1. Inheritance

1.1 What is Inheritance?

- Real-world objects are often related in parent-child relationships.
- Example: Object A (Parent) and Object B (Child) share properties.
- In programming, this relationship is mimicked using **Inheritance**.

1.2 Real-Life Example: Car Hierarchy

- **Parent Class:** Car (Generic)
    - Common attributes:
        - Brand
        - Model
        - IsEngineOn
        - CurrentSpeed
    - Common behaviors:
        - startEngine()
        - stopEngine()
        - accelerate()
        - brake()
- **Child Classes:**
    - **ManualCar** (inherits Car)
        - Specific attribute: CurrentGear

        - Specific behavior: shiftGear()
    - **ElectricCar** (inherits Car)
        - Specific attribute: BatteryPercentage
        - Specific behavior: chargeBattery()

1.3 C++ Syntax

```
class ManualCar : public Car { ... };
class ElectricCar : public Car { ... };
```

- `public` inheritance maintains access specifiers.
- `private` and `protected` alter accessibility.

1.4 Access Specifiers in Inheritance

- **public:**
    - Public members stay public.
    - Protected members stay protected.
- **protected:**
    - Public and protected members become protected.
- **private:**
    - All inherited members become private.
- **Private members** of parent class are **never inherited**.

// See code section for full code example.

2. Polymorphism

2.1 What is Polymorphism?

- Derived from: **"Poly" (many) + "Morph" (forms)** = many forms.
- **One stimulus → different responses** based on object/situation.

2.2 Two Real-Life Scenarios:

- **Scenario 1:**
    - Different animals (Duck, Human, Tiger) all have a `run()` behavior.
    - Each performs it differently.
- **Scenario 2:**
    - Same human `run()`s differently based on context (tired vs chased).

2.3 Types of Polymorphism in Programming:

- **Static Polymorphism** – Compile-time
    - Achieved via **Method Overloading**
- **Dynamic Polymorphism** – Runtime
    - Achieved via **Method Overriding**

3. Static Polymorphism (Method Overloading)

- Same method name, different parameter lists.
- Overloaded method is resolved at **compile time**.

Example:

```
class ManualCar {
  void accelerate();               // no parameter
  void accelerate(int speed);     // with parameter
};
```

- Allows the same behavior to adapt based on passed arguments.

Rules:

- Method name: Same
- Return type: Can be same or different (but not used for overloading)
- Parameters:
    - Vary in number **or** type

4. Dynamic Polymorphism (Method Overriding)

- Same method signature is redefined in child classes.
- Achieved using **virtual functions** in C++.
- Resolved at **runtime**.

Example:

```
class Car {
  virtual void accelerate() = 0; // Abstract
};

class ManualCar : public Car {
  void accelerate() override; // Manual-specific logic
};

class ElectricCar : public Car {
  void accelerate() override; // Electric-specific logic
};
```

5. Combined Use of OOP Pillars

- Final code demonstrates:

// See code section for full code example.

- **Abstraction** (Hiding implementation details)
- **Encapsulation** (Private/protected members)
- **Inheritance** (Manual/Electric inherit Car)
- **Polymorphism** (Method overriding & overloading)

Additional Concepts:

- **Protected:**
  - Inaccessible outside class, but accessible in child class.
- **Operator Overloading (Homework):**
  - Concept asked as homework: What is operator overloading?
  - Why is it available in C++ but not in Java/Python?

Conclusion & Practice

- Understanding OOPs is best done via **real-world relatable examples**.
- Practice suggestion: Modify/add features to existing car classes.
- **Homework:**
  1. Define **Operator Overloading**.
  2. Why is it not supported in Java/Python?