SOLID Principles

Sammi Aldhi Yanto

A nice guy, Gopher, and Kotlin Enthusiast

Introduction to SOLID

- SOLID principles are an object-oriented approach that are applied to software structure design
- It is conceptualized by Robert C. Martin (also known as Uncle Bob)
- The SOLID principles is not a particular law or rule that we are obliged to obey, but rather a principle that is meant to assist us in writing neat code
- It also ensures that the software is modular, tolerant of change, easy to understand, debug, refactor, and the basic components can be reused in the form of other system software

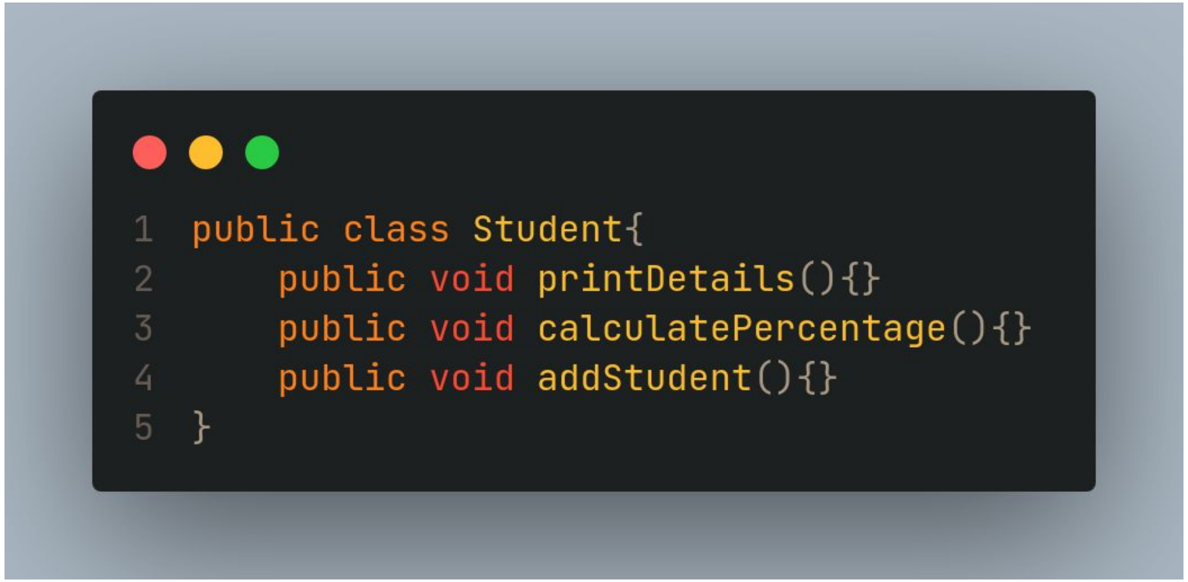
The word SOLID acronym for

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Single Responsibility Principle

- *Every Java class must perform a single functionality*
- *Responsibility means that if a class has 2 (two) unrelated functionality to make a change, then we have to divide the different functionality in one way into two different classes*
- *Each class that has been separated based on its functionality will only handle one responsibility*
- *We just need to focus on each class that has been separated*

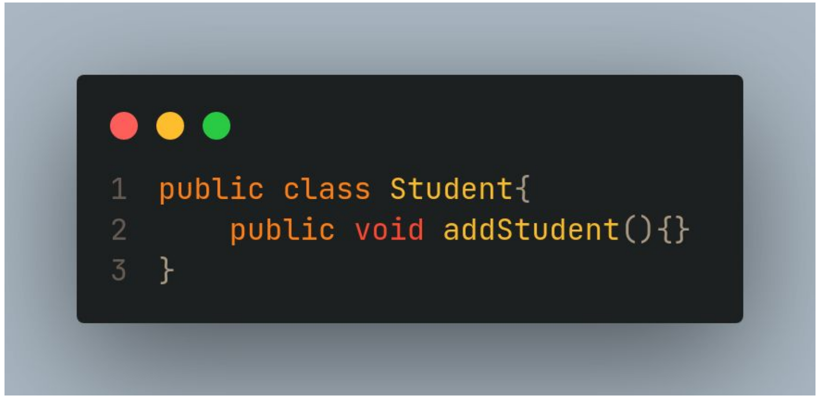
Let's understand the SRP through an example.



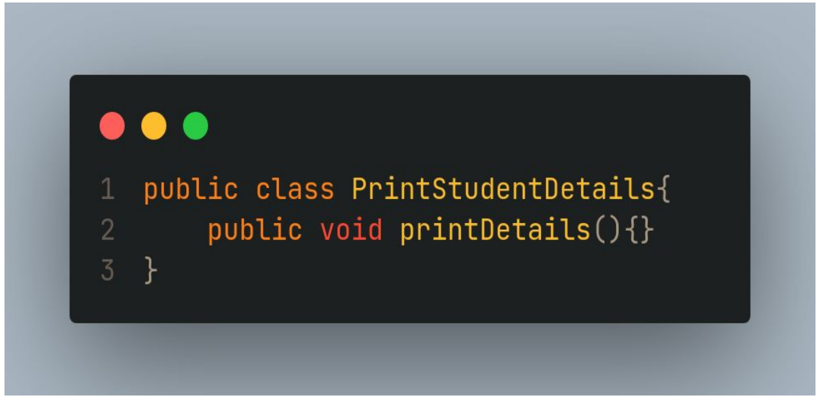
```java
public class Student{
    public void printDetails(){}
    public void calculatePercentage(){}
    public void addStudent(){}
}
```

Ooops!

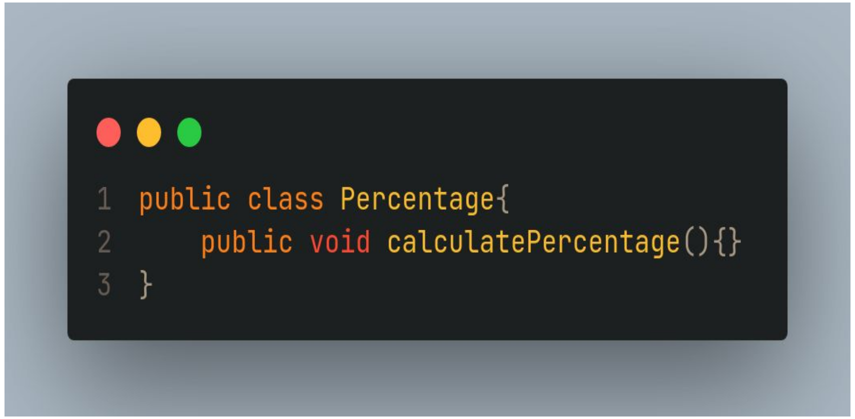
The code snippet violates the single responsibility principle. To achieve the goal of the principle, we should implement a separate class that performs a single functionality only.



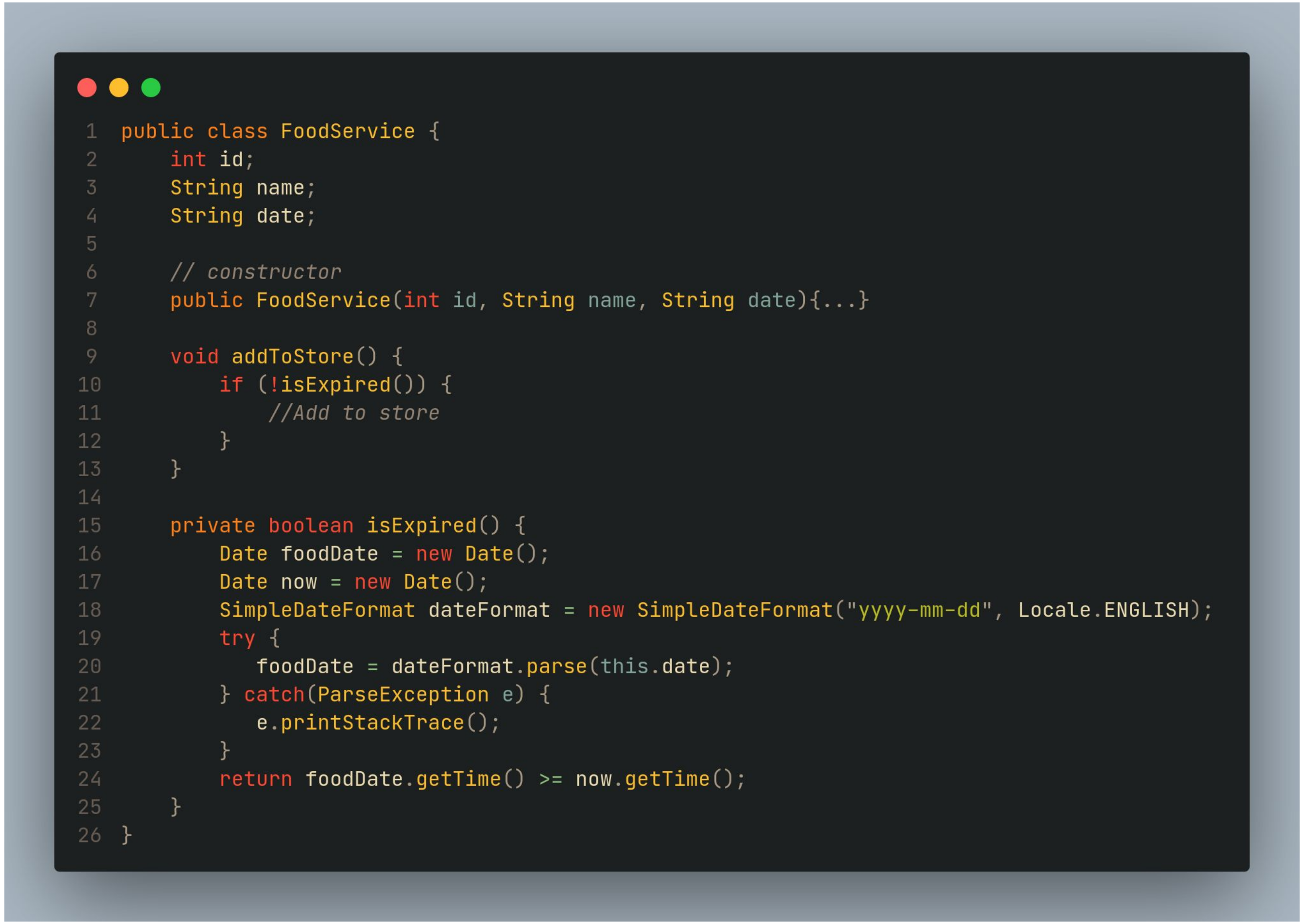
```java
public class Student{
    public void addStudent(){}
}
```



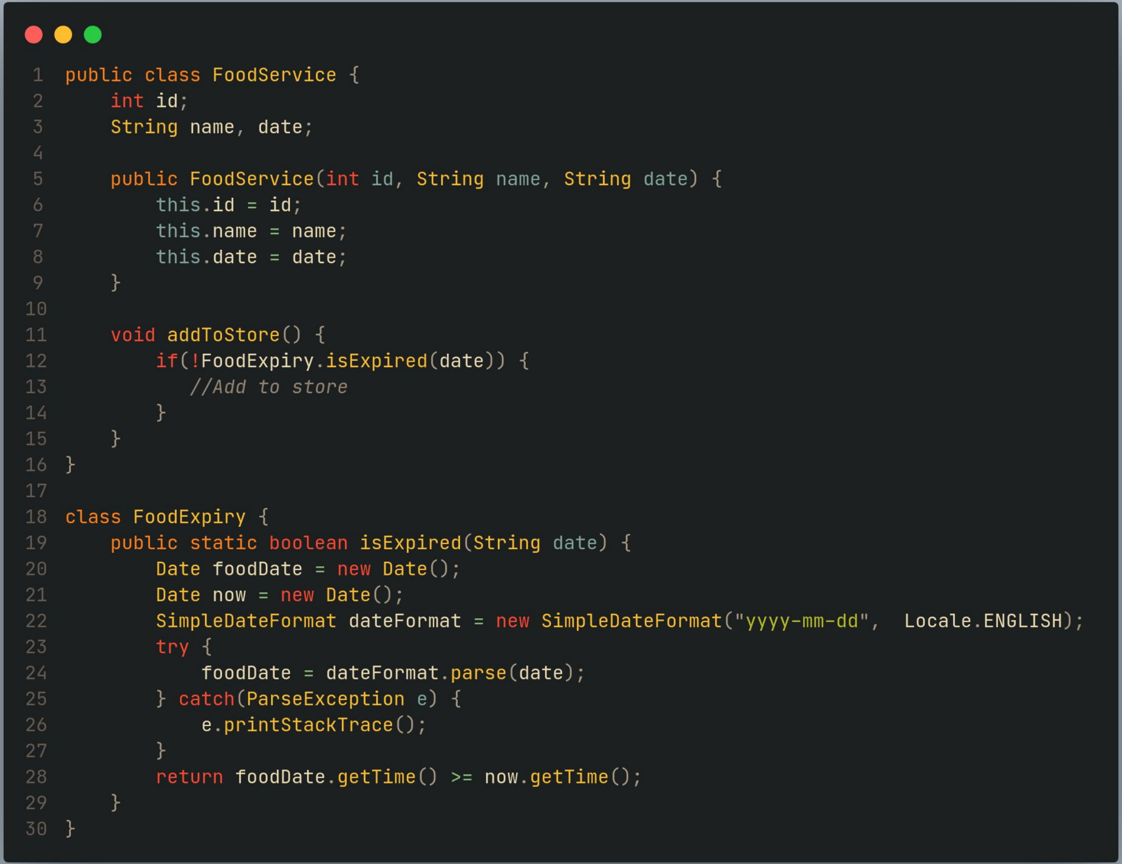
```java
public class PrintStudentDetails{
    public void printDetails(){}
}
```



```java
public class Percentage{
    public void calculatePercentage(){}
}
```



```java
public class FoodService {
    int id;
    String name;
    String date;

    // constructor
    public FoodService(int id, String name, String date){...}

    void addToStore() {
        if (!isExpired()) {
            //Add to store
        }
    }

    private boolean isExpired() {
        Date foodDate = new Date();
        Date now = new Date();
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-mm-dd", Locale.ENGLISH);
        try {
           foodDate = dateFormat.parse(this.date);
        } catch(ParseException e) {
           e.printStackTrace();
        }
        return foodDate.getTime() >= now.getTime();
    }
}
```

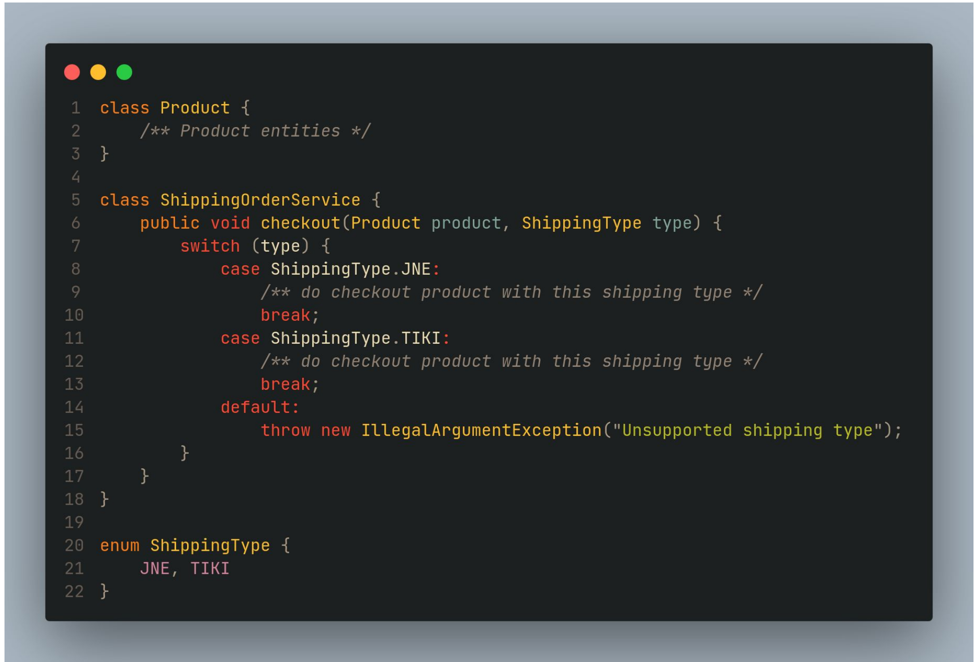


```java
public class FoodService {
    int id;
    String name, date;

    public FoodService(int id, String name, String date) {
        this.id = id;
        this.name = name;
        this.date = date;
    }

    void addToStore() {
        if(!FoodExpiry.isExpired(date)) {
           //Add to store
        }
    }
}

class FoodExpiry {
    public static boolean isExpired(String date) {
        Date foodDate = new Date();
        Date now = new Date();
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-mm-dd",  Locale.ENGLISH);
        try {
            foodDate = dateFormat.parse(date);
        } catch(ParseException e) {
            e.printStackTrace();
        }
        return foodDate.getTime() >= now.getTime();
    }
}
```

Open-Closed Principle

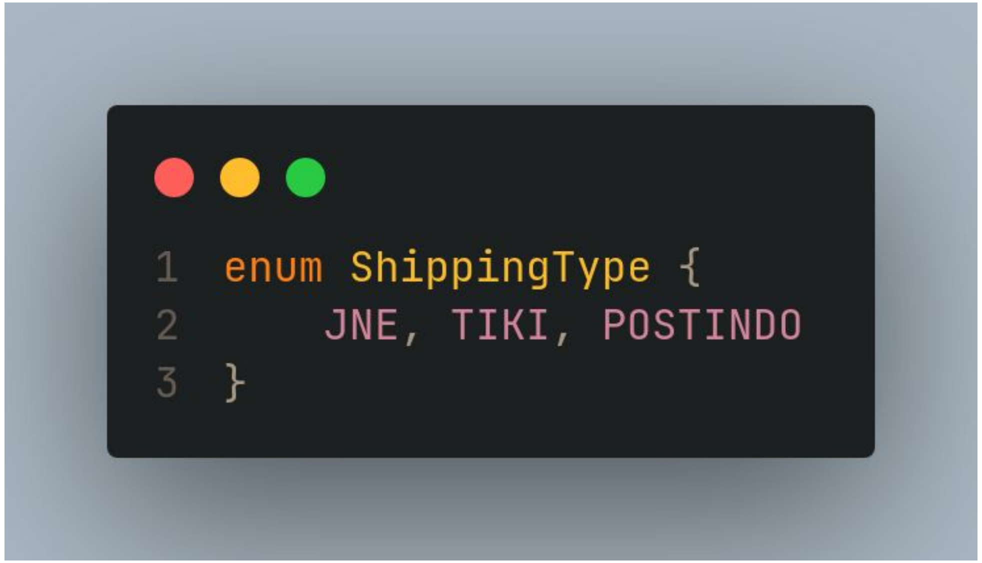
- *the module should be open for extension but closed for modification*
- *The extension allows us to implement new functionality to the module*

Let's understand the OCP through an example

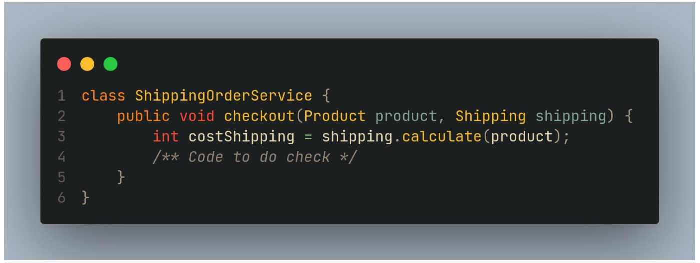


```java
class Product {
    /** Product entities */
}

class ShippingOrderService {
    public void checkout(Product product, ShippingType type) {
        switch (type) {
            case ShippingType.JNE:
                /** do checkout product with this shipping type */
                break;
            case ShippingType.TIKI:
                /** do checkout product with this shipping type */
                break;
            default:
                throw new IllegalArgumentException("Unsupported shipping type");
        }
    }
}

enum ShippingType {
    JNE, TIKI
}
```

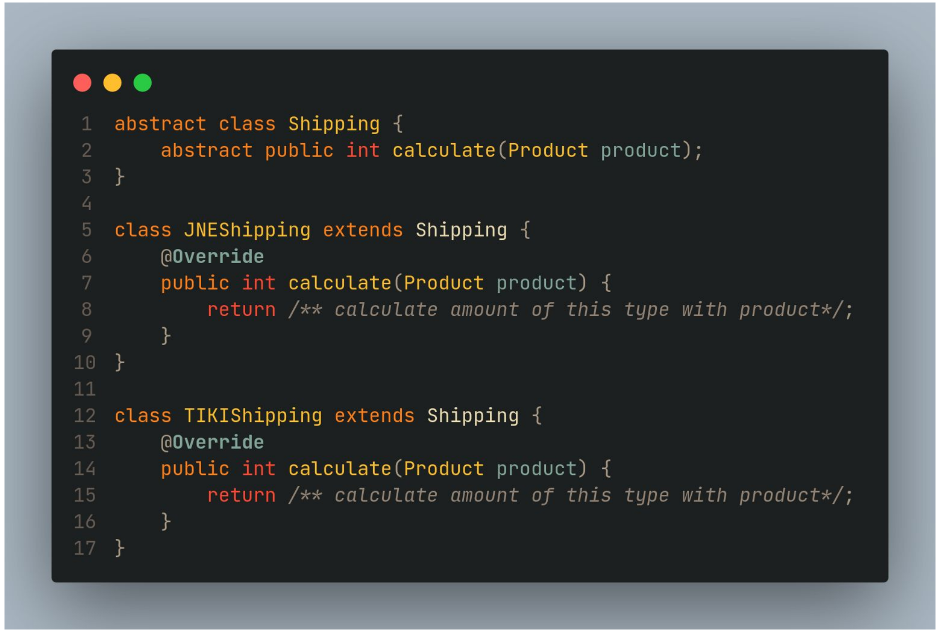
Ooops!



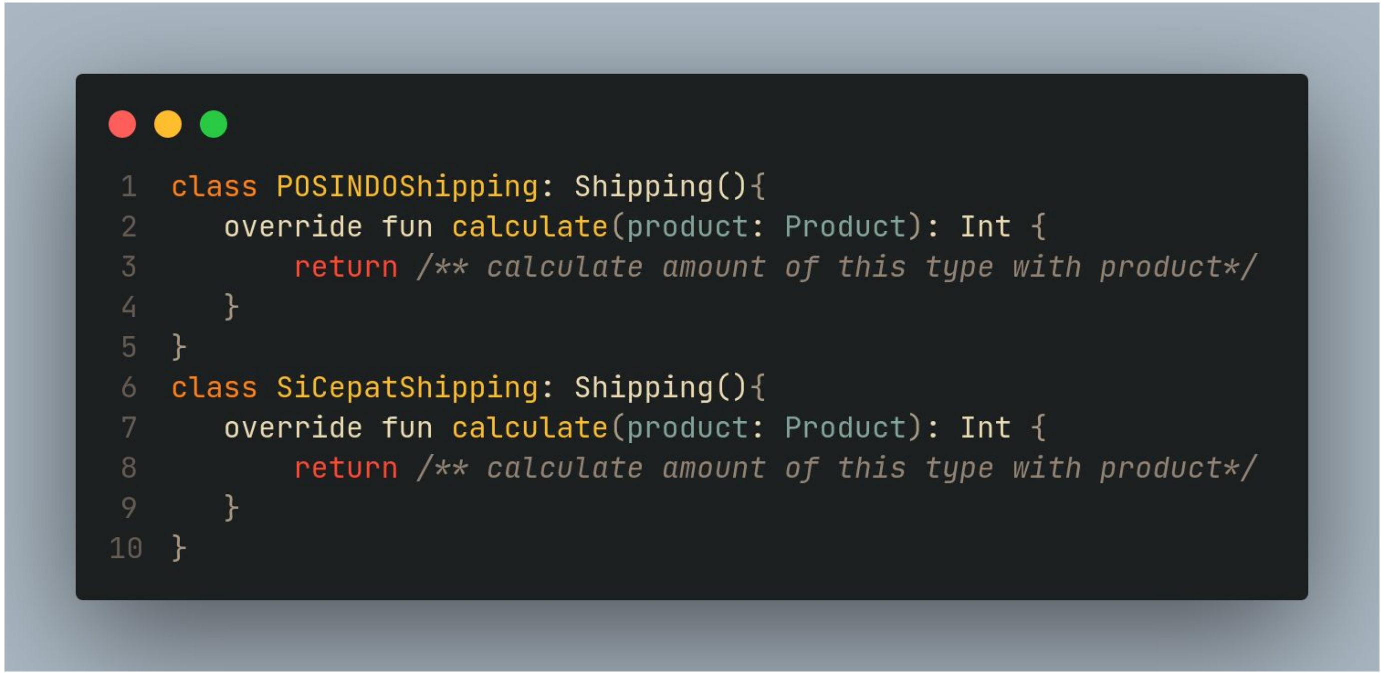
```
enum ShippingType {
    JNE, TIKI, POSTINDO
}
```



```java
class ShippingOrderService {
    public void checkout(Product product, Shipping shipping) {
        int costShipping = shipping.calculate(product);
        /** Code to do check */
    }
}
```



```java
abstract class Shipping {
    abstract public int calculate(Product product);
}

class JNEShipping extends Shipping {
    @Override
    public int calculate(Product product) {
        return /** calculate amount of this type with product*/;
    }
}

class TIKIShipping extends Shipping {
    @Override
    public int calculate(Product product) {
        return /** calculate amount of this type with product*/;
    }
}
```



```kotlin
class POSINDOShipping: Shipping(){
   override fun calculate(product: Product): Int {
       return /** calculate amount of this type with product*/
   }
}
class SiCepatShipping: Shipping(){
   override fun calculate(product: Product): Int {
       return /** calculate amount of this type with product*/
   }
}
```

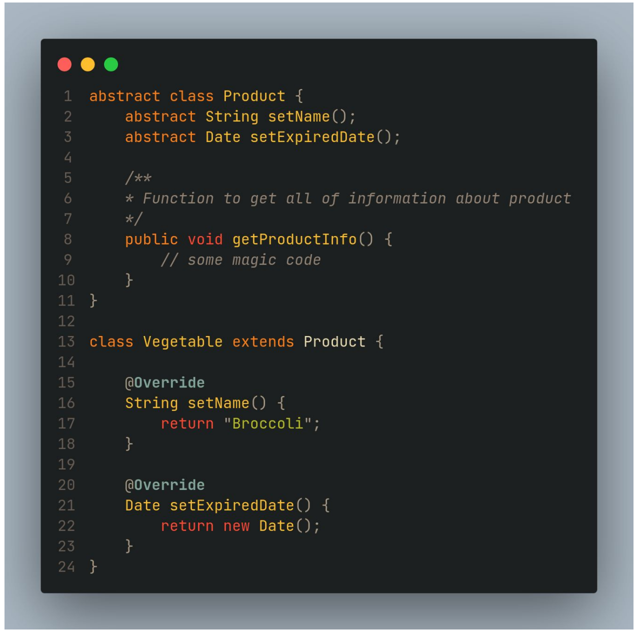
Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) was introduced by Barbara Liskov
- *Simply put, Liskov's substitution is the rule that applies to hierarchical inheritance*
- *All SubClass can run at least the same way as its SuperClass.*

there are several rules that must be obeyed

- **Contravariant** is a condition in which the parameters of a function in the SubClass must have the same type and number as the function in its SuperClass. Meanwhile, **Covariant** is a condition to return the value of functions that are in SubClass and SuperClass
- **Preconditions** and **Postconditions**. These are actions that must exist before or after a process is executed
- The **Constraints** here are the restrictions the SuperClass imposes on changing the state of an object
    - For example, suppose that the SuperClass has an object that has a fixed value, then the SubClass is not allowed to change the state of the object's value

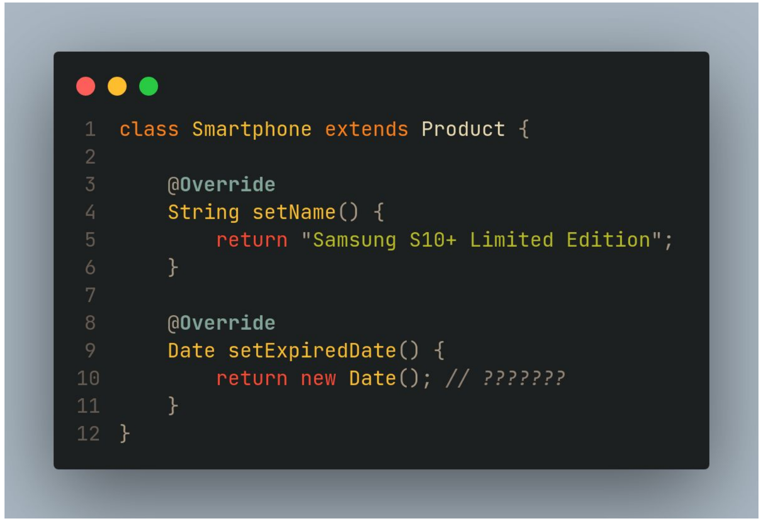
Let's understand the LSP through an example.



```java
abstract class Product {
    abstract String setName();
    abstract Date setExpiredDate();

    /**
    * Function to get all of information about product
    */
    public void getProductInfo() {
        // some magic code
    }
}

class Vegetable extends Product {

    @Override
    String setName() {
        return "Broccoli";
    }

    @Override
    Date setExpiredDate() {
        return new Date();
    }
}
```

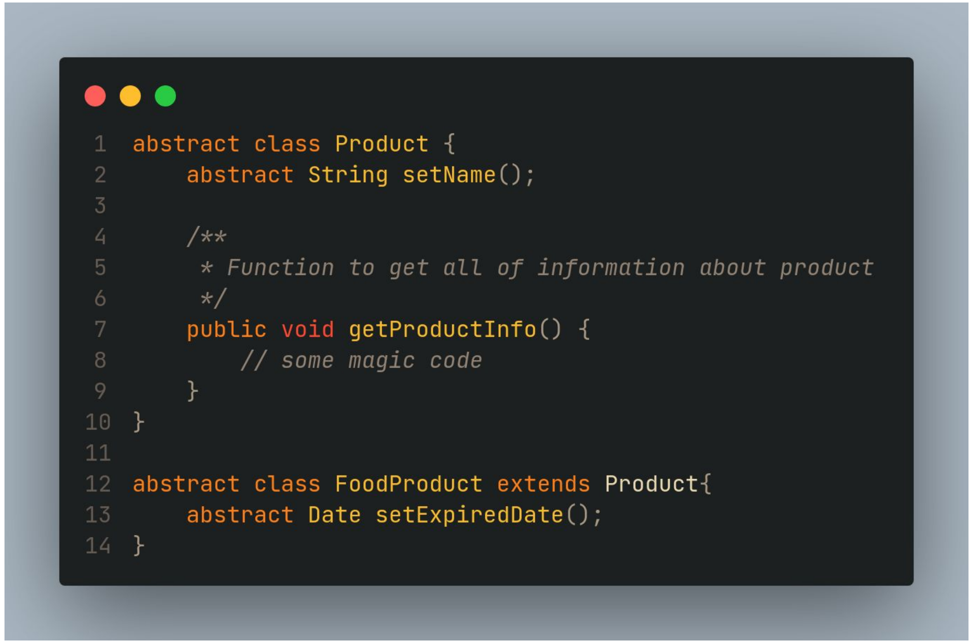
Ooops!

- In this case the Product class becomes irrelevant to be inherited to the Smartphone class and this certainly violates the SubClass rules

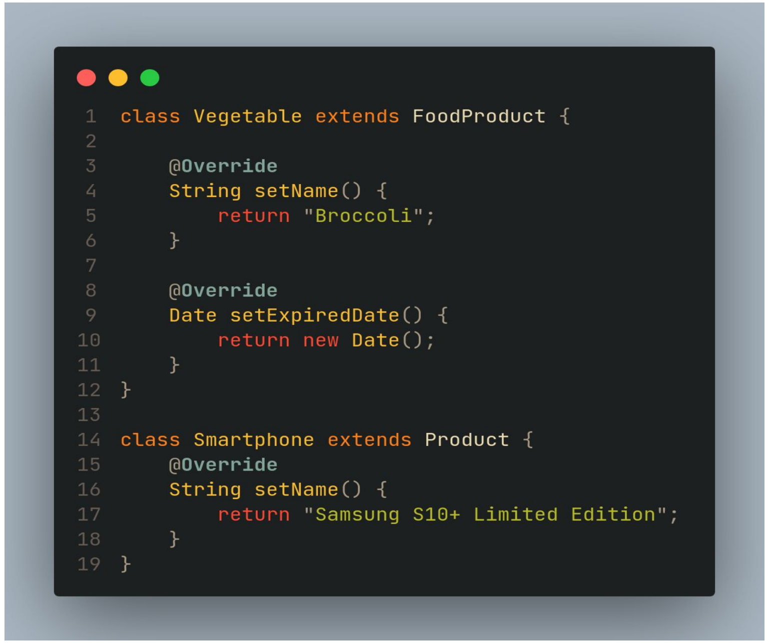


```java
class Smartphone extends Product {

    @Override
    String setName() {
        return "Samsung S10+ Limited Edition";
    }

    @Override
    Date setExpiredDate() {
        return new Date(); // ???????
    }
}
```

This change keeps the Product class a SuperClass



```java
abstract class Product {
    abstract String setName();

    /**
     * Function to get all of information about product
     */
    public void getProductInfo() {
        // some magic code
    }
}

abstract class FoodProduct extends Product{
    abstract Date setExpiredDate();
}
```

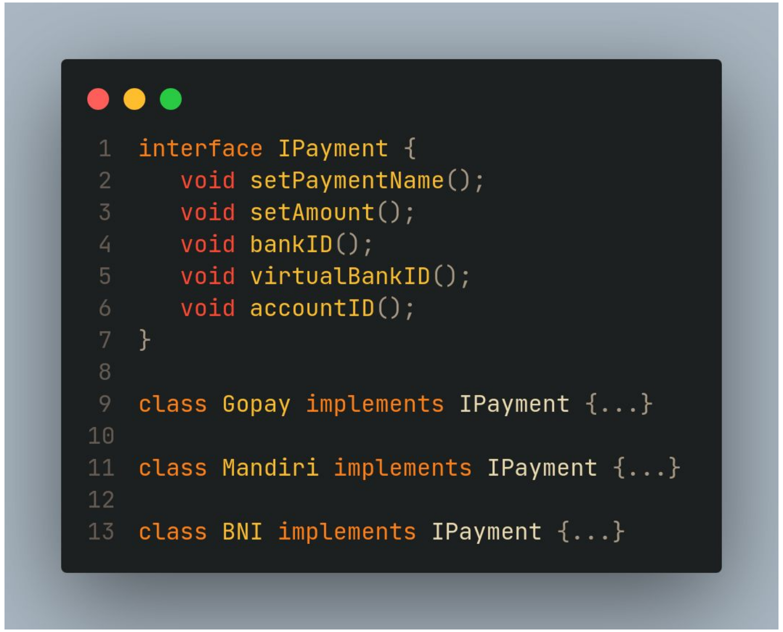


```java
class Vegetable extends FoodProduct {

    @Override
    String setName() {
        return "Broccoli";
    }

    @Override
    Date setExpiredDate() {
        return new Date();
    }
}

class Smartphone extends Product {
    @Override
    String setName() {
        return "Samsung S10+ Limited Edition";
    }
}
```

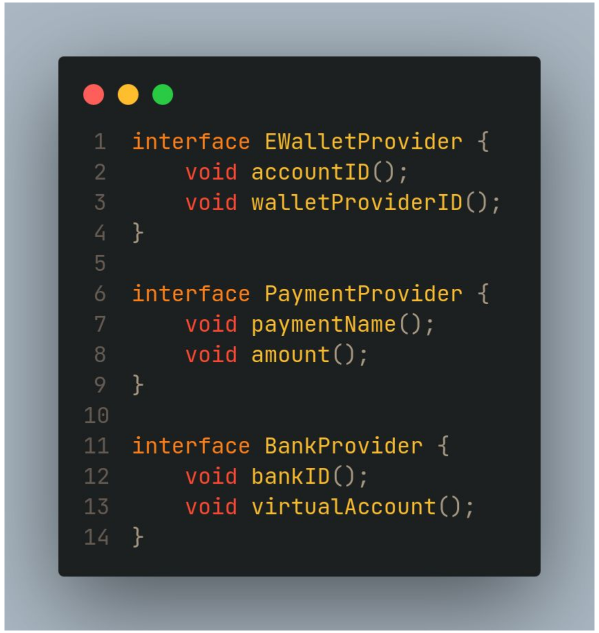
Interface Segregation Principle

- The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use
- The goal of the interface segregation principle is similar to the single responsibility principle
- This principle itself aims to reduce the number of dependencies of a class on unneeded interface classes

Let's understand the ISP through an example

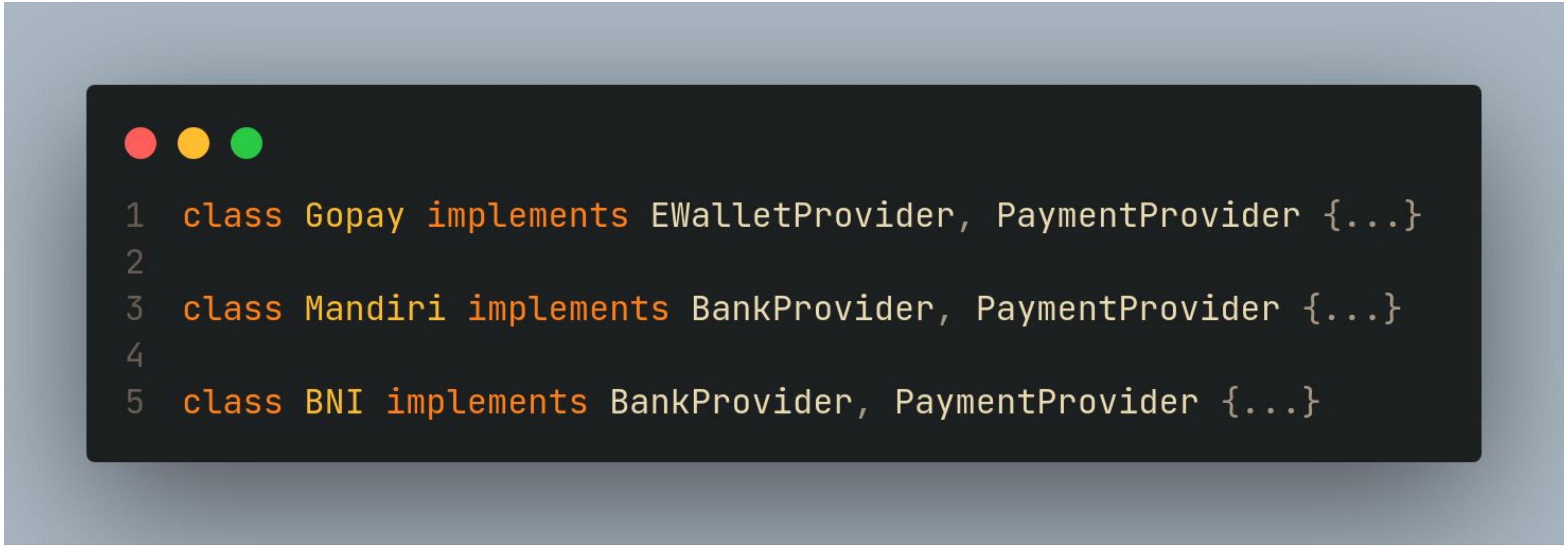


```
interface IPayment {
    void setPaymentName();
    void setAmount();
    void bankID();
    void virtualBankID();
    void accountID();
}

class Gopay implements IPayment {...}

class Mandiri implements IPayment {...}

class BNI implements IPayment {...}
```

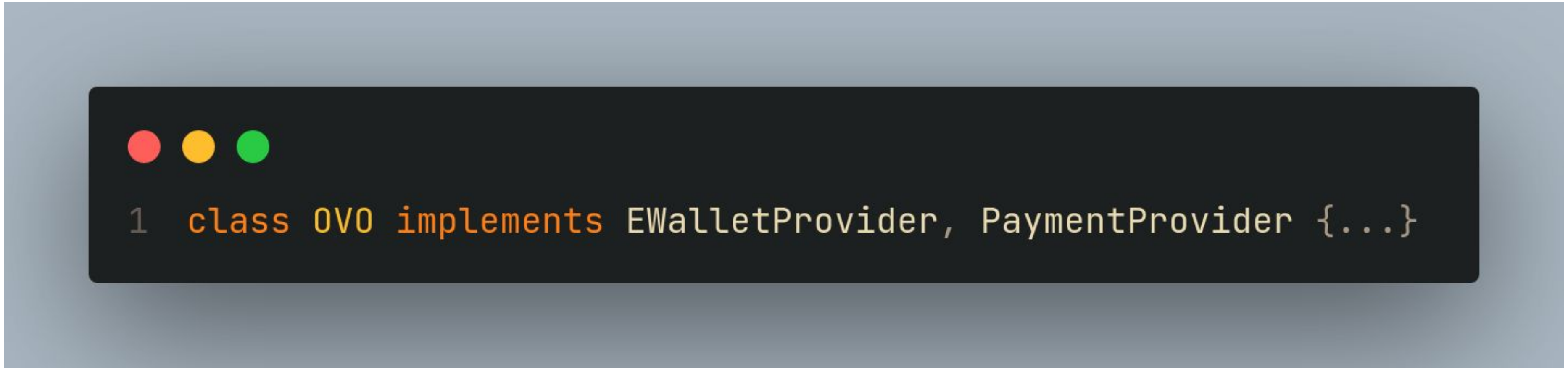


```
interface EWalletProvider {
     void accountID();
     void walletProviderID();
}

interface PaymentProvider {
     void paymentName();
     void amount();
}

interface BankProvider {
     void bankID();
     void virtualAccount();
}
```

By breaking the interface into several small interfaces, we can easily customize the needs of each class.



```java
class Gopay implements EWalletProvider, PaymentProvider {...}

class Mandiri implements BankProvider, PaymentProvider {...}

class BNI implements BankProvider, PaymentProvider {...}
```

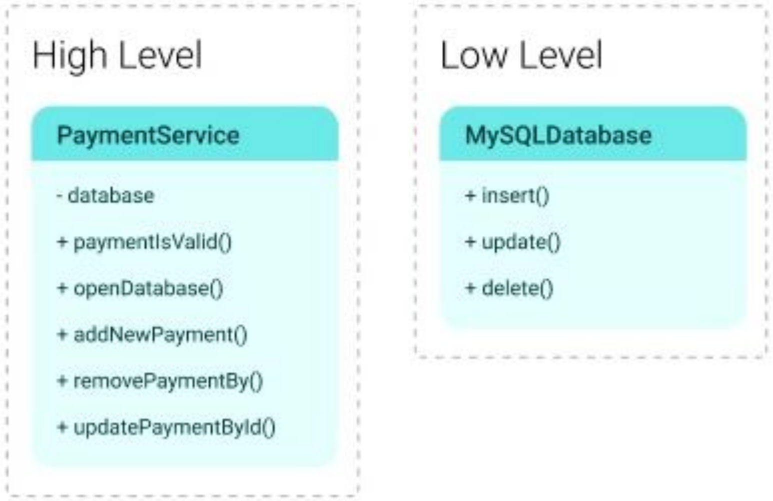


```
class OVO implements EWalletProvider, PaymentProvider {...}
```

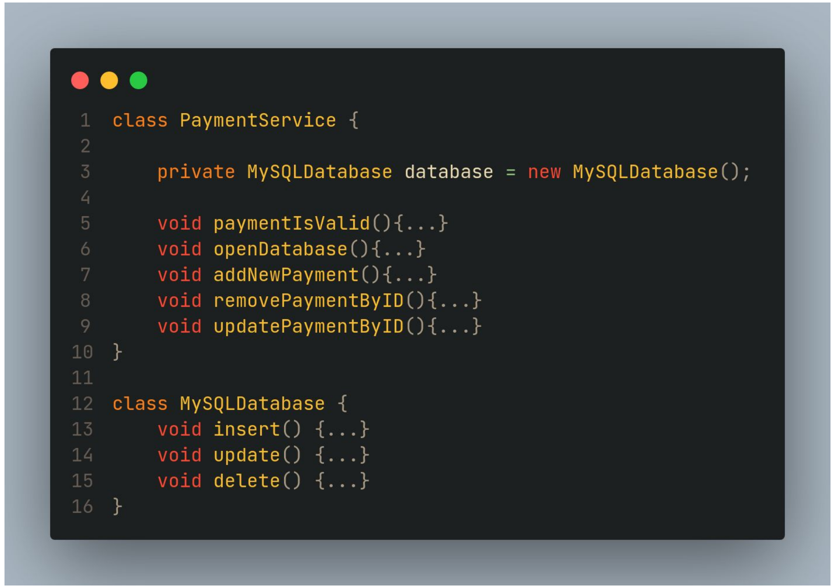
Dependency Inversion Principle

- The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. *High-level modules should not depend on the low-level module but both should depend on the abstraction (Robert Cecil Martin)*
- ***High-level modules*** *are classes that deal with bundles of functionality*
- ***Low-level modules*** *are responsible for more detailed operations. At the lowest level allows this module to be responsible for writing information to the database*

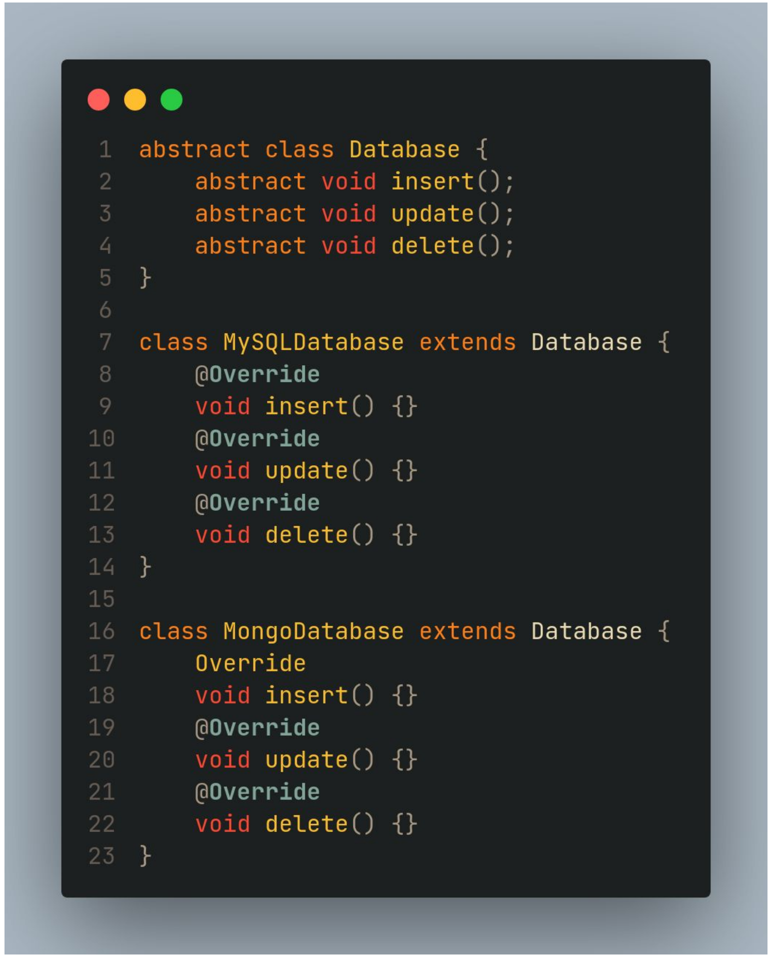
Let's understand the DIP through an example



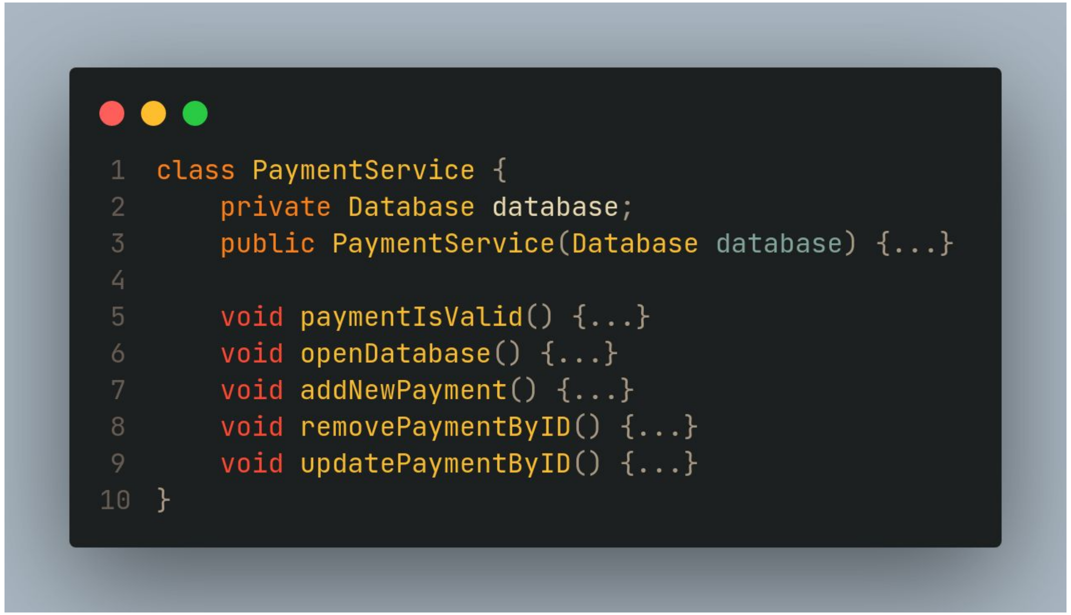
The problem is that this class depends on the database class and has a direct reference to that class as its property. Another problem that will arise is when we need a change to the MySQLDatabase class, where changes to that class can affect the above class, namely PaymentService.



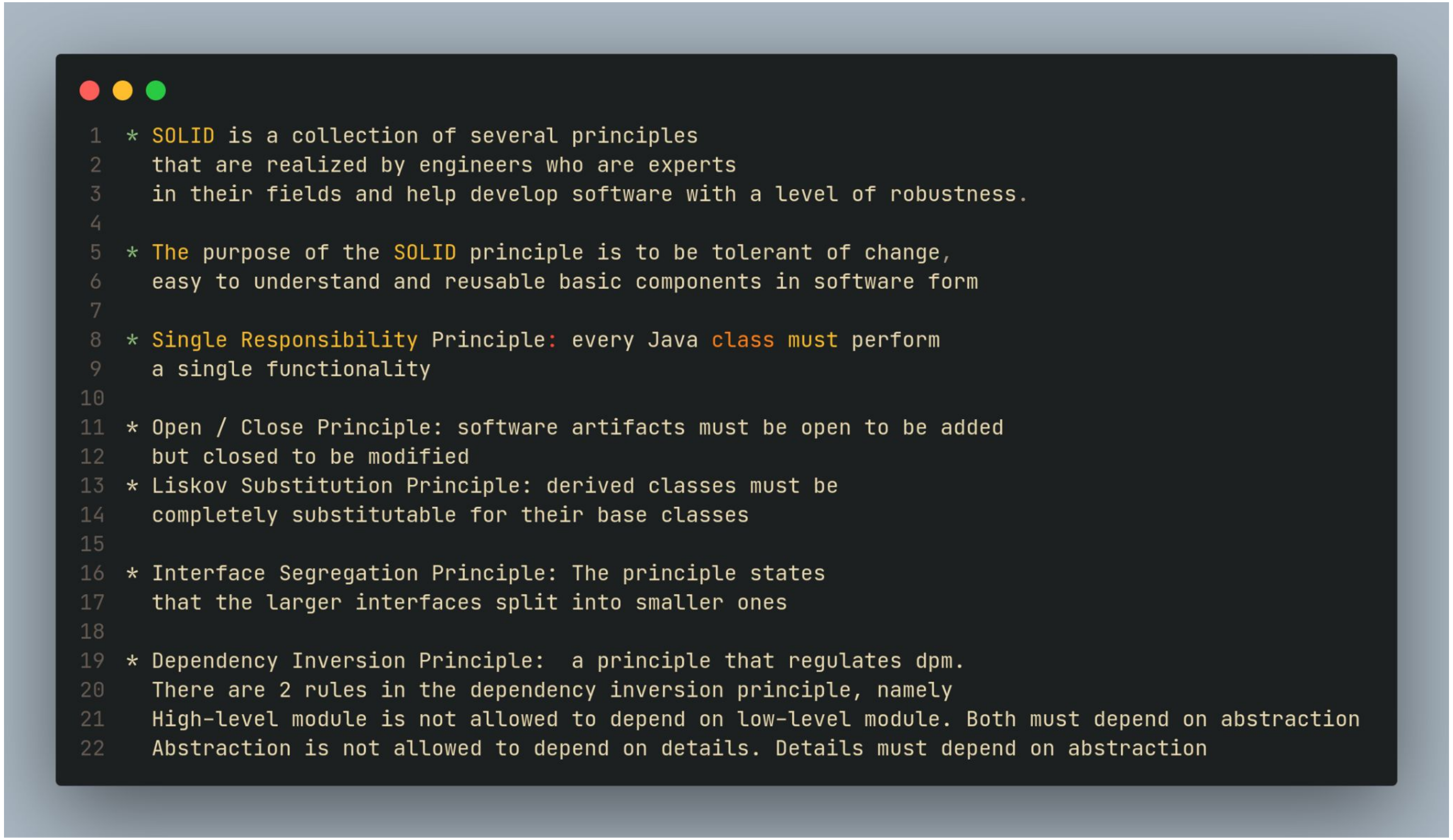
```
class PaymentService {

    private MySQLDatabase database = new MySQLDatabase();

    void paymentIsValid(){...}
    void openDatabase(){...}
    void addNewPayment(){...}
    void removePaymentByID(){...}
    void updatePaymentByID(){...}
}

class MySQLDatabase {
    void insert() {...}
    void update() {...}
    void delete() {...}
}
```



```java
abstract class Database {
    abstract void insert();
    abstract void update();
    abstract void delete();
}

class MySQLDatabase extends Database {
    @Override
    void insert() {}
    @Override
    void update() {}
    @Override
    void delete() {}
}

class MongoDatabase extends Database {
    Override
    void insert() {}
    @Override
    void update() {}
    @Override
    void delete() {}
}
```



```java
class PaymentService {
    private Database database;
    public PaymentService(Database database) {...}

    void paymentIsValid() {...}
    void openDatabase() {...}
    void addNewPayment() {...}
    void removePaymentByID() {...}
    void updatePaymentByID() {...}
}
```



```
* SOLID is a collection of several principles
  that are realized by engineers who are experts
  in their fields and help develop software with a level of robustness.

* The purpose of the SOLID principle is to be tolerant of change,
  easy to understand and reusable basic components in software form

* Single Responsibility Principle: every Java class must perform
  a single functionality

* Open / Close Principle: software artifacts must be open to be added
  but closed to be modified
* Liskov Substitution Principle: derived classes must be
  completely substitutable for their base classes

* Interface Segregation Principle: The principle states
  that the larger interfaces split into smaller ones

* Dependency Inversion Principle:  a principle that regulates dpm.
  There are 2 rules in the dependency inversion principle, namely
  High-level module is not allowed to depend on low-level module. Both must depend on abstraction
  Abstraction is not allowed to depend on details. Details must depend on abstraction
```

Any Questions?