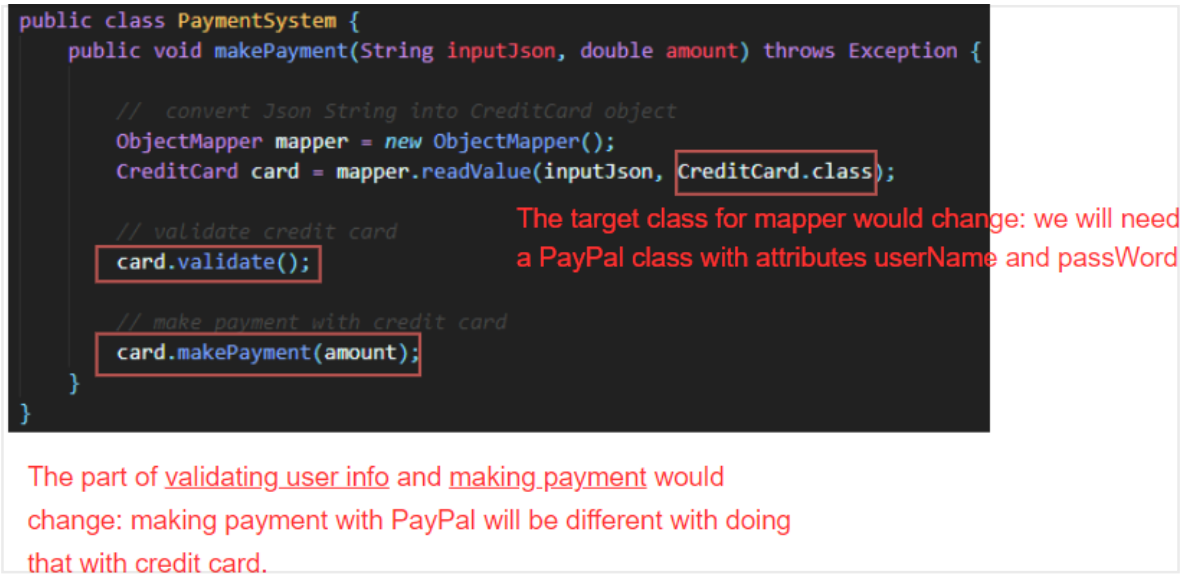
Dependency Inversion Principle(DIP)

SOLID Design in Practice — D: Dependency Inversion Principle

Imagine you are designing an online payment system for an e-commerce website, and you have coded up a simple backend for accepting payments using **credit card** shown below.

```
class CreditCard {
    String cardNo;
    int expMonth;
    int expYear;
    String cvc;

    CreditCard(String cardNo, int expMonth, int expYear, String cvc) {
        // this.attribute = attribute
    }

    public void validate() throws Exception {
        // Implementation of validating info of credit card
    }

    public void makePayment(double amount) throws Exception {
        // Implementation of making payment with credit card
    }
}

public class PaymentSystem {
    public void makePayment(String inputJson, double amount) throws Exception {

        //  convert Json String into CreditCard object
        ObjectMapper mapper = new ObjectMapper();
        CreditCard card = mapper.readValue(inputJson, CreditCard.class);

        // validate credit card
        card.validate();

        // make payment with credit card
        card.makePayment(amount);
    }
}
```

It works really well. However, one day your manager asked you to modify the system to make it support **PayPal** as well. Which part of the code we will have to change to support making payment with PayPal?



So pretty much everything. If we were to make payments in other ways: Debit card, Gift card, Venmo, Stripe, Apple Pay..., we will have to keep modifying the code. That will be a lot of work to do!

What's wrong with the original code? Instantiating **CreditCard** object inside the method of **PaymentSystem** class indicates that we are **tightly coupling** our payment system to this particular payment method. That is a bad practice should be avoided!

Dependency inversion principle

To fix this problem, we now introduce the dependency inversion principle. In object-oriented design, the **dependency inversion principle** is a specific methodology for **loosely coupling** software modules. The principle states:

- High-level modules should not depend upon low-level modules. Both should depend on **abstractions** (**interfaces**).
- **Low-level module** implements a basic functionality can be used anywhere
- **High-level module** is a module implements some complex business logic

2. Abstractions should not depend on details (**implementations**). Details should depend on abstractions.

Fix the code

According to this design principle, we should replace the concrete implementation(**CreditCard**) with an abstract interface**(PaymentService)**, such that our high level module(**PaymentSystem**) is not tightly coupled to any concrete class. To make payment using different methods, we can pass in a different implementation of such interface!

```java
public class PaymentSystem {

    PaymentService paymentService;

    public void setPaymentService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void makePayment(String inputJson, double amount) throws Exception {
        paymentService.validate(inputJson);
        paymentService.makePayment(amount);
    }
}
```

The **PaymentService** interface simply specifies behaviors that need to be implemented by all payment methods (see the **CreditCardPaymentService** and **PayPalPaymentService** below)**.**

```java
interface PaymentService {
    public void validate(String inputJson) throws Exception;
    public void makePayment(double amount) throws Exception;
}

class CreditCardPaymentService implements PaymentService {

    CreditCardClient creditCardClient;

    public void validate(String inputJson) throws Exception {
        // Implementation of validating credit card
        // creditCardClient = ...
    }

    public void makePayment(double amount) throws Exception{
        // Implementation of making payment with credit card
        // creditCardClient.makePayment(amount);
    }
}

class PayPalPaymentService implements PaymentService {

    PayPalClient payPalClient;

    public void validate(String inputJson) throws Exception {
        // Implementation of validating PayPal account
        // payPalClient = ...
    }

    public void makePayment(double amount) throws Exception{
        // Implementation of making payment with PayPal
        // payPalClient.makePayment(amount);
    }
}
```

Now the code has became a lot more scalable!

Summary:

To summarize, instantiating objects (with **new** keyword) inside the method usually indicate **tight coupling** of the modules. In other words, changes in one class will force the changes in the classes it closely depend on. It trades software extensibility and scalability for its simplicity.

On the contrary, loose coupling means classes work independently. It can be achieved by applying the **dependency inversion principle**, usually with the help of **Interfaces**. Classes interact through interfaces rather than some concrete implementations to increases the flexibility of a system, and make the code more ‘robust’.