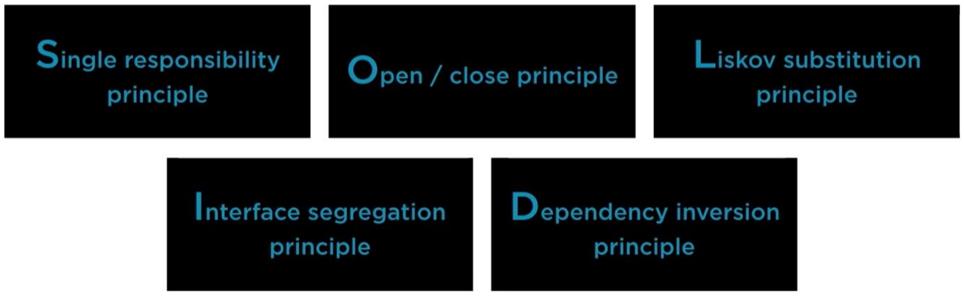
The Solid Principles



Single responsibility principle

Open / close principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

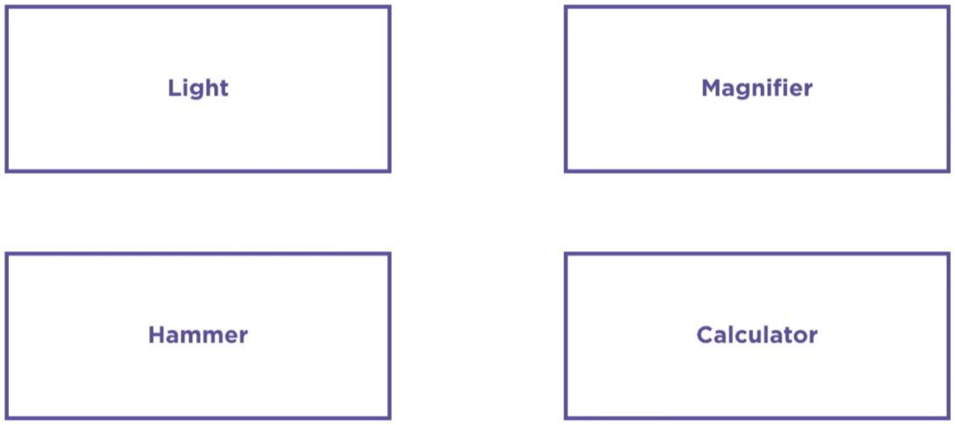
- Firstly introduced by Robert C. Martin, also known as Uncle Bob
- First Five Object Oriented Design Principles
- Make it easy to write extendable, maintainable and testable code

1. Single responsibility principle:

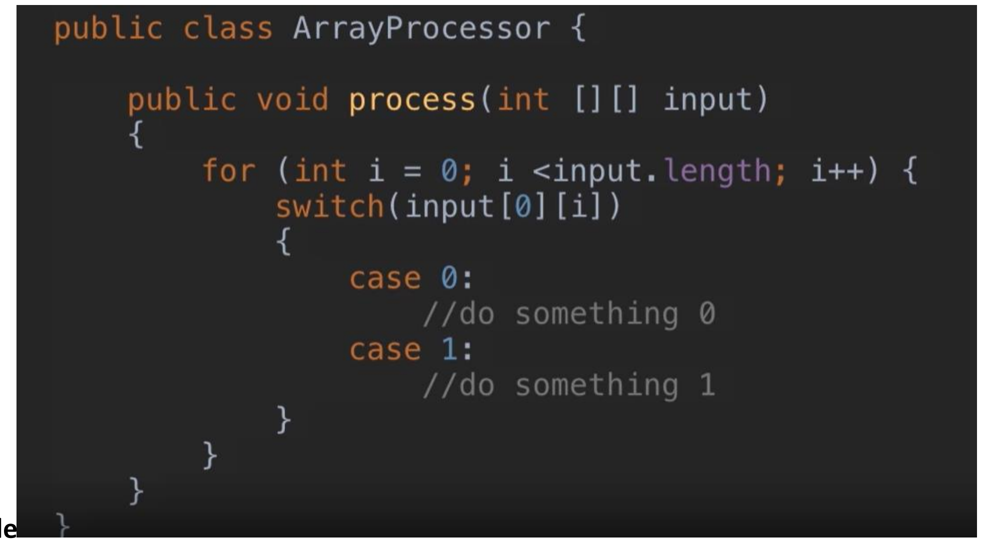
a. Every class should have only one responsibility
b. Responsibility=reason to change the class
c. Results in short components/classes

For example, if you want to write code for “Multi-purpose Knife”

Instead of single class containing all the functionality, your classes should look like this:



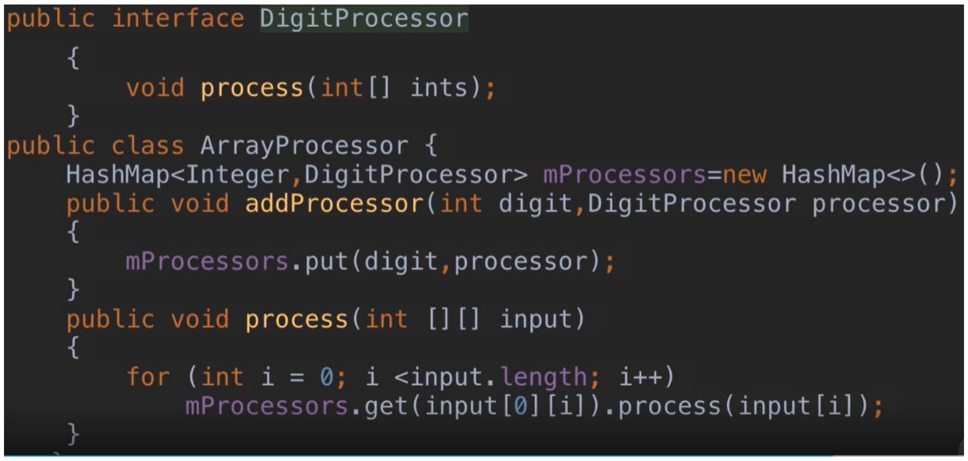
2. **Open /Close principle**
    a. Should be open for extension
    b. Close for modification
    c. Add new features using inheritance but should not changes existing class



```
public class ArrayProcessor {

    public void process(int [][] input)
    {
        for (int i = 0; i <input.length; i++) {
            switch(input[0][i])
            {
                case 0:
                    //do something 0
                case 1:
                    //do something 1
            }
        }
    }
}
```

Example

In this example, if we want to handle digit 3, 4 etc. later on, we would need to modify this class. Instead, it should do like this

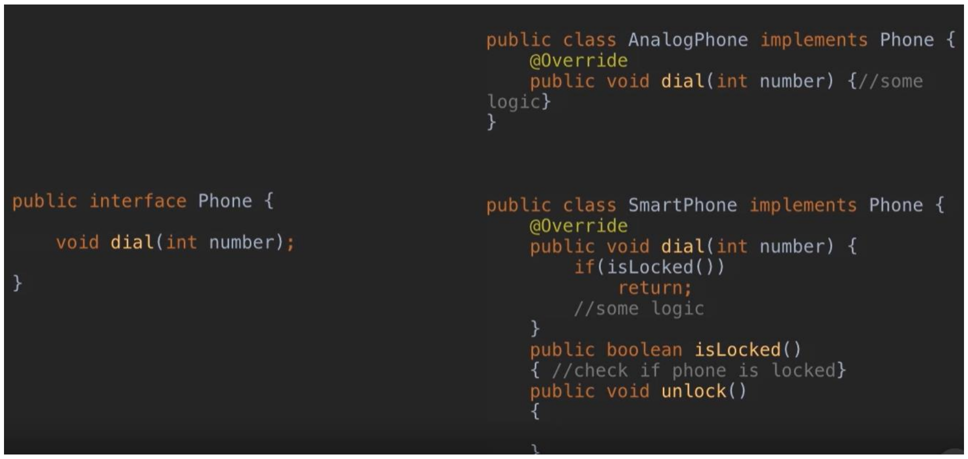


```
public interface DigitProcessor
    {
        void process(int[] ints);
    }
public class ArrayProcessor {
    HashMap<Integer,DigitProcessor> mProcessors=new HashMap<>();
    public void addProcessor(int digit,DigitProcessor processor)
    {
        mProcessors.put(digit,processor);
    }
    public void process(int [][] input)
    {
        for (int i = 0; i <input.length; i++)
            mProcessors.get(input[0][i]).process(input[i]);
    }
```

3. **Liskove Substitution principle:**
    a. A method that takes class "Y" as parameter
        i. Must be able to work with any subclass of "Y"

Example:

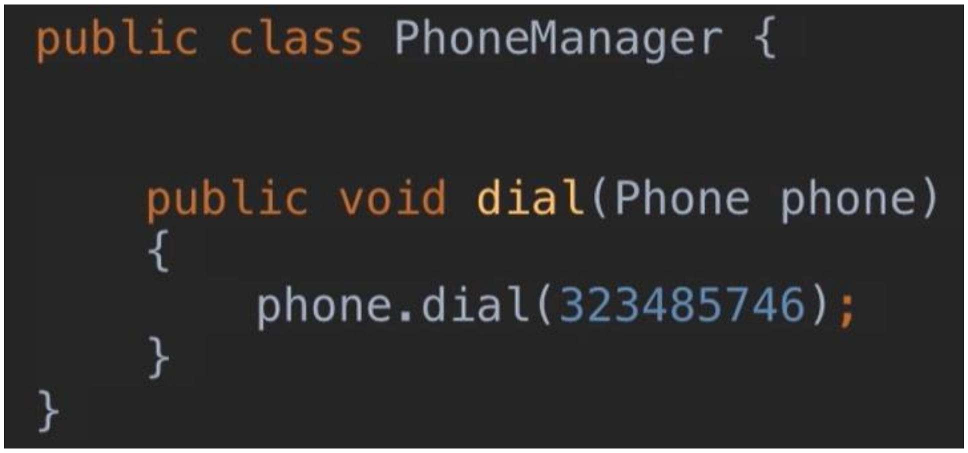
Take a look at Phone class and its children classes.



```
public interface Phone {

    void dial(int number);

}
```

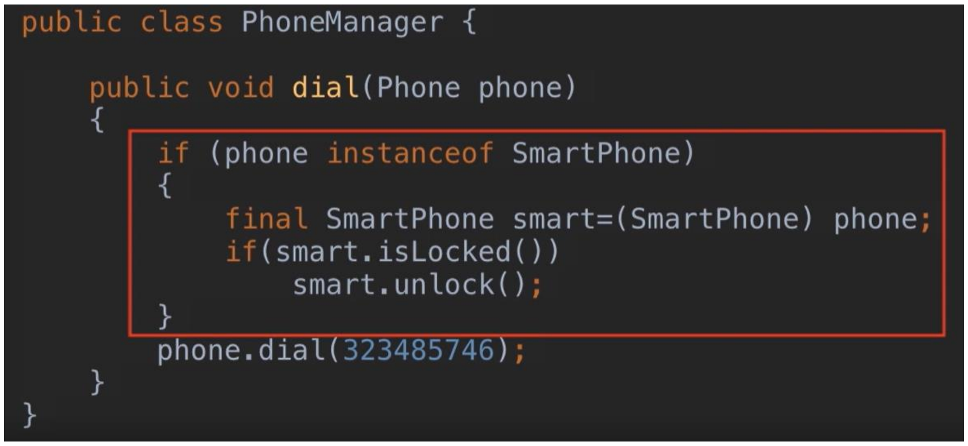
```
public class AnalogPhone implements Phone {
    @Override
    public void dial(int number) {//some
logic}
}


public class SmartPhone implements Phone {
    @Override
    public void dial(int number) {
        if(isLocked())
            return;
        //some logic
    }
    public boolean isLocked()
    { //check if phone is locked}
    public void unlock()
    {
```

If we want to dial a phone, we would do like this



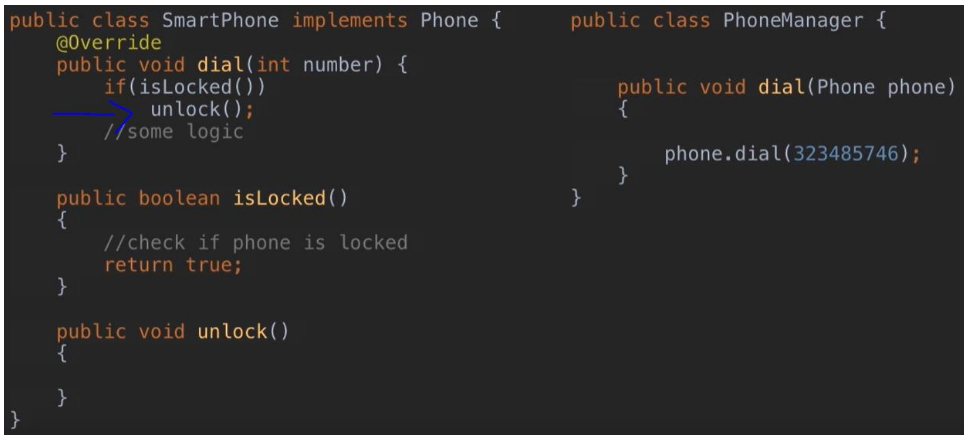
```java
public class PhoneManager {


    public void dial(Phone phone)
    {
        phone.dial(323485746);
    }
}
```

But if the phone is smartphone and it is locked, we cannot dial the number. Hence, the property is violated. We could handle it in phone Manager class as:



```java
public class PhoneManager {

    public void dial(Phone phone)
    {
        if (phone instanceof SmartPhone)
        {
            final SmartPhone smart=(SmartPhone) phone;
            if(smart.isLocked())
                smart.unlock();
        }
        phone.dial(323485746);
    }
}
```

But adding this, we violate our second property which states that if we add new phones in future, we would have to modify our phone Manager class. Instead, the right solution is to handle it in Smartphone class like this:

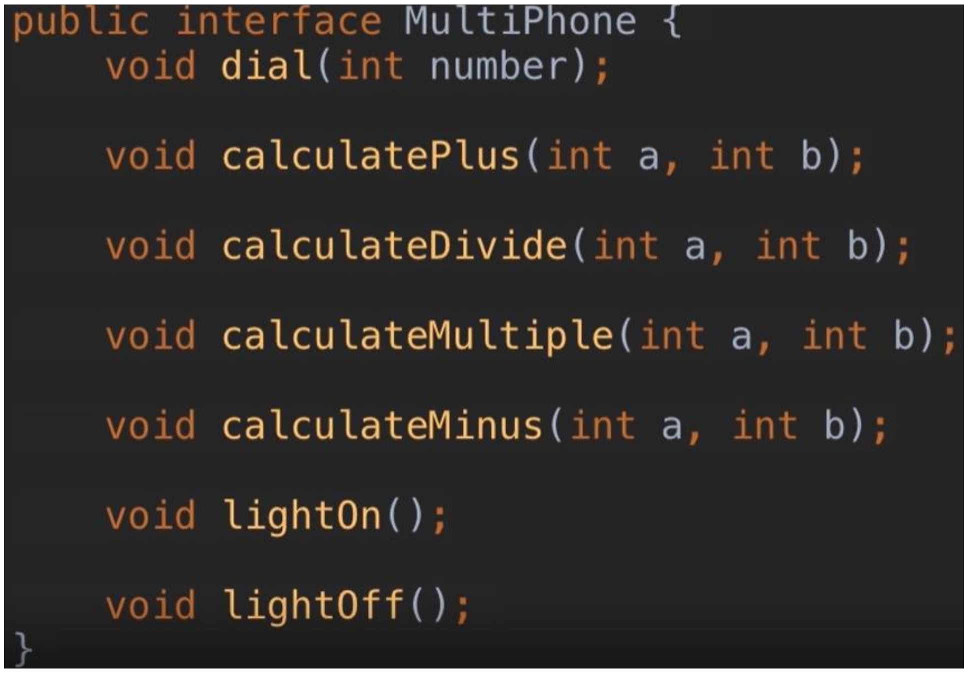


```java
public class SmartPhone implements Phone {
    @Override
    public void dial(int number) {
        if(isLocked())
            unlock();
        //some logic
    }

    public boolean isLocked()
    {
        //check if phone is locked
        return true;
    }

    public void unlock()
    {

    }
}
```

```java
public class PhoneManager {


    public void dial(Phone phone)
    {

        phone.dial(323485746);
    }
}
```

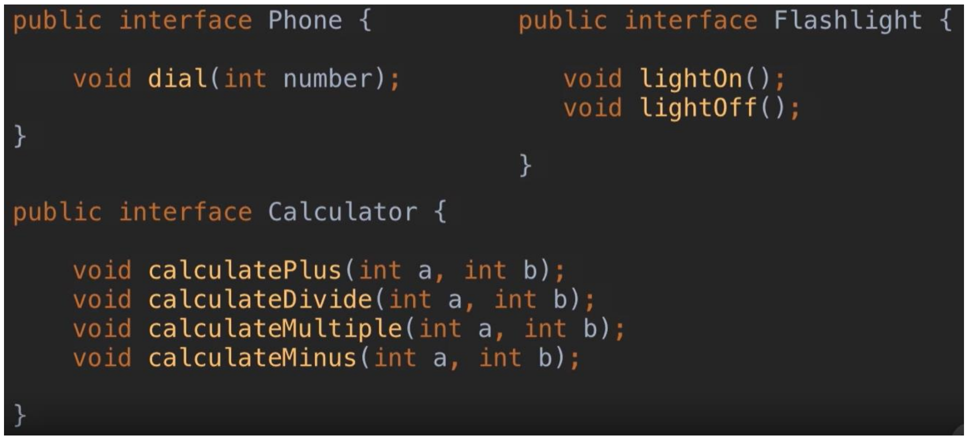
4. **Interface Segregation principle:**
    a. Complex interfaces make it harder to extend small parts of our system
    b. Complex interfaces should be split

Example:



```java
public interface MultiPhone {
    void dial(int number);

    void calculatePlus(int a, int b);

    void calculateDivide(int a, int b);

    void calculateMultiple(int a, int b);

    void calculateMinus(int a, int b);

    void lightOn();

    void lightOff();
}
```

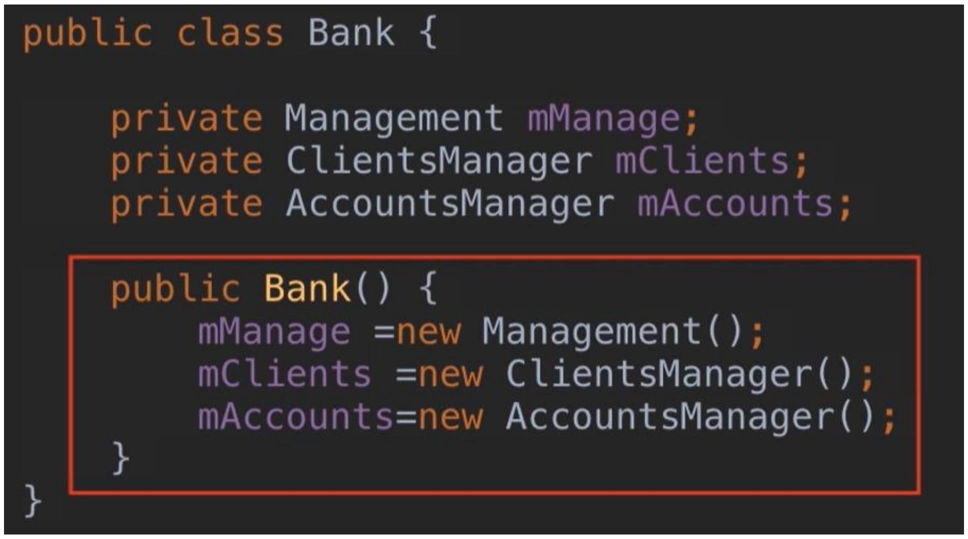
Here, we defined a very complex interface which implements a lot of different methods. If we want to implement calculator class, we would end up with empty definitions of dial, and lightOn, lightOff methods which is violating the principle. Instead, we can break the interface in three interfaces.



```java
public interface Phone {          public interface Flashlight {

    void dial(int number);            void lightOn();
                                      void lightOff();
}
                                  }

public interface Calculator {

    void calculatePlus(int a, int b);
    void calculateDivide(int a, int b);
    void calculateMultiple(int a, int b);
    void calculateMinus(int a, int b);

}
```

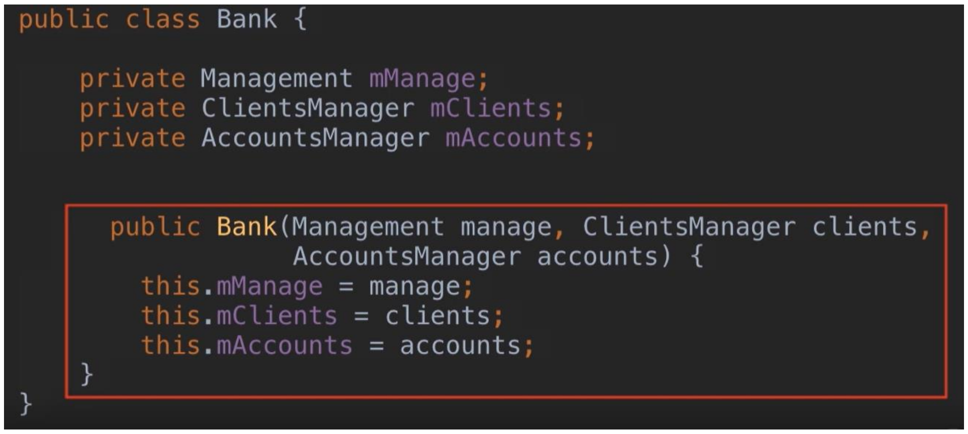
5. **Dependency Inversion principle:**
    a. No hidden dependency
    b. Let calling class create the dependency, instead of letting the class itself create the dependency

Example:



```java
public class Bank {

    private Management mManage;
    private ClientsManager mClients;
    private AccountsManager mAccounts;

    public Bank() {
        mManage =new Management();
        mClients =new ClientsManager();
        mAccounts=new AccountsManager();
    }
}
```

These are hidden dependencies because the calling class of Bank Constructor does not know the hidden dependencies inside the bank constructor. The right way to do is the following way:



```java
public class Bank {

    private Management mManage;
    private ClientsManager mClients;
    private AccountsManager mAccounts;


      public Bank(Management manage, ClientsManager clients,
                  AccountsManager accounts) {
        this.mManage = manage;
        this.mClients = clients;
        this.mAccounts = accounts;
    }
}
```

In this way, we are passing the dependency to constructor as parameter, hence calling class knows on what instances Bank is dependent on. This is also very important in testing as we know each component and can test independently.