Types of inheritance in Java

what are the types of inheritance that is supported by java ? different types of inheritance

give me the proper deep explaination and detailed explained so that i can understand it well

Sure Piyush! Let's understand **Inheritance** in Java **deeply** — with types, examples, diagrams, limitations, and why each is used.

🚀 What is Inheritance in Java?

Inheritance is an **Object-Oriented Programming (OOP)** concept where **one class acquires (inherits)** the properties and behaviors (fields + methods) of another class.

✔️ It helps in:

- Code reusability
- Reducing duplication
- Making code more maintainable
- Achieving polymorphism

📌 Class that inherits → **Child / Subclass / Derived class**
📌 Class being inherited → **Parent / Superclass / Base class**

📌 Types of Inheritance Supported in Java

Java supports **5 types** of inheritance:

Inheritance Type	Supported in Java?	Meaning
Single Inheritance	✅ Yes	One parent → One child
Multilevel Inheritance	✅ Yes	Chain of classes
Hierarchical Inheritance	✅ Yes	One parent → Multiple children
Hybrid Inheritance	⚠️ Yes (Using Interfaces Only)	Combination of two or more types
Multiple Inheritance	❌ Not directly (Classes) but ✔️ via Interfaces	A class inherits from multiple parents

Let's understand each one in **deep detail** 👇

1️⃣ Single Inheritance

➡️ One subclass inherits from one superclass.

📌 Structure:

```
css

A → B
```

📌 Example:

```
java

class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}

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.eat(); // Inherited from Animal
        d.bark();
    }
}
```

✔️ Very common
✔️ Simple parent-child relationship

2️⃣ Multilevel Inheritance

➡️ One class inherits another that also inherits another.
Like a family tree line.

📌 Structure:

```
css

A → B → C
```

📌 Example:

```
java

class Animal {
    void eat() { System.out.println("Eating"); }
}

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

class BabyDog extends Dog {
    void weep() { System.out.println("Weeping"); }
}
```

```
public class Test {
    public static void main(String[] args) {
        BabyDog bd = new BabyDog();
        bd.eat();
        bd.bark();
        bd.weep();
    }
}
```

✔️ Features + Behavior continue to pass through chain
⚠️ Long chain can become complex

3️⃣ Hierarchical Inheritance

➡️ Multiple subclasses inherit the same parent class.

📌 Structure:

```
markdown

          Animal
        /    |     \
     Dog    Cat    Tiger
```

📌 Example:

```
java

class Animal {
    void eat() { System.out.println("Animal eats"); }
}

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

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

✔️ Common behavior reusable for all children
⚠️ Methods unique to one child not available to others

4️⃣ Multiple Inheritance (NOT allowed in Java using classes)

➡️ A single class tries to inherit from multiple classes

📌 Structure:

```
css

   A      B
     \   /
       C   ← (Not allowed)
```

❌ Java does not support this with classes because of:

🔹 The Diamond Problem

If both parent classes have a method with the same signature —> ambiguity.

But...

✔️ Java supports Multiple inheritance using Interfaces

📌 Example:

```java
java

interface A {
    void show();
}

interface B {
    void show();
}

class C implements A, B {
    public void show() {
        System.out.println("Resolved in class C");
    }
}
```

✔️ Multiple inheritance through interfaces
✔️ No ambiguity because subclass must override methods

5️⃣ Hybrid Inheritance (Combination)

➡️ Combination of Hierarchical + Multiple + Multilevel
⚠️ Supported **only through Interfaces**

📌 Example Structure:

```css
css

     A (Interface)
   /       \
  B         C
   \       /
       D (Class)
```

✔️ Useful in large applications
✔️ Common in Spring Boot projects (Interface-based architecture)

🏁 Summary Table

Type	Example Structure	Supported?	How?
Single	A → B	✔️ Yes	Classes
Multilevel	A → B → C	✔️ Yes	Classes
Hierarchical	A → B, A → C	✔️ Yes	Classes

Type	Example Structure	Supported?	How?
Multiple	A → C ← B	❌ (Classes) but ✔️ Interfaces	Interfaces
Hybrid	Combination	✔️ Interfaces only	Interfaces

❓ Why Multiple Inheritance is not supported with Classes?

To avoid **ambiguity** and **complexity** during method calling.

Example Diamond Problem:

```csharp
class A { void show(){} }
class B { void show(){} }
class C extends A, B { }  // Who's show()?
```

Java → Smart decision → Interface-based solution 👍

⭐ Benefits of Inheritance in Java

- Code reuse → no need to rewrite common logic
- Enables polymorphism → dynamic behavior
- Improves structure and maintainability
- Avoids duplication → bug-free softwar

🎯 Real-Life Example of Inheritance

```nginx
Vehicle
↓
Car
↓
Sedan
```

Vehicle properties like wheelCount, speed → reused in all child classes

---

Would you like diagrams drawn for each type?
Or if you want, I can provide:
✔️ Interview questions & answers
✔️ Code practice problems with solutions
✔️ A mind map PDF for revision

Should I create a **visual diagram** version for quick memory? 😄

what is the difference between encapsulation and abstraction

Here you go Piyush — a **crystal clear, deep, and easy-to-understand** explanation of the difference between **Encapsulation** and **Abstraction** in Java 👇

🔹 Encapsulation vs Abstraction — Detailed Explanation

Feature	Encapsulation	Abstraction
Definition	Wrapping data (variables) and methods (behavior) into a single unit (class).	Hiding unnecessary details and showing only the essential functionality to the user.
Main Goal	Protect data from unauthorized access.	Reduce complexity and show only relevant features.
How it is achieved?	Using **classes**, **private data members**, **getters/setters**.	Using **abstract classes**, **interfaces**, and **abstract methods**.
Focus	Data Protection (Security).	Functionality Representation (Design).
Access Level	Controls access to internal data using accessibility modifiers.	Shows only what is needed and hides internal working.
Real-World Example	ATM machine stores your balance privately and allows access only through PIN → encapsulation of data.	ATM only shows simple UI → “Withdraw”, “Check Balance” but hides how it processes internally.
Relationship	**Encapsulation is implemented to achieve abstraction.**	Abstraction uses encapsulation to hide complexity internally.
Visibility of Implementation	Internal working **not directly visible** (private members).	Internal working **completely hidden** behind the interface.
Example in Code	Private variables + getters/setters hide data.	Abstract methods provide behavior without implementation.

✔️ Encapsulation Example

```java
class Account {
    private double balance; // private data -> hidden

    public void deposit(double amount) {
        balance += amount;
    }

    public double getBalance() {
        return balance; // controlled access
    }
}
```

➡️ Data is safe — outside world cannot access `balance` directly.

✔️ Abstraction Example

```
java

abstract class Vehicle {
    abstract void start(); // only functionality, no details
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with key/push button");
    }
}
```

➡️ User only knows it starts — not how engine, fuel, spark plug work internally.

🧠 Quick Memory Trick

Concept	What does it hide?
Encapsulation	Hides the **data**
Abstraction	Hides the **implementation details**

🎯 Summary in 2 Lines

> **Encapsulation** = Data hiding + getter/setter control
> **Abstraction** = Implementation hiding + show only useful features

---

If you want, I can also provide:
✔️ 4–6 exam-style definitions
✔️ Small diagrams for better memory
✔️ Java interview questions based on these concepts

Would you like a quick **real-life analogy chart** too?