Samuel Edet Ukim

OOP HW3

Part A: Conceptual Questions

Inheritance Definition

- **Inheritance:** In my own words, inheritance is a mechanism in object-oriented programming where a new class (the derived or child class) can acquire the properties (data members) and behaviors (methods) of an existing class (the base or parent class). The derived class can then extend or modify these inherited features, allowing for code reuse and the creation of a hierarchy of related classes. It establishes an "is-a" relationship (e.g., a Car "is a" Vehicle).

- **Difference from composition or aggregation:**
  - **Composition:** Represents a "has-a" relationship where one object contains another object as a part of itself, and the contained object's lifecycle is often tied to the containing object (e.g., a `Car` *has a* `Engine`; if the `Car` is destroyed, the `Engine` is also likely unusable in that context).
  - **Aggregation:** Also represents a "has-a" relationship, but the related objects have independent lifecycles and can exist independently (e.g., a `Car` *has a* `Driver`; the driver can exist even if the car doesn't, and vice versa).
  - **Inheritance:** Focuses on an "is-a" relationship and the reuse of base class characteristics, allowing the derived class to be treated as a specialized type of the base class.

Types of Inheritance

- **Single Inheritance:** A derived class inherits from only one base class.
  - **Appropriate Example:** Creating a hierarchy of animals where `Dog` inherits from a single `Animal` base class, acquiring general animal characteristics and behaviors.
- **Multiple Inheritance:** A derived class inherits from two or more base classes.
  - **Appropriate Example:** Creating a `FlyingCar` class that inherits from both a `Car` class (providing driving capabilities) and an `Airplane` class (providing flying capabilities). This can be useful when an object naturally exhibits characteristics of multiple distinct types.

Overriding Methods

- **How method overriding helps:** Method overriding allows a derived class to provide its own specific implementation of a method that is already defined in its base class. This enables the derived class to tailor the inherited behavior to its specific needs or extend it with additional functionality while maintaining the same method signature (name, parameters, and return type).

- **Why override instead of adding a new method:** Overriding is used when the derived class needs to perform a similar action to the base class but in a different way or with additional steps. By keeping the same method name, it allows for polymorphism, where objects of different derived classes can be treated uniformly through a base class interface, and the correct specific implementation will be executed at runtime. Adding a new method would require the calling code to know the specific type of the object to invoke the specialized behavior.

Real-World Analogy

- **Real-Life Scenario:** A "Smartphone" inherits characteristics from a "Phone."
- **Alignment with OOP Inheritance:** Just like a derived class inherits from a base class, a smartphone inherits the fundamental functionality of a phone (making and receiving calls) from the general concept of a phone. However, a smartphone extends this base functionality by adding new attributes (e.g., touchscreen, apps, internet connectivity) and overriding or adding new behaviors (e.g., browsing the web, running applications) compared to a basic phone. The "Smartphone" "is a" "Phone" but with specialized features.

Part B: Minimal Coding (C++)

C++

```
#include <iostream>
#include <string>

class Vehicle {
public:
    std::string brand;

    Vehicle(std::string b) : brand(b) {}

    virtual void drive() {
        std::cout << "Vehicle is driving." << std::endl;
    }
};

class Car : public Vehicle {
public:
    int doors;

    Car(std::string b, int d) : Vehicle(b), doors(d) {}

    void drive() override {
        std::cout << "Car (brand: " << brand << ") with "
<< doors << " doors is driving on the road." << std::endl;
    }
};

int main() {
    Vehicle genericVehicle("Generic");
    Car myCar("Toyota", 4);

    genericVehicle.drive(); // Output: Vehicle is driving.
    myCar.drive();        // Output: Car (brand: Toyota)
with 4 doors is driving on the road.

    return 0;
}
```

Part B: Brief Explanation (for the Coding Example)

The `Car` class inherits publicly from the `Vehicle` class, meaning it gains access to the `brand` attribute and the `drive()` method of `Vehicle`. `Car` adds a new attribute, `doors`, which is specific to cars. The `drive()` method is overridden in the `Car` class to provide a more specialized behavior that indicates it's a car driving and also utilizes the `brand` attribute inherited from `Vehicle`.

Part C: Short Reflection & Discussion

- **When to Use Inheritance:**
    - **Clearly Beneficial:** When there is a clear "is-a" relationship between classes and significant code reuse can be achieved by inheriting common attributes and behaviors. For example, creating different types of employees (e.g., `SalariedEmployee`, `HourlyEmployee`) that inherit from a general `Employee` class with common attributes like `name` and `employeeId`.
    - **Potentially Overkill/Fragile Design:** When the "is-a" relationship is weak or forced, or when the base class has many unrelated features that derived classes don't need. This can lead to the "fragile base class problem," where changes in the base class can unintentionally break derived classes. For example, inheriting `ElectricKettle` from a very general `KitchenAppliance` class that has many methods irrelevant to a kettle might be overkill.
- **Method Overriding vs. Overloading:**
    - **Method Overriding (Runtime Polymorphism):** Occurs when a derived class provides a method with the same name, parameters, and return type as a method in its base class. The specific method to be executed is determined at runtime based on the actual type of the object being called.
    - **Method Overloading (Compile-Time Polymorphism):** Occurs when a class has multiple methods with the same name but different parameter lists (number, types, or order of parameters). The compiler determines which overloaded method

to call based on the arguments provided in the method invocation.

    - **Why inheritance relies heavily on overriding for real flexibility:** Inheritance allows derived classes to inherit the *structure* and *basic behavior* of the base class. However, to create truly specialized and useful derived classes that behave appropriately for their specific type, they often need to *modify* or *extend* the inherited behavior. Method overriding provides this crucial mechanism for tailoring the base class's functionality to the unique requirements of each derived class, enabling polymorphism and more flexible and adaptable code.

- **Inheritance vs. Interfaces/Abstract Classes:**
    - **Inheritance (from a concrete base class):** Provides both an interface (the public methods) and a partial or complete implementation that derived classes can reuse and extend. It establishes an "is-a" relationship and can include state (data members).
    - **Implementing an Interface:** Defines a contract (a set of methods that a class must implement) without providing any implementation details. A class can implement multiple interfaces, signifying that it adheres to multiple contracts or roles. It focuses on "can-do" relationships (e.g., a `Car` can "be driven," a `Bird` can "fly").
    - **Inheriting from an Abstract Base Class:** Similar to inheriting from a concrete base class, but abstract base classes can have abstract methods (declared but not implemented), forcing derived classes to provide their own implementations. They can also contain concrete methods and state. They often serve as a blueprint for a family of related classes.
- **Pitfalls of Multiple Inheritance:**
    - **Diamond Problem:** If a class `D` inherits from two classes `B` and `C`, which both inherit from a common superclass `A`, and `A` has a method `foo()`, `D` can inherit multiple implementations of `foo()`. This creates ambiguity for the compiler about which `foo()` to call when invoked on an object of type `D`.
    - **Mitigation Strategy:**
        - **Virtual Inheritance (C++):** In C++, virtual inheritance ensures that only one shared base class subobject is created when multiple inheritance occurs through a common ancestor, thus resolving the ambiguity of the diamond problem.
        - **Interface-Based Design:** Favoring the implementation of multiple interfaces over inheriting from multiple concrete classes can mitigate the diamond problem by separating interface from implementation. Classes can implement multiple interfaces, defining what they *can do*, without inheriting potentially conflicting implementations. Any shared implementation can be handled through composition.