PART A: Conceptual Questions

1. Definition

Polymorphism is the ability of different classes to respond to the same function call in their own way. In simpler terms, it means one interface can be used for different data types or objects—allowing the same method name (like `draw()` or `speak()`) to behave differently depending on the object calling it.

Polymorphism is considered one of the pillars of OOP alongside encapsulation and inheritance because it promotes flexibility and reusability. It lets developers write generic code that can work with objects of different types, reducing duplication and making it easier to extend systems without modifying existing code.

2. Compile-Time vs. Runtime

Compile-time polymorphism (method overloading) allows multiple methods in the same class to have the same name but differ in parameters, with the correct one chosen at compile time.

Runtime polymorphism (method overriding) allows a derived class to provide a specific implementation of a method already defined in its base class, with the correct method chosen during program execution.

Runtime polymorphism requires an inheritance relationship because it relies on virtual functions and base class pointers or references to enable dynamic method resolution.

3. Method Overloading

A class might have multiple methods with the same name but different parameter lists (method overloading) to allow users to perform similar operations in different ways, improving flexibility and usability. For example:

A `Print` class might offer:

`print()`: prints a default message

`print(string message)`: prints a custom message

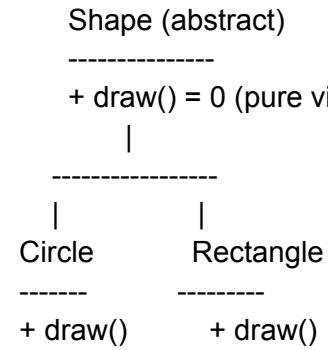
`print(int copies, string message)`: prints the message multiple times

4. Method Overriding

A derived class overrides a base class's method by redefining it with the same name and signature, allowing the derived class to provide specialized behavior specific to its role. This means when a method is called on a base class reference pointing to a derived object, the derived version executes, enabling runtime polymorphism.

In languages like **C++**, the `virtual` keyword is used in the base class to indicate that a method can be overridden. This tells the compiler to use **dynamic dispatch**, ensuring the correct version of the method (from the derived class) runs at runtime. Without `virtual`, C++ would default to **static dispatch**, and the base class's version would always run—even if the object is actually a derived instance—thus breaking polymorphism.

PART B: Minimal Demonstration



In C++ programming, an abstract class like “Shape” with a pure virtual method draw()allows polymorphism. A shape* can point to either a “Circle” or a “Rectangle” object, and at runtime, the correct draw() method is called based on the actual object’s type. This is known as dynamic dispatch or runtime polymorphism.

PART C: Overloading vs. Overriding Distinctions

1. Overloaded Methods

In the Calculator class with multiple calculate() methods accepting different parameter types—such as calculate(int, int) and calculate(double, double)—compile-time resolution is used by the compiler to determine which method to call based on the arguments provided at the time the code is compiled. This is known as **method overloading**, a form of **compile-time polymorphism**. The compiler examines the number and types of arguments passed in each method call and selects the most appropriate version of `calculate()` before the program runs

2. Overridden Methods

In the Shape example, where draw() is overridden in derived classes (e.g., Circle, Rectangle), the decision for which method to call occurs at runtime, if the method is marked as virtual (in C++) or uses polymorphic behavior. This is known as runtime polymorphism.
This matters for flexible code design because it allows developers to write general code that works with base class references or pointers, but still executes the correct derived class behavior. For example, an array of Shape* in C++ can store different types like Circle or Rectangle, and calling draw() on each will invoke the correct version. This enables extensibility without changing existing code—supporting the Open-Closed Principle—and allows systems to grow and adapt more easily.

PART D: Reflection & Real-World Applications

1. Practical Example

In a game, polymorphism is essential when dealing with various game entities like Player, Enemy, and NPC, all derived from a base class GameObject with a virtual method update(). Each class overrides update() to define specific behaviors—like movement, AI, or interactions—while the game engine simply loops through a list of GameObject* and calls update() on each one. This reduces code duplication by avoiding repeated conditional logic (e.g., if type == Player then...), and improves design by keeping entity-specific behavior encapsulated, making the system easier to extend and maintain.

2. Potential Pitfalls

Pitfall of Method Overloading:
One possible confusion is ambiguous method resolution, especially when implicit type conversions are involved. For example, calling calculate(5.0) might confuse the compiler if both calculate(int) and calculate(double) exist, leading to unexpected behavior or compilation errors.
Pitfall of Runtime Polymorphism:
A potential issue is performance overhead and debugging complexity, since virtual function calls require dynamic dispatch (vtable lookup in C++), which is slower than direct calls. It also makes tracing and debugging harder, as it's not always obvious at runtime which overridden method is being executed.

3. Checking Understanding

Polymorphism allows the Triangle class to override methods like draw() from the base Shape class, so existing code that uses Shape references or pointers does not need to change. For example, if you have a function that loops through a list of Shape* and calls draw() on each one, the new Triangle objects can be added to the list, and the correct draw()behavior will be called at runtime—without modifying that loop or any existing logic. This supports extensibility and adheres to the Open-Closed Principle.