Interfaces

In object-oriented design, **interfaces** play a foundational role in building systems that are **extensible**, **testable**, and **loosely coupled**.
They allow different parts of the system to interact through well-defined contracts without needing to know how the behavior is actually implemented.

1. What is an Interface?

At its core, an **interface defines a contract**: a set of method signatures that any implementing class must fulfill. It declares **what** a class can do, but not **how** it does it.

Real-World Analogy

Consider a remote control. It exposes a standard set of buttons:

- `play()`
- `pause()`
- `volumeUp()`
- `powerOff()`

The person using the remote doesn't care whether it controls a TV, a soundbar, or a projector. The **interface remains the same**, but the **underlying device behaves differently**.
This is exactly how interfaces work in software design.

2. Key Properties of Interfaces

Defines behavior without dictating implementation
An interface specifies *what* operations are expected, but not *how* they are carried out. This gives freedom to implementers to provide customized logic while still honoring the contract.

Enables polymorphism
Different classes can implement the same interface in different ways. This allows your code to work with different implementations interchangeably.

Promotes decoupling

Code that depends on interfaces is insulated from changes in concrete implementations. This reduces the ripple effect of changes and increases testability and maintainability.

3. Example: Payment Gateway Interface

Let's say you're designing a payment processing module that supports multiple providers like **Stripe**, **Razorpay**, and **PayPal**.

You can define a generic interface:

Python

```
from abc import ABC, abstractmethod

class PaymentGateway(ABC):
    @abstractmethod
    def initiate_payment(self, amount):
        pass
```

Java

```
public interface PaymentGateway {
    void initiatePayment(double amount);
}
```

This interface doesn't care how the payment is processed—it only mandates that all implementing classes must define a method called initiatePayment().

Now you can create multiple implementations:

Java

```
public class StripePayment implements PaymentGateway {
    public void initiatePayment(double amount) {
        System.out.println("Processing payment via Stripe: $" + amount);
    }
}

public class RazorpayPayment implements PaymentGateway {
    public void initiatePayment(double amount) {
```

```
        System.out.println("Processing payment via Razorpay: ₹" + amount);
    }
}
```

Python

```
class StripePayment(PaymentGateway):
    def initiate_payment(self, amount):
        print(f"Processing payment via Stripe: ${amount}")

class RazorpayPayment(PaymentGateway):
    def initiate_payment(self, amount):
        print(f"Processing payment via Razorpay: ₹{amount}")
```

Both StripePayment and RazorpayPayment implement the same interface, but the actual logic for processing the payment is different.

Usage: Loose Coupling in Action
Now let's say you have a CheckoutService that processes payments. Instead of hardcoding a specific payment gateway, you inject the interface:

Python

```
class CheckoutService:
    def __init__(self, payment_gateway):
        self.payment_gateway = payment_gateway

    def set_payment_gateway(self, payment_gateway):
        self.payment_gateway = payment_gateway

    def checkout(self, amount):
        self.payment_gateway.initiate_payment(amount)
```

Java

```
public class CheckoutService {
    private PaymentGateway paymentGateway;

    public CheckoutService(PaymentGateway paymentGateway) {
        this.paymentGateway = paymentGateway;
    }

    public void setPaymentGateway(PaymentGateway paymentGateway) {
        this.paymentGateway = paymentGateway;
```

```
    }

    public void checkout(double amount) {
        paymentGateway.initiatePayment(amount);
    }
}
```

Now you can plug in any payment gateway at runtime:

Python

```
if __name__ == "__main__":
    stripe_gateway = StripePayment()
    checkout_service = CheckoutService(stripe_gateway)
    checkout_service.checkout(120.50)  # Output: Processing payment via Stripe: $120.5

    # Switch to Razorpay
    razorpay_gateway = RazorpayPayment()
```

Java

```
public class Main {
    public static void main(String[] args) {
        PaymentGateway stripeGateway = new StripePayment();
        CheckoutService service = new CheckoutService(stripeGateway);
        service.checkout(120.50);  // Output: Processing payment via Stripe: $120.5

       // Switch to Razorpay
        PaymentGateway razorpayGateway = new RazorpayPayment();
        service.setPaymentGateway(razorpayGateway);
        service.checkout(150.50);  // Output: Processing payment via Razorpay: ₹150.5
    }
}
   checkout_service.set_payment_gateway(razorpay_gateway)
    checkout_service.checkout(150.50)  # Output: Processing payment via Razorpay: ₹150.5
```

No change required in CheckoutService