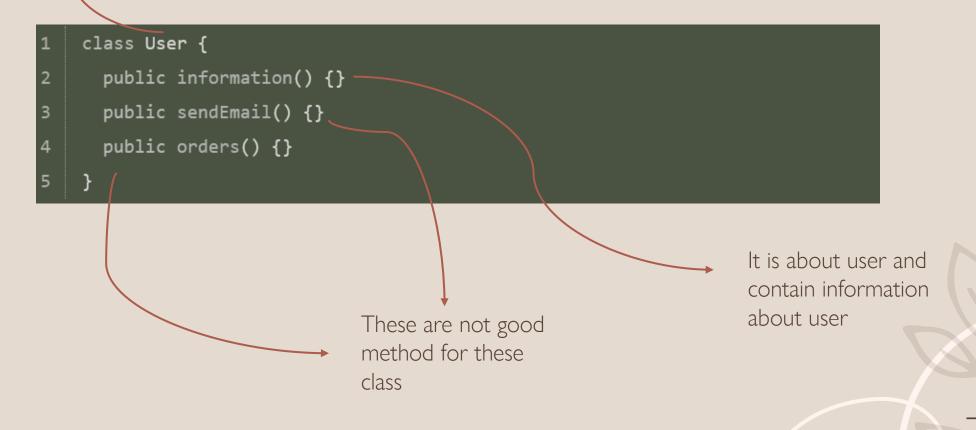




Single Responsibility Principle (S)

It should contain something related to user

A class/module should have one and only one reason to change, meaning that a class /module should only have one job.



solution

```
class User {
 public information() {}
class Email {
 public send(user: User) {}
class Order {
 public show(user: User) {}
```

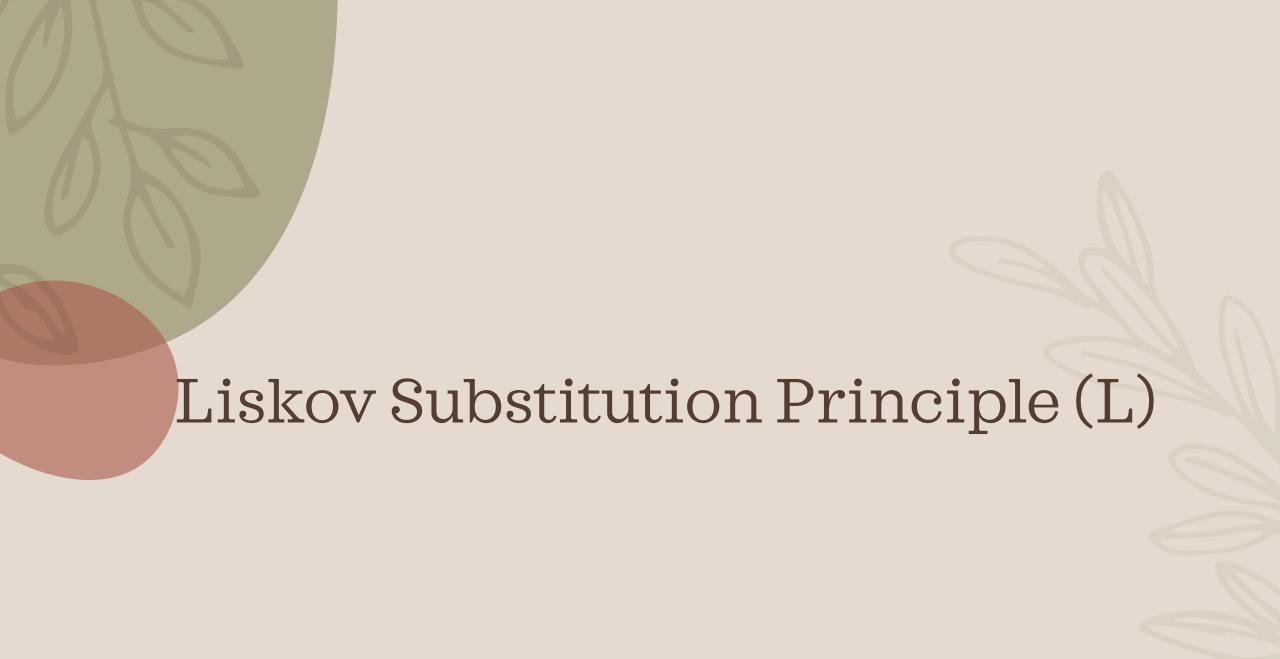
Open-Closed Principle (O) Objects or entities should be open for extension, but closed for modification.

```
What should we do for add new character
```

```
void attack(Character character) {
    if(character.name == 'Mario') {
        // logic for a mario attack
    } else if(character.name == 'Lugi') {
       // logic for a lugi attack
void jump(Character character) {
    if(character.name == 'Mario') {
       // logic for a mario jump
   } else if(character.name == 'Lugi') {
      // logic for a lugi jump
```

solution

```
abstract class GameCharacter {
    //.. other things
    abstract void attack() {}
    abstract void jump() {}
class Mario extends GameCharacter {
   void attack() {
     // attack pattern for mario
    void jump() {
    // logic for rendering Mario
class Lugi extends GameCharacter {
    void attack() {
     // attack pattern for mario
    void jump() {
    // logic for rendering Mario
```



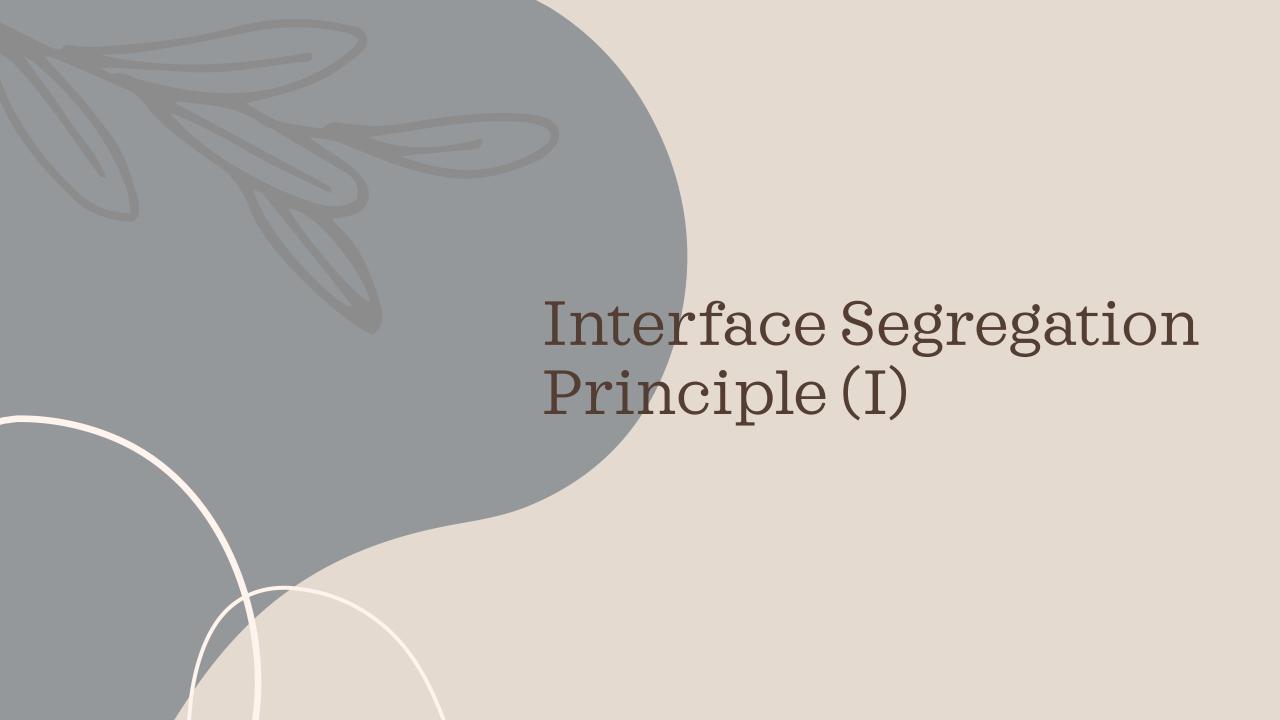
objects of a superclass should be replaceable by objects of a subclass without affecting the correctness of the program

```
class A { ... }
x = new A;
y = new A;
z = new A;
```

```
1 class B extends A { ... }
```

```
1  x = new A new B;
2
3  // ...
4
5  y = new A new B;
6
7  // ...
8
9  z = new A new B;
```

This code should works correctly



A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

```
1 interface Animal {
2    fly();
3    run();
4    eat();
5 }
```

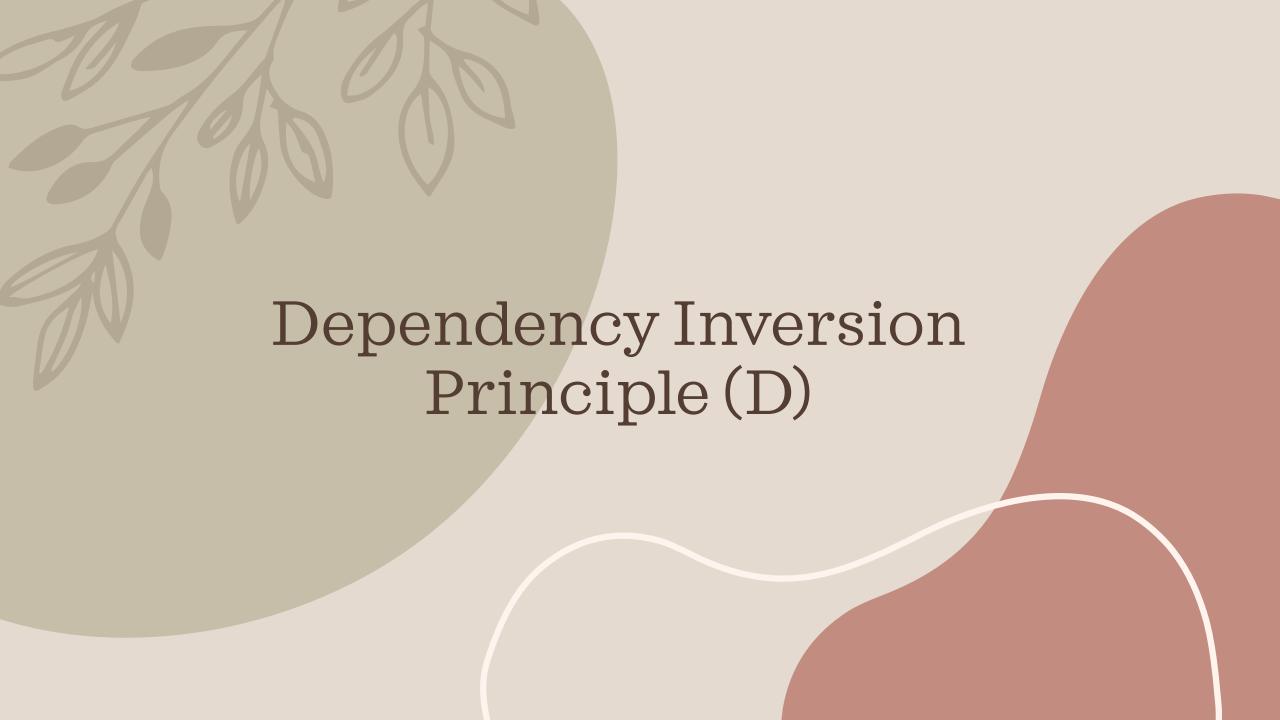
```
class Dolphin implements Animal {
        public fly() {
             return false;
         public run() {
         public eat() {
10
             // Eat
11
12
13
```

solution.

```
interface Animal {
   run();
   eat();

interface FlyableAnimal {
   fly();
}
```

```
class Dolphin implements Animal {
         public run() {
         public eat() {
            // Eat
10
     class Bird implements Animal, FlyableAnimal {
11
12
         public run() { /* ... */ }
13
        public eat() { /* ... */ }
14
        public fly() { /* ... */ }
```



Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

```
class MySql {
        public insert() {}
         public update() {}
         public delete() {}
    class Log {
        private database;
         constructor() {
10
             this.database = new MySql;
12
```

solution

```
interface Database {
       insert();
       update();
        delete();
    class MySql implements Database {
        public insert() {}
        public update() {}
        public delete() {}
    class FileSystem implements Database {
        public insert() {}
        public update() {}
        public delete() {}
10
11
12
    class MongoDB implements Database {
13
        public insert() {}
14
15
        public update() {}
        public delete() {}
16
17
```

solition

```
class Log {
  private db: Database;

public setDatabase(db: Database) {
  this.db = db;

}

public update() {
  this.db.update();
}
```

```
logger = new Log;
logger.setDatabase(new MongoDB);
// ...
logger.setDatabase(new FileSystem);
// ...
logger.setDatabase(new MySql);
logger.setDatabase(new MySql);
logger.update();
```

thank you