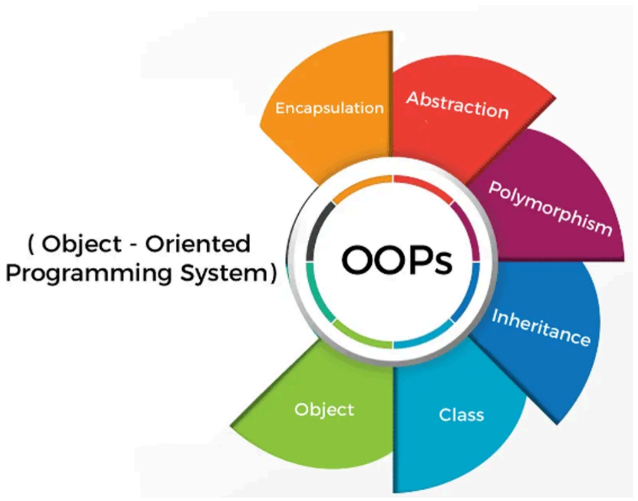
Object-oriented concept in Java:



Programming Paradigms

- To develop a computer application, we use different **programming paradigms** (also called *styles of programming*). A programming paradigm defines **how a program is structured and how problems are solved**, not the syntax itself.
- Common programming paradigms include:
  - **Procedural Programming**
  - **Functional Programming**
  - **Object-Oriented Programming (OOP)**

Important Note:

- Programming paradigms are **not programming languages**. Instead, **programming languages are classified based on the paradigms they support**.

- Java supports **multiple programming paradigms**, such as:
  - Procedural (using methods and control structures)
  - Object-Oriented (core paradigm)

    - Functional (using lambda expressions and streams)

- In a **single application**, we may use **different paradigms** to solve different types of problems.

What is OOP?

- **Object-Oriented Programming (OOP)** is a programming paradigm that is centered around the concept of **objects**.
- In OOP:
    - Software is designed using **objects**
    - Objects are created from **classes.**
    - A **class** acts as a **blueprint or template** for creating objects.
    - Objects represent **real-world entities.**

Key Concepts in OOP

1. Objects

- Objects are the **core building blocks** of OOP.
- An **object** is an instance of a class and represents a real-world entity.
- Each object has **two main components**:

    a) State (Attributes)

    - Represents the **data** of an object
    - Also called **fields** or **properties.**
    - Example (Human object):
        - name
        - age
        - height

        - weight

b) Behavior (Methods)

- Represents the **actions** an object can perform
- Defines how an object can **operate on its data**
- Example (Human object):
    - walk()
    - talk()
    - run()
    - sleep()

---

2. Classes

- A **class** is a **blueprint or template** used to create objects.
- A class defines:
    - **Attributes** (what data the object will have)
    - **Methods** (what actions the object can perform)
- From one class, we can create **multiple objects**, each with **different data values**.

Example:

- Class: `Human`
- Objects: You, your friend, your teacher
  (All are humans, but with different names, ages, and heights)

Real-World Analogy of OOP

- In the real world, everything can be seen as an **object**, such as:
    - Humans

    - Animals
    - Vehicles
    - Trees
    - Books
- Even though all humans share common characteristics, their **values differ**.
- This similarity + variation is perfectly modeled using **classes and objects** in OOP.

Application of OOP in Real Projects

- OOP is widely used in real-world software development.

  Example: Banking Application

  Possible classes:

    - Customer
    - Account
    - Loan
    - Transaction
    - Card

  Customer class

  - Properties: name, address, phone number
  - Behaviors: openAccount(), transferFunds()

  By modeling software around real-world entities, OOP makes applications:

  - Easier to understand
  - Easier to manage
  - Easier to scale

Features of Object-Oriented Programming (OOP)

- OOP provides several powerful features that improve **code quality, reusability, and maintainability**.

1. Encapsulation

Feature:

- Encapsulation is the process of **bundling data (attributes) and methods (behaviors)** into a single unit called a **class**, and restricting direct access to data.

Benefits:

- **Data Hiding:** Prevents unauthorized or accidental access
- **Controlled Access:** Data is accessed only through valid methods
- **Improved Security:** Protects object integrity

---

2. Inheritance

Feature:

- Inheritance allows a **child class (subclass)** to acquire properties and methods of a **parent class (superclass)**.
- The subclass can:
    - Reuse existing functionality
    - Add new features
    - Modify inherited behavior

Benefits:

- **Code Reusability:** Avoids duplication

- **Extensibility:** Easy to extend existing systems
- **Maintainability:** Changes in the parent class propagate automatically

---

3. Polymorphism

Feature:

- Polymorphism allows **one interface to represent multiple forms**.
- It enables a method to behave differently based on the object that calls it.
- Polymorphism can be achieved through:
  - **Method Overriding:**
    Subclass provides its own implementation of a superclass method
  - **Method Overloading:**
    Same method name with different parameters in the same class

Benefits:

- **Flexibility:** The same method works for different objects
- **Scalability:** Easy to add new behaviors
- **Clean Code:** Reduces conditional logic

---

4. Abstraction

Feature:

- Abstraction hides **complex implementation details** and exposes only **essential functionality**.
- It focuses on **what an object does**, not **how it does it**.

Benefits:

- **Simplified Interfaces:** Reduce complexity for users
- **Better Readability:** Cleaner and more understandable code
- **Error Reduction:** Limits direct interaction with internal logic

---

What is a Class?

- A **class** in Java is a **blueprint or template** used to create objects.
- It defines:
    - **Attributes (fields)** → represent the *state* of an object
    - **Methods** → represent the *behavior* of an object.
- Every object created from a class is called an **instance** of that class, and each instance has its **own copy of instance variables**.

Basic Syntax of a Class:

```
public class ClassName {
        // instance variables (represent the state of an object)
        // methods (represent the behavior of an object)
}
```

Explanation:

- `public` → access modifier (optional)
- `ClassName` → user-defined name
    - Must be a **valid Java identifier**
    - Should follow **PascalCase** naming convention
- `{ }` → class body
    - All fields, methods, constructors, and blocks must be inside these braces.

Components of a Java Class

A class may contain:

1. Fields (variables/attributes)
2. Methods (functions)
3. Constructors
4. Instance Initializers (blocks)
5. Static Initializers (blocks)

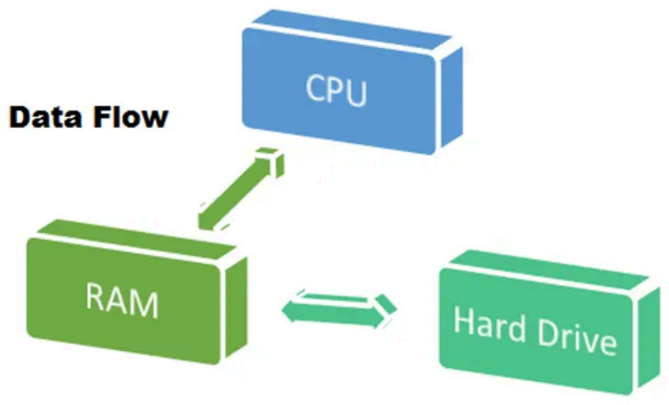
Computer Memory (Basic Understanding):

Main Components of a Computer System

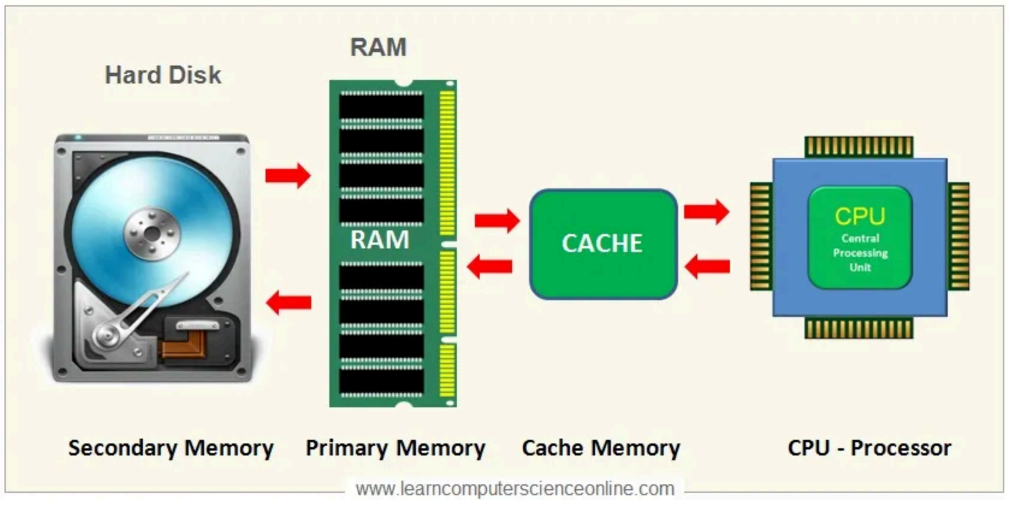
- **CPU (Processor)** – Executes instructions
- **RAM (Primary Memory)** – Temporary memory used during execution
- **Hard Disk (Secondary Memory)** – Permanent storage (files, programs, .class files)

Types of Memory:

Component	Role
Processor (CPU)	Executes instructions
Primary Memory (RAM)	Holds data and code during execution
Secondary Memory (Hard Disk)	Stores programs permanently



Computer System - Primary , Secondary And Cache Memory



Instance Variables in Java

What is an Instance Variable?

- An **instance variable** is a variable:
    - Declared **inside a class**
    - Declared **outside methods, constructors, and blocks**
    - Belongs to an **object**, not the class
- Each object gets its **own copy** of instance variables.

Key Characteristics of Instance Variables

1. **Runtime Allocation**
   Memory is allocated **when an object is created**, not at compile time.

2. **Object-Specific**
   Each object has its own independent copy.

3. **Default Initialization**
   If not explicitly initialized:

   - `int` → `0`
   - `double` → `0.0`
   - `boolean` → `false`
   - reference → `null`

Example: Instance vs Local Variable:

```
package com.masai;
public class Main {

  int x = 10;   // instance variable

  public static void main(String[] args) {

      int y = 10; // local variable

      System.out.println(y); // OK

      // System.out.println(x); // ERROR
  }
}
```

Why Error?

- `main()` is **static**
- `x` is **non-static**

- static methods **cannot directly access non-static members**

Static vs Non-Static Members

Two Types of Members in a Class

1. Static Members

- Belong to the **class**
- Shared by **all objects**
- Loaded when the **class is loaded**
- Memory is allocated at **Method area(MetaSpace)**.

2. Non-Static Members

- Belong to the **object.**
- Created **only when an object is created**

JVM Memory View:

Component	Stored In
Class metadata, static variables	Method Area (Metaspace)
Objects & instance variables	Heap Memory
Local variables, method calls	Stack Memory

Why Do We Need Objects?

1. To allocate memory for non-static members dynamically.

2. To access instance variables and methods.

Object Creation in Java:

Syntax:

```
ClassName refVar = new ClassName();
```

Example:

```
Main obj = new Main();
```

What Happens When `new` is used?

1. JVM allocates memory in the heap
2. Instance variables get default values.
3. Constructor is executed
4. Reference variable stores the address of the object

Accessing Non-Static Members:

```
Main obj = new Main();
System.out.println(obj.x);
```

Object and Reference Variable:

- **Object** → Actual memory in heap
- **Reference variable** → Stores the address of the object

Multiple Objects Example:

```
Main obj1 = new Main();

obj1.x = 50;
```

```
Main obj2 = new Main();

System.out.println(obj2.x); // 10
```

- Each object has its **own copy** of instance variables.

Multiple References to Same Object:

```
Main obj1 = new Main();

obj1.x = 50;

Main obj2 = obj1;

System.out.println(obj2.x); // 50
```

- Both references point to the same object.

NullPointerException:

- Occurs when:
  - Reference variable is `null`
  - You try to access the instance members

Example:

```
Main obj = null;
System.out.println(obj.x); // Runtime Error
```

Garbage Collection (GC)

- **Garbage Collection (GC)** is a process in Java that **automatically manages memory**.
- Java developers do **not** need to delete objects manually. The **JVM (Java Virtual Machine)** automatically removes unused objects from memory.

Main purpose of GC:

- To free heap memory by destroying objects that are no longer needed, preventing memory leaks.
- An object is considered **garbage** if:
    - It is no longer referenced
    - It cannot be accessed by any part of the program

Example:

```
Main obj1 = new Main();
obj1= null; //  object becomes eligible for GC
```

When Does an Object Become Eligible for GC?

- An object becomes eligible for garbage collection when **no references point to it**.

Common Scenarios:

1. **Nullifying the reference**

    ```
    Main obj1 = new Main();
    obj1= null; //  object becomes eligible for GC
    ```

2. **Reassigning reference**

    ```
    Main obj1 = new Main();
    obj1 = new Main(); // first object eligible for GC
    ```

3. **Object created inside a method**

```
void create() {
        Main obj = new Main()
} // obj goes out of scope
```

4. **Anonymous object**

```
new Main(); // eligible immediately
```

How Garbage Collection Works

1. JVM identifies **reachable objects**
2. Unreachable objects are marked as garbage
3. The garbage collector removes them
4. Memory is reclaimed and reused

- GC runs **automatically**, but timing is **not guaranteed**.

Requesting Garbage Collection

- You can *request* GC, but the JVM may ignore it.

```
System.gc();
Runtime.getRuntime().gc();
```

- These are **requests**, not commands.

finalize() Method and GC

- This method belongs to the **java.lang.Object** class.
- We need to override this method inside our class to perform the cleanup logic.

```
protected void finalize() {
   // cleanup code
}
```

- Called before the object is garbage collected.
- The finalize() method is deprecated and should not be used.

- There is no guarantee when or even if it will be called.
- Modern Java uses try-with-resources or explicit close() methods.
- We will talk about this method once again when we discuss the **Inheritance** and the method **overriding** concepts.

Task:

Q1/- How many objects are eligible for GC?

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

Q2/- Is the following object eligible for GC?

```
Student s1 = new Student();
Student s2 = s1;
Student s3 = s2;
s1 = null;
s2 = null;
```

State and Behavior of an Object

- **State** → Instance variable values
- **Behavior** → Methods operating on state

Example:

Song.java

```
package com.masai;
public class Song {

  String artist;
  String title;

// Behavior: play method that uses the state
  void play() {
     System.out.println(artist + " is singing " + title);
  }

  public static void main(String[] args) {
     // Creating first object
     Song track1 = new Song();
     track1.artist = "Lata";
     track1.title = "Vande Mataram";
     track1.play(); // Output: Lata is singing Vande Mataram

     // Creating second object
     Song track2 = new Song();
     track2.artist = "Sukhwindar";
     track2.title = "Jai Ho";
     track2.play(); // Output: Sukhwindar is singing Jai Ho

  }
}
```

Static Members in Java:

Key Points

- Declared using `static`
- Only **one copy** exists.
- Shared by all objects
- Belong to the **class**

Static vs Non-Static (Summary Table):

Static	**Non-Static**
Belongs to the class	Belongs to the object
Access using `ClassName.member`	Access using an `object.member`
Static variable initializes with default values at the time of class loading.	A non-static variable is initialized with default values at the time of object creation
All objects of a class share a single copy of static variables	Each object has one local copy of the non-static variables.

Example:

For a Banking Application:

- **Static variables:** bankName, branchName, ifscCode, phone, bankAddress
- **Non-static Variables: (each customer's) accountNumber, name, phone, address**

Main.java:

```
package com.masai;
public class Main{

        static int x;
        int y;

        public static void main(String[] args){

                Main obj1 = new Main();

                obj1.x=10;
                obj1.y=20;

                Main obj2 = new Main();

                obj2.x=50;
                obj2.y=80;

                System.out.println(obj1.x); //50
                System.out.println(obj1.y); //20
                System.out.println(obj2.x); //50
                System.out.println(obj2.y); //80

        }
}
```

Static method vs Non-static method

- The Common functionality of all objects in the application must be defined as **static**.
- Non-static functionality belongs to a particular object.

Example: For an ATM application:

- **Static method:** To launch a welcome screen
- **Non-static method:** To launch a transaction screen

Accessing static members of a class:

- We can access static members of a class by using the following 3 ways:
    1. Directly by their name (only inside the same class)
    2. By using the object
    3. **By using the class name (even inside another class).**

Example1:

<u>Main.java:</u>

```
package com.masai;
public class Main
{
        static int i=10;

        public static void fun1(){
                System.out.println("inside static fun1 of Demo");
        }

        public static void main(String[] args){

                //using directly
                System.out.println(i);
                fun1();

                //by using class name
                System.out.println(Main.i);
                Main.fun1();

                //using by an object
                Main obj=new Main();
                System.out.println(obj.i);
                obj.fun1();
        }
}
```

Example2:

<u>Main.java:</u>

```java
package com.masai;
public class Account
{
        static String bankName;

        long accountNumber;
        String accountHolderName;
        double balance;

        public static void main(String[] args){

                        Account.bankName="SBI";

                        Account ac1=new Account();
                        ac1.accountHolderName="Ramesh";
                        ac1.accountNumber=13422323432L;
                        ac1.balance=5000;

                        Account ac2=new Account();
                        ac2.accountHolderName="Amit";
                        ac2.accountNumber=25422323432L;
                        ac2.balance=6000;


                        System.out.println("Account 1 details");
                        System.out.println(ac1.bankName);
                        System.out.println(ac1.accountHolderName);
                        System.out.println(ac1.balance);

                        System.out.println("=====================");

                        System.out.println("Account 2 details");
                        System.out.println(ac2.bankName);
                        System.out.println(ac2.accountHolderName);
                        System.out.println(ac2.balance);
        }
}
```

Multiple Java Classes Collaboration

- A Java class **may exist without** `main()`
- Such a class:

- Can be compiled
- Can have objects created
- Cannot be executed directly

- In Java, we can also create an empty class also.

Example:

A.java:

```
public class A{
        //Empty class
}
```

- We can also generate the **.class** file also for this empty Java class.
- But we can not execute the above Java class, because it does not have a main method.
- In fact, we can create an object for this empty class also.
- Remember, the above class is empty, but when we try to create an object of that class, that object will not be empty; it will have some properties in it.

Creating one class object inside the main method of another class:

- In this case, both classes should be in the same **path/location.**

Example1:

A.java:

```
package com.chitkara;
public class A {

        int i=10;

        void funA() {
                System.out.println("inside funA of A class");
```

```
		}
	}
```

Demo.java

```
	package com.chitkara;

	public class Demo {
		int i=10;

		A a1= new A();

		A a2;

		public static void main(String[] args) {

			Demo d1 = new Demo();

			System.out.println(d1);//Demo object address

			System.out.println(d1.i);//10

			System.out.println(d1.a1); //A object address

			System.out.println(d1.a2);//null

			d1.a1.funA();
//			d1.a2.funA(); // NullPointerException

//			A a5 = new A();
//			d1.a2= a5;

			d1.a2 = new A();

			d1.a2.funA();

		}
	}
```

Example2:

A.java:

```
public class A{

        int i = 10;
        static int j = 20;

        void funA(){
                System.out.println("inside funA of A class");
        }
}
```

Demo.java:

```
public class Demo{

        public static void main(String[] args){

                //creating class A object
                A a1= new A();
                System.out.println(a1.i);
                a1.funA();

                //Accessing the static member.
                System.out.println(A.j);

        }
}
```

- **Note:** We can access the **static** members of a class from anywhere inside the class (inside static context or inside non-static context) because they are already loaded into RAM (into the method area). To access the non-static members from the static context(static method or static area), we need to create an object of the class.

- All non-static members are sharable among themselves, i.e., we can access them directly from any non-static method.

Example:

Employee.java:

```
package com.masai;
public class Employee{

        String id="E-1s11";
        String name="Ramesh";
        double salary=25000.00;
```

```
		String address="Hyderabad";

		public void displayDetails(){

			System.out.println("Employee Details");
			System.out.println("-------------------");
			System.out.println("Employee Id :"+id);
			System.out.println("Employee Name :"+name);
			System.out.println("Employee Saslary:"+salary);
			System.out.println("Employee Address:"+address);

		}
	}
```

Main.java:

```
	package com.masai;
	public class Main{

		public static void main(String[] args){

				Employee emp=new Employee();
				emp.displayDetails();
		}
	}
```

Relationships in Java

- In Object-Oriented Programming (OOP), a relationship describes how one class is connected to another class. Relationships help us model real-world connections between objects and design well-structured applications.

- Java mainly supports the following types of relationships:

1. IS-A Relationship (Inheritance)

- Represents **inheritance**
- One class **extends** another class.
- Shows a **parent–child** relationship

"Child is a type of parent."

Example:

- **Dog** IS-A **Animal**
- **SavingsAccount** IS-A **Account**

Key Points:

- Achieved using `extends`
- Promotes **code reusability**
- Supports **method overriding**

2. HAS-A Relationship (Association)

- One class **uses or contains** an object of another class
- Achieved using **instance variables**
- Represents **object collaboration**

“One object has another object.”

Example:

- **Car** HAS-A **Engine**
- **Student** HAS-A **Address**

The Has-A relationship is also called an Association in Object-Oriented Programming.

Example:

Address.java

```
package com.masai;
public class Address{

        String city;
```

```
                    String state;
                    String pincode;
            }
```

Employee.java:

```
package com.masai;
public class Employee{

        String empId;
        String empName;
        double salary;

        //Employee has Address
        Address address;

        public static void main(String[] args){

        Employee emp1 = new Employee();

        System.out.println(emp1); //Employee object hashcode i.e Employee@323232
        System.out.println(emp1.empId); // null
        System.out.println(emp1.empName); // null
        System.out.println(emp1.salary); // 0.0
        System.out.println(emp1.address); // null


        Employee emp2 = new Employee();
        emp2.empId = "Emp-01";
        emp2.empName = "Ram";
        emp2.salary = 60000.00;

        emp2.address = new Address();
        emp2.address.city = "Coimbator";
        emp2.address.state = "Tamilnadu";
        emp2.address.pincode = "434322";

        System.out.println(emp2.empId); // Emp-01
        System.out.println(emp2.empName); // Ram
        System.out.println(emp2.salary); // 60000.00
        System.out.println(emp2.address); //Address object hashcode i.e Address@232423
        System.out.println(emp2.address.city); //Coimbator
        System.out.println(emp2.address.state); //Tamilnadu
        System.out.println(emp2.address.pincode); //434322

        }
}
```

Note: We can define the Address class inside the Employee class as a **static** member.

Example1:

Address.java

```
                    package com.masai;
                    public class Address{

                              String city;
                              String state;
                              String pincode;
                    }
```

Employee.java

```
package com.masai;
public class Employee{

          String empId;
          String empName;
          double salary;

          //defining Address class as a static member
          static Address address;

          public static void main(String[] args){

          Employee emp1 = new Employee();

          System.out.println(emp1); //Employee object hashcode i.e Employee@323232
          System.out.println(emp1.empId); // null
          System.out.println(emp1.empName); // null
          System.out.println(emp1.salary); // 0.0

          System.out.println(Employee.address); // null

          Employee emp2 = new Employee();
          emp2.empId = "Emp-01";
          emp2.empName = "Ram";
          emp2.salary = 60000.00;

          Employee.address = new Address();
//address = new Address(); //within the same class, we can access the static members directly
//emp2.address=new Address(); we can access the static members with the help of an object, also

          Employee.address.city = "Coimbator";
          Employee.address.state = "Tamilnadu";
          Employee.address.pincode = "434322";
          System.out.println(emp2.empId); // Emp-01
          System.out.println(emp2.empName); // Ram
          System.out.println(emp2.salary); // 60000.00
```

```
            System.out.println(emp2.address); //Address object hashcode i.e Address@232423
            System.out.println(emp2.address.city); //Coimbator
            System.out.println(emp2.address.state); //Tamilnadu
            System.out.println(emp2.address.pincode); //434322
            }
}
```

Example2: Object collaboration:

Address.java

```
                        package com.masai;
                        public class Address{

                                    String city;
                                    String state;
                                    String pincode;

                                    public void printAddress(){
                                                System.out.println("City :"+city);
                                                System.out.println("State :"+state);
                                                System.out.println("Pincode :"+pincode);
                                    }
                        }
```

Employee.java:

```
package com.masai;
public class Employee{

            String empId;
            String empName;
            double salary;
            Address address = new Address();

            public void showDetails(){

                        System.out.println("Employee Id :"+empId);
                        System.out.println("Employee Name :"+empName);
                        System.out.println("Employee Salary :"+salary);
                        System.out.println("Employee Address :");
                        address.printAddress();
            }

            public static void main(String[] args){

                        Employee emp1 = new Employee();
                        emp1.showDetails();

                        Employee emp2 = new Employee();
```

```
                emp2.empId = "Emp-01";
                emp2.empName = "Ramesh";
                emp2.salary = 50000;
                emp2.address.city = "Coimbator";
                emp2.address.state = "Tamilnadu";
                emp2.address.pincode = "434322";

                emp2.showDetails();
        }
}
```

Example3: Defining one class as a static member of another class:

A.java:

```
        package com.masai;
        public class A {

                int x = 10;

                void funA(){
                        System.out.println("inside funA of A class");
                }
        }
```

Here:

- `x` is a non-static (instance) variable
- `funA()` is a non-static method
- Both belong to an object of class A

Main.java:

```
        package com.masai;
        public class Main {

                static int j=200;
                static A a1 = new A();

                public static void main(String[] args)
                {

                        Main obj=new Main();
                        //System.out.println(obj.j);// 200
                        //System.out.println(obj.a1); //A@23232
                        //obj.a1.funA();
                        System.out.println(Main.a1);
                        Main.a1.funA();
                        //System.out.println();
                }
        }
```

Note: The **println()** method belongs to the **"java.io.PrintStream"** class. "**out**" is a variable of this **PrintStream** class, which is **statically** defined inside the **System** class.

Example:

```
public class System{

        static PrintStream out = new PrintStream(...);

}
//that's why we can use System.out.println();
```

Important Note

- **`println()` is a method of the PrintStream class**
- **`out` is a static variable of type `PrintStream`**
- **`out` is defined inside the `System` class**

Types of HAS-A Relationship

- HAS-A relationship has two types:

    1. Aggregation (Weak HAS-A)
    2. Composition (Strong HAS-A)

a) Aggregation (Weak HAS-A)

- Objects have an independent life cycle
- A child object can exist independently (without a parent).

Example:

- **College** HAS-A **Student**
- **Department** HAS-A **Teacher**
- **Library** Has-A **Book**
- **Team** Has-A **Player**

If:

- College closes.
- Students still exist.
- That's **Aggregation**

Use Aggregation when:

- Objects are **logically related**
- Both objects can exist **independently.**
- Example: Teacher – Department

Aggregation Example in Java

Student.java

```
class Student {

  int roll;
  String name;
```

```
    void show() {
        System.out.println(roll + " " + name);
    }

}
```

College.java: (Aggregation)

```
class College {

    String collegeName;
    Student student; // Aggregation


    void display() {
        System.out.println(collegeName);
        student.show();
    }
}
```

Main.java:

```
public class Main {

public static void main(String[] args) {

        Student s = new Student();
        s.roll = 101;
        s.name = "Amit";

        College c = new College();
        c.collegeName = "Chitkara";

        c.student = s; // object assigned
        c.display();
    }
}
```

Why is this is Aggregation?

- The `Student` object is created **outside** the `College.`
- A `Student` can exist **without** `College.`

  Weak association
  Independent life cycle

b) Composition (Strong HAS-A)

- Composition represents a **Has-A relationship where the child object cannot exist without the parent**.
- If the parent object is destroyed, the child object **is also destroyed**, then it is **Composition**.
- Objects are tightly coupled.

Example:

- **Car** HAS-A **Engine**
- **House** HAS-A **Room**
- **Human** Has-A **Heart**

Use Composition when:

- Objects are tightly coupled
- A child has no meaning without a parent.
- Example: House – Room

  Room has no meaning without a house.

Composition Example in Java:

Engine.java

```
class Engine {
    void start() {
        System.out.println("Engine started");
    }

}
```

<u>Car.java:</u> (Composition)

```
class Car {

    Engine engine = new Engine(); // created inside class

    void drive() {

        engine.start();
        System.out.println("Car moving");

    }

}
```

<u>Main.java:</u>

```
public class Main {

    public static void main(String[] args) {

        Car car = new Car();
        car.drive();

    }
}
```

Why is this a composition?

- `The Engine` class is created inside the `Car class.`
- Engine cannot exist without Car.

- When the Car object is destroyed, the Engine object will be destroyed automatically.

  Strong association
  Dependent life cycle

Golden Rule:

- **Aggregation**: Parent only uses child (Created outside parent)
- **Composition**: Parent creates and owns child (Created inside parent)

Tricky Case 1: Created inside but still Aggregation

```
class College {

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

- The student still exists independently in the real world.
- College does not *own* student life.
- So logically: Still **Aggregation**
- Creation place alone doesn't change real-world meaning.

Tricky Case 2: Passed from outside but Composition

```
class Car {

        Engine engine;

         void setEngine(Engine e) {
            engine = e;
         }
}
```

- If this **Engine** is used only by this **Car** and destroyed with it,
- Then logically: Composition
- Even though it was created outside.

Note: Composition is actually a special, **stronger** form of **Aggregation**. So, Composition cannot exist without Aggregation.

- All compositions are aggregations.
- But not all aggregations are compositions.
- Composition is a **subset** of aggregation.

Because composition also satisfies the aggregation rule:

- Parent HAS-A child

But adds stronger rule:

- Child life cycle depends on parent

Example: Stronger Composition

```
class Car {

    private final Engine engine = new Engine();


    void drive() {
        engine.start();
        System.out.println("Car moving");
    }

}
```

- Here, the `final` keyword prevents reassignment.

Example: Stronger Aggregation

To enforce aggregation:

- Object must come from outside.
- Parent should not create it.
- Object may be shared.

```
class Car {

  private Engine engine;

  void setEngine(Engine engine) {
      this.engine = engine;
  }

}
```

- An engine can exist independently.

Final Thought: Aggregation vs Composition is decided by **real-world ownership (logically)**, not by where the object is created.

Activity Task1:

Identify Aggregation or Composition and justify.

1. Library HAS-A Books
2. Company HAS-A Employees

3. House HAS-A Rooms
4. University HAS-A Departments
5. Car HAS-A Wheels
6. Team HAS-A Players
7. Laptop HAS-A Battery
8. School HAS-A Teachers
9. Building HAS-A Floors
10. Person HAS-A Heart

Activity Task2:

- Answer the following questions:

1. If many cars use the same engine, then relationship?
2. If each engine belongs to one car only, then relationship?

Activity Task3: Aggregation

Create two classes:

- `Teacher`
- `Department`

A department **has a teacher**, but a teacher can exist independently.

So, implement **Aggregation**.

Requirements

1. Teacher has:
    - id
    - name
2. Department has:
    - department name
    - teacher object
3. Display department and teacher details.
4. Teacher object must be created **outside** the department.

Expected Output:

Department: Computer Science

Teacher: 101 Rahul

Activity Task4: Composition:

Create two classes:

- `House`
- `Room`

- A house owns rooms, and rooms cannot exist without a house. Implement **Composition**.

Requirements

1. The `Room` class should have a method `showRoom()` with the message: “Room is available”.
2. The room object must be created **inside** the `House` class.
3. `House` class should contain a **house name**.

4. House should display:
    - house name
    - room details.

Expected Output:

```
House Name: Green Villa
Room is available
```

Methods in Java

What is a Method?

- In Java, a **method** is a block of code designed to perform a **specific task**. A method executes only when it is **called**.

Advantages of Using Methods

- **Modularity** – break large programs into smaller parts
- **Reusability** – write once, use multiple times
- **Maintainability** – easy to update and debug

- **Readability** – improves code clarity.

Basic Structure of a Method:

```
[access_modifier] return_type methodName(parameter_list) throws ExceptionList {

        // method body
        return value;
}
```

Explanation:

- **Access Modifier –** defines visibility (`public`, `private`, `protected`, default)
- **Return Type –** type of value returned (`int`, `double`, `Object`, etc.)
    - Use `void` if no value is returned.d
- **Method Name –** usually a verb (e.g., `calculate()`, `printDetails()`)
- **Parameter List –** inputs to the method (optional)
- **Exception List –** exceptions that the method may throw (optional)

Naming Rule

- Method names → **verbs that describe the action.**
- Variable names → **nouns**

Types of Methods in Java

1. Concrete Method

A concrete method has:

- Method declaration
- Method body (implementation)

Example:

```
public void sayHello() {

        System.out.println("Welcome to Java");
}
```

- Can exist in both concrete classes and **abstract** classes.

2. Abstract Method

- An abstract method only has a declaration (method signature), but no body. The method must be implemented in a **subclass**. Abstract methods are found in **abstract classes** or **interfaces**.

  Example:

  ```
  public abstract void sayHello();
  ```

  Rules:

  - Must be declared using the `abstract` keyword
  - Must be implemented by child classes
  - Can exist only in **abstract classes** or **interfaces**

Concrete vs Abstract Methods:

Concrete Method	**Abstract Method**
Has body	No body
No `abstract` keyword	Uses `abstract`
Optional override	Must be overridden
Can exist anywhere	Only in an abstract class/interface

Method Signature

- A method signature uniquely identifies a method.

Method Signature consists of

1. Method name
2. Number of parameters
3. Type of parameters
4. Order of parameters

Not part of signature

- Access modifier
- Return type
- Parameter names
- throws clause

Method Signature Examples:

Method Declaration	**Method Signature**
public int add(int a, int b)	add(int, int)

Method Declaration	Method Signature
void add(double a, int b)	add(double, int)
public void sayHello()	sayHello()

Same signature: NOT allowed
Different parameter order/type: allowed

Methods with Parameters

- Methods can accept parameters that provide input to the method. This allows the method to perform an action based on the provided data. Parameters can be of any data type, including primitive types and reference types (e.g., objects, arrays, etc.).

Example 1: Method with a primitive parameter

Demo.java:

```
public class Demo {

  void fun1(int i) {
     System.out.println("Inside fun1 of Demo");
     System.out.println("The value of i is: " + i);
  }

  public static void main(String[] args) {

     Demo d1 = new Demo();
     d1.fun1(20);

     byte x = 20;
     d1.fun1(x);  // Implicit typecasting to int

     double n1=100.00;
         //d1.fun1(n1); //Error
         //d1.fun1((int)n1); //valid
  }
}
```

Note: In this example main() method is calling the fun1() method on the Demo class object.

Method with Class Type Parameter: (Reference type)

- As we can take any primitive type as a parameter to a function, we can also take a class reference also as a parameter.
- If a method takes a concrete class as a parameter, then to call that method, we can pass the following 3 things as arguments:
    1. **Same class object**
    2. **Its child class object** //we will talk about the child classes in the upcoming session
    3. **null**

Example:

A.java:

```
package com.masai;
public class A{

        int i=10;

        void funA(){
                System.out.println("inside funA of A");
        }
}
```

Demo.java:

```
package com.masai;
public class Demo
{
        //method with class as parameter
        void fun1(A a1){   // here to call this method, we can pass 3 possible values
                      // 1.same class obj, 2 .child class obj, 3. null
                System.out.println("inside fun1 of Demo");
                System.out.println("the value of a1 is : "+a1);
                a1.funA();
        }
        public static void main(String[] args)
        {
                Demo d1=new Demo();
                //A a5=new A();
                //d1.fun1(a5);

                d1.fun1(new A());
        }
}
```

- To the above method call, if we supply the null value, then the fun1() method will be called, but inside the method, when control reaches a1.funA(), it will throw a runtime exception called “**NullPointerException”**.
- So, it is always recommended to check the null value inside the method definition before accessing any member from the supplied object.

Example:

A.java:

```
        package com.masai;
        public class A{
                int i=10;

                void funA(){
                        System.out.println("inside funA of A");
                }
        }
```

Demo.java:

```
        package com.masai;
        public class Demo
        {

                void fun1(A a1){  //never use such identifiers

                        if(a1 != null){
                                        System.out.println("inside fun1 of Demo");
                                        System.out.println("the value of a1 is : "+a1);
                                        a1.funA();
                                }else
                                        System.out.println("supplied value is null");

                }
                public static void main(String[] args)
                {
                        Demo d1=new Demo();
                        d1.fun1(new A());
                        d1.fun1(null);
                }
        }
```

Methods with Return Type:

- A method must have a return type, at least **void**.
- The void return type indicates that the method does not return any value; it will perform some task there only.
- When a method has a return type other than **void**, then the author of that method should not close the method body without returning an appropriate value.

Example: method with return type as primitive:

Demo.java:

```
package com.masai;
public class Demo
{
        int fun1(){

                System.out.println("inside fun1 of Demo");
                //return 100;

                //int x=200;
                //return x;

                //we can return any value which is compatible with the int type(smaller than int)

                byte b=10;
                return b;

                //long x=20;
                //return x; //Error
                //return (int)x; //OK
        }

        public static void main(String[] args)
        {
                Demo d1=new Demo();
```

```
		d1.fun1(); //here method will be called, but the returned value will be unreferenced
			// hence, it will reach the garbage collector
			//to utilize the returned value, we need to hold that value inside a variable
	//the variable on which we hold that value should be either of the same type or bigger
	// than the specified type.

		int x=d1.fun1();

		System.out.println("Returned value is "+x);

		long y=d1.fun1(); //implicit typecasting

		System.out.println("Returned value is "+x);

		//byte b= d1.fun1(); // Error
		byte b = (byte)d1.fun1(); //explicit type casting
	}
}
```

Example: method with return type as Class:

- We can define a method with a class as a return type also, if a method is defined to return a class type, then that method should return one of the following :
    1. **Same class object**
    2. **The child class object of the specified type**
    3. **null**

A.java:

```
		package com.masai;
		public class A{

			int i=10;

			void funA(){
				System.out.println("inside funA of A");
			}
		}
```

Demo.java:

```
package com.masai;
public class Demo
{

        public A fun1(){ //return type as class A type

                System.out.println("inside fun1 of Demo");

                //A a1=new A();
                //return a1;
                return new A();
        }
        public static void main(String[] args)
        {
            Demo d1=new Demo();
            d1.fun1(); //here returned A class object will reach the GC.

            //to hold the returned value, we have 2 options:
            //1.to the same class variable
            //2.to the parent class variable
            A a1= d1.fun1(); //to the same class variable
            a1.funA();

            Object obj = d1.fun1(); //to the parent class variable
        }
}
```

Note: In Java, there is a class called the **Object** class, which acts as a parent/super class of any Java class directly or indirectly. This **Object** class belongs to the "java.lang" package.

To the variable of the Object class, we can assign any class object.

Polymorphism:

- Polymorphism is a fundamental concept in Object-Oriented Programming (OOP), which means **"many forms."** In Java, it allows the same method or function to behave differently based on the context. This ability to define multiple behaviors for the same method name in the same or different classes is what makes polymorphism powerful and flexible.

Advantages of Polymorphism:

- **Flexibility**: Allows objects of different types to be treated uniformly, enabling more versatile code.
- **Code Reusability**: Promotes the use of the same method for different object types, reducing code duplication.
- **Maintainability**: Changes in base classes don't affect existing code, simplifying maintenance.
- **Simplicity**: Reduces the need for complex conditional statements, making code more readable.
- **Extensibility**: New classes can be added without modifying existing code, supporting easy system expansion.

Types of Polymorphism in Java:

1. **Static Polymorphism (Compile-Time Polymorphism)**:
    - **Definition**: Static polymorphism occurs when the method to be invoked is determined at **compile time**.
    - **How it works**: This is achieved through **method overloading** and **constructor overloading**. Method overloading involves defining multiple methods with the same name but different parameter lists (either in number, type, or both).
    - **Example**:
        - Multiple methods with the same name but different parameter types or counts in the same class.
    - **Key Points**:
        - It is resolved at **compile time**.
        - Commonly achieved through **method overloading** or **constructor overloading**.
        - The method signature (**number**, **type**, or **order** of parameters) is what distinguishes overloaded methods.
2. **Dynamic Polymorphism (Run-Time Polymorphism)**:
    - **Definition**: Dynamic polymorphism occurs when the method to be invoked is determined at **runtime**, based on the object type.
    - **How it works**: This is achieved through **method overriding**. Method overriding occurs when a subclass provides its specific implementation of a method that is

already defined in the superclass with the same method signature (name, return type, and parameters).

- **Example**: Method overriding is commonly used in **inheritance**, where a subclass overrides a method of its superclass.
- **Key Points**:
    - It is resolved at **runtime**.
    - Achieved through **method overriding** in a subclass.
    - The method invoked is determined by the **actual object type** (not the reference type) at runtime.
    - Allows different behaviors for the same method in different objects.

Example of Static Polymorphism (Method Overloading):

Calculator.java:

```
package com.masai;
public class Calculator {

  // Add two integers
  public int add(int a, int b) {
      return a + b;
  }

  // Add three integers
  public int add(int a, int b, int c) {
      return a + b + c;
  }

  // Add two double values
  public double add(double a, double b) {
      return a + b;
  }
}
public class Main {

  public static void main(String[] args) {
      Calculator calc = new Calculator();

      // Calling overloaded methods
      System.out.println("Sum of two integers: " + calc.add(10, 20));
      System.out.println("Sum of three integers: " + calc.add(10, 20, 30));
      System.out.println("Sum of two doubles: " + calc.add(10.5, 20.5));
```

```
   }
}
```

Example2:

Demo.java

```
package com.masai;
public class Demo
{
        void fun1(byte b){
                System.out.println("inside fun1(byte) of Demo");
                //500 lines of code

         }

        void fun1(int i){
                System.out.println("inside fun1(int) of Demo");
                //10000 lines of code

         }
        public static void main(String[] args)
        {
                Demo d1=new Demo();
                byte x=20;
                d1.fun1(x);  // it will give priority to the nearest one.
        }
}
```

Disadvantages of Static Polymorphism:

- One major disadvantage of **static polymorphism (method overloading)** is that **the compiler can sometimes get confused and enter an ambiguous state**, resulting in a **compile-time error**.
- This happens when the compiler **cannot uniquely decide which overloaded method should be called** based on the arguments supplied.

Example:

Demo.java:

```
package com.masai;
public class Demo {

	void fun1(int i, float j) {
		System.out.println("inside fun1(int, float) of Demo");
	}
	void fun1(float f, int j) {
		System.out.println("inside fun1(float, int) of Demo");
	}

	public static void main(String[] args) {

		Demo d1 = new Demo();
		// d1.fun1(10, 10.55);//Error passing int and double
		d1.fun1(10, 10.55f); // fun1(int, float) will be called
		d1.fun1(10.55f, 10); // fun1(float,int) will be called
		d1.fun1(10, 10); // Error
	}
}
```

Why Does Ambiguity Occur?

- Method overloading is resolved at **compile time.**
- The compiler selects the method based on:
  - Number of parameters
  - Type of parameters
  - Order of parameters
- If **two or more overloaded methods are equally suitable**, the compiler fails.

Note: There are multiple overloaded println() methods defined inside the java.io.PrintStream class.

Such as:

```
println(primitives)
println(Object)
println(String)
println(char[])
```

Try it:

```
System.out.println(null);
System.out.println((String)null);
```

Main Method Revisited:

```
public static void main(String[] args) {
  // program starts here
}
```

Why `public`?

- JVM must access it from anywhere

Why `static`?

- JVM calls it without creating an object

Why `void`?

- JVM doesn’t expect a return value

Why `String[] args`?

- Command-line arguments

What if `static` is removed?

- Code compiles
- A runtime error occurs.

Can we overload the main method?

- Yes, but JVM uses only the standard signature.

Using Command Line argument:

- The JVM initially calls the main method by passing an empty String array object**.**

  Example:

  ```
  String[] args = {};
  ```

- Command-line arguments are **values passed to a Java program at runtime** through the command prompt.
- These values are received by the **`main()`** **method.**

- They are stored inside a **String array.**

Passing Arguments from the Command Line

- While running the program, **values separated by space** are passed as command-line arguments.

  ```
  java Demo 10 20
  ```

- Internally, the JVM creates:

  ```
  String[] args = {"10", "20"};
  ```

- Each space-separated value becomes **one element.**

- All values are stored as **String.**

Important Points About Command Line Arguments

- CLA are **always Strings**

- Space ( ) is the default separator.

- Conversion is required for numeric operations.s
- `args. length` tells how many arguments are passed

Example:

Demo.java:

```
package com.masai;
public  class Demo {

	public static void main(String[] args) {

		System.out.println(args); //String array object address
		System.out.println(args.length);// number of argument supplied at runtime //separated by
		space

		if(args.length == 2) {


			String snum1 = args[0];
			String snum2= args[1];

			int num1= Integer.parseInt(snum1);
			int num2= Integer.parseInt(snum2);

			int result = num1+num2;
			//int result = Integer.parseInt(args[0]) + Integer.parseInt(args[1]);

			System.out.println("The Sum of 2 number is: "+result);

		}else {
			System.out.println("please pass 2 numbers as CLA");
		}


	}

}
```

Steps to Pass Command Line Arguments (CLA) in Eclipse:

Step 1: Open Run Configurations

1. Right-click on the Java file (Demo.java)
2. Click Run As -> Run Configurations…

Step 2: Locate Program Arguments Option

1. In the Run Configurations window:
    - Select Java Application
    - Select your class name.
2. Click on the Arguments tab

Step 3: Enter Command Line Arguments

1. In the Program arguments textbox:

    10 20

2. Arguments must be:
    - Space separated
    - Written without quotes (for numbers)

Step 4: Run the Program

1. Click Run
2. Output appears in the console.