Polymorphism

Polymorphism allows the same method name or interface to exhibit different behaviors depending on the object that is invoking it.

The term "polymorphism" comes from Greek and means "many forms." In programming, it allows us to write code that is generic, extensible, and reusable, while the specific behavior is determined at runtime or compile-time based on the object's actual type.

Polymorphism lets you call the same method on different objects, and have each object respond in its own way.

You write code that targets a common type, but the actual behavior is determined by the concrete implementation.

Real-World Analogy
Think of a remote control. Whether it operates a TV, an air conditioner, or a projector, the button press action remains the same for the user. Internally, though, each device responds differently.

That's polymorphism at work—the same interface (remote control) triggers different behaviors based on the receiver (device type).

Why Polymorphism Matters
Encourages loose coupling: You interact with abstractions (interfaces or base classes), not specific implementations.
Enhances flexibility: You can introduce new behaviors without modifying existing code, supporting the Open/Closed Principle.
Promotes scalability: Systems can grow to support more features with minimal impact on existing code.
Enables extensibility: You can "plug in" new implementations without touching the core business logic.
Two Types of Polymorphism
1. Compile-time Polymorphism (Static Binding)
Also known as method overloading, this occurs when:

You have multiple methods with the same name in the same class.
Each method has a different parameter list (number, type, or order).
The method to call is determined by the compiler at compile time.
Example:

Java

```
public class Calculator {
   public int add(int a, int b) {
       return a + b;
   }

   public double add(double a, double b) {
       return a + b;
   }

   public int add(int a, int b, int c) {
       return a + b + c;
   }
}
```

When you call add(), the compiler selects the appropriate method based on the arguments passed.

2. Runtime Polymorphism (Dynamic Binding)
Also known as method overriding, this happens when:

A subclass overrides a method defined in its superclass or interface.
The method to invoke is determined at runtime, based on the actual object type.
Example:
Suppose you're designing a system that sends notifications. You want to support email, SMS, push notifications, etc.

You start by defining a common interface.

Java

```
public interface NotificationSender {
    void sendNotification(String message);
}
```

Python

```
class NotificationSender(ABC):
    @abstractmethod
    def send_notification(self, message):
        pass
```

Now, you implement it in multiple ways:

Java

```
public class EmailSender implements NotificationSender {
    public void sendNotification(String message) {
        System.out.println("Sending EMAIL: " + message);
    }
}

public class SmsSender implements NotificationSender {
    public void sendNotification(String message) {
        System.out.println("Sending SMS: " + message);
    }
}
```

Python

```
class EmailSender(NotificationSender):
    def send_notification(self, message):
        print("Sending EMAIL:", message)

class SmsSender(NotificationSender):
    def send_notification(self, message):
        print("Sending SMS:", message)
```

And use it like this:

Java

```
public void notifyUser(NotificationSender sender, String message) {
    sender.sendNotification(message);
}
```

Python

```
def notify_user(sender, message):
    sender.send_notification(message)
```

You can pass any implementation of NotificationSender, and the correct behavior will be triggered based on the object passed.

This is runtime polymorphism, where the decision of which method to execute is made during execution, not at compile time.

Polymorphism in LLD Interviews
Polymorphism is especially useful in Low-Level Design when:

You want to plug in different behaviors without modifying the core logic
You need to support extensible systems with new types of objects (e.g., different payment providers, transport types, etc.)
You want to design to interfaces or base classes and allow flexibility in how objects behave
For example: if you're designing a PaymentProcessor interface, you can have multiple implementations like CreditCardProcessor, PayPalProcessor, and UPIProcessor. The payment system doesn’t need to care which one it's using, it just calls processPayment().