1. Introduction to Java

History and Origins

- **The "Green Team":** Java was originally developed by a team known as the "Green Team" at **Sun Microsystems**, led by **James Gosling**.
- **Initial Name:** It was originally called "Oak" (after an oak tree outside Gosling's office), then "Green," and finally renamed to **Java**.
- **Original Purpose:** While we know it as a web language today, Java was initially designed for **consumer electronics** (like cable boxes and toasters) to communicate with each other.
- **The Shift:** The boom of the World Wide Web in the mid-90s shifted Java's focus to the internet because of its portable nature.
- **Current Stewardship:** In **2010**, Oracle Corporation acquired Sun Microsystems. Today, Oracle manages the standard, though OpenJDK exists as an open-source alternative.

Java Editions

Java is not a "one size fits all" platform; it comes in three main flavors :

1. **Java SE (Standard Edition):** The core platform used for desktop applications and learning the language. It contains basics like `String`, `ArrayList`, and `Math`.
2. **Java EE (Enterprise Edition):** Built on top of SE, this includes tools for large-scale web sites and server-side apps (e.g., banking systems).
3. **Java ME (Micro Edition):** A stripped-down version for resource-constrained devices like set-top boxes, old mobile phones, and embedded controllers.

- Java was mainly developed to create software for consumer electronic devices that could be controlled by a remote

Features of Java

1. Simple

- Java is considered simple because it removes many complex features present in C and C++. It does not require header files, as class declarations and implementations are written together and accessed using simple import statements.
- Java avoids pointer arithmetic, preventing direct memory manipulation and reducing errors like crashes and security issues.
- Instead of structures and unions, Java uses classes to organize data and behavior in a clear way.

- It also does not support operator overloading (except for string concatenation), which avoids confusing code, and eliminates virtual base classes, making inheritance simpler

2. Object oriented programming language

Organizes programs in terms of collection of objects that represent an instance of a class

4 pillars of OOPs – Abstraction, Encapsulation, Inheritance, Polymorphism.

Abstraction

Abstraction is the process of **hiding implementation details** and showing only the **essential features** of an object. It focuses on *what an object does* rather than *how it does it*. In Java, abstraction is achieved using **abstract classes** and **interfaces**.

Encapsulation

Encapsulation is the technique of **wrapping data and methods into a single unit (class)** and restricting direct access to data. It is achieved using **access modifiers** like `private`, `public`, and `protected`, ensuring **data hiding and security**.

Inheritance

Inheritance is a mechanism where a **new class (subclass)** acquires the properties and behaviors of an **existing class (superclass)**. It represents an **is-a relationship** and promotes **code reusability**.

Polymorphism

Polymorphism means **one interface, many forms**. It allows the same method name to perform **different actions** depending on the object. In Java, it is achieved using **method overloading** and **method overriding**.

3. Distributed

- Java is considered **distributed** because it provides an extensive set of built-in libraries to support **network communication** using TCP/IP protocols such as **HTTP and FTP**.
- These libraries allow Java applications to easily communicate over a network, send and receive data, and interact with remote systems.
- Using classes like URL, URLConnection, and networking APIs, Java programs can **open and access remote objects or resources over the Internet** in much the same way as accessing local files.

4. Robust

- Java is considered robust because it is designed to reduce errors and handle problems reliably. It provides a strong type-checking mechanism that detects many errors at compile time, thereby minimizing unexpected issues during runtime.
- Java also has a powerful exception-handling mechanism that allows errors to be caught and handled gracefully, preventing abrupt program termination.
- Java offers automatic memory management through garbage collection, which automatically allocates and deallocates memory, eliminating problems such as memory leaks and dangling pointers.

5. Secure

Java is considered **secure** because it was designed for use in **networked and distributed environments**, where security is critical. Java prevents unauthorized access through features like **bytecode verification**, **restricted direct memory access**, and **controlled execution by the JVM**. Programs run inside a secure environment that prevents viruses and tampering, enabling the development of **virus-free and tamper-free systems**.

6. Portable

- Java's platform independence ensures that applications can run on any device or system with a compatible JVM.
- Enables developers to create applications that can be deployed across various devices without modification.
- Ensures consistent behavior and performance across different environments.

7. Interpreted

Java is considered interpreted because the source code is first compiled into bytecode, which is then executed by the Java Virtual Machine (JVM) rather than directly by the operating system. The JVM interprets the bytecode line by line, allowing the same program to run on different platforms without modification. This approach improves portability and makes Java platform-independent.

8. High Performance

- Java's Just-In-Time (JIT) compiler optimizes bytecode to native machine code at runtime, enhancing execution speed.
- Efficient memory management and garbage collection mechanisms contribute to improved performance along with support for multithreading and concurrent processing by leveraging modern hardware capabilities.

Common Terms of Java :

Java Virtual Machine (JVM):
The Java Virtual Machine (JVM) enables Java's **platform independence**. A Java program passes through three phases: writing, compiling, and running.

- The compiler converts source code into **bytecode**, which the JVM executes during the running phase.
- The JVM loads and verifies bytecode, manages memory, performs garbage collection, enforces security, and converts bytecode into machine code using interpretation and JIT compilation. This allows Java programs to run on different platforms without modification.

Java Development Kit (JDK):
The Java Development Kit (JDK) is a complete package required to **develop Java applications**. It includes the Java compiler, JRE, debugging tools, and documentation utilities. The JDK supports the entire development process, from writing and compiling code to testing and debugging programs.

Java Runtime Environment (JRE):
The Java Runtime Environment (JRE) provides the environment needed to **run Java programs**. It includes the JVM and core libraries but does not contain development tools like the compiler. The JRE ensures consistent program execution across platforms by managing class loading, memory, and security during runtime.

Java Virtual Machine (JVM) :

The **Java Virtual Machine (JVM)** is a **platform-dependent execution environment** that converts Java bytecode (.class file) into machine-level instructions and executes them. JVM can be viewed in three ways.

1. It is a **specification**, where the behavior and structure of the JVM are defined, but the actual implementation is left to different vendors.

2. It is an **implementation**, where this specification is implemented by providers such as Oracle and others; this implementation is available as part of the **Java Runtime Environment (JRE)**.

3. It is a **runtime instance**, meaning that whenever a Java program is executed using the java command, a new instance of the JVM is created to run that program.

Implementation of a Java Applications

Example 1: Sum and Average of Numbers

Program

```
public class SumAverage {
    public static void main(String[] args) {
        int a = 10, b = 20, c = 30;
        int sum = a + b + c;
        double avg = sum / 3.0;

        System.out.println("Sum = " + sum);
        System.out.println("Average = " + avg);
    }
}
```

Steps

- Save file as → `SumAverage.java`
- Compile → `javac SumAverage.java`
- Run → `java SumAverage`

Output

```
Sum = 60
Average = 20.0
```

Example 2: Factorial of a Number (Using Loop)

Program

```
public class Factorial {
    public static void main(String[] args) {
        int n = 5;
        int fact = 1;

        for (int i = 1; i <= n; i++) {
            fact = fact * i;
        }

        System.out.println("Factorial of " + n + " = " + fact);
    }
}
```

Output

```
Factorial of 5 = 120
```

Example 3: Prime Number Check

Program

```
public class PrimeCheck {
    public static void main(String[] args) {
        int num = 13;
        boolean isPrime = true;

        for (int i = 2; i <= num / 2; i++) {
            if (num % i == 0) {
                isPrime = false;
                break;
            }
        }

        if (isPrime && num > 1)
            System.out.println(num + " is a Prime Number");
        else
            System.out.println(num + " is not a Prime Number");
    }
}
```

Output

```
13 is a Prime Number
```

Example 4: Method Usage (Function Call)

Program

```
public class Square {
    static int square(int n) {
        return n * n;
    }

    public static void main(String[] args) {
        int number = 6;
        System.out.println("Square = " + square(number));
    }
}
```

Output

```
Square = 36
```

Example 5: Using Command-Line Arguments

Program

```
public class CmdArgs {
    public static void main(String[] args) {
```

```
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);

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

Run

```
java CmdArgs 5 7
```

Output

```
Sum = 12
```

Abstraction, Encapsulation and Composition in Java

Data Abstraction in Object-Oriented Programming

Data abstraction in object-oriented programming is the process of **providing functionality to users while hiding implementation details**. The user only knows *what an entity does* and not *how it performs internally*. In Java, abstraction is implemented using **abstract classes and interfaces**, where interfaces provide **100% abstraction** by exposing only method declarations.

Advantages of Java Abstraction

1. Reduces Complexity
Abstraction hides unnecessary implementation details, allowing users and developers to focus only on essential functionality. This simplifies system design and makes large programs easier to understand.

2. Avoids Code Duplication
Common behaviors can be defined in abstract classes or interfaces and reused by multiple classes. This prevents rewriting the same logic repeatedly and promotes consistent implementation.

3. Eases the Burden of Maintenance
Since implementation details are hidden, changes made to internal logic do not affect users of the class. This makes updating and maintaining software easier and less error-prone.

4. Increases Security and Confidentiality
Abstraction restricts access to internal data and methods, exposing only what is necessary. This prevents misuse of data and enhances system security.

- Abstraction defines an object in terms of its properties (attributes), behavior (methods), and interfaces (means of communicating with other objects).
- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Since classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT)**

Example:

Example 1: Bank Account (ADT Example)

A **BankAccount** is an abstract data type where the user knows **what operations can be performed**, not how they are internally implemented.

- **Attributes:** accountNumber, balance
- **Methods (Behavior):** deposit(), withdraw(), checkBalance()
- **Interface:** methods used by the user to interact with the account

```
abstract class BankAccount {
    protected double balance;

    abstract void deposit(double amount);
    abstract void withdraw(double amount);

    public double checkBalance() {
        return balance;
    }
}
```

Here, the user does not know how deposit or withdraw works internally—only what the operations do.

Example 2: Vehicle Abstraction

A **Vehicle** is defined by its essential features without implementation details.

- **Attributes:** speed
- **Methods:** start(), stop()
- **Interface:** how users interact with the vehicle

```
abstract class Vehicle {
    int speed;
    abstract void start();
    abstract void stop();
}
```

Different vehicles implement these behaviors differently, but the abstraction remains the same.

Abstract class in Java:

- In Java, we can achieve Abstraction using Abstract classes and interfaces.
- Interfaces allow 100% abstraction (complete abstraction).
- An Abstract class is a class whose objects can’t be created.

- An Abstract class is created through the use of the ***abstract*** keyword. It is used to represent a concept.
- An abstract class can have abstract methods (methods without body) as well as non-abstract methods or concrete methods (methods with the body). A non-abstract class cannot have abstract methods.

- The class has to be declared as abstract if it contains at least one abstract method.
- An abstract class does not allow you to create objects of its type. In this case, we can only use the objects of its subclass.
- Using an abstract class, we can achieve 0 to 100% abstraction.
- There is always a default constructor in an abstract class, it can also have a parameterized constructor.
- The abstract class can also contain final and static methods.

Examples :

Example 1: Shape Abstraction

```
abstract class Shape {          // Abstract class
    abstract void draw();       // Abstract method

    public void display() {     // Concrete method
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {    // Subclass
    @Override
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Main {
    public static void main(String[] args) {
        Shape s = new Circle(); // Polymorphism
        s.draw();
        s.display();
    }
}
```

Output

```
Drawing a Circle
Drawing a shape
```

Example 2: Employee Abstraction

```
abstract class Employee {           // Abstract class
    abstract void calculateSalary();

    public void showRole() {         // Concrete method
```

```
        System.out.println("Employee role");
    }
}

class Manager extends Employee {    // Subclass
    @Override
    void calculateSalary() {
        System.out.println("Salary calculated for Manager");
    }
}

class Main {
    public static void main(String[] args) {
        Employee emp = new Manager();
        emp.calculateSalary();
        emp.showRole();
    }
}
```

Output

```
Salary calculated for Manager
Employee role
```

Encapsulation :

Encapsulation is the process of **wrapping data (variables) and functions (methods) into a single unit called a class**. In encapsulation, the data is **not directly accessible from outside the class**; instead, only the methods defined within the class can access and modify it. These methods act as an **interface between the object's internal data and the external program**, controlling how the data is used. This protection of data from direct access is known as **data hiding or information hiding**, which improves security, prevents accidental modification of data, and makes the program easier to maintain

Advantages of Encapsulation

1. Data Hiding
Encapsulation restricts direct access to data members by hiding implementation details. Data can only be accessed through specific methods, which protects it from accidental or unauthorized modification and improves security.

2. Increased Flexibility
Encapsulation allows variables to be made **read-only or write-only** using access modifiers and getter/setter methods. This gives better control over how data is accessed and modified based on system requirements.

3. Reusability
Encapsulated classes can be reused easily in different programs or modules. Since the internal implementation is hidden, changes can be made without affecting other parts of the program, supporting easy modification for new requirements.

4. Easy Testing
Encapsulated code is easier to test because each class functions as an independent unit. Unit testing becomes simpler as methods can be tested without exposing internal data.

5. Freedom to the Programmer
Encapsulation gives programmers the freedom to change or improve internal implementation details without impacting the rest of the system, as long as the external interface remains unchanged.

Disadvantages of Encapsulation

1. Increased Complexity
Encapsulation can increase complexity when many classes, methods, and access controls are used unnecessarily. Excessive use of getters and setters may make the code harder to manage and understand.

2. Reduced Understanding of System Behavior
Since internal details are hidden, it may become difficult for developers to understand how the system works internally, especially during debugging or when maintaining large applications.

3. Limited Flexibility
Encapsulation may restrict direct access to data, which can limit flexibility in certain situations where quick access or modification of internal data is required.

Example: Encapsulation with Private Data (Error Demonstration)

```
class Student {

    private int marks;     // private variable

    public int getMarks() {    // Getter
        return marks;
    }

    public void setMarks(int m) {   // Setter
        this.marks = m;
    }
}

class Main {

    public static void main(String[] args) {
```

```
        Student s = new Student();

        s.marks = 90;                  // ERROR: marks has private access
        System.out.println(s.marks);   // ERROR: marks has private access
    }
}
```

Why Errors Occur

- `marks` is declared as **private**
- Private variables **cannot be accessed directly outside the class**
- Access is allowed **only through public methods** (`getMarks()`, `setMarks()`)

Correct Way (Using Encapsulation)

```
s.setMarks(90);
System.out.println(s.getMarks());
```

Composition :

Composition is a design technique used to implement a **"has-a" relationship**, where one class contains an object of another class as an instance variable. It represents a strong association between two objects, meaning one object **uses or depends on another object** to perform its functionality.

Composition and inheritance are both important design techniques in object-oriented programming, but they serve different purposes. **Inheritance** is used to represent an **"is-a" relationship**, where a subclass inherits the properties and behavior of a superclass. In contrast, **composition** focuses on reusing functionality by including objects rather than inheriting them.

In composition, an **instance variable of one class refers to an object of another class**, allowing functionality to be reused without creating a rigid class hierarchy. This approach improves flexibility, reduces tight coupling, and makes the program easier to modify and maintain. Because of these advantages, composition is often preferred over inheritance for achieving code reusability in real-world applications.

Example: Composition (Has-a Relationship)

Step 1: Create a class to be used by another class

```
class Engine {

    public void start() {
```

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

Step 2: Use the above class inside another class (Composition)

```
class Car {

    private Engine engine;   // has-a relationship

    Car() {
        engine = new Engine();   // composition
    }

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

Step 3: Main Class

```
class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.drive();
    }
}
```

Output

```
Engine starts
Car is moving
```

Why this is Composition

- `Car` **has an** `Engine`
- `Engine` is used as an **instance variable**
- Functionality is reused **without inheritance**
- Represents a **has-a relationship**

Real-World Meaning

A **Car has an Engine**, but a Car **is not** an Engine.

Benefits of using Composition:

- Composition allows us to reuse the code.

- In Java, we can use multiple Inheritance by using the composition concept.
- The Composition provides better test-ability of a class.
- Composition allows us to easily replace the composed class implementation with a better and improved version.
- Composition allows us to dynamically change our program's behavior by changing the member objects at run time.

Class, Attributes, Methods and Objects

A **class** in Java is a blueprint or template used to create objects. It represents a real-world or logical concept and defines the structure and behavior that its objects will have. A class specifies **what data an object can store** and **what actions it can perform**, but it does not represent a real entity by itself. Memory is allocated only when objects of the class are created. In object-oriented design, a class helps organize code into reusable, modular units by grouping related data and behavior together.

Structure of a class :

```
class ClassName {

    // attributes (variables)

    // methods (functions)

}
```

A **class attribute** (also called an instance variable or field) represents the **state** of an object. Attributes store information about an object and describe its properties. Each object created from a class has its own copy of these attributes. Attributes are usually declared as `private` to protect the data and maintain encapsulation. For example, in a `Student` class, attributes like `id` and `name` describe the student's identity and details.

```
class Student {

    int id;          // class attribute

    String name;     // class attribute



    void displayDetails() {   // class method

        System.out.println(id + " " + name);

    }

}
```

A **class method** defines the **behavior** of an object. Methods specify the actions that an object can perform and often operate on the object's attributes. They are used to read, modify, or process the data stored in the attributes. Methods help objects interact with each other and perform meaningful tasks within a program. For instance, a method like `displayDetails()` can be used to print a student's information.

Object:

An **object** in Java is a real-world entity created from a class. It represents a **specific instance** of a class and occupies memory in the system. If a class is considered a **blueprint**, then an object is the **actual thing built from that blueprint**. Every object has its own identity, state, and behavior. The **identity** is the unique reference name used to access the object, the **state** is defined by the current values of its attributes, and the **behavior** is represented by the methods the object can perform. Objects allow programs to model real-world entities and interact with them in a meaningful way.

An object contains **state (attributes)** and **behavior (methods)** defined by its class. Once an object is created, its attributes and methods can be accessed using the **dot (.) operator**. In Java, an object is created using the **`new` keyword**, which allocates memory for that object. Multiple objects can be created from the same class, and each object will have its **own separate memory and state**, even though they share the same class structure.

Example of usage of an object in Java :

```
class Student {
    int rollNo;
    String name;
    void display() {
        System.out.println(rollNo + " " + name);
    }
}
public class main{
        public static void main(String[] args)
        {
                Student s1 = new Student(); //object of class Student
                s1. rollNo = 1; // initializing attribute of class Student
                s1.name  = “John”; // initializing attribute of class Student
                s1.display(); //calling method of the class Student    }      }
```

Explanation of the Example

- `Student` is a **class** that defines attributes (`rollNo`, `name`) and a method (`display()`).
- `s1` and `s2` are **objects** of the `Student` class.
- Each object has:
  - **Identity** → `s1`, `s2`
  - **State** → different values of `rollNo` and `name`
  - **Behavior** → `display()` method
- Objects access attributes and methods using the **dot operator**.
- Multiple objects are created from one class, but each object has its **own memory**.

Java Object : Key Characteristics

- Created using **new** keyword
- Has its own **memory**
- Can access class attributes and methods using dot operator
- Multiple objects can be created from one class

Class vs Object in Java

Class	Object
Blueprint	Instance
Logical entity	Physical entity
No memory	Occupies memory

Complete example of class and object in Java :

```
class Car {
    String brand;     // attribute
    int speed;        // attribute

    void showDetails() {   // method
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
    }
}

public class Main {
    public static void main(String[] args) {

        // Creating objects of Car class
        Car c1 = new Car();
        Car c2 = new Car();

        // Setting state for first object
        c1.brand = "Toyota";
        c1.speed = 120;

        // Setting state for second object
       c2.brand = "BMW";
       c2.speed = 180;
       // Calling methods using dot operator
       c1.showDetails();
       c2.showDetails();  }     }
```

Explanation of the Example

In this program, `Car` is a **class** that represents a real-world concept of a car. It defines two **attributes**, `brand` and `speed`, which represent the **state** of a car object. The method `showDetails()` defines the **behavior** of the class and is used to display the car's information.

Inside the `main` method, two **objects** of the `Car` class are created using the `new` keyword: `c1` and `c2`. These objects have their own **identity**, which is given by the reference variables `c1` and `c2`. Each object occupies its **own memory** and holds its own **state**, even though both objects are created from the same class.

The attributes of each object are assigned different values, showing that multiple objects created from one class can store different data. The method `showDetails()` is accessed using the **dot (.) operator**, demonstrating how an object can access the attributes and methods of its class. This example clearly shows how a class acts as a blueprint and objects are the actual instances created from it.

Access Modifiers:

Access modifiers in Java are keywords used to control the **visibility and accessibility** of classes, variables, methods, and constructors. They define **where a particular member can be accessed from**, helping protect data and enforce proper object-oriented design. By using access modifiers, Java supports **encapsulation**, which means hiding internal details of a class and exposing only what is necessary.

Java provides **four access modifiers**:

1. **Public** members are accessible from **anywhere** in the program. If a class, method, or variable is declared as `public`, it can be accessed from any other class, even from different packages. This modifier provides the **widest level of access** and is typically used for methods that form part of an application's interface.

2. **Private** members are accessible **only within the same class** in which they are declared. They cannot be accessed from outside the class, not even by subclasses. This modifier provides the **highest level of data protection** and is commonly used for variables to prevent direct access and modification.

3. **Protected** members are accessible within the **same package** and also in **subclasses**, even if the subclass is in a different package. This modifier is useful when a class wants to share data or behavior with its child classes while still restricting access from unrelated classes.

4. The **default access modifier** (when no keyword is used) allows access **only within the same package**. Members with default access are not visible outside their package.

This level of access is also known as *package-private* and is useful when classes within the same package need to work closely together.

Modifier	Same Class	Same Package	Subclass (diff pkg)	Everywhere
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

Why Access Modifiers are Important?

- Provide **security**
- Support **encapsulation**
- Control **data visibility**
- Improve **code maintainability**

Interfaces

An **interface** in Java is a collection of **abstract methods and constants** that a class must implement. It defines a **contract** that specifies what a class **can do**, but it does not define **how the behavior is implemented**. In other words, an interface describes the **"what" and not the "how"**. Any class that implements an interface must provide implementations for all its abstract methods.

Interfaces are used in Java to achieve **100% abstraction** and to support **multiple inheritance**. Since Java does not allow a class to inherit from more than one class, a class can implement multiple interfaces, thereby allowing it to inherit behavior from multiple sources in the form of method declarations.

An interface in Java contains only **method signatures** and constants. By default, all methods declared in an interface are **public and abstract**, and all variables are **public, static, and final**. Interfaces do not have constructors, and they cannot be instantiated directly.

In **Java 8 and later versions**, interfaces were enhanced to allow **default methods**, which provide a default implementation. This allows interfaces to evolve without breaking existing implementing classes. However, the primary purpose of interfaces remains to define behavior without implementation.

Example:

P.T.O

```
interface Animal {
    void sound();      // abstract method
    void eat();        // abstract method
}

class Dog implements Animal {

    public void sound() {
        System.out.println("Dog barks");
    }

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

public class Main {
   public static void main(String[] args) {

       Animal a = new Dog();   // interface reference
       a.sound();
       a.eat();
   }
}
```

Explanation of the Example

In this example, `Animal` is an **interface** that declares two abstract methods: `sound()` and `eat()`. The interface specifies **what actions** an implementing class must perform, but it does not define **how** those actions are implemented.

The `Dog` class **implements** the `Animal` interface and provides concrete implementations for both methods. Since all interface methods are implicitly `public`, the implementing methods in the `Dog` class must also be declared `public`.

In the `main` method, an object of `Dog` is created and referenced using the `Animal` interface type. This demonstrates **abstraction**, where the program interacts with the object through the interface rather than the concrete class. It also shows how interfaces support **loose coupling** and allow different implementations to be used interchangeably.

Uses of interfaces in Java :

1. An **interface is used to achieve abstraction** in Java by defining what actions a class must perform without specifying how those actions are implemented. This allows programmers to focus on *what the system does* rather than *how it does it*, making the design more flexible and easier to understand.
2. Interfaces **support the functionality of multiple inheritance** in Java. Since a class cannot extend more than one class, Java allows a class to implement multiple interfaces. This enables a class to inherit behaviors from multiple sources in the form of method declarations, thereby overcoming the limitations of single inheritance.
3. Interfaces can also be used to achieve **loose coupling** between components of a program. When classes depend on interfaces rather than concrete implementations, changes in one class do not directly affect other classes. This improves maintainability, scalability, and ease of testing.
4. It is possible to have **multiple implementations of the same interface**. Different classes can implement the same interface in different ways, while still adhering to the same method structure. This allows the program to switch between different implementations at runtime, promoting flexibility and extensibility in application design.

Difference between class and interfaces :

Feature	Class	Interface
Keyword used	class	interface
Object creation	✓ Yes	✗ No
Variables	Instance variables allowed	Only constants (public static final)
Methods	Concrete + abstract	Abstract (default/static allowed from Java 8)
Method implementation	Can exist	Must be provided by implementing class
Inheritance	Extends **one** class	Class can implement **multiple** interfaces
Constructor	✓ Allowed	✗ Not allowed
Access modifiers for methods	Any	Always public
Multiple inheritance	✗ Not supported	✓ Supported
Purpose	Represents **what an object is**	Represents **what an object can do**

Frequently Asked Questions about Interfaces in Java

1. **Can we instantiate an interface directly?**
   No, an interface cannot be instantiated directly. An interface does not have a constructor, and there is no default constructor provided by Java. Therefore, objects of an interface cannot be created.
2. **Can we have data members in an interface?**
   Yes, interfaces can have data members. However, all data members in an interface are implicitly **public, static, and final**. This means they belong to the interface itself, are shared by all implementing classes, and their values cannot be changed. These variables act as constants and provide a guarantee that their values remain the same.
3. **Can a class with all methods implemented still be abstract?**
   Yes, a class can be declared abstract even if it implements all methods of an interface. If creating an object of that class does not make sense in the application's domain, the class can be made abstract.
4. **Can we specify an interface method as private?**
   No, interface methods cannot be private. Interface methods are meant to be accessible to implementing classes. *(Note: From Java 9 onwards, private methods are allowed inside interfaces for internal use.)*
5. **Can we specify an interface method as protected?**
   No, interface methods cannot be protected. An interface is meant to be accessible to all implementing classes, so restricting access using `protected` is not allowed.
6. **Can an interface extend another interface?**
   Yes, an interface can extend one or more other interfaces using the `extends` keyword.
7. **Can a class implement more than one interface?**
   Yes, a class in Java can implement multiple interfaces. This is done using the `implements` keyword, with interface names separated by commas.
8. **Can a class override only some methods of an interface?**
   Yes, but if a class does not implement all methods of an interface, then the class must be declared **abstract** and cannot be instantiated.

Constructors, Destructors and Garbage Collector

Constructors :

A **constructor** in Java is a special member of a class that is used to **initialize an object when it is created**. It is automatically invoked at the time of object creation using the `new` keyword. A constructor has the **same name as the class** and is syntactically similar to a method, but it **does not have any explicit return type**, not even `void`.

The main purpose of a constructor is to **assign initial values to the instance variables** of a class and to perform any necessary **start-up operations** required to create a fully initialized object. Constructors ensure that an object is in a **valid and usable state** as soon as it is created.

In Java, **every class has a constructor**, whether it is explicitly defined or not. If a programmer does not define any constructor, Java automatically provides a **default constructor** that initializes instance variables to their default values (such as `0` for numeric types, `false` for boolean, and `null` for object references). However, once a programmer defines **any constructor**, Java **does not provide the default constructor automatically**.

Each time an object is created using the `new` operator, the corresponding constructor of the class is **invoked automatically** to initialize the data members of that object. Constructors are executed only **once per object creation**.

3 types of constructors :

Default Constructor

A **default constructor** is a constructor that **does not take any parameters**. If a programmer does not define any constructor in a class, the Java compiler automatically provides a default constructor. This constructor initializes the instance variables of the class with their **default values**, such as `0` for numeric data types, `false` for boolean, and `null` for object references. The default constructor is mainly used to create an object with default initialization.

Parameterized Constructor

A **parameterized constructor** is a constructor that **accepts parameters**. It is used to initialize the instance variables of a class with **user-defined or specific values** at the time of object creation. Parameterized constructors help ensure that objects are created in a meaningful and fully initialized state. Constructors do not have return value statements, but internally they **return the reference of the current class object**.

Copy Constructor

A **copy constructor** is used to create a **new object by copying the values of an existing object**. Java does not provide a built-in copy constructor, but it can be created by defining a constructor that takes an object of the same class as a parameter. Copy constructors are useful when an exact copy of an object is required without sharing the same memory location.

Example :

```
class Book {
    String title;
    int price;
    // 1. Default Constructor
    Book() {
        title = "Not Assigned";
        price = 0;
    }
    // 2. Parameterized Constructor
    Book(String t, int p) {
        title = t;
        price = p;
    }
// 3. Copy Constructor
   Book(Book b) {
      title = b.title;
      price = b.price;
 }
```

```
        void display() {
        System.out.println("Title: " + title);
        System.out.println("Price: " + price);
    }
}
public class Main {
    public static void main(String[] args) {
        // Object created using default constructor
        Book b1 = new Book();
        // Object created using parameterized constructor
        Book b2 = new Book("Java Programming", 500);
        // Object created using copy constructor
        Book b3 = new Book(b2);
        b1.display();
        b2.display();
       b3.display();
    }
}
```

In this example, the Book class contains three constructors. The **default constructor** initializes the object with default values. The **parameterized constructor** initializes the object with specific values provided at the time of object creation. The **copy constructor** creates a new object by copying the values of an existing object (b2). Although b2 and b3 have the same data, they are **separate objects with different memory locations**. This example clearly shows how different types of constructors are used to initialize objects in different ways.

Garbage Collector

Garbage Collection in Java is the automatic process of identifying and removing **unused or unreachable objects** from memory in order to free up space. An object becomes eligible for garbage collection when it is **no longer referenced** by any part of the program. By reclaiming this unused memory, Java helps prevent memory leaks and improves application performance.

One of the most important features of the Java programming language is its **automatic garbage collection mechanism**. Unlike languages such as C or C++, where memory allocation and de-allocation must be handled manually by the programmer, Java manages memory automatically. This reduces programming errors and makes Java applications safer and easier to maintain.

The **Garbage Collector** is a background program that runs inside the Java Virtual Machine (JVM). It continuously monitors objects in memory and determines which objects are **no longer in use**. Objects that do not have any active references are considered garbage. These unreferenced objects are automatically removed, and the memory occupied by them is **reclaimed** so it can be reused for creating new objects.

Garbage collection runs automatically, and programmers **do not need to explicitly free memory**. This automatic memory management is one of the key reasons Java is considered a robust and reliable programming language.

There are certain actions to be performed before an object is destroyed like:

- Closing all the database connections or files
- Releasing all the network resources
- Other Housekeeping tasks
- Recovering the heap space allocated during the lifetime of an object
- Release of release locks

Java provides a mechanism called finalization to do this through finalize( ) method.

finalize() method

The **`finalize()` method** in Java is a method that is called by the **Java Runtime Environment (JVM)** just before an object is removed from memory by the garbage collector. It is used to perform **cleanup operations**, such as releasing resources, before an object is destroyed. The `finalize()` method is defined in the `Object` class, and it can be overridden by a class to specify custom cleanup behavior.

The `protected` keyword is used so that the `finalize()` method cannot be accessed directly by code outside the class hierarchy. This ensures that only the JVM or subclasses can invoke

it. The method is **called just before garbage collection**, and **not when an object goes out of scope**. Simply losing scope does not guarantee that the object will be garbage collected.

The execution of the `finalize()` method is **not guaranteed**, because garbage collection itself is not guaranteed to run at any specific time. Therefore, programmers should not rely on `finalize()` for critical resource management. Due to its unpredictability and performance issues, the `finalize()` method has been **deprecated since Java 9**, and its use is strongly discouraged in modern Java applications.

Parameter Passing – Value Types and Reference Types

When working with **methods/functions**, we often need to pass data from one method to another.
This data is passed using **parameters and arguments**.

An **argument** is the actual value that is passed to a function when it is called.
A **parameter** is the variable that receives that value inside the function.

Types of Parameters

1. Formal Parameter

- These are variables declared in the function definition.
- They act as placeholders to receive values.
- They exist only inside the function.

Example:

```
void add(int a, int b) // a and b are formal parameters
```

2. Actual Parameter (Argument)

- These are the real values passed to the function when calling it.
- They match the formal parameters in number, order, and type.

Example:

```
add(5, 10);   // 5 and 10 are actual parameters
```

Parameter Passing Techniques

There are two main ways to pass parameters to functions:

Pass by Value (Value Types)

In **pass by value**, a **copy of the actual parameter** is passed to the function.

- **Changes made to formal parameter do not get transmitted back to the caller.**
- **Any modifications to the formal parameter variable inside the called function affect only the separate storage location and will not be reflected in the actual parameter in the calling environment.**

Explanation

When a function is called:

- The system creates a **new memory location** for the formal parameter.
- The value of the actual parameter is **copied** into this new location.
- The function works only on this copied value.
- After the function finishes, the copied value is destroyed.
- The original variable remains unchanged.

So, the function cannot change the original variable.

Example:

```
public class PassByValueExample {

    static void modify(int num) {
        System.out.println("Inside function before change: " + num);
        num = 100;  // modifying formal parameter
        System.out.println("Inside function after change: " + num);
    }

    public static void main(String[] args) {
        int number = 50;   // actual parameter
        System.out.println("Before function call: " + number);

        modify(number);   // passing value

        System.out.println("After function call: " + number);
    }
}
```

Output

```
Before function call: 50
Inside function before change: 50
Inside function after change: 100
After function call: 50
```

Why Output Remains 50?

- `number` (actual parameter) has value **50**.
- A copy of 50 is given to `num` (formal parameter).
- Inside the function, only the copy (`num`) changes to 100.
- The original variable `number` is not affected.

Pass by Reference (Reference Types / Alias)

In **pass by reference**, the function receives a **reference (memory address)** of the actual parameter instead of a copy.

- **Non-primitive type variables store references to objects.**
- **Changes made to formal parameter do get transmitted back to the caller through parameter passing.**
- **Any changes to the formal parameter are reflected in the actual parameter in the calling environment as the formal parameter receives a reference (or pointer) to the actual data.**

Explanation

When a reference type variable (like an object or array) is passed:

- The function receives the **reference to the original object**.
- Both the actual parameter and formal parameter refer to the **same memory location**.
- If the object is modified inside the function, the original object is also modified.
- This is called **aliasing**, because two variables refer to the same object.

So, changes inside the function affect the original data.

Example :

```
class Student {
    int marks;
}

public class PassByReferenceExample {

    static void modify(Student s) {
        System.out.println("Inside function before change: " + s.marks);
        s.marks = 90;   // modifying object data
        System.out.println("Inside function after change: " + s.marks);
    }

    public static void main(String[] args) {
        Student st = new Student();
        st.marks = 50;   // actual parameter

        System.out.println("Before function call: " + st.marks);

        modify(st);   // passing reference

        System.out.println("After function call: " + st.marks);
    }
}
```

Output

```
Before function call: 50
Inside function before change: 50
Inside function after change: 90
After function call: 90
```

Why Output Changes to 90?

- `st` stores a reference to the `Student` object.
- The reference is passed to `s`.
- Both `st` and `s` refer to the **same object**.
- When `s.marks` is changed to 90, the original object is modified.
- Therefore, `st.marks` also becomes 90.

Understanding Object, Reference, and Variable

Object

- An **object** is the real data created in memory.
- It is stored in the **heap memory**.
- It contains actual values (data members / fields) and methods.
- Example: When we write `new Student()`, a Student object is created in the heap.

The object holds the real data.

Reference

- A **reference** is the **memory address** of the object.
- It tells where the object is located in heap memory.
- It does not store actual data — it only points to the object.
- Similar to a pointer in C/C++.

The reference connects the variable to the object.

Variable

- A **variable** stores the reference value.
- In Java, object variables do not store the object directly.
- They store the reference (address) of the object.
- Object reference variables are usually stored in **stack memory** (if local variables).

The variable holds the reference, not the object itself.

Example Explanation :

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

What happens step-by-step?

1. `new Student()`
   - Creates a Student object in **heap memory**.
   - Memory is allocated for its data members.
2. The address (reference) of that object is returned.
3. `Student s`
   - A variable `s` is created (in stack memory if inside a method).
4. `s` stores the reference (memory address) of the Student object.

Method Overloading

Method Overloading is a feature in Java that allows a class to have **more than one method with the same name**, provided their **argument lists are different**.

It means:

- Same method name
- Different parameters

Other Names of Method Overloading

Method Overloading is also known as:

- **Compile-Time Polymorphism**
- **Static Polymorphism**
- **Early Binding**

It is called compile-time polymorphism because the method call is resolved at **compile time**, not runtime.

Rules for Method Overloading

To overload a method, the **argument list must be different** in one of the following ways:

1. Changing the Number of Parameters

```
int add(int a, int b)
int add(int a, int b, int c)
```

2. Changing the Data Type of Parameters

```
 int add(int a, int b)
double add(double a, double b)
```

3. Changing the Order of Parameters

```
void show(int a, String b)
void show(String b, int a)
```

NOTE : All overloaded methods must have same **METHOD NAME.**

What is Method Signature?

A **method signature** includes:

- Method name
- Parameter list (number, type, order)

Compiler decides which method to call (compile-time polymorphism) based on the **method signature : method name + parameter list (number, type, order)**

What is NOT Method Overloading?

- Changing only the **return type** is NOT overloading.

Benefits of using Method Overloading

1. Method overloading increases the readability of the program.
2. This provides flexibility to programmers so that they can call the same method for different types of data.
3. This makes the code look clean.
4. This reduces the execution time because the binding is done in compilation time itself.
5. Method overloading minimises the complexity of the code.
6. The code can be used again, which saves memory.

Example :

```
class Calculator {
   // Instance Variables
   int result;

   // Overloading by changing number of parameters
   int add(int a, int b) {
      result = a + b;
      return result;
```

```
}

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

// Overloading by changing data type of parameters
double add(double a, double b) {
    return a + b;
}

// Overloading by changing order of parameters
void display(int a, String message) {
    System.out.println("Integer: " + a + ", Message: " + message);
}

void display(String message, int a) {
    System.out.println("Message: " + message + ", Integer: " + a);
}

// Static method overloading
static int multiply(int a, int b) {
    return a * b;
}

static double multiply(double a, double b) {
    return a * b;
}
```

```
    // Constructor Overloading
    Calculator() {
        System.out.println("Default Constructor Called");
    }

    Calculator(int value) {
        result = value;
        System.out.println("Parameterized Constructor Called. Value = " + result);
    }
}

public class Main {
    public static void main(String[] args) {

        // Constructor Overloading
        Calculator obj1 = new Calculator();
        Calculator obj2 = new Calculator(50);

        System.out.println();

        // Method Overloading (Number of Parameters)
        System.out.println("Addition (2 ints): " + obj1.add(10, 20));
        System.out.println("Addition (3 ints): " + obj1.add(10, 20, 30));

        // Method Overloading (Different Data Type)
        System.out.println("Addition (double): " + obj1.add(5.5, 4.5));

        // Method Overloading (Order of Parameters)
```

```
        obj1.display(100, "Hello");
        obj1.display("World", 200);

        System.out.println();

        // Static Method Overloading
        System.out.println("Multiply (int): " + Calculator.multiply(4, 5));
        System.out.println("Multiply (double): " + Calculator.multiply(2.5, 4.0));
    }
}
```

Output :

```
Default Constructor Called
Parameterized Constructor Called. Value = 50

Addition (2 ints): 30
Addition (3 ints): 60
Addition (double): 10.0
Integer: 100, Message: Hello
Message: World, Integer: 200

Multiply (int): 20
Multiply (double): 10.0
```

Class Attributes and Methods

Types of Methods in Java

In Java, methods are mainly of two types:

1. **Instance Methods**
2. **Static Methods**

By default, methods are considered **instance methods** unless declared as `static`.

Instance Methods

Definition

Instance methods are methods that **require an object of the class** to be created before they can be called.

- They belong to an **object** of the class.
- They can access **instance variables (non-static variables)** directly.
- To invoke an instance method, we must create an object of the class.

Static Methods

Definition

Static methods are methods that can be called **without creating an object** of the class.

- They belong to the **class**, not to any object.
- They are declared using the `static` keyword.
- They are accessed using the **class name**.
- They cannot directly access non-static (instance) variables.

Types of Attributes (Variables) in Java

In Java, attributes (variables inside a class) are mainly of two types:

1. **Instance Variables**
2. **Static Variables**

By default, attributes are considered **instance variables** unless declared as `static`.

Instance Variables

Definition

Instance variables are variables that **belong to an object** of a class.

- They are declared inside a class but **outside methods**.
- They require an **object to be created** before they can be accessed.
- Each object gets its **own separate copy** of instance variables.
- They store data specific to an object.

Static Variables

Definition

Static variables are variables that **belong to the class**, not to objects.

- Declared using the `static` keyword.
- Shared by all objects of the class.
- Can be accessed **without creating an object**.
- Accessed using the **class name**.

The static keyword can be used for :

- Variable (also known as a class variable)
- Method (also known as a class method)
- Block

- Class

Class variables (or static fields)

- **Variables that are common to all objects**
- They are associated with the **class**, rather than with any **object**

- Every instance of the class shares a class variable, which is in one fixed location in memory
- Any object can change the value of a class variable
- Class variables can also be manipulated without creating an object

Static Method and Attribute: Example

```
class Employee {

    static String company = "TCS";   // static variable (class attribute)

    static void displayCompany() {   // static method

        System.out.println("Company Name: " + company);

    }

}

public class Main {

    public static void main(String[] args) {

        // No object creation required

        Employee.displayCompany();

        // Changing static variable

        Employee.company = "Infosys";

        Employee.displayCompany();

    }

}
```

Output :

Company Name: TCS

Company Name: Infosys

Explanation :

- company belongs to the class Employee, not to objects.
- displayCompany() is also static.
- We accessed both using the class name: Employee.company
- If multiple objects are created, they all share the same company.

P.T.O

Instance attributes and methods : Example

```
class Student {
    String name;   // instance variable
    int marks;     // instance variable
    void display() {   // instance method
        System.out.println("Name: " + name);
        System.out.println("Marks: " + marks);
        System.out.println();
    }
}
public class Main {
    public static void main(String[] args) {
        // First object
        Student s1 = new Student();
        s1.name = "Rahul";
        s1.marks = 85;
        // Second object
        Student s2 = new Student();
        s2.name = "Anita";
        s2.marks = 92;
    s1.display();
    s2.display();
  }
}
```

Output :

Name: Rahul

Marks: 85

Name: Anita

Marks: 92

Explanation :

- name and marks are instance variables.
- Each object (s1, s2) has its own separate data.
- Changing s1.marks does NOT affect s2.marks.
- The method display() is called using object reference: s1.display().

Static vs Instance Method

Feature	Static Method	Instance Method
Belongs to	Class	Object
Called using	Class name	Object
Object required	✗ No	✓ Yes
Access instance variables	✗ No	✓ Yes
Access static variables	✓ Yes	✓ Yes
Use of this	✗ Not allowed	✓ Allowed
Memory	One copy	One per object

Static Attribute vs Instance Attribute

Feature	Static Attribute	Instance Attribute
Belongs to	Class	Object
Copies in memory	One	One per object
Accessed using	Class name	Object
Value shared	Yes	No
this keyword	✗ Not allowed	✓ Allowed

Java Static Block

A **static block** in Java is used to **initialize static data members** of a class.

- It is written using the keyword `static`.
- It is executed **before the main method**.
- It runs when the class is **first loaded into memory**.
- It is executed **only once**, no matter how many objects are created.

Why We Use Static Block

- To initialize static variables.
- To perform tasks that should run only once.
- To execute configuration or setup code before the program starts.

Example of Static Block

```
class Test {

    static {
        System.out.println("Static block is invoked");
    }

    public static void main(String[] args) {
        System.out.println("Hello main");
    }
}
```

Output :

```
Static block
is invoked
Hello main
```

Explanation :

1. When the program starts, the JVM loads the `Test` class.
2. At the time of class loading, the **static block executes first**.
3. After that, the `main()` method runs.
4. The static block executes **only once**, even if multiple objects are created.

Static Class

A class can be declared **static only if it is a nested class** (a class inside another class).

- We **cannot declare a top-level class as static**.
- Only **inner (nested) classes** can be declared static.
- Such classes are called **Static Nested Classes**.

Important Rules

- A static nested class **does not require an object of the outer class**.
- It behaves like a normal class but is grouped inside another class.
- It **cannot access non-static members** of the outer class directly.
- It can access **only static members** of the outer class.

Example of Static Nested Class

```
class Outer {

    static int x = 10;       // static member
    int y = 20;              // non-static member

    static class Inner {

        void display() {
            System.out.println("Value of x: " + x);

            // System.out.println(y); ✗ Not allowed
        }
    }
}

public class Main {
    public static void main(String[] args) {

        // No need to create Outer object
        Outer.Inner obj = new Outer.Inner();
        obj.display();
    }
}
```

Output

```
Value of x: 10
```

Explanation

- `Inner` is a static nested class inside `Outer`.
- Since it is static: It does **not need** an object of `Outer`.
- It can access only: Static members of `Outer` (`x`)
- It cannot access: Non-static members (`y`) directly.

Inheritance

Inheritance is one of the most important pillars of **Object-Oriented Programming (OOP)**. It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class.

In simple words, inheritance allows a new class to reuse the properties and behavior of an existing class.

It helps in:

- Reducing code duplication
- Improving code organization
- Making programs more structured and hierarchical

IS-A Relationship

Inheritance represents an **IS-A relationship**, also known as a **parent-child relationship**.

This means the child class is a specialized form of the parent class.

For example:

- A Dog **IS-A** Animal
- A Car **IS-A** Vehicle
- A Manager **IS-A** Employee

The child class automatically gets access to the non-private members of the parent class.

Acquiring Properties (Data and Methods)

When one class inherits another class:

- It gets access to:
  - Instance variables (data members)
  - Methods (functions)

This allows the child class to:

- Use parent class methods directly
- Add new methods of its own
- Modify existing behavior if required

This mechanism promotes **code reusability**, meaning we do not need to rewrite the same code again in multiple classes.

Superclass (Parent / Base Class)

The class whose features are inherited is called:

- Superclass
- Parent class
- Base class

This class contains common properties that can be shared among multiple child classes.

It usually defines:

- General attributes
- Common behavior

Example:
If we have a class `Vehicle`, it may contain:

- speed
- fuelType
- start()
- stop()

These common features can be reused by subclasses like `Car`, `Bike`, or `Truck`.

Subclass (Child / Derived Class)

The class that inherits the properties of another class is called:

- Subclass
- Child class
- Derived class

The subclass:

- Inherits accessible members of the superclass
- Can add its own unique features
- Can provide its own implementation of inherited methods

This makes the subclass more specific compared to the superclass.

Reusability of Code

One of the biggest advantages of inheritance is **code reusability**.

Instead of writing the same methods again in multiple classes:

- Write common code once in the parent class.

- Allow child classes to reuse it.

This results in:

- Less code duplication
- Easier maintenance
- Better readability

If a change is needed in common behavior, we only update the parent class, and all child classes automatically reflect the change.

Syntax :

class Superclass-name

{

//methods and fields of super class

}

Syntax:

class subclass-name **extends** Superclass-name

{

//methods of super class which are overridden

//methods and fields of subclass

}

Example :

Parent Class (Superclass)

```
class Animal {
    String name;

    void eat() {
        System.out.println("Animal is eating");
    }

    void sleep() {
        System.out.println("Animal is sleeping");
    }
}
```

This is the **superclass**.
It contains common properties and behaviors that all animals can have:

- `name` (data/field)
- `eat()` method
- `sleep()` method

Child Class (Subclass)

```
class Dog extends Animal {

    void bark() {
        System.out.println("Dog is barking");
    }
}
```

- `Dog` **extends** `Animal`
- Dog **IS-A** Animal
- Dog inherits `name`, `eat()`, and `sleep()`

Dog also has its own specific behavior:

- `bark()`

Main Class to Test

```
public class TestInheritance {
 public static void main(String[] args) {

        Dog d = new Dog();

        d.name = "Tommy";
        System.out.println(d.name);

        d.eat();      // inherited method
        d.sleep();    // inherited method
        d.bark();     // child class method
    }
}
```

Output :

```
Tommy
Animal is eating
Animal is sleeping
Dog is barking
```

Advantages :

1. Reusability and Saves Time

Inheritance allows a subclass to reuse the parent class's code, reducing duplication and saving development time.

2. Enhances Readability:

It organizes code in a clear parent-child structure, making programs easier to understand.

3. Overriding

A subclass can provide its own version of a parent class method to change its behavior.

Types of Inheritance in Java :

1. Single Level Inheritance

One class inherits another class.

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

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

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // inherited
        d.bark();  // own method
    }
}
```

`Dog` inherits from `Animal`.

2. Multilevel Inheritance

A class inherits from a class that itself inherits another class.

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

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

class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy is weeping");
    }
}

public class Test {
    public static void main(String[] args) {
        Puppy p = new Puppy();
        p.eat();   // from Animal
        p.bark();  // from Dog
        p.weep();  // own method
    }
}
```

`Puppy` inherits properties of both `Dog` and `Animal`.

3. Hierarchical Inheritance

Multiple classes inherit from the same parent class.

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

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

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

public class Test {
    public static void main(String[] args) {
    Dog d = new Dog();
        d.eat();
        d.bark();
```

```
        Cat c = new Cat();
        c.eat();
        c.meow();
    }
}
```

Both `Dog` and `Cat` inherit from `Animal`.

4. Multiple Inheritance (Not Supported in Java with Classes)

Multiple inheritance means one class inherits from more than one class.

Java does **NOT support this using classes** to avoid ambiguity (diamond problem).

This is NOT allowed:

```
class A {}
class B {}
class C extends A, B {}  // Error
```

But Java supports multiple inheritance using **interfaces**:

```
interface A {
    void show();
}

interface B {
    void display();
}

class C implements A, B {
    public void show() {
        System.out.println("Show method");
    }

    public void display() {
        System.out.println("Display method");
    }
}

public class Test {
    public static void main(String[] args) {

        C obj = new C();

        obj.show();      // from interface A
        obj.display();   // from interface B
    }
}
```

5. Hybrid Inheritance (Partially Supported in Java)

Two or more types of inheritance—such as single, multiple, multilevel, or hierarchical—are combined to create a complex class hierarchy

It typically merges patterns like **hierarchical and multiple** inheritance or **single and multilevel** inheritance.

Java does not support multiple inheritance via classes; must use interfaces to achieve hybrid structures.

```
// Base class
class Person {
    String name;

    void setName(String name) {
        this.name = name;
    }

    void showName() {
        System.out.println("Name: " + name);
    }
}

// Multilevel inheritance
class Employee extends Person {
    int empId;

    void setEmpId(int empId) {
        this.empId = empId;
    }

    void showEmpId() {
         System.out.println("Employee ID: " + empId);
    }
}

// Multiple inheritance using interfaces
interface Coder {
    void code();
}

interface Trainer {
    void train();
}

class Developer extends Employee implements Coder,
Trainer {

    public void code() {
        System.out.println("Developer is coding");
    }

    public void train() {
        System.out.println("Developer is training
juniors");
    }
}
```

```
// Main class
public class TestHybrid {
    public static void main(String[] args) {

        Developer dev = new Developer();

        dev.setName("Rahul");
        dev.setEmpId(101);

        dev.showName();      // from Person
        dev.showEmpId();     // from Employee
        dev.code();          // from Coder interface
        dev.train();         // from Trainer interface
    }
}
```

Output :

```
Name: Rahul
Employee ID: 101
Developer is coding
Developer is training juniors
```

Constructors and Inheritance :

When an object of a subclass is created, the constructor of the subclass automatically calls the **default constructor of the superclass**. This happens because the parent class must be initialized before the child class. Therefore, in inheritance, object construction happens **from top to bottom** (superclass → subclass).

If the superclass has a **parameterized constructor**, then the subclass must explicitly call it using the `super()` keyword. Otherwise, a compile-time error will occur.

The `super` keyword is used to refer to the **immediate parent class** of the current class. It can be used to:

- Call the superclass constructor
- Access superclass methods
- Access superclass variables

When calling a superclass constructor using `super()`, it **must be the first statement inside the subclass constructor**. This ensures that the parent class is initialized before the child class starts its own initialization.

If `super()` is not written explicitly, Java automatically inserts a call to the **default (no-argument) constructor** of the superclass.

Method Overriding :

If a subclass (child class) has the same method as declared in the parent class, it is known as **method overriding** in Java.

In other words, when a subclass provides its own specific implementation of a method that is already defined in the parent class, it is called method overriding.

Method overriding allows the child class to modify the behavior of the parent class method according to its requirement.

Usage of Java Method Overriding

- It is used to provide a specific implementation of a method that is already provided by its superclass.
- It is used to achieve **runtime polymorphism**, where the method call is resolved at runtime based on the object.

Rules for Java Method Overriding

- The method must have the **same name** as in the parent class.
- The method must have the **same parameters** as in the parent class.
- There must be an **IS-A relationship** (inheritance).
- The return type must be the same (or covariant).
- The access modifier cannot be more restrictive than the parent method.

Example :

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

class Dog extends Animal {

    // Overriding method
    void sound() {
        System.out.println("Dog barks");
    }
}

public class TestOverride {
    public static void main(String[] args) {

        Animal a = new Dog();  // Parent reference, child object
        a.sound();             // Calls overridden method
    }
}
```

Output :

```
Dog barks
```

Explanation of Example :

- `Dog` class overrides the `sound()` method of `Animal`.
- Both methods have the same name and same parameters.
- There is an IS-A relationship (`Dog` IS-A `Animal`).
- Even though the reference is of type `Animal`, the method of `Dog` is executed.
- This demonstrates **runtime polymorphism**.

Abstract class

Introduction

- It provides **implementation reuse** and can also provide a **default implementation** for some methods.
- It is used to create a superclass that defines a **generalized form** that will be shared by all of its subclasses.
- The superclass defines the common structure, while each subclass fills in the specific details.
- The superclass determines the nature of methods that the subclasses must implement.
- If a method has no body in the superclass, it becomes the **responsibility of the subclass** to provide the implementation. This is known as **subclass responsibility**.

Such methods are called **abstract methods**, and they do not contain a method body.

Creation and Usage of Abstract Class in Java

- An abstract class is created using the **`abstract` keyword** at the beginning of the class declaration.
- It may contain **abstract methods** (methods without body) as well as normal (concrete) methods.
- If a class contains at least one abstract method, the class must be declared as abstract.
- An abstract class **cannot be instantiated** (object cannot be created directly).
- If a class extends an abstract class, it must either:
  - Provide implementation of all abstract methods, or
  - Declare itself as abstract.
- It can have both **static and non-static variables and methods** like a normal class.
- It cannot be declared as **final**, because abstract classes are meant to be extended.
- A class can extend only **one abstract class**, since Java does not support multiple inheritance with classes.

P.T.O

Example of Abstract Class

```
abstract class Vehicle {

    String brand;   // non-static data member

    // Constructor
    Vehicle(String brand) {
        this.brand = brand;
    }

    // Abstract method (no body)
    abstract void start();

    // Concrete method
    void display() {
        System.out.println("Brand: " + brand);
    }

    // Static method
    static void info() {
        System.out.println("Vehicles have engines.");
    }
}

class Car extends Vehicle {

    Car(String brand) {
        super(brand);
    }

    // Providing implementation of abstract method
    void start() {
        System.out.println("Car starts with a key.");
    }
}

public class TestAbstractUsage {
    public static void main(String[] args) {

        Vehicle.info();  // static method
        Vehicle v = new Car("Toyota");  // abstract class reference
        v.display();   // concrete method from abstract class
        v.start();     // implemented method in subclass
  }
}
```

Output

```
Vehicles have engines.
Brand: Toyota
Car starts with a key.
```

Explanation

- `Vehicle` is an abstract class created using the `abstract` keyword.
- It contains:
  - An abstract method `start()`
  - A concrete method `display()`
  - A static method `info()`
- Object of `Vehicle` cannot be created directly.
- `Car` extends `Vehicle` and provides implementation for `start()`.
- Since `Car` implements all abstract methods, it is not required to be abstract.

Abstract vs Interface :

Feature	Abstract Class	Interface
Keyword	abstract class	interface
Object Creation	Cannot create object	Cannot create object
Methods	Can have abstract + concrete methods	By default abstract methods (can have default & static methods from Java 8)
Variables	Can have instance variables	Only public static final (constants)
Constructors	✓ Allowed	✗ Not allowed
Access Modifiers	Can use any (private, protected, public)	Methods are public by default
Multiple Inheritance	✗ Not supported (for classes)	✓ A class can implement multiple interfaces
Purpose	Partial abstraction	Full abstraction (possible)

Points to Note :

Object Creation :

- **Abstract Class:**
  We cannot create an object directly from an abstract class because it may contain abstract methods (incomplete methods).
  Object can only be created using its subclass.
- **Interface:**
  We also cannot create an object of an interface directly because it only defines method declarations.
  A class must implement the interface to create objects.

Variables

- **Abstract Class:**
  Can have normal instance variables (non-static), static variables, final variables, etc.
  It behaves like a normal class regarding variables.
- **Interface:**
  Variables are by default **public, static, and final**.
  That means they are constants and their values cannot be changed.

Constructors

- **Abstract Class:**
  Constructors are allowed.
  They are used to initialize variables when a subclass object is created.
- **Interface:**
  Constructors are not allowed because interfaces cannot be instantiated.

Single Rooted Hierarchy and Object class

Introduction :

Same Base Class

In Java, all classes directly or indirectly inherit from the `Object` class.
This means every class has a common root, forming a **single-rooted hierarchy**.
Because of this, all objects share some common methods like `toString()`, `equals()`, and `hashCode()`.

Common Interface

Since every class extends `Object`, all objects can be treated as `Object` type.
This provides a common interface for handling different types of objects in a uniform way.
For example, any object can be stored in an `Object` reference variable.

It Enables Easy Memory Management

Because all objects are derived from a single root (`Object`), Java can manage memory more efficiently.
Implementation of Garbage Collector is became easy since required implementation is provided in the base class, enabling to send messages to every object.

Simplifies Argument Passing (Generic Handling)

Since all classes inherit from `Object`, we can pass different types of objects to a method using an `Object` reference.
This allows generic handling of various object types without writing separate methods for each type.

Example idea:

```
void display(Object obj) {
    System.out.println(obj);
}
```

This method can accept `String`, `Integer`, or any user-defined class object.

Singly-Rooted Hierarchy

Java follows a singly-rooted class hierarchy because every class ultimately extends `Object`. This design is common in most object-oriented programming languages and ensures consistency and uniform behavior across all classes.

Object Class :

- **Object Class as Superclass**

The `Object` class defined by Java is the **superclass of all other classes**, if no other superclass is explicitly mentioned.
If you do not extend any class, Java automatically extends the `Object` class for you.

Example:

```
class Student {
}
```

This is internally treated as:

```
class Student extends Object {
}
```

- **Root of Class Hierarchy**

`Object` is the **root of the entire class hierarchy** in Java.
Every class directly or indirectly inherits from it.
All objects, including arrays, get common methods like:

- `toString()`
- `equals()`
- `hashCode()`
- `getClass()`

Because of this, all Java objects share some common behavior.

- **Object Reference Variable**

A reference variable of type `Object` can refer to an object of **any class**, since every class is a subclass of `Object`.

Example:

```
Object obj;

obj = "Hello";      // String object
obj = 10;           // Integer object (autoboxing)
obj = new Student();
```

This allows **generic handling** of different types of objects.

- **Package**

The `Object` class is defined in the **`java.lang` package**.
This package is automatically imported in every Java program, so we do not need to import it explicitly.

Some Methods of Object class :

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

- **public final Class<?> getClass()**

  Returns the runtime class of this Object.

- **public int hashCode()**

  Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap.

  The general contract of hashCode is:

  Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer. This integer need not remain consistent from one execution of an application to another execution of the same application.

- **public boolean equals(Object obj)**

  Indicates whether some other object is "equal to" this one.

  The equals method implements an equivalence relation on non-null object references:

  Default Behavior: Compares memory reference (same as ==).

Often overridden for content comparison.

- **public String toString()**

  Returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

  The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object

  Default return : *ClassName@hashcode*

P.T.O

Overriding equals() Method

The `equals()` method is used to compare the **content of objects**, not their memory addresses.

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

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Overriding equals method
    public boolean equals(Object obj) {

        if (this == obj)          // same reference
            return true;

        if (obj == null || getClass() != obj.getClass())
            return false;

        Student s = (Student) obj;

        return id == s.id && name.equals(s.name);
    }
}

public class TestEquals {
    public static void main(String[] args) {

        Student s1 = new Student(1, "Rahul");
        Student s2 = new Student(1, "Rahul");

        System.out.println(s1.equals(s2));  // true
    }
}
```

Explanation

- By default, `equals()` compares memory addresses.
- Here, we override it to compare `id` and `name`.
- So even though `s1` and `s2` are different objects, it returns `true` because their data is the same.

Overriding toString() Method

The `toString()` method returns the **string representation of an object**.

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

    Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Overriding toString method
    public String toString() {
        return "Employee ID: " + id + ", Name: " + name;
    }
}

public class TestToString {
    public static void main(String[] args) {

        Employee e = new Employee(101, "Anita");

        System.out.println(e);  // automatically calls toString()
    }
}
```

Output

```
Employee ID: 101, Name: Anita
```

Explanation

- By default, `toString()` prints class name + hashcode.
- After overriding, it prints meaningful object information.
- `System.out.println(e)` automatically calls `e.toString()`.

Array, List and Stack

Collections Framework in Java

The **Collections Framework** is a unified architecture for storing and manipulating groups of objects.
A **collection** is simply an object that represents a group of objects (like a list, set, or queue).

It provides ready-made classes and interfaces to store, retrieve, and manipulate data efficiently.

Primary Advantages of Collections Framework

1. Reduces Programming Effort

The framework provides built-in data structures (like `ArrayList`, `HashSet`, `HashMap`) and algorithms (like sorting and searching).
Developers do not need to implement these structures from scratch, which saves time and reduces errors.

2. Increases Performance

The framework offers optimized and high-performance implementations of data structures.
Since different implementations follow the same interface, we can easily switch from one implementation to another (e.g., `ArrayList` to `LinkedList`) to improve performance based on requirements.

3. Provides Interoperability

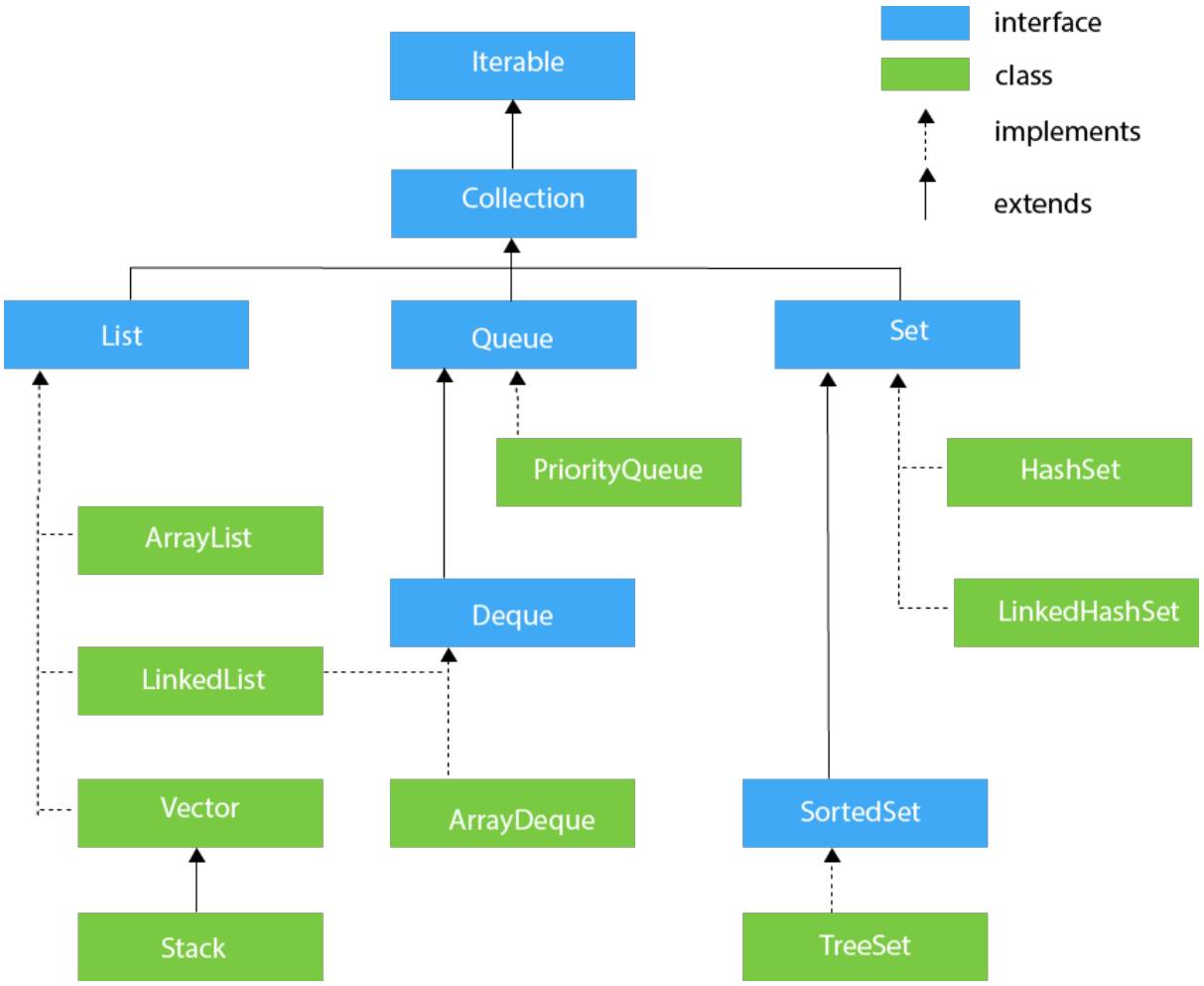
It establishes a common set of interfaces (like `Collection`, `List`, `Set`, `Map`).
This allows collections to be passed between different APIs easily because they follow a standard structure.

4. Reduces Learning Effort

Instead of learning different custom collection implementations for different libraries, developers only need to understand the standard framework interfaces and classes.

5. Fosters Software Reuse

Since the framework provides standard interfaces and reusable algorithms, developers can reuse existing code across different projects.
This promotes consistency and maintainability in applications.



Java Collections Framework Hierarchy Diagram

Array in Java

An array is an object that stores a **fixed number of similar elements** in **contiguous memory locations**.
All elements in an array must be of the **same data type**.

Arrays help in:

- Storing multiple values in a single variable
- Accessing elements using an index
- Managing data efficiently

Can Be of Any Type

Arrays can store:

- Primitive types (`int`, `double`, `char`, etc.)
- Reference types (`String`, objects, etc.)

Example:

```
int[] numbers;
String[] names;
```

Types of Arrays

One-Dimensional Array (1D)

Stores elements in a single row.

Two-Dimensional Array (2D)

Stores elements in rows and columns (matrix form).

Declaration of 1D Array

Different valid syntaxes:

```
dataType[] arr;
dataType []arr;
dataType arr[];
```

Instantiation (Memory Allocation):

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

Example:

```
int[] arr = new int[5];
```

Declaration of 2D Array

Different valid syntaxes:

```
dataType[][] arr;
dataType [][]arr;
dataType arr[][];
```

Instantiation:

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

Example:

```
int[][] matrix = new int[3][3];
```

Important Points

- Array index starts from **0**.
- Size of array is fixed after creation.
- Elements are accessed using index: `arr[0]`, `arr[1]`, etc.
- In 2D arrays, elements are accessed as: `arr[row][column]`.

For-Each Loop in Java

The **for-each loop** (enhanced for loop) is used to traverse array elements one by one.
It automatically picks each element from the array and stores it in a variable, then executes the loop body.

It is mainly used when:

- We only need to read elements
- We do not require index values

```
for(data_type variable : array) {
    // body of loop
}
```

- `data_type` → type of array elements
- `variable` → stores each element temporarily
- `array` → name of the array

Example

```
public class TestForEach {
    public static void main(String[] args) {

        int[] numbers = {10, 20, 30, 40};

        for(int num : numbers) {
            System.out.println(num);
        }
    }
}
```

Output

```
10
20
30
40
```

Arrays Class (java.util.Arrays)

The `Arrays` class belongs to the **java.util package** and is a part of the Java Collection Framework.

It provides **utility methods** to perform operations on arrays such as:

- Sorting
- Searching
- Converting to string
- Comparing arrays

Important Points

- It contains only **static methods**.
- Methods are accessed using the class name (`Arrays.methodName()`).
- It extends `java.lang.Object`.

Syntax (Class Declaration)

```
public class Arrays extends Object
```

Syntax to Use Arrays Methods

```
Arrays.functionName();
```

Example:

```
Arrays.sort(arr);
```

Commonly Used Functions

sort()

Sorts the array in ascending order.

```
import java.util.Arrays;

int[] arr = {5, 2, 8, 1};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
```

binarySearch()

Searches an element in a sorted array.

```
int index = Arrays.binarySearch(arr, 8);
System.out.println(index);
```

toString()

Returns string representation of array.

```
System.out.println(Arrays.toString(arr));
```

equals()

Compares two arrays.

```
int[] a1 = {1,2,3};
int[] a2 = {1,2,3};

System.out.println(Arrays.equals(a1, a2));
```

asList()

Converts array into List.

```
String[] names = {"A", "B", "C"};
System.out.println(Arrays.asList(names));
```

List in Java

The **List** interface in Java is used to maintain an **ordered collection** of elements.
It preserves the insertion order, meaning elements are stored and retrieved in the order they were added.

Features of List

- Provides **index-based methods** to insert, update, delete, and search elements.
- Allows **duplicate elements**.
- Allows storing **null values**.
- Elements can be accessed using index (starting from 0).

Package and Inheritance

- The `List` interface is available in the **`java.util` package**.
- It extends the **`Collection` interface**.

Declaration

```
public interface List<E> extends Collection<E>
```

Implementation Classes of List

- `ArrayList`

- `LinkedList`
- `Stack`
- `Vector`

Each implementation has different performance characteristics but follows the same List interface.

Few Common Methods of List

- `size()` → Returns number of elements
- `clear()` → Removes all elements
- `add(E e)` → Adds an element
- `add(int index, E e)` → Adds element at specific index
- `addAll()` → Adds all elements of another collection
- `contains()` → Checks if element exists
- `containsAll()` → Checks if all elements exist
- `equals()` → Compares two lists
- `hashCode()` → Returns hash code
- `isEmpty()` → Checks if list is empty
- `indexOf()` → Returns index of element

Example: Programming Demo

```
import java.util.*;

public class ListDemo {
    public static void main(String[] args) {

        List<String> list = new ArrayList<>();

        list.add("Apple");
        list.add("Banana");
        list.add("Apple");   // duplicate allowed
        list.add(null);      // null allowed

        System.out.println("List: " + list);
        System.out.println("Size: " + list.size());
        System.out.println("Contains Banana? " + list.contains("Banana"));
        System.out.println("Index of Apple: " + list.indexOf("Apple"));

        list.remove(1);  // remove element at index 1
        System.out.println("After removal: " + list);
    }
}
```

Output:

```
List: [Apple, Banana, Apple, null]
Size: 4
Contains Banana? true
```

```
Index of Apple: 0
After removal: [Apple, Apple, null]
```

ArrayList in Java

ArrayList is a resizable array implementation of the List interface.
It uses a dynamic array internally to store elements.

- It has no fixed size limit (size grows automatically).
- Elements can be added or removed at any time.
- It maintains insertion order.
- It allows duplicate elements.
- It allows null values.
- It is found in the `java.util` package.
- Since it implements the List interface, all List methods can be used.

Declaration

```
public class ArrayList<E>
        extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, Serializable
```

Creating a Generic ArrayList

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

(From Java 7 onward, we can also write:)

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

Primitive Types Not Allowed

We cannot create an ArrayList of primitive types such as `int`, `float`, `char`, etc.
Instead, we must use their wrapper classes.

Incorrect:

```
ArrayList<int> al = new ArrayList<int>();   // ❌ Not allowed
```

Correct:

```
ArrayList<Integer> al = new ArrayList<Integer>();   // ✅ Allowed
```

Few Common Methods

- add()

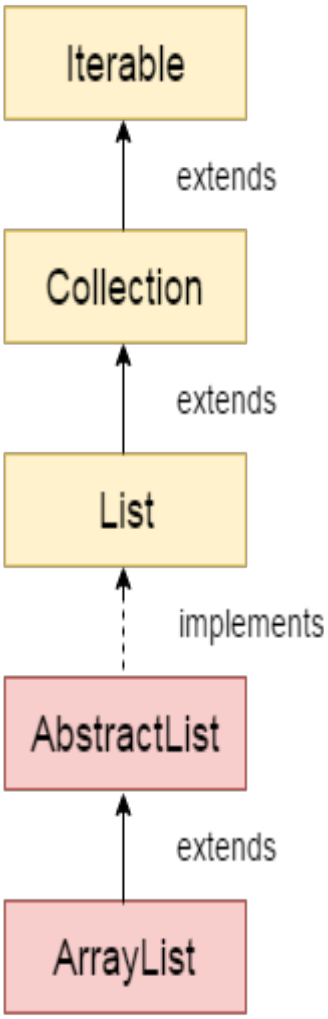
- addAll()
- clear()
- clone()
- contains()
- remove()
- removeAll()
- replaceAll()
- size()
- get()
- set()

Example: Programming Demo

```
import java.util.*;

public class ArrayListDemo {
    public static void main(String[] args) {

        ArrayList<String> list = new ArrayList<>();

        list.add("Apple");
        list.add("Banana");
        list.add("Apple");   // duplicate allowed
        list.add(null);      // null allowed

        System.out.println("List: " + list);

        list.remove("Banana");
        System.out.println("After removal: " + list);

        System.out.println("Contains Apple? " +
list.contains("Apple"));
        System.out.println("Size: " + list.size());
    }
}
```

Output (Example)

```
List: [Apple, Banana, Apple, null]
After removal: [Apple, Apple, null]
Contains Apple? true
Size: 3
```



Hierarchy of ArrayList in Java Collections Framework

Introduction to Stack Class in Java

- Java Collection framework provides a Stack class that models and implements a **Stack data structure**.
- The class is based on the basic principle of last-in-first-out [LIFO].

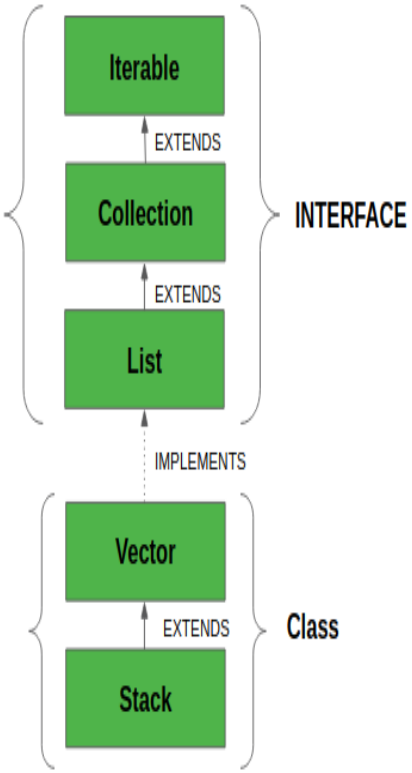
Declaration:

public class Stack<E> extends Vector<E>

Implemented interfaces :

- **Serializable:** It is a marker interface that classes must implement if they are to be serialized and deserialized.
- **Cloneable:** This is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.

- **Iterable<E>:** This interface represents a collection of objects which is iterable — meaning which can be iterated.
- **Collection<E>:** A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.
- **List<E>:** The List interface provides a way to store the ordered collection. It is a child interface of Collection.
- **RandomAccess:** This is a marker interface used by List implementations to indicate that they support fast (generally constant time) random access.



Hierarchy of List Interface with Vector and Stack in Java Collections Framework

OO Development Process, System Design and Frameworks

Object Oriented Development Process

The essence of application concepts lies in identifying and organizing the important ideas, entities, and relationships within a problem domain rather than focusing immediately on how they will be written in a specific programming language.
It emphasizes understanding the problem clearly before implementing the solution.

This approach encourages software developers to work and think in terms of the application domain throughout the entire software life cycle — from requirements gathering and analysis to design, implementation, testing, and maintenance.
It ensures that development is driven by business needs rather than technical constraints.

It is a conceptual process that remains independent of any programming language until the final implementation stage.
The main goal is to model real-world concepts first and worry about coding details later.
This helps in building solutions that truly reflect the real-world system being represented.

Application concepts represent a way of thinking.
Their greatest benefit is that they help developers and customers express abstract ideas clearly and communicate effectively with each other.
This shared understanding reduces misunderstandings, avoids ambiguity, and improves overall system design quality.

Object-Oriented (OO) concepts such as classes, objects, relationships, and interactions are often used to express these designs.
They help structure the system logically and also serve as documentation that explains how the system is organized and how different parts interact.

By focusing on application concepts, software development becomes more organized, clearer, and easier to maintain over time.
It also improves scalability, because systems designed around clear concepts can be extended more easily.
Changes in requirements can be handled more effectively since the underlying conceptual model remains stable.

Additionally, this approach promotes modularity and separation of concerns, making systems more flexible and adaptable.
It supports better testing and validation because the conceptual model provides a clear reference for expected system behavior.

Overall, concentrating on application concepts leads to better analysis, clearer design decisions, improved communication among stakeholders, and higher quality software systems.

Object Oriented Methodology

This approach emphasizes how to systematically go about developing a system or application.
It focuses on analyzing the problem domain, identifying objects, defining their responsibilities, and modeling their interactions before implementation.

It encourages and facilitates the reuse of software components.
By designing systems using modular and well-defined objects, existing components can be reused in new applications, reducing development time and cost.

It employs the international standard **Unified Modeling Language (UML)** defined by the Object Management Group (OMG).
UML provides a standardized way to visualize, specify, construct, and document the components of a software system through diagrams such as class diagrams, sequence diagrams, and use case diagrams.

A system can be developed on a component basis.
This means that individual components (classes or modules) can be independently developed, tested, and reused in other systems.
Component-based development promotes scalability, maintainability, and easier system enhancement.

This approach asks the analyst to determine:

- What are the objects in the system?
- What responsibilities does each object have?
- How do objects relate to one another?
- How do they collaborate to achieve system goals?
- How do they behave and change over time?

It also focuses on defining object attributes (data) and methods (behavior), ensuring that each object has clear responsibilities and minimal dependency on others.
By clearly identifying relationships such as association, inheritance, and aggregation, the overall structure of the system becomes more organized.

Overall, this approach improves clarity, promotes modular design, supports reuse, and helps build systems that are easier to understand, modify, and extend over time.

This approach emphasizes how to systematically go about developing a system or application.
It provides a structured method for analyzing requirements, designing solutions, and implementing systems using object-oriented principles.

It encourages and facilitates the reuse of software components.
By designing systems as a collection of independent and reusable objects or modules, existing components can be reused in other applications, reducing development time, cost, and effort.

It employs the international standard **Unified Modeling Language (UML)** defined by the Object Management Group (OMG).
UML provides a common visual language to model systems using diagrams such as:

- Use Case Diagrams
- Class Diagrams
- Sequence Diagrams
- Activity Diagrams

These diagrams help in clearly representing system structure and behavior.

A system can be developed on a component basis.
This means the system is divided into smaller, manageable components that can be developed and tested independently.
Component-based development supports scalability, maintainability, and effective reuse of existing system components.
It also enables sharing of components across different projects.

This approach asks the analyst to determine:

- What are the objects in the system?
- What responsibilities does each object have?
- How do objects relate to one another?
- What interactions occur between objects?
- How do objects behave and change over time?

It focuses on defining:

- Attributes (data of the object)
- Methods (behavior of the object)
- Relationships (association, inheritance, aggregation, composition)

By clearly identifying object responsibilities and interactions, the system becomes more organized, modular, and easier to understand.

Overall, this approach improves clarity in design, supports reuse, enhances communication among stakeholders, and leads to systems that are easier to maintain and extend over time.

Stages of Object Oriented Methodology

System Conception

Software development begins with business analysis or when users conceive an application and formulate tentative requirements.
At this stage, the main objective is to identify the need for the system and define its purpose.

The focus is on:

- Understanding business goals
- Identifying stakeholders
- Defining the scope of the application
- Clarifying initial expectations

This stage lays the foundation for the entire development process.

Analysis

During analysis, the analyst works closely with the requestor or client to fully understand the problem.
Problem statements are often incomplete, ambiguous, or incorrect, so careful clarification is necessary.

The analysis model represents a precise abstraction of what the system must do, not how it will be implemented.
It should avoid technical or implementation decisions.

The analyst must:

- Understand the big picture of the application
- Identify users and stakeholders
- Determine what problems the system will solve
- Identify when, where, and why the system is needed
- Define system functionality and constraints
- Understand and model the workflow of the application

The output of analysis is a clear and well-structured representation of system requirements.

System Design

In the system design stage, the development team devises a high-level strategy called the system architecture to solve the application problem.

This stage focuses on:

- Dividing the system into major subsystems
- Defining system components and their interactions
- Choosing architectural patterns or frameworks

- Deciding on technologies and platforms
- Establishing performance, security, and scalability strategies

System design translates requirements into a structured solution framework.

Class Design

During class design, detailed refinement of the analysis model takes place according to the system design strategy.

The class designer:

- Defines classes and their attributes
- Specifies methods and responsibilities
- Establishes relationships between classes
- Designs data structures and algorithms
- Applies object-oriented principles such as encapsulation and inheritance

The focus is on how each class will internally function while maintaining alignment with the overall architecture.

Implementation

In the implementation stage, developers translate the designed classes and relationships into a specific programming language, database structure, or hardware platform.

This stage involves:

- Writing code based on class design
- Integrating components
- Performing testing and debugging
- Ensuring traceability between implementation and design

It is important to follow good software engineering practices so that:

- The system remains flexible and extensible
- Code reflects the design clearly
- Future modifications are easier to handle

System Design

System design is the first design stage where the basic approach to solving the problem is defined.
In this stage, developers decide how the problem will be solved, starting at a high level and gradually adding more detail.

It focuses on defining the overall structure and style of the system.

A high-level strategy, known as the **system architecture**, is applied to build the solution. System architecture determines how the system is organized into subsystems and how these subsystems interact.

It also provides the foundation and context for detailed design decisions made in later stages.

Decisions:

1. Estimating performance
2. Making a Reuse plan
3. Breaking system into sub-systems
4. Identifying Concurrency
5. Allocation of Sub Systems to hardware
6. Manage data storage
7. Handling global resources
8. Choosing a software control strategy
9. Handling boundary conditions
10. Set trade-off priorities
11. Select an architectural Style

Estimating Performance:

Preparing a rough performance estimate means making a quick and approximate calculation to check whether the system idea is feasible.
The goal is not high accuracy, but to get a reasonable understanding of system capacity and limitations.

It involves making simplifying assumptions.
Instead of worrying about detailed technical factors, we approximate, estimate, and make educated guesses.

This helps answer questions like:

- Can the system handle expected load?
- Will performance be acceptable?
- Is storage capacity sufficient?

Example: Online Movie Ticket Booking System

Suppose a cinema chain has 20 theaters.
Each theater has 5 screens.
Each screen has 200 seats.

Total seats per show across all theaters:
$20 \times 5 \times 200 = 20{,}000$ seats per show.

Assume during peak time, all seats are booked within 10 minutes.

That means:
20,000 bookings ÷ 10 minutes = 2,000 bookings per minute.

If each booking requires 2 database operations (store booking + update seat availability),
Total database operations per minute ≈ 4,000 operations.

This rough estimate helps determine if:

- The server can handle 2,000 user requests per minute
- The database can process 4,000 operations per minute

Rough Storage Estimate

Assume each booking record requires 500 bytes of storage.
If 20,000 bookings occur per show:

20,000 × 500 bytes = 10,000,000 bytes
≈ 10 MB per show.

If there are 5 shows per day:
10 MB × 5 = 50 MB per day.

This quick estimation helps determine approximate storage needs.

Making a reuse plan:

Reuse is one of the major advantages of object-oriented development, but it does not happen automatically.
It requires proper design, planning, and discipline to make components reusable.

There are two different aspects of reuse:

1. **Using existing things**
   This involves reusing already developed components, libraries, frameworks, or patterns in new applications.
   It saves development time, reduces cost, and improves reliability since reused components are often tested and stable.
2. **Creating reusable things**
   This involves designing and developing components in such a way that they can be reused in future projects.
   It requires careful design, clear interfaces, low coupling, and high cohesion.

It is much easier to reuse existing components than to design new reusable components.
Designing reusable software requires extra effort, proper abstraction, and generalization.

In practice:

- Most developers reuse existing components.
- Only a smaller group of developers focus on creating reusable components for others.

Reusable things include:

- **Models** – Conceptual designs that can be applied to similar problems.
- **Libraries** – Prewritten collections of classes and functions.
- **Frameworks** – Structured platforms that provide reusable architecture and components.
- **Patterns** – Proven design solutions to common problems.

Effective reuse improves productivity, reduces duplication of effort, enhances consistency, and leads to more reliable and maintainable systems.

Libraries:

A class library is a collection of reusable classes designed to perform related tasks in many different applications.
It helps developers avoid rewriting common functionality and improves productivity and reliability.

Libraries are usually organized into packages or modules so that users can easily find relevant classes.
Each class should have clear documentation describing its purpose, methods, and usage.

Qualities of a Good Class Library

- **Coherence** – Organized around a clear and focused theme or domain.
- **Completeness** – Provides all necessary functionality related to its purpose.
- **Consistency** – Maintains uniform naming conventions and method signatures.
- **Efficiency** – Provides optimized implementations and may offer alternatives that balance time and space.
- **Extensibility** – Allows users to extend or customize classes when needed.
- **Genericity** – Uses generic (parameterized) classes to make components flexible and reusable.

A class library performs a set of specific and well-defined operations and can be reused across multiple systems.

Examples include:

- Network protocol libraries
- Compression libraries
- Image manipulation libraries
- String utility libraries
- Regular expression libraries

Using class libraries improves code reuse, reduces errors, and speeds up development.

Patterns:

A pattern is a best practice or a proven solution to a recurring problem in software development.
It is a time-tested approach that has been applied successfully in many situations.

Patterns provide guidance on how to solve common problems effectively.
They include recommendations on when to use them and explain the trade-offs involved.

Patterns can be applied at different stages of development, including:

- Analysis patterns
- Architectural patterns
- Design patterns
- Implementation patterns

A pattern usually involves a small number of classes and well-defined relationships between them.

The main advantage of using patterns is that they are already validated solutions.
They have been studied, refined, and successfully used in past projects.

Software Architecture Patterns define the overall structure of a system.
Examples:

- Layered Pattern
- Client-Server Pattern

Design Patterns focus on solving object-level design problems.
The well-known *Gang of Four (GoF)* categorized them into three types:

- **Creational Patterns** – Deal with object creation.
- **Structural Patterns** – Define relationships between classes and objects.
- **Behavioral Patterns** – Define how objects interact and communicate.

Using patterns improves design quality, promotes reuse, and provides a common vocabulary for developers.

Framework:

A framework provides a skeletal structure or ready-made architecture for building applications.
It defines the overall structure of the system and provides predefined classes, interfaces, and components that developers can extend.

The skeletal structure must be elaborated to build a complete application.
This elaboration usually involves specializing abstract classes and adding application-specific behavior.

A framework consists of more than just a collection of classes.
It also defines a flow of control and a set of rules or conventions that guide how components interact.
This concept is often referred to as *inversion of control* — the framework controls the execution and calls the developer's code when needed.

Frameworks are commonly used in Java development to build applications and web applications.
They provide reusable design structures for a specific type of software.

A framework is essentially a set of cooperating classes that together form a reusable design for a particular category of applications.

It acts like a skeleton that developers build upon by writing their own code.
Unlike libraries, where the developer calls functions as needed, frameworks manage the overall program flow and invoke the developer's code at appropriate times.

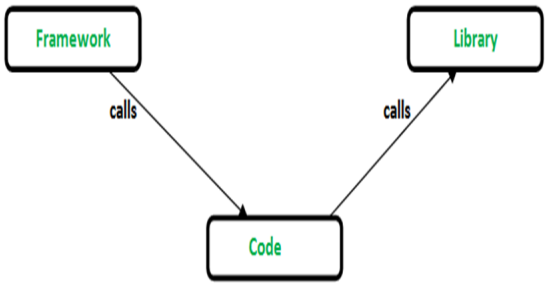
Frameworks provide architectural guidance by partitioning the design into abstract classes and predefined components.
Developers typically customize a framework by:

- Subclassing framework classes
- Implementing required interfaces
- Composing instances of framework components

This allows the framework to remain generic while enabling application-specific customization.

In summary:

- **Libraries** → You call the library.
- **Frameworks** → The framework calls your code.



Frameworks promote consistency, reuse, faster development, and adherence to a structured architecture.

Frameworks in Java:

- Collections: Provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).
- Swing: The part of JFC (Java Foundation Classes) built on the top of AWT and written entirely in Java. The javax.swing API provides all the component classes like JButton, JTextField, JCheckbox, JMenu, etc.
- AWT: An abstract window toolkit that provides various component classes like Label, Button, TextField, etc., to show window components on the screen. All these classes are part of the Java.awt package.
- Spring, Hibernate, Grails, Play, JavaServer Faces (JSF), Google Web Toolkit (GWT), Quarkus

Architectural Patterns

Definition

An architectural pattern is a general and reusable solution to a commonly occurring problem in software architecture.
It provides a high-level blueprint for designing the structure of an application.

Architectural patterns define:

- System structure
- Component interaction
- Data flow
- Overall behavior of the application

They guide developers in organizing code and system components effectively.

Purpose of Architectural Patterns

Architectural patterns help:

- Define the fundamental structure of a software system.
- Improve system scalability, maintainability, and performance.
- Reduce development risk by using proven design solutions.
- Ensure consistency and clarity in large systems.

Some patterns are better suited for:

- High scalability (e.g., Microservices, Space-Based)
- High agility and flexibility (e.g., Event-Driven)
- Simplicity and maintainability (e.g., Layered Architecture)

Choosing the correct architecture depends on:

- Business requirements
- Performance needs
- Team size and expertise
- Deployment environment

Problems Addressed by Architectural Patterns

Architectural patterns help address major software engineering challenges such as:

- Hardware performance limitations
- System scalability
- High availability and fault tolerance
- Security concerns
- Business risk minimization

- System maintainability and evolution

Common Architectural Patterns :

Layered Architecture (N-Tier Architecture), Event-Driven Architecture, Microkernel Architecture, Microservices Architecture, Space-Based Architecture, MVC (Model-View-Controller)

MVC Architechture :

MVC is an architectural pattern used in software engineering for structuring applications.
It was first introduced by **Trygve Reenskaug** in 1979 while working on Smalltalk at Xerox PARC.

It helps to decouple data access and business logic from the way information is presented to the user.
This separation ensures that changes in the user interface do not directly affect the core business logic and vice versa.

MVC is a way of designing and building applications by clearly separating application logic from presentation logic.
This makes the system more organized, maintainable, and easier to modify.

The application is divided into three main logical components:

Model – Handles data and business logic. It represents the application's core functionality and manages data operations.

View – Responsible for displaying data to the user. It represents the graphical or user interface part of the application.

Controller – Manages user input and interactions. It processes requests, communicates with the model, and selects the appropriate view.

MVC is considered a design pattern that distinguishes clearly between:

- The data model
- The processing/control logic
- The user interface

It neatly separates the graphical interface shown to the user from the code that handles user actions.
This ensures that calculations and business rules remain completely independent from the presentation layer.

By completely separating calculations and interface, MVC improves clarity, modularity, and maintainability of software systems.

Architecture:

Consists of Data model, presentation information and control information.

Model: The model represents data and the rules that govern access to and updates of this data. In enterprise software, a model often serves as a software approximation of a real-world process.

View: Renders the contents of a model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed. This can be achieved by using a push model, in which the view registers itself with the model for change notifications, or a pull model, in which the view is responsible for calling the model when it needs to retrieve the most current data. Represents the presentation layer of application. It is used to visualize the data that the model contains.

Controller: The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in an enterprise web application, they appear as GET and POST HTTP requests. Depending on the context, a controller may also select a new view -- for example, a web page of results -- to present back to the user

The MVC pattern needs all these components to be separated as different objects.

Example:

Consider an **Online Shopping Application**.

When a user opens the website and searches for a product:

Model
The Model contains the product data, prices, stock details, and business rules.
When a search request is made, the Model retrieves the matching products from the database and applies any necessary logic (like filtering or sorting).

View
The View displays the product list on the screen.
It shows product names, prices, images, and availability to the user.
The View does not handle how the data is fetched — it only presents it.

Controller
The Controller receives the user's search input (for example, typing "laptop" and clicking search).
It processes this request, calls the Model to get matching products, and then sends the results to the View to display.

Advantages:

- Faster Development Process
- Model can be reused with multiple views
- Less dependency among components – loose coupling
- The modification does not affect the entire model
- Application becomes more understandable
- Not a single massive codebase

MVC Architecture in Java:

The **Model** contains simple Java classes (POJOs).
These classes represent the data and business logic of the application.
They interact with the database, perform calculations, and return processed data.

The **View** is responsible for displaying data to the user.
In Java web applications, the view is usually implemented using **JSP (JavaServer Pages)** or HTML pages.
The view does not contain business logic; it only presents the data.

The **Controller** contains the **Servlets** (or related request-handling components).
It receives user requests, processes them, calls the model to get or update data, and forwards the result to the view.

Flow of User Request in MVC (Servlet-Based Application)

1. A client (browser) sends a request to the server.
2. The request is received by the **Controller (Servlet)**.
3. The controller processes the request and calls the **Model**.
4. The model performs business logic and returns the required data.
5. The controller sets the data as request attributes.
6. The controller forwards the request to the **View (JSP)**.
7. The view displays the data and sends the response back to the browser.

This clear separation ensures:

- Business logic remains in Model
- Request handling remains in Controller
- Presentation remains in View

Simple Code Example (MVC using Servlet)

1. Model (Java Class)

```
// Model class
public class Student {

    private String name;
    private int marks;

    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    public String getName() {
        return name;
    }

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

2. Controller (Servlet)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class StudentController extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, IOException {

        // Create model object
        Student student = new Student("Rahul", 85);

        // Send data to view
        request.setAttribute("studentData", student);

        // Forward to JSP (View)
        RequestDispatcher rd = request.getRequestDispatcher("student.jsp");
        rd.forward(request, response);
    }
}
```

3. View (student.jsp)

```
<%@ page import="yourpackage.Student" %>
<%
    Student s = (Student) request.getAttribute("studentData");
%>

<html>
<body>
    <h2>Student Details</h2>
    Name: <%= s.getName() %><br>
    Marks: <%= s.getMarks() %>
</body>
</html
```