SOLID Principles

S - Single Responsibility Principle
O - Open / Closed Principle
L - Liskov Substitution Principle
I - Interface Segmented Principle
D - Dependency Inversion Principle

Advantages of following these Principles:

Help us to write better code:
- Avoid Duplicate code ✓
- Easy to maintain ✓
- Easy to understand ✓
- Flexible software ✓
- Reduce Complexity

Single Responsibility Principle

A class should have only 1 reason to change

Marker Entity:

```
class Marker {
    String name;
    String color;
    int year;
    int price;

    public Marker(String name, String color, int year, int price) {
        this.name = name;
        this.color = color;
        this.year = year;
        this.price = price;
    }
}
```

let's say we have a class marker which just represents a marker with the mentioned properties

```java
class Invoice {

    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        int price = ((marker.price) * this.quantity);
        return price;
    }

    public void printInvoice() {
       //print the Invoice
    }

    public void saveToDB() {
        // Save the data into DB
    }
}
```

Here we have a class Invoice, which takes in Marker & quantity. It calculates ① price, ② prints Invoice, saves invoice to DB ③

It has 3 reasons to change. But it should have only 1 reason to change.

```java
class Invoice {

    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        int price = ((marker.price) * this.quantity);
        return price;
    }
}
```

It only has calculating the price as rest

```java
class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDB() {
        // Save into the DB
    }
}
```

It saves invoice to DB

```java
class InvoicePrinter {
    private Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void print() {
        //print the invoice
    }
}
```

It prints invoice

So to tackle this we have to divide this class into 3, and each of these 3 classes have a single responsibility

Open Closed Principle

Open for extension & closed for modification

Let's say we have an fully functional & tested class running live

```java
class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDB() {
        // Save into the DB
    }
}
```

Now we change it to add some functionality. The problem here is that it can break system

```java
class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

     public void saveToDB() {
         // Save Invoice into DB
     }

    public void saveToFile(String filename) {
        // Save Invoice in the File with the given name
    }
}
```

So here what we do is make an interface & implement it in different classes to not modify only one class

```java
interface InvoiceDao {

    public void save(Invoice invoice);
}

class DatabaseInvoiceDao implements InvoiceDao {

    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}

class FileInvoiceDao implements InvoiceDao {

    @Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```

It has necessary functions

We make different classes to implement it

*Now if we want to extend functionality we can make a new class

Liskov Substitution Principle

If B is a subtype of class A, then we should be able to replace object of A with B without breaking the behaviour of program.

Subclass should extend capability of parent class not narrow it down.

① Here lets say we have an interface bike.

```
interface Bike {

    void turnOnEngine();
    void accelerate();
}

class MotorCycle implements Bike {

    boolean isEngineOn;
    int speed;

    public void turnOnEngine() {
        //turn on the engine!
        isEngineOn = true;
    }

    public void accelerate() {
        //increase  the speed
        speed = speed + 10;
    }
}
```

Now motorcycle implements bike & its behaviour is fine.

② But here we changed the behaviour and now if we want to use Bicycle in motorcycle class/ code, we can't use it.

```
class Bicycle implements Bike {

    public void turnOnEngine() {
        throw new AssertionError( detailMessage: "there is no engine");
    }

    public void accelerate() {
        //do something
    }
}
```

Interface Segmented Principle

Interfaces should be such that client should not implement unnecessary functions they do not need.

```
interface RestaurantEmployee {
    void washDishes();
    void serveCustomers();
    void cookFood();
}

class waiter implements RestaurantEmployee {

    public void washDishes(){
        //not my job
    }

    public void serveCustomers() {
        //yes and here is my implemenation
        System.out.println("serving the customer");
    }

    public void cookFood(){
        // not my job
    }
}
```

3 functions

useful

No use for this class

Here as you see that the waiter class implements RestaurantEmployee. But but, it has to implement methods which are not useful to it.

functions specific to waiter class

```
interface WaiterInterface {
    void serveCustomers();
    void takeOrder();
}

class waiter implements WaiterInterface {

    public void serveCustomers() {
        System.out.println("serving the customer");
    }

    public void takeOrder(){
        System.out.println("taking orders");
    }
}
```

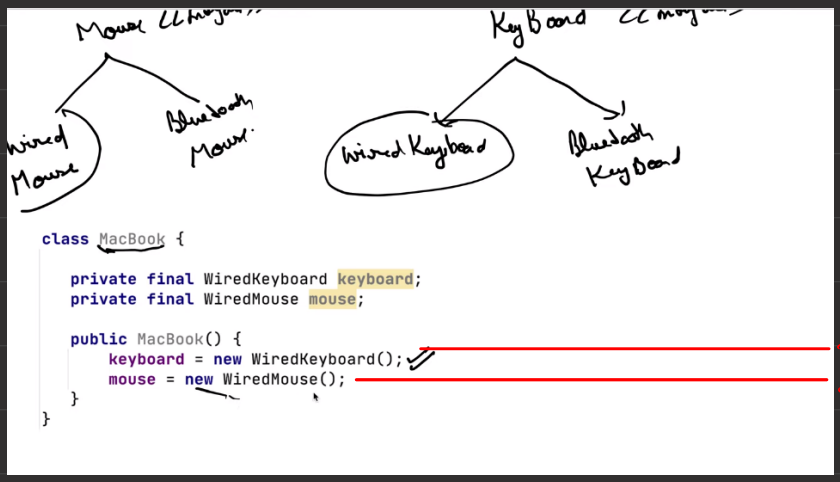
Similar

```
interface ChefInterface {
    void cookFood();
    void decideMenu();
}
```

So what we do is break interfaces / classes further down into smaller pieces to reduce redundancy

Dependency Inversion Principle

Class should depend on interfaces rather than concrete classes.




```java
class MacBook {

    private final WiredKeyboard keyboard;
    private final WiredMouse mouse;

    public MacBook() {
        keyboard = new WiredKeyboard();
        mouse = new WiredMouse();
    }
}
```

Here we have 2 interfaces Mouse & Keyboard, from which 2 classes each are implemented

} now here we are using concrete class thus limiting its functionality & ↑ duplicacy

```java
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

    public MacBook(Keyboard keyboard, Mouse mouse) {
        this.keyboard = keyboard;
        this.mouse = mouse;
    }
}
```

} Here we have corrected it by using interface specific system, because behaviour will be same, just our will differ.

Also this is much more flexible

wired keyboard, wireless mouse

wireless keyboard wired mouse