Heshawa Cooray

Part A: Conceptual Questions

Definition Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables flexibility by letting a single interface represent multiple types.

Polymorphism is a core OOP principle because it allows for extensibility, reduces code duplication, and enhances maintainability by supporting dynamic behavior changes.

Compile-Time vs. Runtime

- **Compile-time polymorphism (method overloading)**: The same function name is used but with different parameters, resolved during compilation.
- **Runtime polymorphism (method overriding)**: A derived class provides a specific implementation of a method defined in the base class, determined at runtime.
- **Which requires inheritance?** Runtime polymorphism requires inheritance because the derived class must override a base class method.

Method Overloading

- Multiple methods with the same name but different parameters improve usability by allowing different input types.
- Example: A calculate() function in a Calculator class that accepts (int, int) or (double, double), making it easier to work with different number types.

Method Overriding

- A derived class overrides a base class method to provide specialized behavior.
- In languages like C++, virtual ensures the base method is properly overridden and enables runtime polymorphism.

Part B: Minimal Demonstration

```
class Shape {
public:
  virtual void draw() = 0; // Pure virtual function
};

class Circle : public Shape {
public:
  void draw() override {
    cout << "Drawing a Circle" << endl;
  }
};

class Rectangle : public Shape {
public:
  void draw() override {
    cout << "Drawing a Rectangle" << endl;
  }
};

int main() {
  Shape* shapes[] = { new Circle(), new Rectangle() };
  for (Shape* s : shapes) {
    s->draw();
  }
  return 0;
}
```

Explanation

- Shape is an abstract base class with a draw() method.
- Circle and Rectangle override draw() with specific behavior.
- A list of Shape* stores different objects, demonstrating runtime polymorphism.

Part C: Overloading vs. Overriding Distinctions

Overloaded Methods

- calculate(int, int) and calculate(double, double) in a Calculator class are resolved at compile time based on argument types.

Overridden Methods

- The draw() method in Shape is overridden by Circle and Rectangle.
- The decision to call Circle::draw() or Rectangle::draw() happens at **runtime**.
- This matters because it enables flexible and extensible designs without modifying existing code.

Part D: Reflection & Real-World Applications

Practical Example

- In a game engine, different entities (players, enemies, objects) inherit from a base Entity class with a render() method.
- Using polymorphism, a list of Entity* can call render() on all objects without knowing their specific types.

Potential Pitfalls

- **Method overloading**: Too many overloaded methods can cause confusion and ambiguity in function selection.
- **Runtime polymorphism**: Overuse can impact performance and make debugging harder due to indirect function calls.

Checking Understanding

- Adding Triangle to Shape does not require modifying existing code, polymorphism ensures that existing functions handling Shape references will automatically work with Triangle.