  - **void drive()** — A method (function that performs an action).Interactive Idea:

```
class Car {
   int speed;        //variable
   String color;
      void drive()    //method
       {
      System.out.println("Driving");
      }
 }
```

- Activity: create a class for a gadget (e.g., Mobile, Fan) with one property and one method.

General Form of a Class

Class: A class is a **template** or **blueprint** from which objects are created

- It is a **user defined datatype** which contains common attributes and methods.
- A class in Java can contain:
  - **Data members or variables**
  - **Methods or member functions**

General Form of a Class:

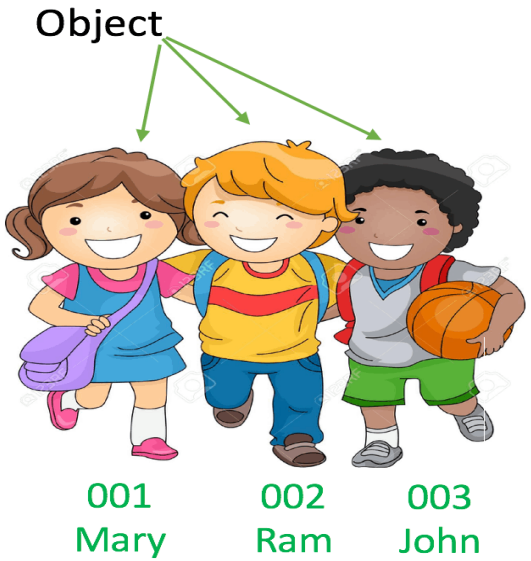
```
class class_name
    {
      //data members or variables
      dataType variable1;
      dataType variable2;
        ....
     // methods or member functions
      returntype methodName1(){..}
      returntype methodName2(){..}
       ....
     }
```

Ex.

```
class Student {
    int id;                 // data members
    String name;

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

Class Student

```
Int rno;
String name;

Void dips(){}
```



Declaring Objects

Creating an Object:

- An object is an **instance of a class**
- An **object** can be created using **new** keyword.
- The **new** keyword is used to allocate memory at runtime.

Syntax:

ClassName obj = new ClassName();

Here.

ClassName → name of the class(user-defined)
obj → reference variable
new → keyword to create object

Ex.

Car c1 = new Car();
Student s1=new Student();

Assigning Values to Object Fields:

- Once the object is created, we can assign values to its variables (fields) using the dot operator **(.)**

Syntax:

obj.variableName = value;

```
Car c1 = new Car();   // object creation

    c1.speed = 80;        // assigning value

    c1.drive();           // method call
  }
}
```

Creating an object

```
// creating class
class Employee
 {
    // data members
    int eid;
    String name;
    double sal;

    // method
    void display() {
        System.out.println("Employee ID: " + eid);
        System.out.println("Name: " + name);
        System.out.println("Salary: " + sal);
    }
}
```

```
// main class to use Employee
public class TestEmployee {
    public static void main(String[] args) {
        // creating object
        Employee e1 = new Employee();

        // assigning values to data members
        e1.eid = 101;
        e1.name = “Sachin";
        e1.sal = 55000.50;

        // accessing method
        e1.display();
    }
}
```

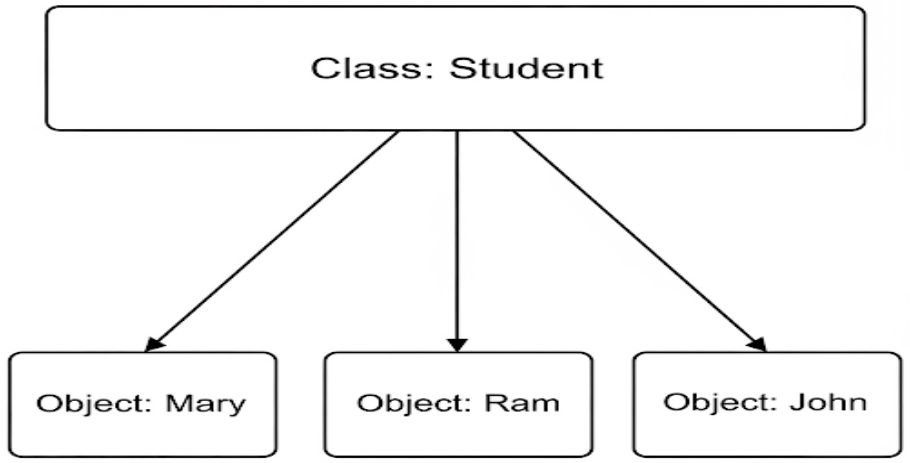
Creating multiple objects

```
// creating class
class Student {
    // data members
    int sid;
    String name;
    float cgpa;

    // method
    void display() {
        System.out.println("Student ID: " + sid);
        System.out.println("Name: " + name);
        System.out.println("Cgpa: " + cgpa;
    }
}
```

```
// main class to use Student
public class TestStudent {
    public static void main(String[] args) {
        // creating first object
        Student s1 = new Student();
        s1.sid = 201;
        s1.name = "Rahul";
        s1.cgpa = 8.5;

        // creating second object
        Student s2 = new Student();
        s2.sid = 202;
        s2.name = "Priya";
        s2.cgpa = 9.2;

        // accessing methods
        s1.display();
        s2.display();
    }
}
```

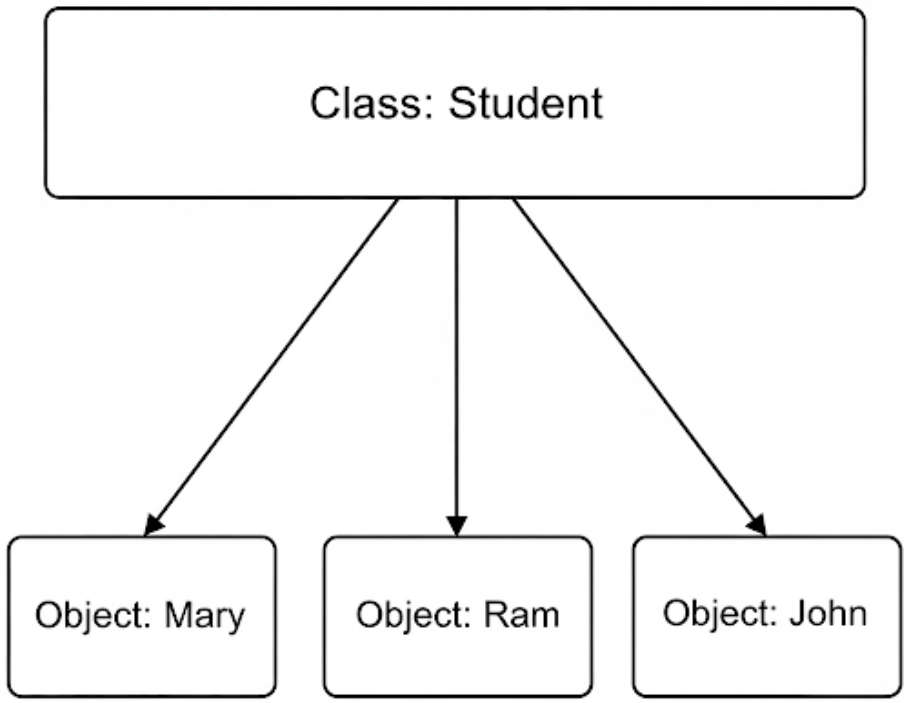
Creating object: Reading data using a method

```
// Define a class named "Student"
class Student {
    // Declare instance variables
     int rno;
     String name;

    // Method to read student details
    void read() {
     Scanner scanner = new Scanner(System.in);
       System.out.print("Enter Rno and Name: ");
       rno= scanner.nextInt();
       name = scanner.nextLine();
     }

    // Method to display student details
    void disp() {
       System.out.println("Student Name: " + name);
       System.out.println("Student Age: " + rno);
    }
}
```




```
// Main class to run the program
public class StudentTest {
    public static void main(String[] args) {

        // Create an object of the Student class
        Student s1 = new Student();

        s1.read();  //reading data using method
        s1.disp();
    }
}
```

Object vs class



Class:

- Template for declaring and creating objects
- Does not hold memory directly
- **Bind data as well as methods** together as a single unit
- Logical Entity (definition only)

Object:

- Object is an instance of a class .
- **Memory space is assigned to them.**
- Each **object has its own value**
- **Objects are like variables** of the class
- Physical Entities that occupy memory at runtime

Methods in Java

- A **block of code** designed to perform a specific task.
- It is used to achieve the **reusability** of code.

There are **two types of methods** in Java:

1. **Predefined Method**
2. **User-defined Method**

1.Predefined Method:

- Method **already defined in the Java class** libraries
- Also known as the **standard library method** or **built-in method.**
- Usage: We can directly use these methods in the **program at any point.**
- Some pre-defined methods are **max(), length(), equals(), compareTo(), sqrt(),** etc

Ex.

```
int result = Math.max(10, 20); // returns 20
```

Methods in Java

2) User-defined Method

- The method **written by the programmer** is known as **a user-defined** method.

Defining a user defined method

Syntax:

```
accessspecifier  datatype methodName(parameterlsit)
 {
   // body
 }
```

Ex.

```
//user defined method
public void deposit(float amt)
{     //method body
       if (amount > 0) {
                balance += amount;
             System.out.println(" Deposited.. New Balance = " + balance);
              } else
               {
                 System.out.println("Invalid deposit amount!");
               }
       }
```

Calling a User defined Method

- To call it, you must **create an object of the class** first.

Syntax:

```
ClassName obj = new ClassName();
obj.methodName(arguments);
```

Ex

```
Demo d = new Demo();
   d.greet("Naveen");
```

```
class Demo {

   // user-defined method
   void greet(String name)
   {
     System.out.println("Hello, " + name + "! Welcome to Java.");
   }

   public static void main(String[] args) {
      // create an object to call non-static method
      Demo obj = new Demo();

      // calling the user-defined method
      obj.greet("Naveen");
      obj.greet("Anusha");
   }
}
```

Initialization the data of an object:

There are 3 ways to initialize object in Java.

1. **By object reference variable**
2. **By method**
3. **By constructor**

```
1)  initialising object data with reference variable
class Student{
        int id;
        String name;
}
class TestStudent2{
    public static void main(String args[]){
          Student s1=new Student();
          s1.id=501;      //initialized through object reference
          s1.name="Naveen";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

501 Naveen

2) Initialization of data through method

The object of Student class are **initialized by invoking** the **read method**.

```
class Student{
        int rollno;
        String name;
          void read(){
            rollno=111;
            name=Karan;
           }
       void disp()
          {
          System.out.println(rollno);
          System.out.println(name);
          }
```

```
 public static void main(String args[]){
     Student s1=new Student();
          s1.read();    //initialize through method
          s1.disp();
         }
}
```

Initialization data through method

Two objects of Student class are **initialized by invoking** the **read method**.

```
class Student{
        int rollno;
        String name;
          void read(int r, String n){
            rollno=r;
            name=n;
           }
       void displ()
          {
          System.out.println(rollno);
          System.out.println(name);
          }
 }
```

```
class TestStudent{
 public static void main(String args[]){
     Student s1=new Student();
     Student s2=new Student();
   s1.read(111,"Karan"); //initialize through method with parameters
   s2.read(222,"Aryan");
        s1.disp();
        s2.disp();
 }
}
```

Initialization of object through Constructor

A constructor is used to initialize the object's data.

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

   Student(int i,String n)  //creating a constructor for initiating the data
    {
                id = i;
                name = n;
   }
   //method to display the values
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   //creating objects and passing values
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   //calling method to display the values of object
   s1.display();
   s2.display();
  }
}
```

Introduction to Constructors

A constructor is a **special method** used to initialize the objects of a class

Properties:

1. Its **name is the same name as the class name**.
2. **Invoked automatically** when the objects are created.
3. They **do not have return type**, not even void.
4. They **cannot be inherited**, though a **derived class can call the base class constructor.**

```
class class_name
{  …

        public  class_name(..)  //constructor
          {
            //code for initializing obj
          }
 };
```

```
class Person
{
  int age; string name;

   Person() // constructor
   {
    name =“Sachin”;     age=40 ; // object initialization
   }

   public static void main (String args[])
   {
      Person p= new Person(); //constructor is invoked
   }

};
```

Types of Constructors

TYPES OF CONSTRUCTORS

1. Default Constructors((No-Argument Constructor))

- A **constructor with out parameters.**
- **Created automatically by compiler if no constructor is defined.**

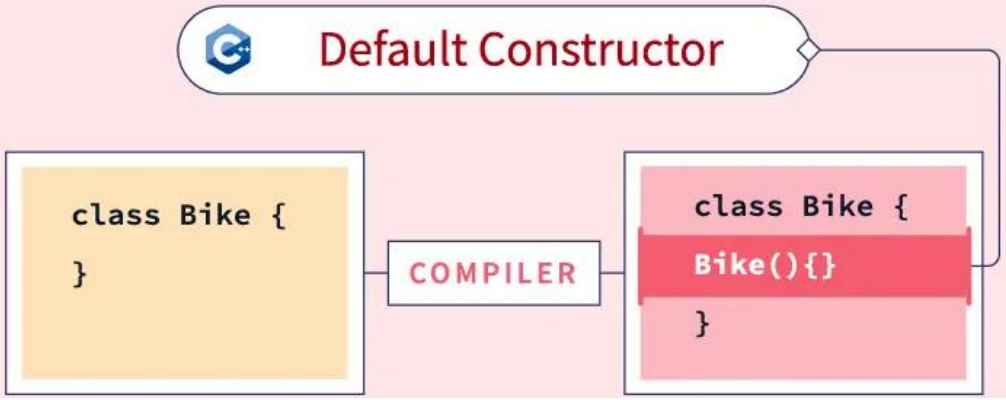
Ex. Bike(){ }

2. Parameterized Constructors

- **A constructor with parameters**
- **Accepts arguments to initialize object.**

Ex. Bike(String name, String model, int year)

3. Copy Constructor (User-defined) – copies values from another object.
creates a new object as a copy of an existing object



Parameterized Constructor:

```
Bike(String name, string model, int  year)
{
 …
}
Bike b= new Bike("Trek", "FX2", 2021);
```

1.Default Constructor

- A constructor **which accepts no parameters** and is called a default constructor.

- If there is no constructor for a class, the **compiler implicitly creates a default constructor.**

```
class class_name
 {
   …
  class_name ()
   {
     //code for initiliztion.
   }
};
```

- **Example**:

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

    // Default constructor
    Student() {
        rollno = 0;
        name = "Unknown";
    }
}
```

2. Parameterized Constructors

- A constructor that **can take parameters** and is called a parameterized constructor.
- It is used to initialize objects with a different set of values.

```
Class classname
{
…

      classname (parameter list) //parametrized constructor
      {
        //code for object initialization using parameters
      }
};
```

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

   // Parameterized constructor
   Student(int r, String n) {
      rollno = r;
      name = n;
   }
 public static void main()
{
        Studetn s1(110, “sachin”);
        Studetn s2(111, “virat”);
        System.out.println(s1.rno);
        System.out.println(s2.name);
}
 }
```

3. Copy Constructor

- A particular constructor used for the **creation of an existing object.**
- A copy constructor is a constructor that **creates a new object as a copy of an existing object.**
- Although Java does not provide a built-in copy constructor, you can create one manually.

```
class Student {
    int rollno;
    String name;
     // Parameterized constructor
    Student(int r, String n) {
        rollno = r;
        name = n;
    }
   // Copy constructor
    Student(Student s) {
        rollno = s.rollno;
        name = s.name;
    }
}
```

Method overloading

- If a class has **more than one method with the same name** but different parameter lists, it is called method overloading.".

Different ways to overload the method

1. By changing **the number of parameters**
2. By Changing the **Type of Parameters**
3. By Changing the **Order of Parameters**

1) By Changing the Number of Parameters

```
public class Adder {

   int add(int a, int b) {
         return a + b;
     }
   int add(int a, int b, int c) {
         return a + b + c;
     }
psvm(String args[])
{
   Adder a=new Adder();
    a.add(10,20); //two arguments
    a.add(10,20,30);//three arguments

 }
```

Overloading methods and constructors

2) By Changing the Type of Parameters

Changing **type** : Java chooses the **correct method at compile time.**

```
public class Calculator {
  public int add(int a, int b) {
      return a + b;
   }
  public double add(double a, double b) {
      return a + b;
   }
   public static void main(String[] args) {
      Calculator calc = new Calculator();
      S.o.p (calc.add(10, 20));       // Calls add(int, int)
      S.o.p(calc.add(10.5, 20.5));   // Calls add(double, double)
   }
}
```

3. By Changing the Sequence of Parameters

Changing **order** → Java **differentiates methods even with the same parameter count, as long as the sequence differs**

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

   public static void main(String[] args) {
      Calculator calc = new Calculator();
     S.o.p(calc.add(10, 20.5));     // Calls add(int, double)
     S.o.p(calc.add(20.5, 10));     // Calls add(double, int)
   }
}
```

Constructor Overloading

- A class can have **more than one constructor with different parameter lists.**

```
className() {
   // Constructor body
}
className(paramType1 p1) {
   // Constructor body
}
className(paramType1 p1, paramType2 p2) {
   // Constructor body
}
```

```
class Student {
   String name;     int age;
     Student() {
   }
   Student(String name) {   // Constructor with one parameter
      this.name = name;
      this.age = 0;
   }
Student(String name, int age) {  // Constructor with two parameters
      this.name = name;
      this.age = age;
   }
public static void main(String[] args) {
      Student s1 = new Student();
      Student s2 = new Student("Alice");
      Student s3 = new Student("Bob", 20);

      S.o.p(s1.name + " " + s1.age);   // null 0
      S.o.p(s2.name + " " + s2.age); // Alice 0
     S.o.p(s3.name + " " + s3.age); // Bob 20
   }
}
```

new Operator

- The **new operator** in Java is a keyword used to create objects dynamically or to create new arrays

The new operator is used to:

- **Allocate memory** on the heap.
- **Call the constructor**.
- **Return a reference** to the object.

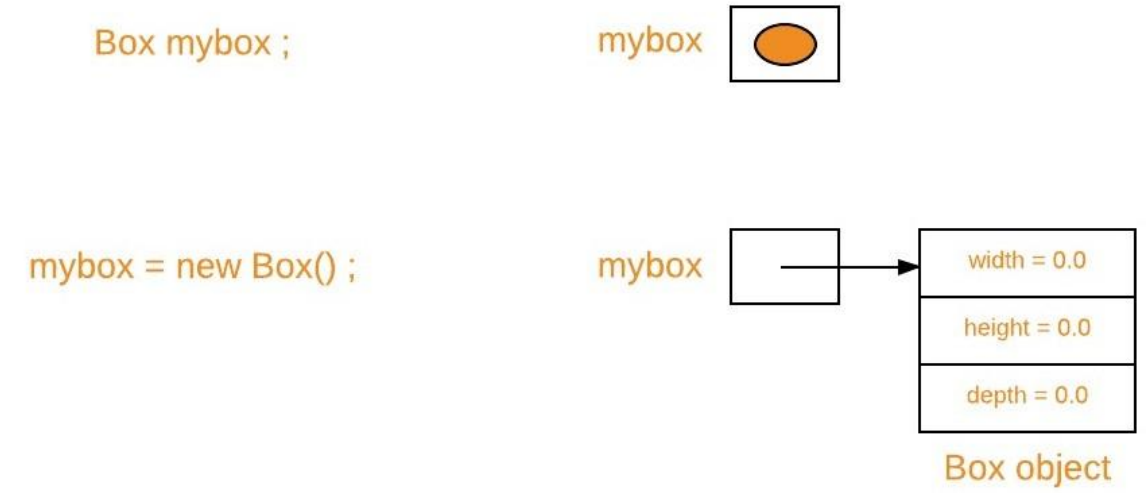
```
class Box
{
   double width;
   double height;
   double depth;
}
```

Syntax :

ref-var-name = new class-name();

Box mybox=new Box();

Box mybox; // only reference
mybox=new Box(); //memory allocated



Declaration, Instantiation and Initialization of an object of type Box

new Operator

Creating an Array Using new operator:

```
int[ ] arr = new int[5]; // creates array of size 5
    arr[0] = 10;
    arr[1] = 20;

   System.out.println(arr[0]); // 10
   System.out.println(arr[1]); // 20
```

👉 new int[5] creates an array in heap memory with default values 0.

Creating a string using new operator:

```
String s1 = new String("Java");
System.out.println(s1);
```

String s1 = new String("Java");

A **new object** is created in **heap memory**

this Keyword

- **this keyword is a special reference variable in Java.**
- **It always refers to the current object of the class.**
- **Avoids confusion when parameter names and instance variables are the same.**

Uses of this

- To differentiate **instance variables** from **parameters**.
- To call another **constructor** in the same class.
- To call **current class methods**.
- To **return the current object**.
- To **pass current object** as a parameter

1) To differentiate **instance variables** from **parameters**.

```
class Person {
    String name;
    Person(String name)
    {
      this.name = name;
    }
}
```

Garbage Collection

- Java handles **Automatic memory management.**
- In C/C++, developers must **manually free memory** using free() / delete.
- Objects with **no active reference** become eligible for **garbage collection (GC)**.
- JVM periodically runs the **Garbage Collector** to free unused memory.
- Developers can request GC using **System.gc(),** but JVM decides the actual execution

An object is **eligible for GC** when:

- **No reference variable** refers to it.
- **Reassigning a reference:**
- It is explicitly set to null.
- It goes out of scope.

Garbage Collection

Cases where objects are garbage collected

1. No reference to an object:

```
class Demo {
    public static void main(String[] args) {
        new String("Hello");  // no reference → eligible for GC
    }
}
```

2. Reassigning a reference:

```
class Test {
    public static void main(String[] args) {
        String s1 = new String("Hello");
        s1 = new String("World");  // "Hello" is now garbage
    }
}
```

3. Nullifying a reference:

```
Employee e = new Employee();
e = null;  // eligible for GC
```

4. Object going out of scope:

```
void show() {
    Student s = new Student("Naveen");
} // after method ends, s becomes garbage
```

Garbage Collection & finalize()

finalize() method:

- finalize() is called **before object is destroyed .**
- finalize() is a method in the Object class.
- Used for **cleanup operations** (like closing files, releasing resources).

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

```
class Demo {
    @Override
    protected void finalize() {
        System.out.println("Finalize called before GC");
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        d = null;   // object eligible for GC
        System.gc(); // request for GC
    }
}
```

Autoboxing and Unboxing

Autoboxing:

Automatic **conversion of a primitive type** (like int, double, char) into its **wrapper class object** (Integer, Double, Character).

Primitive → Wrapper

e.g., int → Integer

Unboxing:

Automatic conversion of a **wrapper object** back into its **primitive type**.

- Wrapper → Primitive

```
Integer i = 5; // Autoboxing
int j = i;    // Unboxing
```

Wrapper Classes

- Wrapper classes in Java are used to **convert primitive data types (like int, double, char, etc.) into objects.**

- Each primitive type has a corresponding wrapper class in java.lang package.

- Useful when working with **collections (e.g., ArrayList, HashMap)** because they only store objects, not primitive.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Wrapper Classes

```
public class WrapperDemo {
    public static void main(String[] args) {
        // Using constructors (older way, deprecated in Java 9+)
        Integer x = new Integer(100);
        Double y = new Double(5.6);

        // Using valueOf() method (recommended way)
        Integer a = Integer.valueOf(50);
        Double b = Double.valueOf(6.78);

        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
}
```

Scanner and BufferedReader Classes

Scanner: **Used for reading input.**

- **Easy to use for basic input (numbers, strings).**
- **Provides methods like nextInt(), nextDouble(), nextLine().**
- **Slower for large input.**
- **Available in java.util package**

Ex.

```
Scanner sc = new Scanner(System.in);
String name = sc.nextLine();
```

BufferedReader: **Used with InputStreamReader.**

- **Faster for large input (file reading).**
- **Must be used with InputStreamReader.**
- **readLine()** reads **a full line of text** from the input
- **Reads input as String → needs parsing for numbers.**

//input using BufferedReader

```
import java.io.*;

public class Example {

    public static void main(String[] args) throws IOException {

        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

        System.out.print("Enter your age: ");

        int age = Integer.parseInt(br.readLine());

        System.out.println("Your age is: " + age);

    }

}
```

Objects and Parameters

- An **object is passed as a parameter** to a method in java.
- To **update object properties** inside methods.
- changes made inside the method affect the original object.

Ex:

Pass a Student object to a method and update their name.

```
class Student {
    String name;

    Student(String name) {
        this.name = name;
    }

    void change(Student s)// object as parameter
    {
        s.name = "Updated";
    }

    public static void main(String[] args) {
        Student st = new Student("Original");
        System.out.println("Before: " + st.name);
        st.change(st);  //passing object as argument
        System.out.println("After: " + st.name);
```

OUTPUT:

Before: Original

After: Updated

Argument Passing, Return Objects

- You can return objects from methods.
- Reinforces modular code and reuse.

```
class Student {
   String name;

   Student(String name) {
      this.name = name;
   }

   // Method returning a Student object
   Student getStudent() {
      return new Student("Ravi");
   }

   public static void main(String[] args) {
      Student s1 = new Student("Initial");
      Student s2 = s1.getStudent();   // call method

      System.out.println("Returned Student: " + s2.name);
   }
}
o/p
Returned Student: Ravi
```

Recursion

- A **method** calling itself is called recursion.
- It is used when a big problem can be divided into smaller, similar sub-problems.
- Ex. factorial, Fibonacci, etc.

```
int fact(int n) {
    if (n <= 1)
        return 1;      // base case
    else
        return n * fact(n - 1);  // recursive case
}
```

A recursive method requires:

1.Base Case – condition to stop recursion.

- Prevents infinite calls.
- Example: if (n <= 1) return 1;

2.Recursive Case – method calls itself with a smaller/simpler input.

- Breaks the problem into sub-problems.
- Example: return n * fact(n - 1);

Recursion

```
//Fibonacci Series Example
class Fibonacci {
    int fib(int n) {
        if (n <= 1) return n;           // Base case
        return fib(n - 1) + fib(n - 2);  // Recursive case
    }

    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();   // Object creation
        int n = 6;
        System.out.print("Fibonacci Series: ");
        for (int i = 0; i < n; i++) {
            System.out.print(f.fib(i) + " ");
        }
    }
}
```

Access Control (Access Modifiers)

Access modifiers define the **scope/visibility** of classes, variables, methods, and constructors.

In Banking apps, **private double balance;** ensures direct access is blocked, but **public void deposit()** allows controlled update

Advantages:

- Provides **data security** (encapsulation).
- Controls **who can access/modify data**.
- Supports **code reusability** with controlled sharing across packages.

Modifier	Where Accessible	Example
public	Anywhere	public void login()
private	Only inside class	private String password;
protected	Same package + subclasses	protected void calculate()
default	Only inside package	String dept;

static

Static means the member **belongs to the class**, not the object.

Allows **variables and methods to belong to the class itself** rather than individual instances

Properties:

1. **Belongs to the class**
2. **Accessed without creating an instance**
3. **Shared among all instances**
4. **Can access other static members directly**
5. static variables are initialized only once

Advantages:

- Saves memory (shared by all objects).
- Allows calling methods/variables **without object creation**.
- Best for **common properties/utility methods**.

```
class Student {
    String name;
    static int count = 0;  // static variable shared by all objects
  Student(String name) {
        this.name = name;
        count++;  // increment static counter
        System.out.println("Student:" +count);
    }
    // Static method - can be called without object
    static void showCount() {
        System.out.println("Total Students: " + count);
    }
    public static void main(..) {
        Student s1=new Student("Ravi");
        Student s2=new Student("Anusha");

        // Display total count using static method
        Student.showCount();
    }
}
```

o/p: Student: 1
Student: 2
Total Students: 2

static

The *static* keyword is a non-access modifier in Java that is applicable for the following:

I. Variables

II. Methods

III. Blocks

IV. Nested Classes

i) Static Variables:

- These are **associated with the class itself** rather than with instances of the class.
- Declared using **static keyword** and are **shared among all instances of the class**.

- We can create static variables at the class level only

```
public class MyClass {
public static int count = 0; // Static field
 }
```

ii) Static Methods:

- **Static methods called with out object**
- **Can't access instance-level members directly** because they don't have access to an instance of the class.

```
class Calculator {
   // static method
   static int square(int n) {
      return n * n;
   }
}

public class Test {
   public static void main(String[] args) {
      // call without creating object
      System.out.println("Square of 5: " + Calculator.square(5));
      System.out.println("Square of 9: " + Calculator.square(9));
```

static

iii) Static Blocks:

- Used to initialize static fields or perform any other static setup when the class is loaded.
- A **static block** in Java runs **only once**, when the class is first loaded into memory.

```
class Database {
    static String connection;
    static { //static block
        connection = "Connected to MySQL Database";
        System.out.println(connection);
    }
}

public class Test {
    public static void main(String[] args) {
    Database db1 = new Database(); //static block runs only once
    Database db2 = new Database(); // static block won't run again
    }
}
```

iv) Static nested class:

A class defined inside another class with static keyword.

- **Accessed** without creating an object of the outer class.
- **Useful for grouping classes that are logically related**

```
class Outer {
    // static nested class
    static class Inner {
        void show() {
            System.out.println("Hello from Inner class!");
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner(); // no Outer object needed
        obj.show();
    }
}
```

Final keyword in Java

The **final** keyword in Java is used to make **variables, methods, or classes** unchangeable.

Usage of final

1. **final variable** → Value **cannot be changed (constant).**
2. **final method** → **Cannot be overridden** by subclasses.
3. **final class → Cannot be inherited.**

i) final variable → creates constant

final variable value cannot be changed once it has been initialized

```
final class Bank {        // final class → cannot be extended
   final double interestRate = 7.5; // final variable → fixed value

   final void showRate() {      // final method → cannot be overridden
      System.out.println("Interest Rate: " + interestRate + "%");
   }
}

public class FinalExample {
   public static void main(String[] args) {
      Bank b = new Bank();
      b.showRate();
   }
}
```

final classes and methods - Preventing inheritance:

ii) final method:

final method → prevents method overriding

A final method cannot overridden.

```
class Bike{
 final void run(){  System.out.println("running");   }
}

class Honda extends Bike{
  void run(){   //can't overridden
     System.out.println("running safely with 100kmph");
  }
  public static void main(String args[]){
  Honda h= new Honda();
  h.run();
  }
}
```

Output:Compile Time Error

iii) final class → prevents inheritance

A final **class can't be extended**

```
final class Bike{
}

class Honda1 extends Bike{  //can't be inherted
  void run(){
      System.out.println("running safely with 100kmph");
   }
  public static void main(String args[]){
  Honda1 h= new Honda1();
  h.run();
  }
}
```

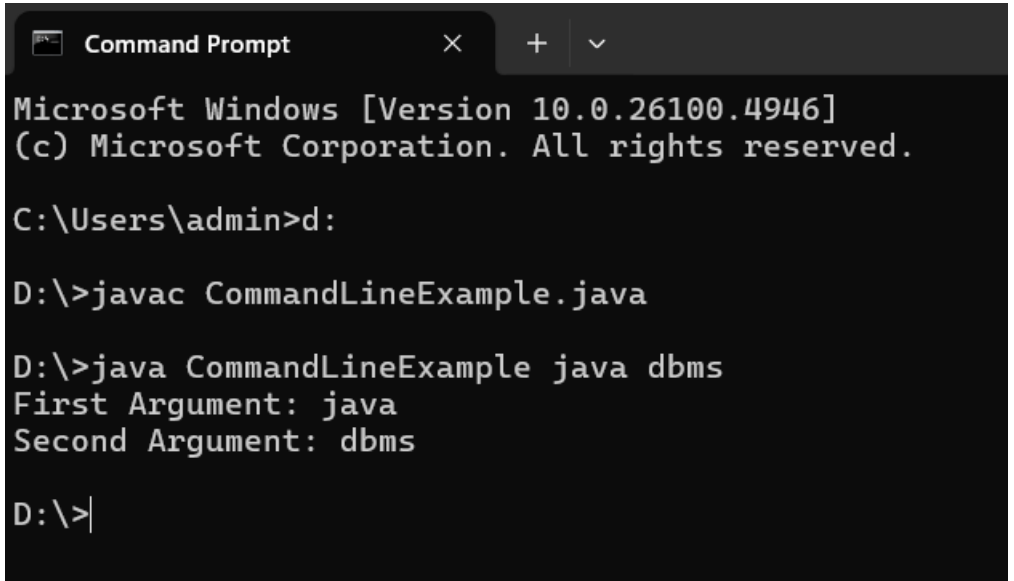
Output:Compile Time Error : can't extend final class Bike

Command Line Arguments

- **Command Line Arguments are** the **values passed to the main()** method when a program is **executed from the terminal.**
- **They are stored in the String[] args array.**

Usage:

- **To pass input values without using Scanner or BufferedReader.**
- **Useful in automation, scripts, batch processing.**




```
public class CommandLineExample {
    public static void main(String[] args) {
        // args[0], args[1] ... contain inputs
        System.out.println("First Argument: " + args[0]);
        System.out.println("Second Argument: " + args[1]);
    }
}
```

Varargs (Variable Arguments)

- **Varargs (...)** allows a method to accept multiple arguments of the same type without explicitly creating arrays.

Usage:

- **Simplifies method calling.**
- **Useful when the number of inputs is unknown (ex: summation, logging**

```
public class VarargsExample {
    // Method with varargs
    static int sum(int... numbers) {
        int result = 0;
        for (int n : numbers) {
            result += n;
        }
        return result;
    }

    public static void main(String[] args) {
System.out.println("Sum: " + sum(10, 20)); // 2 arguments
 System.out.println("Sum: " + sum(5, 10, 15, 20)); // 4 arguments
    }
}
```