* **OOPS concepts(Classes and Objects)**

## ****Java Built-in (Predefined) Classes****

* Java comes with **thousands** of built-in classes, organized into **packages**.
* **Here is main built-in classes** by their **packages**

### 📦 ****1. java.lang****

👉 Automatically imported in every program.  
Contains the **most commonly used** classes.

| **Purpose** | **Class Name** |
| --- | --- |
| Object representation | Object, Class, System |
| Strings & wrappers | String, StringBuilder, StringBuffer, Integer, Double, Boolean |
| Math & numbers | Math, Number |
| Threads & errors | Thread, Runnable, Exception, Error |
| Others | Enum, Runtime |

### 📦 ****2. java.util****

👉 Utility classes — data structures, date/time, etc.

| **Purpose** | **Class Name** |
| --- | --- |
| Collections | ArrayList, HashMap, HashSet, LinkedList, TreeMap, TreeSet |
| Date & time | Date, Calendar, TimeZone |
| Utilities | Random, Scanner, Objects, Optional |

### 📦 ****3. java.io****

👉 Input and Output (File handling).

| **Purpose** | **Class Name** |
| --- | --- |
| File handling | File, FileReader, FileWriter, BufferedReader, BufferedWriter |
| Streams | InputStream, OutputStream, FileInputStream, FileOutputStream |

### 📦 ****4. java.net****

👉 Networking.

| **Purpose** | **Class Name** |
| --- | --- |
| Networking | Socket, ServerSocket, URL, URLConnection |

### 📦 ****5. java.time****

👉 Modern Date & Time API (since Java 8).

| **Purpose** | **Class Name** |
| --- | --- |
| Date-Time | LocalDate, LocalTime, LocalDateTime, Instant, Duration, Period |

### 📦

### ****6. java.sql****

👉 Database connectivity (JDBC).

| **Purpose** | **Class Name** |
| --- | --- |
| JDBC | Connection, Statement, PreparedStatement, ResultSet, DriverManager |

### 📦 ****7. javax.servlet, javax.swing, javafx, etc.****

👉 For **Web** and **GUI** applications.

| **Purpose** | **Example Classes** |
| --- | --- |
| Servlets | HttpServlet, ServletRequest, ServletResponse |
| Swing GUI | JFrame, JButton, JLabel, JPanel |
| JavaFX | Application, Stage, Scene |

## 1.Object ****Class****

### Package:

java.lang.Object

### Description:

* The **Object class is the root (parent)** of all Java classes.
* Every class in Java **directly or indirectly** inherits from it.
* So, if you create a class like this:

class Student {}

Java internally treats it as:

class Student extends Object {}

### ⚙️ ****Important Methods of**** Object ****class****

| **Method** | **Description** |
| --- | --- |
| equals(Object obj) | Compares two objects for equality. |
| hashCode() | Returns a hash code value for the object. Used in hashing (e.g., HashMap). |
| toString() | Returns a string representation of the object. |
| getClass() | Returns the runtime class of the object (an instance of Class). |
| clone() | Creates and returns a copy of this object (used for object cloning). |
| finalize() | Called before an object is destroyed by the garbage collector. (Deprecated now) |
| wait(), notify(), notifyAll() | Used for thread communication (in multithreading). |

### 💡 Example:

class Student {

String name;

int age;

Student(String name, int age) {

this.name = name;

this.age = age;

}

// overriding toString()

public String toString() {

return "Student{name='" + name + "', age=" + age + "}";

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Ratnmala", 24);

System.out.println(s1.toString()); // or simply System.out.println(s1);

}

}

🟢 Output:

Student{name='Ratnmala', age=24}

👉 toString() is from Object class and is **commonly overridden** to give meaningful info.

## 2.Class ****Class****

### Package:

java.lang.Class

* Description:
* Represents the **metadata (structure)** of a class at runtime.
* Each time you create an object, a **Class object** is created internally to describe that object’s type.
* You can use it for **Reflection** — inspecting classes, fields, methods, and constructors at runtime.

### ⚙️ ****Commonly Used Methods of**** Class

| **Method** | **Description** |
| --- | --- |
| getName() | Returns class name. |
| getSuperclass() | Returns the superclass. |
| getDeclaredMethods() | Returns all declared methods. |
| getDeclaredFields() | Returns all declared fields. |
| newInstance() | Creates a new instance (deprecated, use constructor.newInstance()). |

### 💡 Example:

class Student {

String name;

int age;

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student();

Class<?> c = s1.getClass();

System.out.println("Class name: " + c.getName());

System.out.println("Superclass: " + c.getSuperclass().getName());

}

}

🟢 Output:

Class name: StudentSuperclass: java.lang.Object

👉 Here, getClass() returns a **Class object** representing the Student class.

## 3.System ****Class****

### Package:

java.lang.System

### Description:

* The System class provides **system-related utilities** — input/output, system properties, environment variables, and garbage collection.
* It **cannot be instantiated** — all its fields and methods are **static**.

### ⚙️ ****Commonly Used Methods of**** System

| **Method** | **Description** |
| --- | --- |
| System.out.println() | Prints output to the console. |
| System.err.println() | Prints error messages. |
| System.in | Reads input (used with Scanner). |
| System.currentTimeMillis() | Returns current time in milliseconds. |
| System.nanoTime() | Used for precise time measurement. |
| System.exit(status) | Terminates JVM (0 = normal exit). |
| System.gc() | Suggests JVM to run Garbage Collector. |
| System.getenv() | Returns system environment variables. |
| System.getProperty(String key) | Returns system property value. |

### 💡 Example:

public class SystemExample {

public static void main(String[] args) {

System.out.println("Hello, World!"); // prints to console

long start = System.currentTimeMillis();

for (int i = 0; i < 1000000; i++) {}

long end = System.currentTimeMillis();

System.out.println("Time taken: " + (end - start) + " ms");

System.out.println("Java Version: " + System.getProperty("java.version"));

System.out.println("User Directory: " + System.getProperty("user.dir"));

}

}

🟢 Output (example):

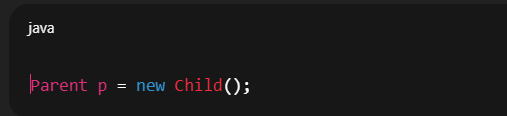
Hello, World!Time taken: 2 msJava Version: 21User Directory: C:\Users\Ratnmala\ecommerce

## 🧩 Summary Table

| **Class** | **Role** | **Key Methods / Uses** |
| --- | --- | --- |
| Object | Base class of all classes | equals(), toString(), hashCode(), getClass() |
| Class | Represents class metadata | getName(), getSuperclass(), getDeclaredMethods() |
| System | Provides system utilities | out.println(), gc(), exit(), getProperty() |

# 1. General Rule of Object Creation

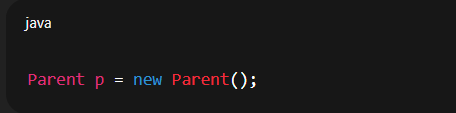
In Java, when you create an object:



* **Constructor executed →** depends on the object created (Child() constructor will run).
* **Reference type →** decides what methods/variables are accessible at **compile-time**.
* **Method execution →** depends on **runtime polymorphism (dynamic dispatch)**.

# 2. Cases of Object Creation

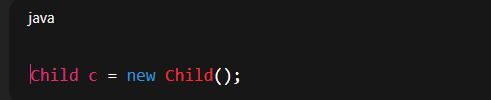
### ✅ Case 1: Parent reference, Parent object



· Calls Parent() constructor.

· Reference type: Parent → you can only access Parent members.

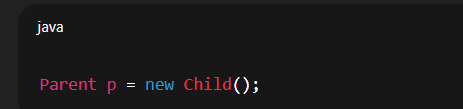
**✅ Case 2: Child reference, Child object**



· Calls Child() constructor (but first Parent() constructor executes automatically → constructor chaining).

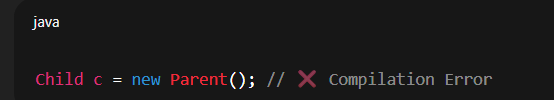
· Reference type: Child → you can access both Parent and Child members.

### ✅ Case 3: Parent reference, Child object (****Polymorphism****)



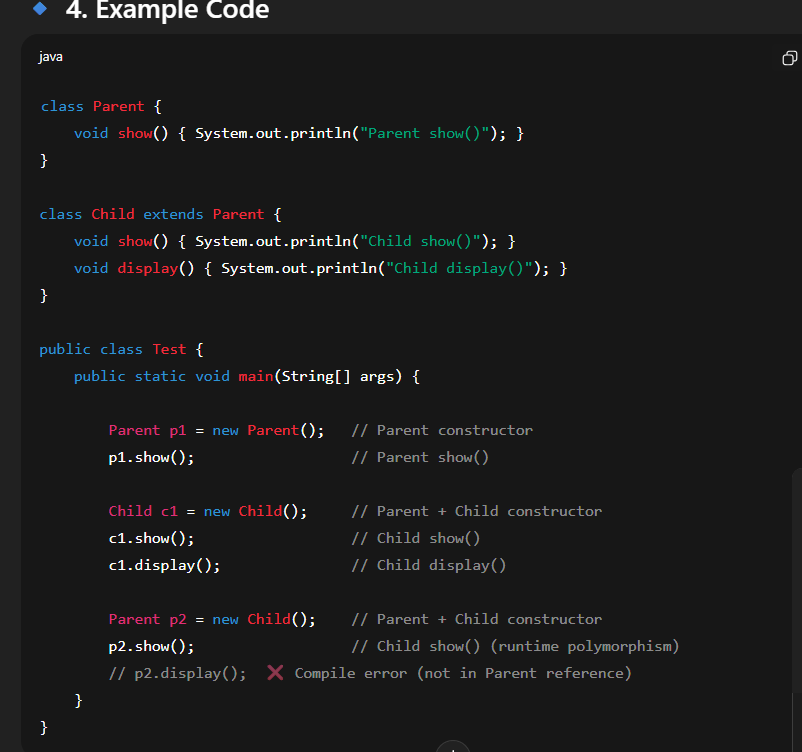
* Calls Child() constructor (and first Parent() constructor).
* Reference type: Parent → only **Parent’s members** are visible at compile-time.
* But overridden methods will execute from Child (runtime polymorphism).
* This is the **most common pattern** in real projects (used in interfaces, abstract classes, dependency injection, Spring framework, etc.).

### ❌ Case 4: Child reference, Parent object



· Not allowed.

· Because **Parent object does not have Child properties**, so unsafe.



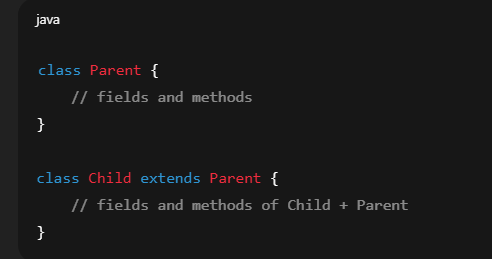
**Day\_4(Inheritance,Association,Abstraction(Abstract Class and interfaces),Polymorphism,Encapsulation**

# ****Inheritance in Java****

👉 **Definition:**  
Inheritance in Java is a mechanism in which **one class (child/subclass)** acquires the **properties and behaviors (fields and methods)** of another class (**parent/superclass**).  
It is one of the **main pillars of Object-Oriented Programming (OOPs)**.

## Why use Inheritance?

* To achieve **code reusability** (common logic written once in parent can be reused by child).
* To achieve **method overriding** (runtime polymorphism).
* To make the program **structured and easy to maintain**.
* 🔹 **Syntax**

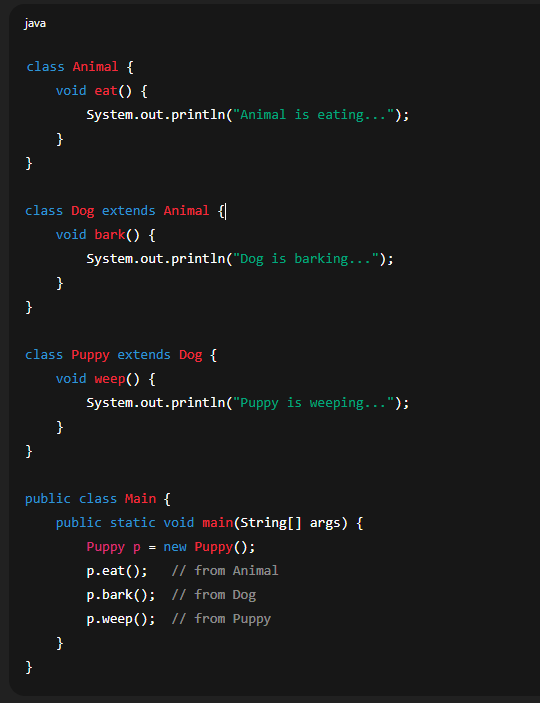


## Types of Inheritance in Java

* Java **does not support multiple inheritance with classes** (to avoid ambiguity), but supports it through **interfaces**.

1. **Single Inheritance** → One class inherits another.
2. **Multilevel Inheritance** → A class inherits another, and then another inherits it.
3. **Hierarchical Inheritance** → Multiple classes inherit the same parent.
4. **Multiple Inheritance (via interfaces only)**.

* **Example**



* **Output**



✅ **Key Point:**

extends keyword is used for class inheritance.

implements keyword is used when a class inherits from an interface.

❖ **Composition :-**

 Without existing container object if there is no chance of existing contained object

then container and contained object are strongly associated and this strongly

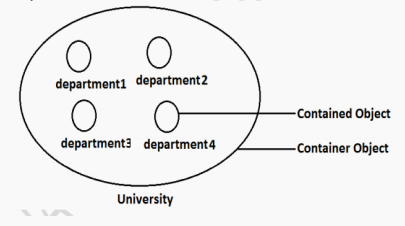
association is called composition.

 **For Example :-**

o University consist of several department without existing university there is

no chance of existing department hence **university on the department**

**are strongly associated** on this association is nothing but composition.



❖ **Aggregation:-**

 Without existing container object if there is a chance of existing contained object

then contained object then container and contained object are **weakly**

**associated** and this weak association is nothing but aggregation.

 **For Example :-**

o Department consist of several professor without existing department there

may be a chance of existing professor object. Hence department and

professor object are weakly associated and this weak association is

nothing but aggregation.

# ****Polymorphism in Java****

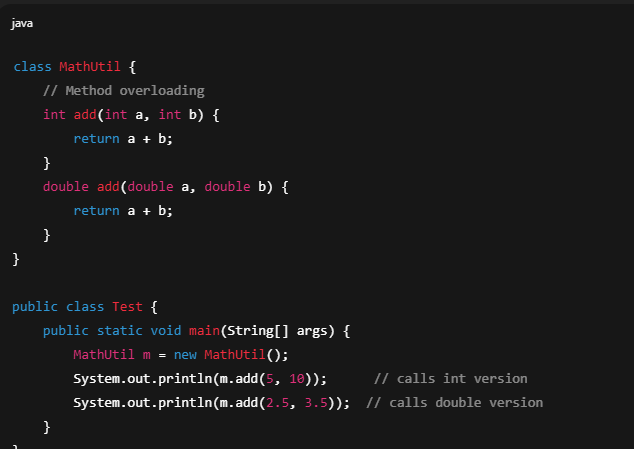
### ✅ Definition

* Polymorphism means **"many forms"**.  
  In Java, it allows one task to be performed in **different ways**.  
  It provides flexibility and reusability.

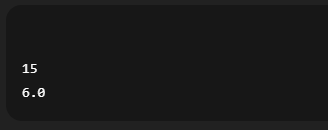
## Types of Polymorphism

### 1️⃣ ****Compile-Time Polymorphism (Method Overloading)****

* Achieved by **overloading methods** (same method name, different parameter list).
* Resolved at **compile-time**.
* **Example:**



* **Output:**

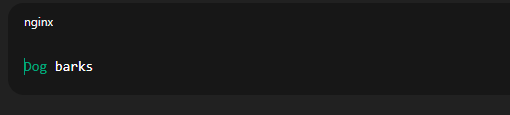


### 2. ****Run-Time Polymorphism (Method Overriding)****

* Achieved by **overriding methods** in the subclass.
* Resolved at **runtime** (dynamic method dispatch).
* Works using **inheritance**.
* **Example:**



* **Output**



## Why Use Polymorphism?

* Code **reusability**.
* Allows **same interface but different implementations**.Makes systems more **flexible and scalable**.

### ****What is Abstraction?****

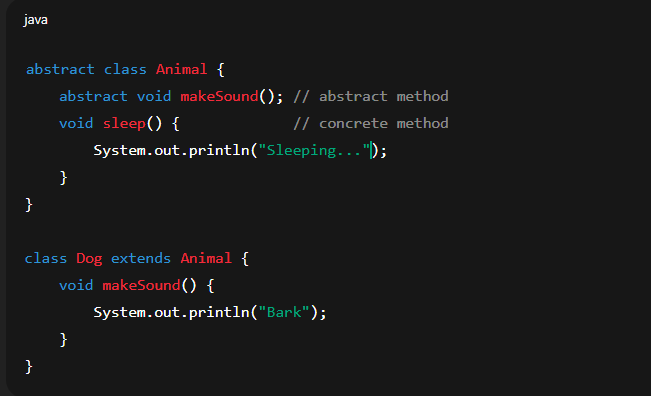
* **Abstraction** is the process of hiding the implementation details and showing only the essential features of an object.
* In Java, we achieve abstraction using:
* **Abstract classes** (abstract class)
* **Interfaces**

### ****Where to Use Abstraction in Java****

Here are the common scenarios:

1. **When defining a common template for a group of classes**

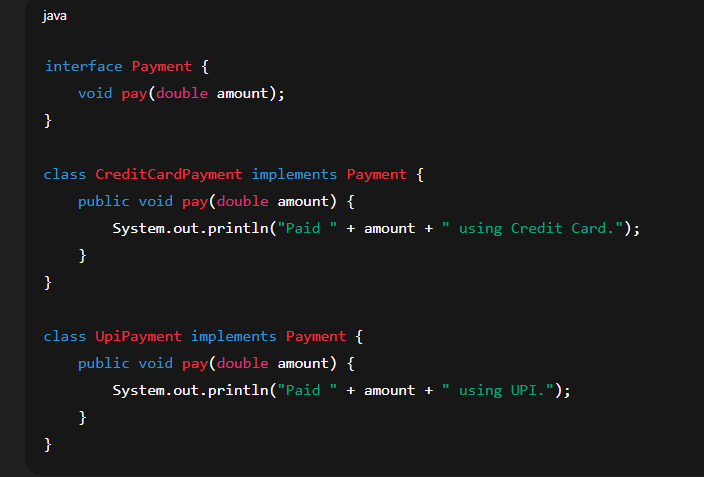
* Example: Animal as an abstract class.



✅ Use abstraction when you want **base behavior + enforce certain methods in subclasses**.

**2.When different implementations share the same contract**

* Use **interfaces** when multiple classes must implement the same behavior but in their own way.
* Example: Payment system.



✅ Use abstraction when you want **polymorphism** (different implementations but common reference type).

**3.When you want to reduce code complexity**

* You don’t need to know how a method works internally, just how to use it.
* Example:  
  In JDBC, you use Connection, Statement, ResultSet interfaces without worrying about their actual implementation (MySQL, Oracle, PostgreSQL, etc.).

**4.When business rules should not depend on implementation details**

* Example: A **notification system** may send SMS, Email, or Push Notification.  
  The higher-level module (like NotificationService) should work only with the abstract type, not the actual implementation.

# ****Interfaces in Java****

### An ****interface**** in Java is a ****blueprint of a class****.

* It contains **abstract methods (by default)** and **static/final variables**.
* Interfaces are used to achieve **abstraction** and **multiple inheritance** in Java.

### ****Key Points****

1. Declared using the interface keyword.
2. Cannot have **constructors** (since they can’t be instantiated).
3. By default:

* All variables in an interface are public static final.
* All methods are public abstract (till Java 7).

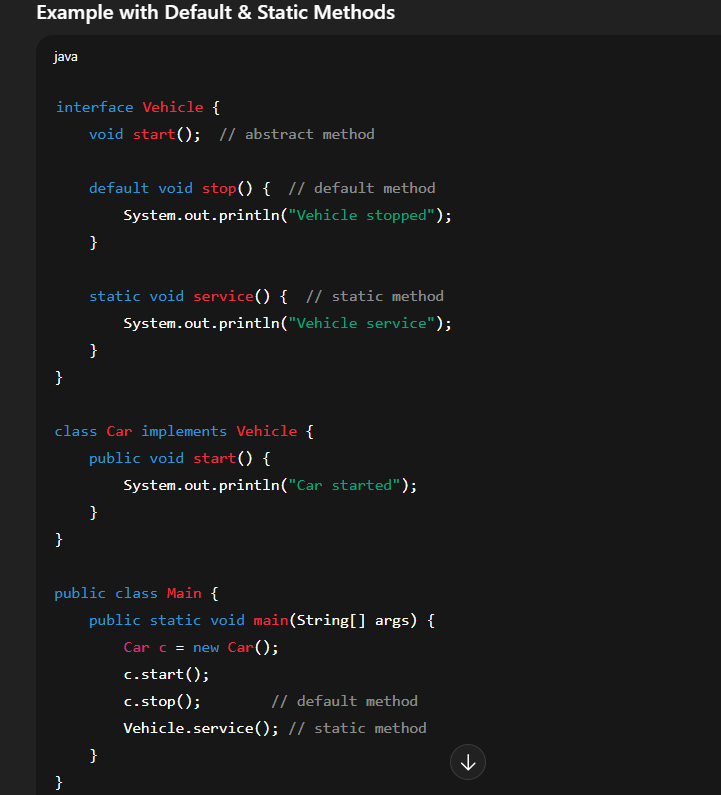
1. From **Java 8 onwards**:

* Interfaces can have **default methods** (with body, using default keyword).
* Interfaces can have **static methods**.

1. From **Java 9 onwards**:

* Interfaces can have **private methods** (to reuse code inside the interface).

1. A class uses implements keyword to implement an interface.
2. A class can implement **multiple interfaces** (supports multiple inheritance).

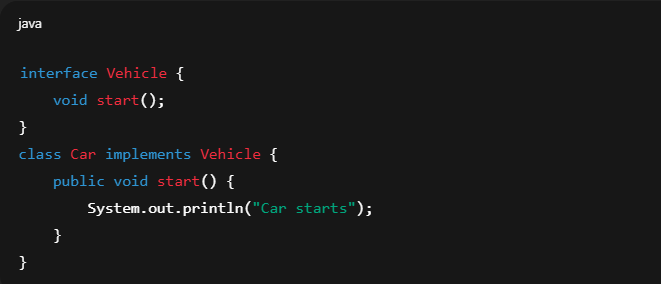


### ****When to Use Interfaces?****

* To achieve **abstraction** (only method declaration, no implementation).
* To achieve **multiple inheritance** (Java does not allow multiple class inheritance).
* To define a **contract** for classes (e.g., Runnable, Comparable).
* **Why do we need** default**,** static**, and** private **methods in Interfaces?**

## ****1. Problem with Old Interfaces (Before Java 8)****

* In **Java 7 and earlier**, interfaces could only have **abstract methods**.
* If you added a new method to an existing interface, **all implementing classes would break** because they must implement the new method.
* **Example** (Java 7 style):



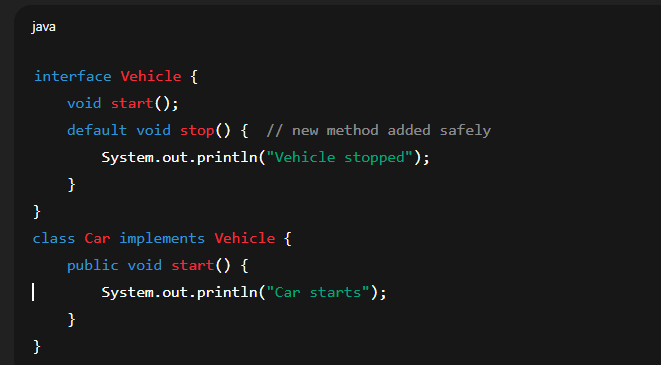
* Now, if later we add:



* Every class (Car, Bike, Bus) that implements Vehicle **must implement** stop().  
  This breaks backward compatibility ❌.

## ****2. Solution → Default Methods (Java 8)****

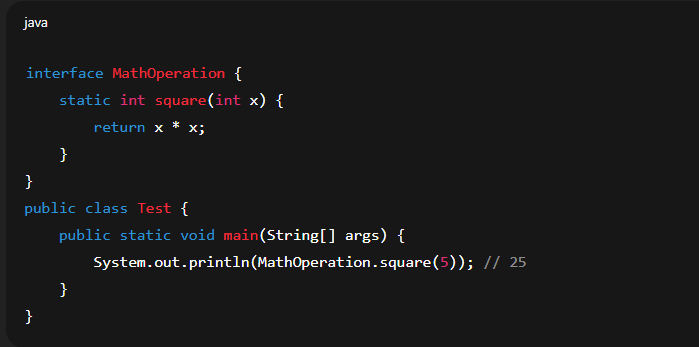
* **Default methods** allow us to **add new functionality** to interfaces without breaking existing classes.
* Old classes don’t need to implement them (they get a default behavior).
* Example:



* Even if we add stop(), old Car works fine without changes.  
  **Reason**: Backward compatibility + code reusability.

## ****3. Static Methods (Java 8)****

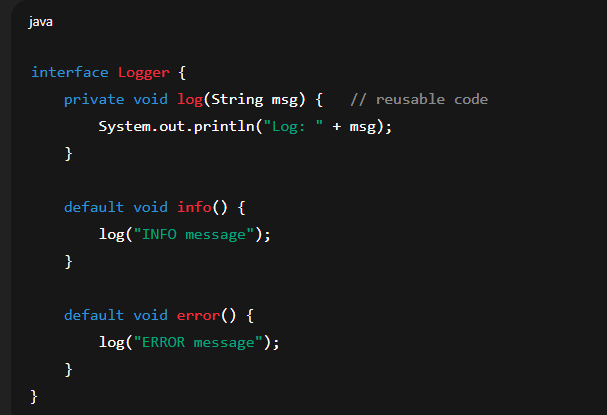
* Sometimes, we want **utility/helper methods** inside an interface.
* Example: validation, calculation, or conversion.
* Instead of writing these in a separate utility class, we can keep them in the interface itself.
* Example:



* **Reason**: Keep utility methods close to the interface → better organization & modularity.

## ****4. Private Methods (Java 9)****

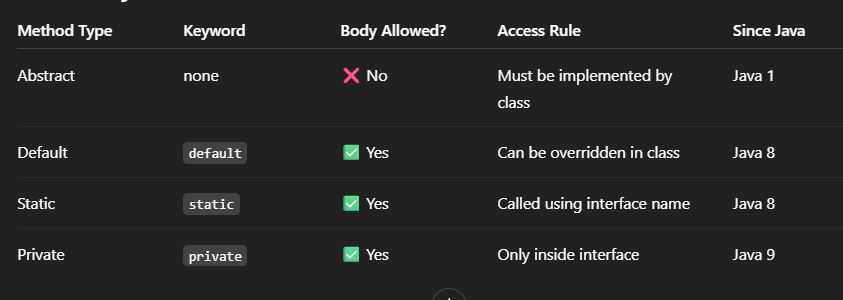
* What if multiple default/static methods in an interface need to share **common logic**?
* Without private methods, we would **duplicate code**.
* With private methods, we can **reuse code inside the interface** safely.
* Example:



* **Reason**: Code reusability + cleaner design inside interfaces.

# ****Final Summary****

* **Default methods** → To add new methods to an interface without breaking old code (backward compatibility).
* **Static methods** → To define utility/helper methods related to the interface.
* **Private methods** → To avoid code duplication inside default/static methods in an interface.



# ****Encapsulation in Java****

* Encapsulation is the process of **wrapping variables (data) and methods (code)** into a single unit (a class) and controlling access to them using **access modifiers**.

👉 In simple terms:

* **Hide the data** (make fields private).
* **Provide access** via **public methods (getters and setters)**.
* **Key Points**
* Achieved using **classes**.
* **Data hiding**: internal details are hidden from the outside world.
* **Controlled access**: data can only be modified through methods.
* Improves **security, maintainability, and flexibility**.

## ****Example: Without Encapsulation:****



* **Output**

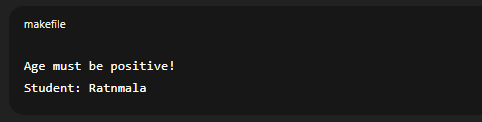


* Without encapsulation, -10 is stored directly.
* Anyone can directly modify fields → no control or validation.

## ****Example: With Encapsulation:****



* **Output**



·With **encapsulation**, invalid age -10 is rejected.

· name and age are **hidden** (private).

· Access given via **getter/setter** methods.

· Validation ensures correctness.

## ****Benefits of Encapsulation****

1. **Data Hiding** → Sensitive info is hidden from direct access.
2. **Validation** → Setters can control what values are allowed.
3. **Flexibility** → You can change implementation without affecting outside code.
4. **Reusability** → Cleaner, modular code.
5. **Security** → Prevents misuse of data.