Object Oriented Programming System

INDEX

⭐ **Important topics**

Introduction

What is OOPs

It's a programming paradigm based on the concept of **"objects,"** which contain both data (fields or attributes) and code (methods or procedures) to manipulate that data.

Or

“OOP is a programming paradigm that revolves around the concepts of objects and classes.

Or

OOP is about **designing software** using **real-world entities** like Car, Student, Account, etc., which have **properties** and **behaviors**.

Need of OOPs

Problem in Procedural Code	OOPs as a Solution
No clear modular structure	Classes group data & logic
Poor code reuse	Inheritance enables reuse
Data is not secure	Encapsulation hides sensitive info
Difficult to manage large code	OOPs offers abstraction & structure
Hard to model real-world entities	OOPs models real-world objects

***Procedural Programming:** Code is written as a sequence of instructions.

Advantages of OOPs

- **Modularity:** Code is divided into classes and objects, making it organized and manageable.
- **Reusability**: Inheritance allows us to reuse existing code, reducing redundancy.
- **Data Hiding (Security)**: Encapsulation hides internal object details, exposing only what's necessary.
- **Maintainability**: Code is easier to update or extend without affecting unrelated parts.
- **Real-world Modelling**: Mirrors real-world entities like Student, Car, etc.
- **Scalability**: Complex and large-scale applications can be designed and maintained efficiently.

Classes & Objects

Class

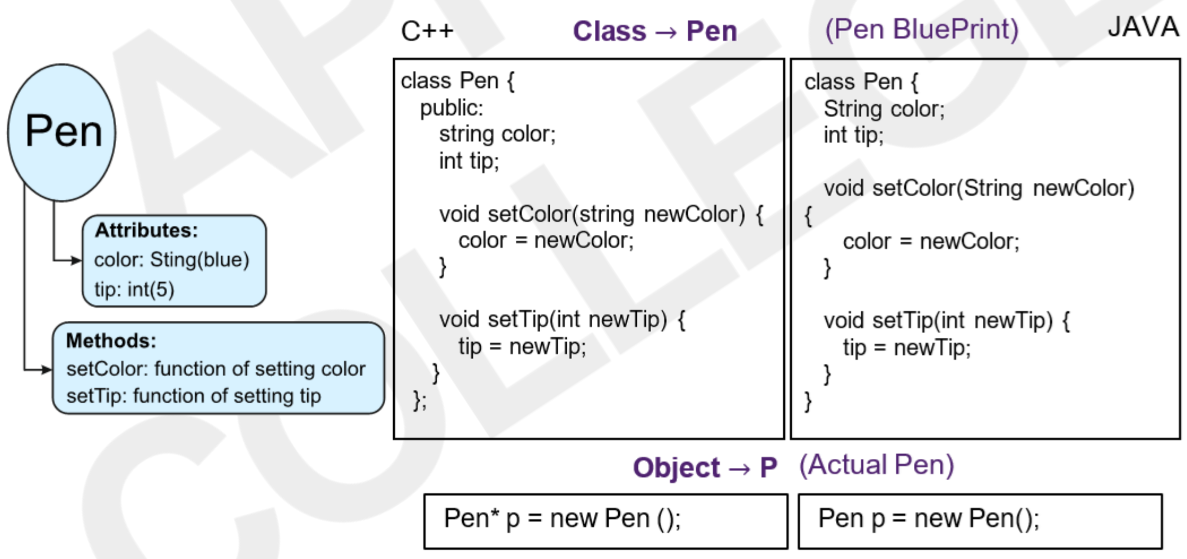
Class is the **blueprint / template** that defines the structure and behavior (data and functions) of objects.

Objects

- Objects are **real world entities** / **instances of class**.
- It contains some **characteristics** (attributes / properties / variables) & **behaviors** (methods) specified in the class template.

NOTE: Class is the blueprint that defines attributes and methods, while an object is a real instance of that blueprint, created in memory and used to perform operations.

Example 1: Class of Pen and Object P



```
class Pen {
  public:
    string color;
    int tip;

    void setColor(string newColor) {
      color = newColor;
    }

    void setTip(int newTip) {
      tip = newTip;
    }
};
```

```
class Pen {
  String color;
  int tip;

  void setColor(String newColor)
{
    color = newColor;
  }

  void setTip(int newTip) {
    tip = newTip;
  }
}
```

```
Pen* p = new Pen ();
```

```
Pen p = new Pen();
```

Example 2: Car

Real-world Concept	OOPs Equivalent
Car Blueprint	Class
Actual Car	Object
Color, Engine	Attributes (Data)
Drive, Brake	Behaviors (Methods)

Code Implementation for Classes and Objects

C++ Code

```
#include <iostream>
using namespace std;

class Car {
public:
    string color;
    int engineCC;

    void drive() {
        cout << "Driving a " << color << " car with " << engineCC << "cc engine." << endl;
    }
};

int main() {
    Car myCar;              // Object created statically
    myCar.color = "Red";
    myCar.engineCC = 2000;
    myCar.drive();

    Car* carPtr = new Car();    // Dynamic memory allocation
    carPtr->color = "Blue";
    carPtr->engineCC = 2200;
    carPtr->drive();
}
```

Java Code

```
class Car {
    private String color;
    private int engineCC;

    void drive() {
        System.out.println("Driving a " + color + " car with " + engineCC + "cc engine.");
    }

    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.color = "Red";
        myCar.engineCC = 2000;
        myCar.drive();
    }
}
```

Access Specifiers / Modifiers

Access specifiers / modifiers define the **visibility** or **access level** of class members (variables & methods).

Types of Access specifiers / modifiers

1. **Private:** Attributes and Methods accessible **inside a class**.
2. **Protected:** Attributes and Methods accessible **inside a class** & **by its derived class**.
3. **Public:** Attributes and Methods accessible **to everyone**.

Access Modifier	Class	Package	Sub Class	World
Private	✓	✗	✗	✗
Protected	✓	✓	✓	✗
Public	✓	✓	✓	✓

The above table applies to Java. In C++, only private, protected, and public exist.

Friend Class

(Only in C++)

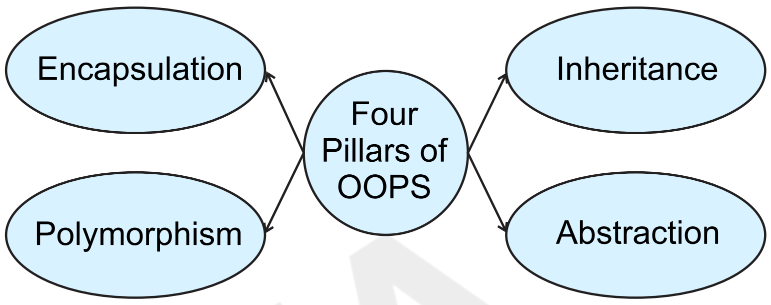
- A **friend class** in C++ is a class that is **granted special access** to the private and protected members of another class.
- Even though the data is private/protected, a friend class **can access it** directly.

Real Life Example: Imagine your **Car** class has a **private engine Number**. You **don't** want **anyone to access it**. **Except** your **Mechanic class**, which needs access for service. So, you make **Mechanic a friend class of Car.**

```
#include <iostream>
using namespace std;

class Car {
private:
    string engineNumber;

public:
    Car() {
        engineNumber = "XYZ1234";
    }

    // Declare Mechanic as a friend class
    friend class Mechanic;
};

class Mechanic {
public:
    void checkEngine(Car c) {
        cout << "Accessing engine number: " << c.engineNumber << endl;
    }
};

int main() {
    Car myCar;
    Mechanic m;
    m.checkEngine(myCar);  // Can access private member
    return 0;
}
```

Core Concepts of OOPs

The **four pillars** of OOP are:



Encapsulation

- **Wrapping** data (attributes / variables) and methods into a **single unit** called class.
- Ensures **data hiding** using private access modifiers.

Real life Example: A capsule hides the bitter medicine inside a protective shell, just like a class hides its data and exposes it only through methods.

Inheritance

- One class / Child class / Derived class **inherits** properties and behavior from another class / parent class / base class.
- Promotes **code reuse** and **hierarchical structure.**

Real life Example: A child inherits traits like eye color and height from their parents.

Polymorphism

- Ability of objects to take on **different forms** or **behave in different ways** depending on the **context** in which they are used.
- Types:
  a. **Compile-time / Static**: Method Overloading.
  b. **Run-time / Dynamic**: Method Overriding.

Real life Example: The same person acts as a student at college, a player on the field, and a child at home - different roles, same individual.

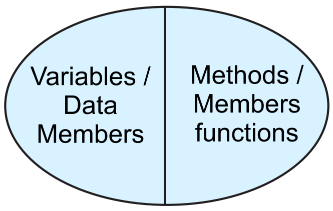
Abstraction

- **Hiding** all the **unnecessary details** and showing only the **relevant features**.
- Achieved via **pure virtual functions** (C++) or **abstract classes / interfaces** (Java).

Real life Example: You use a smartphone by tapping icons, without knowing the internal code behind apps.

Encapsulation

- **Wrapping** data (attributes / variables) and methods into a **single unit** called class and restricting direct access to the internal data.
- Ensures **data hiding** using private access modifiers.
- Think of it as putting everything related to an object inside a capsule (class).



Real life Example: A capsule hides the bitter medicine inside a protective shell, just like a class hides its data and exposes it only through methods.

C++ Code	Java Code

```
class Student {
private:
    int age;

public:
    void setAge(int a) {
        if (a > 0) age = a;
    }

    int getAge() {
        return age;
    }
};

int main() {
    Student s;
    s.setAge(20);          // Valid setter call
    cout << s.getAge();     // Output: 20
    // s.age = 25;  ❌ Not allowed (private)
}
```
	```
	class Student {
	    private int age;

	    public void setAge(int a) {
	        if (a > 0) age = a;
	    }

	    public int getAge() {
	        return age;
	    }

	    public static void main(String[] args) {
	        Student s = new Student();
	        //Encapsulated access
	        s.setAge(20);
	        // Output: 20
	        System.out.println(s.getAge());
	    }
	}
	```

Constructor

- **Special method** which is invoked **automatically** (only called once) at the time of **object creation**.
- Used for **initialisation** of an object.
- Has the same name as class & does not have a return type.
- Memory allocation of objects happens before the constructor call; the constructor is used for initialization.

Types of Constructor
1. **Non-Parameterised / Default**
2. **Parameterised**
3. **Copy Constructor**

Non-Parameterised	Parameterised	Copy Constructor
● Having no parameters. ● Used when we want to create objects without passing any values.	● Accepts parameters to initialize object values at creation time.	● Special constructor which creates a new object as a copy of an existing object.

class Student {
public:
    Student() {
        cout << "Non-Parameterised / Default Constructor called\n";
    }
};
	class Student {
	public:
	    string name;
	    Student(string n) {
	        name = n;
	    }
	};
		class Student {
		public:
		    string name;
		    Student(string n) {
		        name = n;
		    }
		
		    // Copy Constructor
		    Student(const Student &s) {
		        name = s.name;
		    }
		};

class Student1 {
    Student1() {
        System.out.println("Non-Parameterized Constructor called");
    }
}
	class Student2 {
	    String name;
	
	    Student2(String n) {
	        name = n;
	    }
	}
		class Student3 {
		    String name;
		
		    Student3(String n) {
		        name = n;
		    }
		
		    // Copy Constructor
		    Student3(Student3 s) {
		        this.name = s.name;
		    }
		}

⭐ Destructor

(Only in C++)

- A **destructor** is a special member function that is **automatically called** when an object **goes out of scope** or is **explicitly deleted**.
- Used to **free resources** like memory, file handles, etc.
- Same as class name and **prefixed with ~.**
- Has no return type and parameter.

Example:

C++ Code

```
#include <iostream>
using namespace std;

class Student {
public:
    Student() {
        cout << "Constructor called\n";
    }

    ~Student() {
        cout << "Destructor called\n";
    }
};

int main() {
    Student s1;  // Constructor called
}               // Destructor automatically called at end of scope
```

What About Java?

Java **doesn't have destructors**. Instead, it has a **garbage collector**.

Garbage Collector
- It is a **memory management feature.**
- **Automatically frees memory** occupied by objects **no longer in use**.
- Don't need to manually delete objects in Java - the Garbage Collector does it.

Garbage Collection
- Process of **automatically identifying and removing objects from memory** that are **no longer used** by a program.

Need of Garbage Collection
- Detects unreachable objects
- Reclaims their memory and Prevents memory leaks
- Keeps your app efficient and safe

NOTE: Destructor is used in C++ to clean up when the object is no longer needed. Java does not need destructors because it uses automatic garbage collection.

Scope Resolution Operator (::)

(Only in C++)

- Used to **access members** (variables or functions) that are **outside the current scope** or belong to a **specific class / namespace**.

Use Cases
Access Global Variable when Local Variable Shadows it
```
int x = 10;  // global

int main() {
    int x = 20;  // local
    cout << x << endl;      // prints 20
    cout << ::x << endl;   // prints global x = 10 using scope resolution
}
```
Define Class Functions Outside the Class
```
class Student {
public:
    void show();  // function declared
};

void Student::show() {  // defined using scope resolution
    cout << "Student function defined outside the class." << endl;
}
```
Access Namespaces
```
namespace A {
    int value = 5;
}

int main() {
    cout << A::value;  // Access value from namespace A
}
```

This Pointer

- this keyword is a **pointer (in C++) / reference (in Java)** that refers to the **current object -** the object whose method or constructor is being executed.
- Helps to **differentiate** between **class attributes and parameters**, especially when they have the **same name**.
- Available only **inside non-static** member functions.
- **Not available** in **static** methods (no object context).

C++	Java
● Uses -> to access members directly, or *this to dereference the pointer to the current object, allowing member access via the dot operator (.).	● Uses**. (dot)** to access members.
● Pointer (this ->)	● Reference (this.)

class Student {
public:
    string name;
 	
    Student(string name) {
        // referring to the current object's name
        this->name = name;
    }
 	
    void show() {
        cout << "Name: " << this -> name << endl;
    }
};
	class Student {
	    String name;
	
	    Student(String name) {
	        // referring to the current object's name
	        this.name = name;
	    }
	
	    void show() {
	        System.out.println("Name: " + this.name);
	    }
	}

Note: this is only strictly necessary when the parameter name shadows the instance variable name. If the parameter was String n, then name = n; would suffice.

Example

```
class Student {
   int id;
public:
   void setId(int id) {
      this->id = id; // disambiguates between member and parameter
   }
};
```

⭐ Shallow and Deep Copy

1. **Shallow Copy**
   - Copies **reference/address** of dynamic members.
   - Both objects share the **same memory**.
   - Changing data in one object affects the other.
   - Example: Photocopy of a key (looks same, but linked).
2. **Deep Copy**
   - Creates **new memory** and copies the **actual value**.
   - Both objects are **independent**.
   - Modifying one object does not affect the other.
   - Example: Making a new key from scratch (fully separate).

Examples

Shallow Copy	**Deep Copy**
C++	**C++**

#include <iostream>
using namespace std;
 	
class Student {
public:
    int* marks;
 	
    Student(int m) {
        marks = new int(m);
    }
 	
    // Shallow Copy
    Student(const Student& s) {
        marks = s.marks; // copies address only
    }
 	
    void show() {
        cout << "Marks: " << *marks << endl;
    }
};
 	
int main() {
    Student s1(90);
    Student s2 = s1;
    *s2.marks = 50;
 	
    s1.show(); // Marks: 50 ❌
}
	#include <iostream>
	using namespace std;
	
	class Student {
	public:
	    int* marks;
	
	    Student(int m) {
	        marks = new int(m);
	    }
	
	    // Deep Copy
	    Student(const Student& s) {
	        marks = new int(*(s.marks)); // allocates and copies value
	    }
	
	    void show() {
	        cout << "Marks: " << *marks << endl;
	    }
	};
	
	int main() {
	    Student s1(90);
	    Student s2 = s1;
	    *s2.marks = 50;
	
	    s1.show(); // Marks: 90 ✔
	    s2.show(); // Marks: 50 ✔}

Java

```java
class Student {
    int[] marks;

    Student(int m) {
        marks = new int[1];
        marks[0] = m;
    }

    // Shallow Copy
    Student(Student s) {
        this.marks = s.marks;  // copy reference only
    }

    void show() {
        System.out.println("Marks: " + marks[0]);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(90);
        Student s2 = new Student(s1);

        s2.marks[0] = 50;

        s1.show();  // Marks: 50 ✗
        s2.show();  // Marks: 50
    }
}
```

Java

```java
class Student {
    int[] marks;

    Student(int m) {
        marks = new int[1];
        marks[0] = m;
    }

    // Deep Copy
    Student(Student s) {
        this.marks = new int[1];
        this.marks[0] = s.marks[0];  // copy value
    }

    void show() {
        System.out.println("Marks: " + marks[0]);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(90);
        Student s2 = new Student(s1);

        s2.marks[0] = 50;

        s1.show();  // Marks: 90 ✓
        s2.show();  // Marks: 50 ✓
    }
}
```

Inheritance

- One class (Child / Derived) **inherits** properties and behavior from another (parent / base).
- Promotes **code reuse** and **hierarchical structure**.

Real life Example: A child inherits traits like eye color and height from their parents.

C++	Java

```
**// Parent or Base Class**
class Vehicle {
public:
    void start() {
        cout << "Vehicle started" << endl;
    }
};
**// Child or Derived Class**
**// Car inherits from Vehicle**
class Car : public Vehicle {
public:
    void drive() {
        cout << "Car is driving" << endl;
    }
};
```
	```
	**// Parent or Base Class**
	class Vehicle {
	    void start() {
	        System.out.println("Vehicle started");
	    }
	}
	**// Child or Derived Class**
	**// Car inherits from Vehicle**
	class Car extends Vehicle {
	    void drive() {
	        System.out.println("Car is driving");
	    }
	}
	```

Modes of Inheritance

(Only in C++)

- Describes how inheritance affects **access control.**
- When a class inherits from another, it can do so using different **access modes**:

1.	class Derived : public Base // public inheritance
2.	class Derived : protected Base // protected inheritance
3.	class Derived : private Base // private inheritance

- Control how the **members of the base class** are treated in the derived class.

Access Modifier Behaviour

Base Member	Public Inheritance	Protected Inheritance	Private Inheritance
public	public	protected	private
protected	protected	protected	private
private	Not inherited	Not inherited	Not inherited

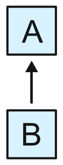
NOTE: Java **does not support inheritance modes like C++** while all inheritance is effectively **public** via extends.

NOTE: Private members are not directly accessible in the derived class, but they still exist within the base class portion of the derived object and can be manipulated via public/protected methods of the base class.

Types of Inheritance

1. Single Inheritance

- A class is allowed to inherit from only one class.
- A derived class inherits from one base class.
- **Parent → Child**



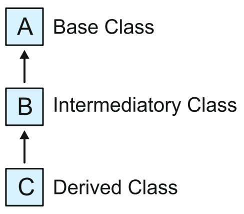
Single Inheritance

C++	Java

```
class A {
public:
    void showA() {
        cout << "Base Class A\n";
    }
};

class B : public A {
public:
    void showB() {
        cout << "Derived Class B\n";
    }
};
```
	```
	class A {
	    public void showA() {
	        System.out.println("Base Class A");
	    }
	}

	class B extends A {
	    public void showB() {
	        System.out.println("Derived Class B");
	    }
	}
	```

2. Multilevel Inheritance

- A class is derived from a class which is already derived from another class.
- **Parent → Child → Grandchild**



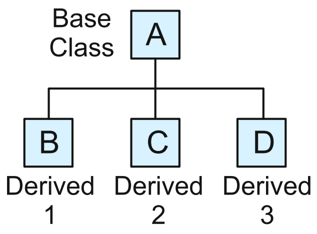
Multilevel Inheritance

C++	Java

```
class A {
public:
    void showA() {
        cout << "Class A (Base)\n";
    }
};

class B : public A {
public:
    void showB() {
        cout << "Class B (Derived from A)\n";
    }
};

class C : public B {
public:
    void showC() {
        cout << "Class C (Derived from B)\n";
    }
};
```
	```
	class A {
	    public void showA() {
	        System.out.println("Class A (Base)");
	    }
	}

	class B extends A {
	    public void showB() {
	        System.out.println("Class  B  (Derived  from
	A)");
	    }
	}

	class C extends B {
	    public void showC() {
	        System.out.println("Class  C  (Derived  from
	B)");
	    }
	}
	```

3. Hierarchical Inheritance

- Multiple classes inherit from a single base class.
- **One base → multiple derived classes**

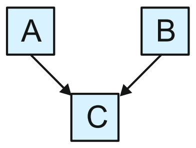


C++	Java

```
class A {
public:
    void showA() {
        cout << "Class A (Base)\n";
    }
};

class B : public A {
public:
    void showB() {
        cout << "Class B (Derived from A)\n";
    }
};

class C : public A {
public:
    void showC() {
        cout << "Class C (Also derived from A)\n";
    }
};
```
	```
	class A {
	    public void showA() {
	        System.out.println("Class A (Base)");
	    }
	}

	class B extends A {
	    public void showB() {
	        System.out.println("Class B (Derived from A)");
	    }
	}

	class C extends A {
	    public void showC() {
	        System.out.println("Class C (Also derived from A)");
	    }
	}
	```

4. Multiple Inheritance

- A single class inherits from two or more base classes.
- **One class inherits from multiple base classes.**



Multiple Inheritance

Does Java support Multiple inheritance?
- No - **Does not** support using **Classes**.
- Yes - **Supports** using **Interfaces**.

Why does Java not support Multiple inheritance using classes?
- It leads to **ambiguity**.
- **Ambiguity** in inheritance occurs when a class **inherits from multiple classes** that have **methods or members with the same name** - making it **unclear** which one should be called.

What is Interface?
- It is a **Contract** that defines a **set of methods (without implementations)** that a class must implement.

C++	Java
Supports multiple inheritance directly using **classes.**	**Supports multiple inheritance** directly using **interfaces.**

```cpp
class A {
public:
    void showA() {
        cout << "Class A\n";
    }
};
class B {
public:
    void showB() {
        cout << "Class B\n";
    }
};
class C : public A, public B {
public:
    void showC() {
        cout << "Class C (Derived from A and B)\n";
    }
};
```

```java
interface A {
    public void showA();
}
interface B {
    public void showB();
}
class C implements A, B {
    public void showA() {
        System.out.println("From Interface A");
    }
    public void showB() {
        System.out.println("From Interface B");
    }
    public void showC() {
        System.out.println("Class C implementing A and B");
    }
}
```

5. Hybrid Inheritance

- A combination of two or more types of inheritance (like multiple + multilevel).

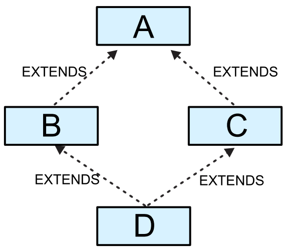
C++	Java

```cpp
class A {
public:
    void showA() { cout << "Class A\n"; }
};

class B : public A {
public:
    void showB() { cout << "Class B (Derived from A)\n"; }
};

class C : public A {
public:
    void showC() { cout << "Class C (Also derived from A)\n"; }
};

class D : public B, public C {
public:
    void showD() { cout << "Class D (Derived from B and C)\n"; }
};
```

```java
interface A {
    public void showA();
}
interface B extends A {
    public void showB();
}
interface C extends A {
    public void showC();
}
class D implements B, C {
    public void showA() {
        System.out.println("From Interface A");
    }
    public void showB() {
        System.out.println("From Interface B");
    }
    public void showC() {
        System.out.println("From Interface C");
    }
    public void showD() {
        System.out.println("Class D implementing B and C");
    }
}
```

⭐ The Diamond Problem

- **Multiple inheritance issue** where a class inherits from two classes that share a common base class, causing ambiguity in inheritance.
- Class B and Class C both inherit from A.
- Class D inherits from both B and C.
- Now, if D accesses something from A, the compiler doesn't know whether to use the copy inherited through B or through C — this is called the diamond problem and it occurs in multiple inheritance.



The Example in C++ (How Diamond Problem Arises)

```
#include <iostream>
using namespace std;

class A {
public:
    void show() {
        cout << "Class A" << endl;
    }
};

class B : public A { };
class C : public A { };

// Diamond Problem arises here
class D : public B, public C { };

int main() {
    D obj;
    // obj.show();      Ambiguity: from B or C?
    obj.B::show();  // Manually resolved
    obj.C::show();
    return 0;
}
```

Virtual inheritance:

- **Virtual inheritance** in C++ ensures that **only one copy** of a base class is inherited **when multiple derived classes share it**.
- It prevents the **diamond problem** by making sure that the **shared base class is not duplicated** in the inheritance chain.

Solution in C++: Virtual Inheritance

```
#include <iostream>
using namespace std;

class A {
public:
    void show() {
        cout << "Class A" << endl;
    }
};

//Virtual Inheritance
class B : virtual public A { };
class C : virtual public A { };
class D : public B, public C { };

int main() {
    D obj;
    obj.show();  // No ambiguity
}
```

What about Java?

- Java's Approach – No Diamond Problem.
- Java **does not allow multiple class inheritance**, so the **diamond problem does not** occur with classes.
- Java uses interfaces for safe multiple inheritance.

Using Interfaces – Java's Safe Multiple Inheritance

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

interface B extends A { }
interface C extends A { }

class D implements B, C {
    public void show() {
        System.out.println("Class D implementing show()");
    }
}
```

- No ambiguity in the above example because D is forced to define its own show() method.

Polymorphism

- Ability of objects to take on **different forms** or **behave in different ways** depending on the **context** in which they are used.

or

- **Polymorphism** means "**many forms":** the ability of a function, method, or object to behave **differently in different contexts**.

Real-life Example: The same person acts as a student at college, a player on the field, and a child at home - different roles, same individual.

Different Behaviors:

1. In Shopping Mall, Behave Like a Customer
2. In Class Room, Behave Like a Student
3. In Bus, Behave Like a Passenger
4. In Home, Behave Like a Son or Daughter

Types of Polymorphism

1. **Compile-time Polymorphism / Static Binding**
   Achieved by:
   - Function/Method Overloading (Java & C++)
   - Operator Overloading (C++ only)

2. **Runtime Polymorphism / Dynamic Binding**
   Achieved by:
   - Method Overriding (Java & C++)
   - Virtual Functions (C++)

Compile-time Polymorphism (Static Binding)

- Also known as **early binding**
- Compile-time polymorphism means that the **method to be executed is determined at compile time**, based on the method **signature** (name + parameters).
- Same method name, **different parameter lists** → the correct method is selected by the **compiler**.

Real life Example: You can call someone using their name, number, or nickname - all are valid ways to "call", depending on what info you have.

1. **Method Overloading:**
   Allows a class to have **more than one method with the same name** but **different parameters** (number, type, or order).

C++	Java

```
class Calculator {
public:
    int add (int a, int b) {
        return a + b;
    }

    double add (double a, double b) {
        return a + b;
    }

    int add (int a, int b, int c) {
        return a + b + c;
    }
};
```
	```
	class Calculator {
	    int add (int a, int b) {
	        return a + b;
	    }

	    double add (double a, double b) {
	        return a + b;
	    }

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

Rules for Method Overloading:

- Method name must be **same**
- Parameters must be **different** (type, number, or order)
- Return type alone doesn't matter

2. Operator Overloading:

- Allows to redefine the behavior of operators (+, -, , etc.) for **user-defined types** (classes/objects).

C++

```
class Complex {
public:
    int real, imag;

    Complex(int r, int i) {
        real = r;
        imag = i;
    }

    // Overload '+' operator
    Complex operator + (const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

    void show() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2, 3), c2(1, 4);
    Complex c3 = c1 + c2;
    c3.show();  // Output: 3 + 7i
}
```

- **Java does not support operator overloading (except for + used with Strings).**

Run-time Polymorphism (Dynamic Binding)

- Also called **dynamic method dispatch / late binding**
- In **runtime polymorphism**, the method that is executed is **determined at runtime**, not at compile-time.
- It allows us to call **the correct overridden method** based on the **actual object type**, even if we are using a **base class reference or pointer**.
- Same method name is called via base type but behaves based on the derived object.

Real life Example: You call **makeSound()** on an Animal, but whether it barks or meows depends on whether the object is a Dog or Cat — this decision is made at runtime

1. Method Overriding:

- **Method overriding** means redefining a **base class method** in a **derived class** using the **same method name, parameters, and return type**.
- It allows a **derived class to provide its own implementation** of a method defined in its base class.

C++	Java
class Animal {	class Animal {
public:	public void speak() {
virtual void speak() {	System.out.println("Animal speaks");
cout << "Animal speaks" << endl;	}
}	}
};	
	class Dog extends Animal {
class Dog : public Animal {	@Override
public:	public void speak() {
void speak() **override** {	System.out.println("Dog barks");
cout << "Dog barks" << endl;	}
}	}
};	
	public class Main {
int main() {	public static void main(String[] args) {
Animal* a = new Dog();	Animal a = new Dog();
a->speak(); **// Output: Dog barks**	a.speak(); **// Output: Dog barks**
delete a;	}
}	}

Rules for Method Overriding:

- Method name must be **same**
- Parameter list must match exactly
- Return type must be same

NOTE: Use of virtual keyword (C++): The **virtual keyword** enables dynamic binding, allowing the correct overridden method to be called at runtime when using base class pointers or references. Without it, the base class method is always called.

Feature	Overloading	Overriding
Definition	Same method name, **different parameters** in same class	Same method name and signature in **parent and child classes**
Type of Polymorphism	Compile-time (Static)	Runtime (Dynamic)
Inheritance Required?	Not required	Required (involves inheritance)
Class Involved	Happens **within the same class**	Happens **between base and derived class**
Parameters	Must be **different** in number / type / order	Must be **exactly same**
Return Type	Can differ (but return type alone can't overload)	Must be same or covariant (in Java)

Virtual Functions

(Only in C++)

- **Member function** in the **base class** that you expect to be **overridden** in **derived classes**.
- Declaring a function **virtual** tells the compiler to use **runtime (dynamic) binding** instead of **compile-time (static) binding**.

Why use virtual Functions?

- Without **virtual**, C++ binds functions at **compile-time** based on the **pointer/reference type**, not the actual object type.
- With **virtual**, C++ defers the decision until **runtime**, so the **correct overridden function** is called.

C++

```
class Animal {
public:
    // Virtual Function
    virtual void speak() {
        cout << "Animal speaks" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() override {
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* a = new Dog(); // base pointer → derived object
    a->speak();  // Output: Dog barks
    delete a;
}
```

- Without **virtual**, this would print **"Animal speaks"**.

Abstraction

- **Hiding** all the **unnecessary / internal implementation details** and showing only the **relevant / essential features**.
- Achieved via **abstract classes with pure virtual functions** (C++) or **abstract classes / interfaces** (Java).

Real life Example: You use a smartphone by tapping icons, without knowing the internal code behind apps.

NOTE: In C++, abstraction can be partially achieved using access specifiers, but full abstraction is implemented through abstract classes with pure virtual functions.

⭐ Abstract Classes and Pure Virtual Functions

(Only in C++)

Abstract Class:
- Class that **cannot be instantiated**.
- Contains **at least one pure virtual function**.
- Provides a **base interface** that must be implemented by derived classes.

Pure Virtual Function:
- Function declared in a base class that **has no definition** in that class and **must be overridden** in derived classes.
- The **= 0** makes a function, a **pure virtual function**.

Abstract Class with Pure Virtual Function

```
#include <iostream>
using namespace std;

// Abstract class
class Shape {
public:
    // Pure virtual function
    virtual void area() = 0;

    void display() {
        cout << "This is a shape.\n";
    }

    // Virtual destructor
    virtual ~Shape() {
        cout << "Shape destroyed\n";
    }
};

// Derived class 1
class Circle : public Shape {
public:
    void area() override {
        cout << "Area of Circle: π * r * r\n";
    }
};
```

```
// Derived class 2
class Rectangle : public Shape {
public:
    void area() override {
        cout << "Area of Rectangle: length * breadth\n";
    }
};

int main() {
    // Shape s; => Not allowed: cannot instantiate abstract class

    Shape* s1 = new Circle();
    s1->area();     // Output: Area of Circle
    s1->display();  // Output: This is a shape.

    Shape* s2 = new Rectangle();
    s2->area();     // Output: Area of Rectangle
    s2->display();  // Output: This is a shape.

    delete s1;
    delete s2;
}
```

- You have a base class Shape, but you don't want anyone to create a generic shape - only specific shapes like Circle or Rectangle.
- So, Shape becomes an **abstract class** with area as a **pure virtual function**, and you **force child classes** to implement the method area ().

⭐ Abstract Class in Java / Interface in Java

(Only in Java)

Abstract Class:

- Cannot be instantiated (i.e., you can't create its objects directly)
- Can contain **abstract methods** (without body)
- Can also contain **concrete methods** (with body)
- Can have **fields**, **constructors**, and **access modifiers**

Abstract Class

```
abstract class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    public abstract void makeSound();  // must be overridden

    public void sleep() {
        System.out.println(name + " is sleeping");
    }
}
```

Interfaces:

- Is a **contract** - it defines **what a class must do**, but not **how**
- Can have **abstract methods** (no body), **default methods** (with body), **static method**
- Cannot have **constructors or instance variables**

Interface
```
interface Flyable {
    void fly();  // abstract method

    default void land() {
        System.out.println("Landing...");
    }

    static void info() {
        System.out.println("Flyable interface");
    }
}
```

```
interface Flyable {
    void fly();  // abstract method

    default void land() {
        System.out.println("Landing...");
    }

    static void info() {
        System.out.println("Flyable interface");
    }
}
```

Feature	Abstract Class	Interface
Inheritance	extends (single class only)	implements (multiple interfaces allowed)
Constructors	Yes	No
Fields	Instance and static fields allowed	Only public static final (constants)
Abstract Methods	Yes	Yes
Concrete Methods	Yes	Yes (via default and static)
Access Modifiers	Any (private, protected, etc.)	Methods are public by default
Instantiation	Cannot be instantiated	Cannot be instantiated
Use Case	Shared base logic	Capability/behavior declaration
Supports Multiple Inheritance	No	Yes

Static Data Member and Function

Static Data Member	Static Member Function
A **static data member** is a variable that is **shared by all objects** of a class.	A **static member function** belongs to the **class** rather than any object.
It is **allocated only once** in memory, and any change made by one object is **reflected across all**.	It can be called **without creating an object**, and it can **only access static members**.

Example:
static int count;
	Example:
	static void showCount() {
	    cout << count;
	}

● Shared among all instances of the class ● Exists even before any object is created	● Called using ClassName::showCount() in C++ ● Called using ClassName.showCount() in Java ● Static member functions cannot access non-static members directly.

MCQs:

1. **Which of the following is not a feature of OOP?**
   A. Encapsulation
   B. Polymorphism
   C. Compilation
   D. Inheritance

2. **Which access modifier allows visibility within the same package only?**
   A. private
   B. protected
   C. public
   D. default

3. **Which access specifier makes class members accessible to derived classes but not outside the class?**
   A. private
   B. protected
   C. public
   D. friend

4. **Which of the following is true about 'protected' members?**
   A. Only accessible within the same class
   B. Accessible in same package and subclasses
   C. Accessible everywhere
   D. Accessible only in subclasses

5. **Which statement is true about constructors in C++?**
   A. Constructor returns a value
   B. Constructor can be virtual
   C. Constructors can be overloaded
   D. Constructors cannot take parameters

6. **Which of the following is a correct way to overload constructors in Java?**
   A. By changing access specifiers
   B. By changing method name
   C. By changing return type
   D. By changing number/types of parameters

7. **Which concept is violated if a constructor calls itself recursively without termination in C++/Java?**
   A. Encapsulation
   B. Abstraction
   C. Infinite recursion (stack overflow)
   D. Overloading

8. **Which inheritance is not supported in Java but is in C++?**
   A. Single
   B. Multilevel
   C. Multiple
   D. Hierarchical

9. **What keyword does Java use to resolve method conflicts in multiple inheritance via interfaces?**
   A. extends
   B. override
   C. implements
   D. super

10. **In C++, what is the default visibility mode for class inheritance?**
    A. public
    B. private
    C. protected
    D. depends on access specifier

11. **Which problem does virtual inheritance solve in C++?**
    A. Constructor overloading
    B. Diamond problem
    C. Ambiguous operator overloading
    D. Memory leak

12. **Function overloading is resolved during which time?**
    A. Runtime
    B. Compile-time
    C. Execution
    D. Interpretation

13. Method overriding requires which condition in Java?
A. Different method names
B. Same method signature in subclass
C. Same return type only
D. Final methods

14. Which method cannot be overridden in Java?
A. Static methods
B. Abstract methods
C. Public methods
D. None

15. In C++, what happens if a virtual function is not overridden in a derived class?
A. Compile-time error
B. Parent's version gets called
C. Runtime error
D. Object slicing occurs

16. Which of the following is true about abstract classes in Java?
A. Can be instantiated
B. Must contain abstract methods
C. Can contain constructors
D. Cannot have variables

17. Which statement is false about interfaces in Java?
A. All methods are public and abstract by default
B. Interfaces support multiple inheritance
C. Can contain constructors
D. Can contain static methods

18. In C++, what makes a class abstract?
A. A constructor
B. A destructor
C. At least one pure virtual function
D. No methods at all

19. Which of the following best represents encapsulation?
A. Hiding data behind methods
B. Using inheritance
C. Overriding methods
D. Global variables

20. Which of the following cannot be achieved using abstraction?
A. Hiding internal logic
B. Code modularity
C. Tight coupling
D. Implementation hiding

21. In Java, what happens if a subclass overrides a method and calls super.method() inside it?
A. Infinite recursion
B. Calls parent class's method
C. Compile-time error
D. Method gets hidden

22. In C++, what happens if you delete a derived class object using a base class pointer without a virtual destructor?
A. Only base destructor called
B. Both destructors called
C. Segmentation fault
D. Compile-time error

23. Which keyword is used in Java to prevent method overriding?
A. final
B. static
C. abstract
D. protected

24. How much memory does a class occupy?
A. Memory equal to all its members
B. Depends on number of member functions
C. Zero until object is created
D. Same as its parent class

25. Is it always necessary to create objects from a class?

A. Yes, for every class

B. No, static members can be accessed without objects

C. Only in C++, not in Java

D. Only for abstract classes

26. Which of the following is true about static methods in Java?

A. Can access instance variables

B. Belong to the object

C. Cannot be called from another static method

D. Can be called without creating object

27. In C++, static members are shared across:

A. Only base class

B. Only objects

C. All objects of the class

D. Inherited classes only

28. What is the size of an empty class in C++?

A. 0

B. 1

C. 2

D. Depends on compiler

29. Which of the following can be overloaded but not overridden?

A. static methods

B. virtual functions

C. constructors

D. destructors

30. In Java, memory for objects is allocated using:

A. malloc

B. new

C. calloc

D. constructor

31. Which C++ operator is used to deallocate memory?

A. free

B. malloc

C. delete

D. clear

32. Java manages memory automatically using:

A. Smart pointers

B. new/delete

C. Garbage Collection

D. Free ()

33. In Java, 'this' keyword refers to:

A. Parent object

B. Static reference

C. Current object

D. Superclass

34. Which of the following can't be virtual in C++?

A. Constructor

B. Member function

C. Destructor

D. Operator overload

35. Which of the following cannot be inherited in Java?

A. final class

B. abstract class

C. interface

D. protected class

36. What will the following Java code output?

```
class Test {
   public static void main(String[] args) {
      Test t1 = new Test();
      Test t2 = t1;
      System.out.println(t1 == t2);
   }
}
```

A. true

B. false

C. Compile error

D. Runtime error

37. In C++, which constructor is invoked if no arguments are passed?

A. Copy constructor
B. Parameterized constructor
C. Default constructor
D. None

38. In Java, what happens when we make an interface reference refer to a class object?

A. Only methods of interface are accessible
B. All methods are accessible
C. Throws error
D. Calls constructor of interface

39. Can we create an object of abstract class in C++ using pointer?

A. Yes, always
B. Yes, but only if not calling pure virtual methods
C. No
D. Yes, but can't instantiate it

40. Can an abstract class have a constructor in Java?

A. No
B. Yes
C. Only if all methods are abstract
D. Only if it has no instance variables

41. What is the output of the following Java code?

```
class Parent {
   void                    show()                {
      System.out.println("Parent"); }
}
class Child extends Parent {
   void show() { System.out.println("Child");
      }
}
class Test {
   public static void main(String[] args) {
      Parent p = new Child();
      p.show();
   }
}
```

A. Compile-time error
B. Child
C. Parent
D. Runtime exception

42. What will the C++ code below print?

```
class Base {
public:
   virtual void show() { cout << "Base\n"; }
};
class Derived : public Base {
public:
   void show() { cout << "Derived\n"; }
};
int main() {
   Base obj;
   Derived d;
   obj = d;
   obj.show();
}
```

A. Base
B. Derived
C. Compile error
D. Undefined

43. Why use abstract class over interface in Java?

A. You need multiple inheritance
B. You want to define method contracts only
C. You want to share common code
D. You want to use default methods

44. What is called automatically when an object goes out of scope in C++?

A. Destructor
B. Garbage collector
C. Free ()
D. Finalize

45. Which of the following is true about finalize() in Java?

A. It must be manually called
B. It is guaranteed to execute before object is garbage collected
C. It is deprecated in Java 9+
D. It is used in C++ for memory cleanup

ANSWER KEY									
1.	(C)	2.	(D)	3.	(B)	4.	(B)	5.	(C)
6.	(D)	7.	(C)	8.	(C)	9.	(D)	10.	(B)
11.	(B)	12.	(B)	13.	(B)	14.	(A)	15.	(B)
16.	(C)	17.	(C)	18.	(C)	19.	(A)	20.	(C)
21.	(B)	22.	(A)	23.	(A)	24.	(C)	25.	(B)
26.	(D)	27.	(C)	28.	(B)	29.	(C)	30.	(B)
31.	(C)	32.	(C)	33.	(C)	34.	(A)	35.	(A)
36.	(A)	37.	(C)	38.	(A)	39.	(C)	40.	(B)
41.	(B)	42.	(A)	43	(C)	44.	(A)	45.	(C)

EXPLANATION

1. Compilation is a programming concept, not an OOP principle. OOP is based on encapsulation, inheritance, and polymorphism.

2. Default (package-private) gives access only within the same package.

3. Protected members are accessible in derived classes but not outside of the class hierarchy.

4. Protected gives access within the same package and to subclasses in other packages.

5. C++ supports constructor overloading with different parameter types.

6. Constructor overloading in Java is done by varying parameter lists.

7. Recursion without a base case leads to stack overflow error.

8. Java does not support multiple inheritance using classes.

9. super is used to specify which interface’s method to call.

10. C++ classes default to private inheritance unless specified otherwise.

11. Virtual inheritance avoids multiple "Base" class copies in diamond structures.

12. Overloading is compile-time polymorphism.

13. Method overriding requires same name, signature, and return type.

14. Static methods are class-level and can’t be overridden.

15. Base class version is called if not overridden.

16. Abstract classes can have constructors and variables.

17. Interfaces cannot have constructors.

18. A class becomes abstract with at least one pure virtual function.

19. Encapsulation hides internal state using access modifiers.

20. Abstraction promotes loose coupling.

21. super.method() calls the overridden method in the parent class.

22. Not using a virtual destructor causes undefined behavior; only base destructor is invoked.

23. The final keyword prevents overriding.

24. In both Java and C++, a class does not occupy memory until an object is created. Memory is only allocated for instances, not for the class blueprint itself (except for static members, which are shared).

25. It is not always required to create objects. You can access static members and use a class as a utility (like Math class in Java) without creating an instance.

26. Static methods are class-level and can be called without an object.

27. Static members are common to all instances of a class.

28. C++ allocates 1 byte to ensure unique addresses for objects.

29. Constructors can't be inherited or overridden but can be overloaded.

30. Java uses the new keyword to allocate memory.

31. delete deallocates memory allocated using new.

32. Java's JVM includes a garbage collector.

33. This refers to the current object instance.

34. Constructors cannot be virtual in C++.

35. Final classes cannot be extended.

36. Both references point to the same object; hence true.

37. Default constructor is used when no arguments are passed.

38. Only methods declared in the interface are accessible.

39. Cannot instantiate abstract class, even via pointer.

40. Abstract classes can have constructors for initialization.

41. This is runtime polymorphism. Child's overridden method is called.

42. Object slicing occurs; Base version is called.

43. Abstract classes allow sharing implementation logic.

44. Destructor is invoked automatically on scope end.

45. finalize() is deprecated due to unpredictability and performance issues.