PART A

1. Inheritance Definition: In your own words, define inheritance. How does it differ from composition or aggregation in creating complex objects?

Inheritance is a way to create a new class by using an existing class. The new class (called the subclass) inherits the properties and behaviors (methods) of the existing class (called the superclass).

Inheritance shows that a Dog is an Animal. The Dog inherits the eat() method from Animal, but also has its method, bark().

Composition shows that a Car has an Engine. The Car class creates and manages its own Engine. If the Car object is destroyed, the Engine inside it is also destroyed.

Aggregation shows that a School has Students, but the students can exist independently of the School. The Student objects can exist outside of the School, and if the School object is destroyed, the Student objects still exist.

2. Types of Inheritance: List two forms of inheritance (e.g., single vs. multiple). Provide a brief example of when each type might be appropriate.

Single Inheritance: A class inherits features from one parent class. Example: A Car is a Vehicle, so it gets the features of a Vehicle.

Multiple Inheritance: A class inherits features from more than one parent class. Example: A HybridCar is both a Vehicle and an Electric, so it gets features from both.

3. Real-World Analogy:Describe a real-life scenario (outside of programming) where an item or concept “inherits” characteristics from another.Discuss how that real-life example aligns with the OOP concept of inheritance.

Vehicle and Car:

A car is a specific type of vehicle. All vehicles can move, and a car inherits that ability to move from the general vehicle category. However, a car also has its unique features, like honk and air conditioning.OOP Inheritance:

Vehicle = Parent class (general properties like make, year).

Car = Child class (inherits moving ability + adds specific features like honking).

So, a Car inherits the common properties of a Vehicle but also adds its unique features.

PART B

Option 1: Minimal Coding

Base Class

Create a simple base class called Vehicle with:A brand attribute (or manufacturer).A method drive() that prints something generic like "Vehicle is driving."

Derived Class

Define a derived class Car that inherits from Vehicle.Add a new attribute (e.g., doors) and override the drive() method to show it’s a car specifically driving.

Short Driver Code

Demonstrate creating a Vehicle object and a Car object.Call drive() on both to illustrate the difference.

```cpp
class Vehicle {
    // Attribute
    string brand;

public:
    // Constructor to initialize brand
    Vehicle(string b) : brand(b) {}

    // Method to print a generic message for driving
    void drive() {
        cout << "Vehicle is driving." << endl;
    }
};

class Car : public Vehicle {
    // New attribute specific to Car
    int doors;

public:
    // Constructor to initialize brand and doors
    Car(string b, int d) : Vehicle(b), doors(d) {}

    // Overriding the drive method for a specific Car message
    void drive() {
        cout << "The " << brand << " car is driving with " << doors << " doors." << endl;
    }
};

int main() {
    // Creating a Vehicle object
    Vehicle v("Vehicle");
    v.drive();  // Calls the Vehicle's drive method

    // Creating a Car object
    Car c("Toyota", 3);
    c.drive();  // Calls the overridden drive method in Car

    return 0;
}
```

PART C:

1. When to Use Inheritance Provide one scenario in which inheritance is clearly beneficial, and one scenario in which inheritance might be overkill or lead to a fragile design.

Beneficial: Modeling different types of vehicles like Car, Truck, and Motorcycle. All share common behaviors (like drive()), but each has unique features. Inheritance allows code reuse and cleaner design.

Overkill: Modeling employees where only a small difference exists between FullTimeEmployee and PartTimeEmployee. Using inheritance could complicate the design. A single Employee class with attributes (like workHours and employmentType) might be simpler and more flexible.

2. Method Overriding vs. Overloading:Differentiate method overriding (runtime polymorphism) and method overloading (compile-time).Why does inheritance rely heavily on overriding for real flexibility?

Method Overriding (Runtime Polymorphism):A subclass provides a specific implementation of a method already defined in the parent class. The method to be executed is determined at runtime based on the object type.Example: A Vehicle class has a drive() method, and a Car class overrides drive() to provide a specific implementation for cars. When the drive() method is called on a Car object, the Car version is executed, not the Vehicle version.

Method Overloading (Compile-time Polymorphism):Multiple methods with the same name but different parameters (type) in the same class. The method to be executed is determined at compile time based on the arguments passed.Example: A Printer class has print(int) and print(double) methods. The method called depends on whether an integer or a double is passed as an argument.

3.Inheritance vs. Interfaces/Abstract Classes:In some languages, we define abstract classes or interfaces.How does the concept of inheritance differ from implementing an interface (or aabstract base class)?

```
// Abstract class example
class Vehicle {
public:
    virtual void drive() = 0;  // pure virtual function
};

class Car : public Vehicle {
public:
    void drive() override { cout << "Car is driving" << endl; }
};

// Interface example (in languages like Java)
interface Drivable {
    void drive(); // No implementation here
}

class Bike implements Drivable {
    void drive() { cout << "Bike is driving" << endl; }
}
```

Inheritance is useful for sharing common functionality and extending behavior. And interfaces are used for defining common behavior that multiple classes can implement without enforcing

any specific implementation. They are particularly helpful in designing flexible, modular, and loosely coupled systems.

4. Pitfalls of Multiple Inheritance:Name one potential problem with multiple inheritance (e.g., diamond problem).Suggest a strategy or approach (like virtual inheritance in C++ or an interface-based design) to mitigate this issue?

The diamond problem occurs when a class inherits from two classes that both inherit from a common base class, leading to ambiguity about which version of the base class's methods to use.

Solution:Virtual Inheritance (C++): Ensures only one copy of the base class is inherited, resolving ambiguity.

Interface-Based Design: Avoids multiple inheritance by using interfaces, where classes implement common methods without inheriting from each other.

PART D

1. Inheritance in Different Languages:Compare how two languages (e.g., Java and C++) handle inheritance.Mention any restrictions (e.g., Java lacks multiple class inheritance, C++ supports it with complexities).

```
##Java
class Animal {
    void eat() { System.out.println("Eats food"); }
}

interface CanBark {
    void bark();
}

class Dog extends Animal implements CanBark {
    public void bark() { System.out.println("Barks"); }
}

##C++
class Animal {
public:
    void eat() { cout << "Eats food\n"; }
};

class Mammal {
public:
    void breathe() { cout << "Breathes air\n"; }
};

class Dog : public Animal, public Mammal {
public:
    void bark() { cout << "Barks\n"; }
};
```

For java

Single Inheritance: A class can inherit from only one class.

No Multiple Inheritance: Prevents complexity like the diamond problem.

Interfaces: A class can implement multiple interfaces for multiple behaviors.

For C++

Supports Multiple Inheritance: A class can inherit from multiple classes.

Virtual Inheritance: Resolves the diamond problem by ensuring only one copy of the base class is inherited.