Samuel Ukim

OOP HW4

Part A: Conceptual Questions

Definition

- **Polymorphism:** In my own words, polymorphism is the ability of objects of different classes to respond to the same method call in their own specific way. It allows treating objects of different types uniformly through a common interface (often a base class or interface), while the actual behavior executed depends on the object's concrete type at runtime.

- **Why often a pillar of OOP:** Polymorphism is considered a pillar of OOP because it promotes flexibility, extensibility, and code reusability. It allows writing more generic code that can work with objects of various related classes without needing to know their exact type at compile time. This makes systems easier to modify and extend with new types of objects without altering existing code significantly.

Compile-Time vs. Runtime

- **Compile-time polymorphism (method overloading):** Compile-time polymorphism is when the compiler determines which specific method to call based on the static types and number/types of arguments provided in the method call.
- **Runtime polymorphism (method overriding):** Runtime polymorphism is when the specific method to execute is determined at runtime based on the actual type of the object on which the method is called, even if it's accessed through a base class reference or pointer.
- Runtime polymorphism requires an inheritance relationship because the base class provides the common interface (the method signature), and derived classes provide their specific implementations of that interface. The runtime system needs to know the actual object's type (which is a subtype of the base type due to inheritance) to dispatch the correct overridden method.

Method Overloading

- **Why multiple methods with the same name but different parameter lists:** A class might have multiple methods with the same name but different parameter lists to provide different ways of performing a conceptually similar operation, catering to various input types or providing different levels of detail or options. This enhances the usability and flexibility of the class.

- **Brief example of simplifying user interactions:** A `MathUtils` class could have multiple `add()` methods: `add(int a, int b)` for integers, `add(double a, double b)` for doubles, and `add(int a, int b, int c)` for adding three integers. This allows the user to simply call `add()` with the appropriate number and types of arguments without needing to remember different function names for each scenario.

Method Overriding

- **How a derived class overrides:** A derived class overrides a base class's method by defining a method with the exact same signature (name, parameter types, and return type). When this overridden method is called on an object of the derived class (accessed through a base class reference or pointer), the derived class's implementation is executed instead of the base class's.

- **Why `virtual` keyword or annotations might be needed:** In some languages like C++, the `virtual` keyword in the base class method declaration is necessary to enable runtime polymorphism (dynamic dispatch). It tells the compiler that this method might be overridden by derived classes and that the correct version to call should be determined at runtime based on the object's actual type. Without `virtual`, the compiler might resolve the method call based on the static type of the reference or pointer, leading to the base class's method being called even on a derived class object. Annotations in other languages serve a similar purpose, explicitly marking a method as intended for overriding.

Part B: Minimal Demonstration (Pseudo-code)

Code snippet

```
// Base class
class Shape {
  virtual method draw() {
    // Optional default behavior
    print "Drawing a generic shape"
  }
}

// Derived class 1
class Circle inherits Shape {
  override method draw() {
    print "Drawing a circle"
  }
}

// Derived class 2
class Rectangle inherits Shape {
  override method draw() {
    print "Drawing a rectangle"
  }
}

// Demonstration
create a list of Shape pointers/references called shapes
create a Circle object called myCircle
create a Rectangle object called myRectangle

add myCircle to shapes
add myRectangle to shapes

for each shape in shapes:
  shape.draw() // The correct draw() method (Circle's or
Rectangle's) is called at runtime
```

Part B: Emphasis on Runtime Choice

In the pseudo-code above, the `shapes` list holds references (or pointers) to `Shape` objects. However, at runtime, some of these references actually point to `Circle` objects, and others point to `Rectangle` objects. When the `draw()` method is called on each element in the `shapes` list, the program dynamically determines the actual type of the object being referenced (either `Circle` or `Rectangle`) and executes the corresponding overridden `draw()` method defined in that specific derived class.

Part C: Overloading vs. Overriding Distinctions

- **Overloaded Methods (Calculator):** In the `Calculator` class with multiple `calculate()` methods accepting different parameter types, **compile-time resolution** is used. When you call `calculate(2, 3)`, the compiler examines the arguments (two integers) and selects the `calculate(int, int)` method based on the static types of the arguments provided in the call. The decision of which `calculate()` method to execute is made before the program runs.

- **Overridden Methods (Shape Example):** In the `Shape` example (or any scenario with method overriding), the decision of which `draw()` method to call occurs at **runtime**. When you call `shape.draw()`, the program needs to determine the actual type of the object that `shape` refers to (e.g., is it a `Circle` or a `Rectangle`?) and then execute the `draw()` method defined in that concrete class.

- **Why this matters for flexible code design:** Runtime polymorphism (overriding) is crucial for flexible code design because it allows you to write code that can work with a variety of objects through a common interface without needing to know their specific types in advance. This promotes loose coupling between different parts of the program, making it easier to extend the system with new types of objects without modifying existing code that relies on the base class interface. The behavior of the code adapts dynamically based on the actual objects it's working with.