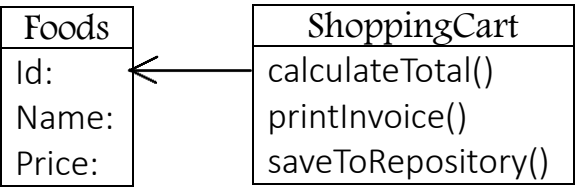
S	➢Single Responsibility Principle (SRP)
O	➢Open-Closed Principle (OCP)
L	➢Liskov Substitution Principle (LSP)
I	➢Interface Segregation Principle (ISP)
D	➢Dependency Inversion Principle (DIP)

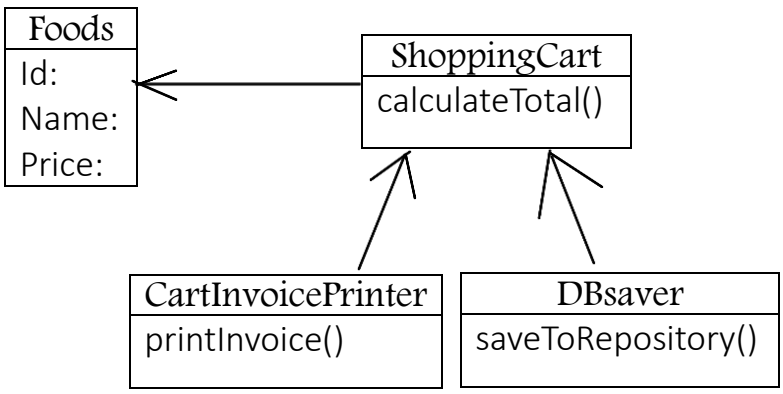
Single Responsibility Principle (SRP):

- A class should have ***only one reason to change***, meaning it should have a ***single responsibility***.
- This principle encourages creating classes that do ***one thing*** well.

Violating SRP



Following SRP




```java
import java.util.HashSet;

class Product {         // Violating SRP
    String name;
    Double price;
    Product (String name, Double price) {
        this.name = name;
        this.price = price;
    }
    @Override
    public String toString() {
        return name + " : " + price;
    }
}

class Cart {
    private final HashSet<Product> set = new HashSet<>();
    private Double total = 0D;
    public void addItem (Product product) {
        set.add(product);
        total += product.price;
    }
    public void removeItem (Product product) {
        set.remove(product);
        total -= product.price;
    }
    public HashSet<Product> getItems () {
        return set;
    }
    public Double calculateTotal () {
        return total;
    }
    public void printInvoice () {
        System.out.println("Printing");
    }
    public void saveToRepository () {
        System.out.println("Saving");
    }
}
```

```java
import java.util.HashSet;

class Product {         // Following SRP
    String name;
    Double price;
    Product (String name, Double price) {
        this.name = name;
        this.price = price;
    }
}
class Cart {
    private final HashSet<Product> set = new HashSet<>();
    private Double total = 0D;
    public void addItem (Product product) {
        set.add(product);
        total += product.price;
    }
    public void removeItem (Product product) {
        set.remove(product);
        total -= product.price;
    }
    public Double calculateTotal () {
        return total;
    }
}
class ManegingDB {
    private final Cart cart;
    ManegingDB (Cart cart){this.cart = cart;}
    public void saveToRepository () {
        System.out.println("Saving");
    }
}
class Invoice {
    private final Cart cart;
    Invoice (Cart cart){this.cart = cart;}
    public void printInvoice () {
        System.out.println("Printing");
    }
}
```

Open-Closed Principle (OCP):

- A class is ***open for extension*** but ***closed for modification***.
- This means that we should be able to ***add new functionality to a class*** or module ***without modifying*** its ***existing code***.
- This is achieved through ***abstraction***, ***inheritance***, and ***polymorphism***.

Benefits of OCP:

- Reduced Risk of Bugs.
- Increased Code Maintainability.
- Enhanced Code Flexibility.
- Improved Testability.

Let's suppose we need to ***write logic*** to ***save data*** to a ***database***. We have many options: we can use ***MySQL***, ***MongoDB***, ***PostgreSQL***, ***H2***, etc. The main difference between ***using OCP*** and ***not using OCP*** is shown below.

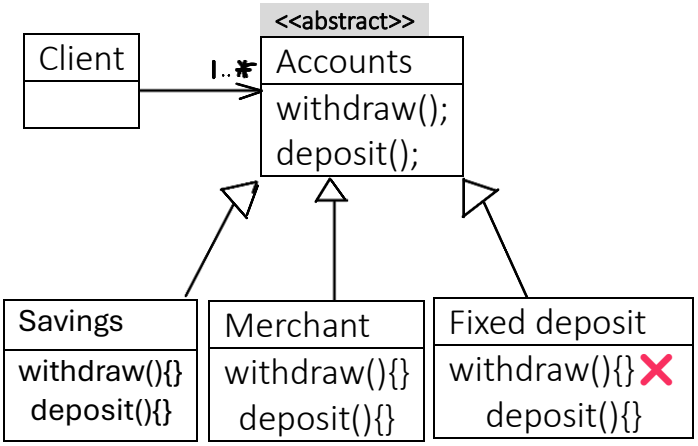
```
class Data {            // Violating OCP
   String info;
   Data(String info) {
      this.info = info;
   }
}
class DatabaseManagement {

   // Saving logic for MySQL
   public void saveMySQL(Data d) {   // tightly coupled
      System.out.println("save to MySQL");
   }

   // Saving logic for MongoDB
   void saveMongoDB(Data d){
      System.out.println("save to MongoDB");
   }
}

class Main {
   public static void main(String[] args) {
      Data d = new Data("My data");
      DatabaseManagement db = new DatabaseManagement();
      db.saveMySQL(d);
      db.saveMongoDB(d);
   }
}
```

```
class Data {            // Following OCP
   String info;
   Data(String info) {
      this.info = info;
   }
}

interface DatabaseManagement {   // parent interface
   void save(Data d);
}

// Logic to save in MySQL
class MySQL implements DatabaseManagement {  // loosely coupled
   @Override
   public void save(Data d) {
      System.out.println("save to MySQL");
   }
}

// Logic to save in MongoDB
class MongoDB implements DatabaseManagement {
   @Override
   public void save(Data d) {
      System.out.println("save to MongoDB");
   }
}

class Main {
   public static void main(String[] args) {
      Data d = new Data("My data");
      DatabaseManagement mongoDB = new MongoDB();
      DatabaseManagement mySql = new MySQL();
      mySql.save(d);
      mongoDB.save(d);
   }
}
```

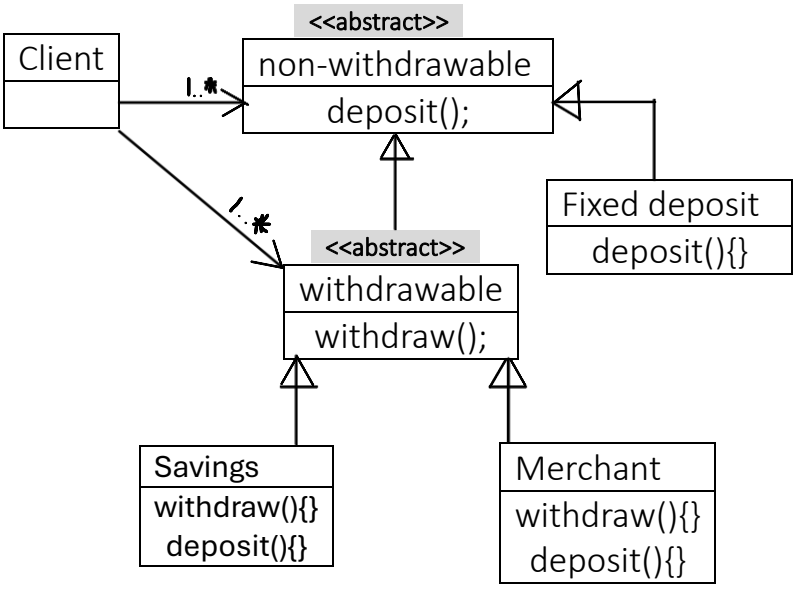
Liskov Substitution Principle (LSP):

- *Sub-classes* should be *substitutable* for their *base-classes*.
- In simpler terms, if a *subclass inherits* from a *superclass*, you should be *able to use instances* of the *subclass anywhere* you would use an *instance* of the *superclass without causing issues*.

Violating LSP



Following LSP




```java
class Clint {
    public static void main(String[] args) {
        ArrayList<Account> accounts = new
ArrayList<>(List.of(new SavingsAccount(),
MerchantAccount(), FixedDepositAccount()));
        for (Account w : accounts) {
            w.deposit(5000);
            try {
                w.withdraw(5000);
            }
            catch (RuntimeException e){
                System.out.println("Exception.");
            }
        }
    }
}
interface Account {
    void deposit (int balance);
    void withdraw (int balance);
}
class FixedDepositAccount implements Account {
    @Override
    public void deposit(int balance) {
        System.out.println("Deposit in FixedAccount");
    }
    @Override
    public void withdraw(int balance) {
        throw new RuntimeException("Exception:::");
    }
}
class SavingsAccount implements Account {
    @Override
    public void withdraw(int balance) {
       System.out.println("Withdraw from SavingsAccount");
    }
    @Override
    public void deposit(int balance) {
        System.out.println("Deposit in SavingsAccount");
    }
}
class MerchantAccount implements Account {
    @Override
    public void withdraw(int balance) {
      System.out.println("Withdraw from MerchantAccount");
    }
    @Override
    public void deposit(int balance) {
        System.out.println("Deposit in MerchantAccount");
    }
}
```

```java
class Clint {
    public static void main(String[] args) {
        ArrayList<WithdrawableAccount> withdrawableAccounts =
new ArrayList<>(List.of(new SavingsAccount(),new
MerchantAccount()));
        ArrayList<DepositableAccount> depositableAccounts =
new ArrayList<>(List.of(new FixedDepositAccount()));
        for (WithdrawableAccount w : withdrawableAccounts) {
            w.deposit(5000);
            w.withdraw(5000);
        }
        for (DepositableAccount w : depositableAccounts)
            w.deposit(5000);
    }
}
interface DepositableAccount {
    void deposit (int balance);
}
interface WithdrawableAccount extends DepositableAccount {
    void withdraw (int balance);
}
class FixedDepositAccount implements DepositableAccount {
    @Override
    public void deposit(int balance) {
        System.out.println("Deposit logic for FixedAccount");
    }
}
class SavingsAccount implements WithdrawableAccount {
    @Override
    public void withdraw(int balance) {
        System.out.println("Withdraw logic for
SavingsAccount");
    }
    @Override
    public void deposit(int balance) {
        System.out.println("Deposit in SavingsAccount");
    }
}
class MerchantAccount implements WithdrawableAccount {
    @Override
    public void withdraw(int balance) {
        System.out.println("Withdraw from MerchantAccount");
    }
    @Override
    public void deposit(int balance) {
        System.out.println("Deposit in MerchantAccount");
    }
}
```