# 🔐 What are Access Modifiers?

Access Modifiers in Java **control the visibility (accessibility)** of classes, variables, methods, and constructors.

There are **4 main types**:

| **Modifier** | **Accessible Within** | **Class** | **Package** | **Subclass (other pkg)** | **World** |
| --- | --- | --- | --- | --- | --- |
| private | ✅ | ❌ | ❌ | ❌ | ❌ |
| *(default)* | ✅ | ✅ | ❌ | ❌ | ❌ |
| protected | ✅ | ✅ | ✅ | ❌ | ❌ |
| public | ✅ | ✅ | ✅ | ✅ | ✅ |

**🧱 1. private**

**📌 Most restrictive** – accessible only within the same class.

✅ Real-World Example:  
class BankAccount {

private double balance;

public void deposit(double amount) {

balance += amount; // balance is private

}

public double getBalance() {

return balance;

}

}  
**💡 Where to Use:**

* To **hide internal data** (like passwords, salary, balance)
* Used for **encapsulation**

🧩 **2. Default (no modifier)**

Accessible only within the same package.  
class Employee { // default class

int id; // default variable

void display() { // default method

System.out.println("Employee ID: " + id);

}

}  
**💡 Where to Use:**

* When classes are **grouped by package**
* For **package-level helpers/utilities**

**🧬 3. protected**

Accessible within:

* Same class
* Same package
* Subclasses (even in different packages via inheritance)

class Animal {

protected String type = "Mammal";

}

class Dog extends Animal {

void printType() {

System.out.println(type); // accessible via inheritance

}

}  
**💡 Where to Use:**

* When you want to **allow subclass access** but still hide from outside world.
* In **frameworks** or **base classes**.

**🌍 4. public**

Accessible **from anywhere**.

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}  
**💡 Where to Use:**

* For **library APIs**, utility classes, main() method
* When method/class needs to be accessed from **anywhere**

**📚 Quick Summary Table**

| **Modifier** | **Use For** |
| --- | --- |
| private | Data hiding, internal logic (e.g., passwords) |
| default | Grouped utility classes in same package |
| protected | Inheritance within or outside package |
| public | API, utility methods, main() class |

**🧠 Student Quiz: Access Modifiers**

**✏️ Q1: What is the most restrictive access modifier in Java?**

* A) public
* B) protected
* C) private
* D) default

✅ **Answer:** C) private

**✏️ Q2: Which modifier allows access from any other class in any package?**

* A) public
* B) protected
* C) default
* D) private

✅ **Answer:** A) public

**✏️ Q3: If no modifier is specified, what's the access level?**

* A) public
* B) protected
* C) private
* D) package-private (default)

✅ **Answer:** D

**✏️ Q4: Which modifier should you use if you want subclass access but not world access?**

* A) public
* B) protected
* C) private
* D) default

✅ **Answer:** B) protected

**✏️ Q5: In which of these can private members be accessed?**

* A) Another class in the same package
* B) Only within the same class
* C) Any subclass
* D) Main class

✅ **Answer:** B

**🔑 What Are Instance Variables?**

**Instance variables** are **non-static variables** declared inside a class but **outside any method or constructor**. Each object gets its own copy of these variables.

class Student {

String name; // instance variable

int age; // instance variable

}  
  
**🎯 Real-World Analogy**

Imagine a **"Student" class**. Every student (object) has a unique:

* Name
* Roll Number
* Age

These are instance variables — unique to **each object**.

Student s1 = new Student();

Student s2 = new Student();  
s1 and s2 have their **own** names and ages.  
  
**🧰 this Keyword**

**👉 Use this to refer to current object’s instance variable (especially when parameter names are same as instance variables).**

**🧠 Real-World Scenario:**

Let’s say you’re assigning data from a form to a student object:

class Student {

String name;

int age;

// Constructor with same parameter names

Student(String name, int age) {

this.name = name; // this.name → instance variable

this.age = age;

}

void show() {

System.out.println("Name: " + this.name);

System.out.println("Age: " + this.age);

}

}  
Here, this.name refers to the class variable, and name refers to the constructor parameter.  
  
**👨‍👧 super Keyword**

**👉 Used in subclasses to refer to superclass members:**

* super.variable
* super.method()
* super() constructor call

🧠 Real-World Example:  
class Person {

String name = "John";

void display() {

System.out.println("Name from Person: " + name);

}

}

class Employee extends Person {

String name = "Alice";

void show() {

System.out.println("Name from Employee: " + name);

System.out.println("Accessing parent name: " + super.name);

super.display(); // calling superclass method

}

}  
  
🧾 Output:  
Name from Employee: Alice

Accessing parent name: John

Name from Person: John  
  
🛠 Mini Example: this vs super  
class Animal {

String sound = "Generic Sound";

}

class Dog extends Animal {

String sound = "Bark";

void printSounds() {

System.out.println(this.sound); // Dog's sound

System.out.println(super.sound); // Animal's sound

}

}  
  
**🧠 Student Quiz: this and super**

**✏️ Q1: What does this refer to in Java?**

* A) Parent class object
* B) Class name
* C) Current object
* D) None

✅ **Answer:** C) Current object

**✏️ Q2: What is the purpose of super?**

* A) To call static methods
* B) To access parent class members
* C) To call private methods
* D) To create objects

✅ **Answer:** B) To access parent class members

**✏️ Q3: What will be printed?**

class A {

String name = "Class A";

}

class B extends A {

String name = "Class B";

void print() {

System.out.println(super.name);

}

}

* A) Class A
* B) Class B
* C) Error
* D) null

✅ **Answer:** A) Class A

**✏️ Q4: What happens if you don’t use this in a constructor where parameter name is same as instance variable?**

* A) It gives error
* B) Nothing is assigned
* C) Local value is assigned
* D) Instance variable is assigned

✅ **Answer:** C) Local value is assigned (shadowing happens)

## **🧬 What is Inheritance in Java?**

**Inheritance** is a mechanism where **one class acquires the properties and behaviors (methods)** of another class.

* The class that is inherited from → **superclass** (parent/base)
* The class that inherits → **subclass** (child/derived)

class Animal {

void eat() {

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

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog barks.");

}

}  
  
Here, Dog **inherits** eat() from Animal.  
  
🧠 Real World Examples of Inheritance:  
✅ Example 1: Vehicles

class Vehicle {

int speed;

void start() {

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

}

}

class Car extends Vehicle {

void drive() {

System.out.println("Driving a car...");

}

}

* Car is a **Vehicle** → It "is-a" relationship.
* Car inherits start() method from Vehicle

✅ Example 2: Employees in a Company

class Employee {

String name;

void work() {

System.out.println("Employee working...");

}

}

class Manager extends Employee {

void manageTeam() {

System.out.println("Manager is managing the team.");

}

}  
Manager **inherits** name and work() from Employee.  
  
📚 Types of Inheritance in Java

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Single** | One subclass inherits one superclass | Dog extends Animal |
| **Multilevel** | Class inherits from a derived class | Puppy → Dog → Animal |
| **Hierarchical** | Multiple subclasses inherit one superclass | Dog & Cat → Animal |
| **Hybrid** | Combination of above (via interfaces only in Java) | Student implements Sports, Academics |
| ❌ **Multiple** | Not supported with classes | class C extends A, B is not allowed |

**📌 Why is Multiple Inheritance Not Supported by Classes?**

Java does not support multiple inheritance with classes to avoid ambiguity and diamond problem.  
**⚠️ The Diamond Problem:**

Imagine this:

class A {

void show() {

System.out.println("From A");

}

}

class B extends A {

void show() {

System.out.println("From B");

}

}

class C extends A {

void show() {

System.out.println("From C");

}

}

// Now what if

// class D extends B, C { } // ❌ Not allowed in Java

* **Ambiguity**: Which show() should class D inherit? From B or C?
* **Java avoids this** by **not allowing** multiple inheritance through classes.

✅ But Java **supports multiple inheritance via interfaces**.

**✅ Real World Analogy: Why No Multiple Inheritance?**

Imagine a student who has **two mentors**, and both mentors give **opposite advice**.  
Whose advice should they follow? This is the **diamond problem** in multiple inheritance.

Java avoids this **confusion** by enforcing **single class inheritance**, and resolving ambiguity through **interfaces**.

**🔷 What is an Interface in Java?**

An **interface** is a **contract** or a **blueprint** for a class. It contains method **signatures** (without implementations), and any class that implements the interface must **define** those methods.

interface Printable {

void print(); // abstract method

}  
  
**📌 Key Points:**

* All methods in an interface are **implicitly public and abstract** (before Java 8).
* Interfaces **cannot have constructors**.
* Interfaces can have:
  + abstract methods (till Java 7)
  + default & static methods (Java 8+)
  + private methods (Java 9+)
* Interfaces are used for **abstraction** and **polymorphism**.

🧠 Real-World Analogy

Think of an **interface** as a **Job Role description** — it tells what actions must be performed (methods), but **not how**.  
For example, a RemoteControl interface can have:  
interface RemoteControl {

void turnOn();

void turnOff();

}  
Any device (like TV, Fan, AC) that wants to behave like a RemoteControl must define turnOn() and turnOff().  
  
✅ Implementing an Interface

interface Drawable {

void draw();

}

class Circle implements Drawable {

public void draw() {

System.out.println("Drawing Circle");

}

}  
  
**🔁 Achieving Multiple Inheritance Using Interfaces**

In Java, **a class can implement multiple interfaces**, which is how **multiple inheritance is achieved**.

interface Printable {

void print();

}

interface Showable {

void show();

}

class Document implements Printable, Showable {

public void print() {

System.out.println("Printing document");

}

public void show() {

System.out.println("Showing document");

}

}  
🔄 Output:  
Printing document

Showing document  
✅ **No ambiguity** because both methods are defined **once** in the implementing class.  
  
**💥 What if Two Interfaces Have Same Method?**

If two interfaces have methods with the **same signature**, the class just needs **one implementation**:

interface A {

void greet();

}

interface B {

void greet();

}

class Hello implements A, B {

public void greet() {

System.out.println("Hello!");

}

}  
  
✅ This still works fine — no diamond problem!

📘 Summary of Types of Inheritance

| **Type** | **Supported in Java?** | **Example** |
| --- | --- | --- |
| Single | ✅ Yes | Dog extends Animal |
| Multilevel | ✅ Yes | Child → Parent → Grandparent |
| Hierarchical | ✅ Yes | Multiple classes from same parent |
| Multiple | ❌ Not via classes | Ambiguity issues |
| Hybrid | ✅ Via interfaces | class A implements B, C |

**🧠 Student Quiz: Inheritance in Java**

**✏️ Q1: Which keyword is used for inheritance in Java?**

* A) inherit
* B) extends
* C) implements
* D) superclass

✅ **Answer:** B) extends

**✏️ Q2: Which of the following types of inheritance is not supported using classes?**

* A) Single
* B) Multiple
* C) Hierarchical
* D) Multilevel

✅ **Answer:** B) Multiple

**✏️ Q3: What is the main reason Java does not support multiple inheritance with classes?**

* A) Memory issues
* B) Compilation errors
* C) Ambiguity / Diamond problem
* D) Slower performance

✅ **Answer:** C) Ambiguity / Diamond problem

**✏️ Q4: Which inheritance type is shown below?**

class A { }

class B extends A { }

class C extends B { }

* A) Single
* B) Multilevel
* C) Hierarchical
* D) Multiple

✅ **Answer:** B) Multilevel  
  
🔹 **Exercise 1: Single Inheritance – Method Access**class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog barks");

}

}

public class Test {

public static void main(String[] args) {

Dog d = new Dog();

d.sound();

d.bark();

}

}  
  
🔹 **Exercise 2: Multilevel Inheritance – Constructor Chaining**class Grandparent {

Grandparent() {

System.out.println("Grandparent Constructor");

}

}

class Parent extends Grandparent {

Parent() {

System.out.println("Parent Constructor");

}

}

class Child extends Parent {

Child() {

System.out.println("Child Constructor");

}

}

public class Test {

public static void main(String[] args) {

Child c = new Child();

}

}  
  
**🔹 Exercise 3: Hierarchical Inheritance – Inherited Method Access**class Vehicle {

void run() {

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

}

}

class Car extends Vehicle {

void wheels() {

System.out.println("Car has 4 wheels");

}

}

class Bike extends Vehicle {

void wheels() {

System.out.println("Bike has 2 wheels");

}

}

public class Test {

public static void main(String[] args) {

Car c = new Car();

c.run();

c.wheels();

Bike b = new Bike();

b.run();

b.wheels();

}

}

# Sorting algorithms

**🔹 1. Selection Sort**

**Concept:**Repeatedly selects the smallest (or largest) item and places it in the correct position.

Real-world Example:  
Imagine organizing a shelf of books by title alphabetically. You start from the left:

* Look through all books to find the one with the earliest title.
* Place that at the start.
* Then, look through the remaining books to find the next in order, and so on.

Key Insight:  
You always "select" the next smallest/largest item from the unsorted part and place it into the sorted part.

**🔹 2. Bubble Sort**

**Concept:**Compare each pair of adjacent items and swap them if they’re in the wrong order. Repeat until the list is sorted.

Real-world Example:  
Think of bubbles rising in soda. The lightest bubbles (smallest numbers) gradually "bubble up" to the top.

Let’s say you’re arranging playing cards from smallest to largest:

* Compare the first two. Swap if out of order.
* Move to the next pair, and so on.
* Repeat the process multiple times. Eventually, the largest card will settle at the end, then second-largest, etc.

Key Insight:  
Multiple passes push larger elements to the end, slowly bubbling the smaller ones to the top.

**🔹 3. Quick Sort**

**Concept:**Pick a "pivot" value. Divide items into two groups: one smaller than the pivot and one larger. Repeat the process for both groups.

Real-world Example:  
Imagine organizing a stack of exam papers by marks:

* Choose one paper’s score as the pivot.
* Make two piles: scores below the pivot and scores above.
* Recursively sort each pile in the same way.

Key Insight:  
Divide the problem around a pivot and recursively conquer each side. Fast in practice, especially with good pivot choice.

**🔹 4. Merge Sort**

**Concept:**Split the list into halves, sort each half, and then merge them together in order.

Real-world Example:  
Suppose two clerks are independently sorting two halves of a customer list. Once done:

* One clerk has a sorted list from A to M.
* The other has a sorted list from N to Z.  
  You merge the two sorted halves into one final sorted list by comparing one item at a time from each.

Key Insight:  
Break the list into the smallest units, sort them, then merge them in sorted order. Stable and efficient.

**✅ Summary Comparison with Analogies:**

| **Algorithm** | **Analogy** | **Strength** |
| --- | --- | --- |
| **Selection Sort** | **Choosing the shortest student one by one** | **Simple logic, good for small data** |
| **Bubble Sort** | **Bubbles floating upward** | **Easy to implement, not very efficient** |
| **Quick Sort** | **Divide papers around a pivot score** | **Fastest in practice with large data sets** |
| **Merge Sort** | **Merging two sorted teams into one** | **Stable, consistent performance** |

## **🔹 1. "IS-A" Relationship (Inheritance)**

**➤ Meaning:**

This denotes inheritance — when one class extends another. It represents a generalization-specialization relationship.

➤ Syntax in Java:

class Dog extends Animal { }

➤ Real-world Example:

* Dog is an Animal
* Car is a Vehicle
* Manager is an Employee

➤ Use Case:

Use IS-A when the subclass truly is a type of the superclass and inherits behavior with possible specialization.

## **🔹 2. "HAS-A" Relationship (Composition/Aggregation)**

**➤ Meaning:**

**This denotes composition — when one class contains an instance of another class as a field.**

**➤ Syntax in Java:**

class Car {

Engine engine; // Car "has an" Engine

}

➤ Real-world Example:

* Car has an Engine
* Library has Books
* Person has an Address

➤ Use Case:

Use HAS-A when one object is made up of or uses another. It’s a part-of or usage relationship.

**🔸 Comparison Table:**

| Feature | IS-A (Inheritance) | HAS-A (Composition) |
| --- | --- | --- |
| Relationship type | Generalization | Association/part-whole |
| Syntax | extends keyword | Object inside another class |
| Coupling | Tight coupling | Loose coupling |
| Flexibility | Rigid (hard to change) | More flexible and reusable |
| Design principle | Inheritance hierarchy | Follows composition over inheritance |
| Real-world example | Dog is an Animal | Car has an Engine |
| Prefer when | Subclass naturally extends behavior | Objects are part of or used by other objects |

**✅ Which One to Prefer and Why?**

Prefer HAS-A (Composition) over IS-A (Inheritance) unless:

* There is a natural inheritance relationship, and
* You don't need to change behavior dynamically or frequently.

Why prefer HAS-A?

* More flexible: You can change components without rewriting entire class hierarchies.
* Better encapsulation: Internal changes don’t affect other classes.
* Avoids fragile base class problem: Changes in parent class don't break child classes unexpectedly.

**🔧 Final Real-world Analogy:**

Imagine you're designing a software system for a transport company.

* Truck is a Vehicle → IS-A → inheritance
* Truck has an Engine, Truck has a Driver → HAS-A → composition

Changing the engine type in a Truck (composition) is easy. Changing the behaviour of all Vehicles may not

Assignments for this week:

**🧩 Assignment 1: Library Management - Access Modifiers**

**🔹 Objective:**

Understand the usage of **public, private, protected, and default access modifiers** in real-world scenarios.

**📝 Task:**

Create a Book class with the following:

* title, author (private)
* id (protected)
* genre (default)
* getBookInfo() method (public) to display book details.

Create another class Librarian that **accesses** book information.

✅ Expected Output:  
Title: Clean Code

Author: Robert C. Martin

ID: 102

Genre: Programming  
**💡 Concepts:**

* Use **private** to encapsulate fields.
* Use **protected/default** to test access from another class in the same package.
* Create **getter/setter** to access private members.

🧩 Assignment 2: **Bank System – Instance Variables, this Keyword  
🔹 Objective:**

Understand how to use this keyword to differentiate between local and instance variables.

**📝 Task:**

Create a class BankAccount with:

* Instance variables: accountHolder, balance
* Constructor with same parameter names
* Use this to assign values
* A method showDetails() to display values

BankAccount(String accountHolder, double balance) {

this.accountHolder = accountHolder;

this.balance = balance;

}  
✅ Expected Output:

Account Holder: Alice

Balance: ₹50000.0  
  
**🧩 Assignment 3: Employee Hierarchy – Inheritance + super Keyword**

**🔹 Objective:**

Practice **single inheritance** and learn how to use the super keyword to invoke superclass methods/constructors.

**📝 Task:**

* Employee class with name and salary
* Constructor and method displayDetails()
* Manager class extends Employee and adds department info
* Use super() to call the parent constructor

✅ Expected Output:

Name: John

Salary: ₹75000.0

Department: Sales