SOLID PRINCIPLES

agenda

Single Responsibility Principle (S)

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

It should contain something related to user

```
1  class User {
2    public information() {}
3    public sendEmail() {}
4    public orders() {}
5  }
```

It is about user and contain information about user

These are not good method for these class

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.

```
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
    }
}
```

What should we do for add new character

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
    }
}
```

Liskov Substitution Principle (L)

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;
```

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

```
x = new A new B;

// ...

y = new A new B;

// ...

z = new A new B;
```

This code should works correctly

Interface Segregation Principle (I)

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.

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

```
class Dolphin implements Animal {
    public fly() {
        return false;
    }

    public run() {
        // Run
    }

    public eat() {
        // Eat
    }
}
```

solution

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

interface FlyableAnimal {
    fly();
}
```

```
class Dolphin implements Animal {
    public run() {
        // Run
    }

    public eat() {
        // Eat
    }
}

class Bird implements Animal, FlyableAnimal {
    public run() { /* ... */ }
    public eat() { /* ... */ }
    public fly() { /* ... */ }
}
```

Dependency Inversion Principle (D)

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() {
        this.database = new MySql;
    }
}
```

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() {}
}

class MongoDB implements Database {
    public insert() {}
    public update() {}
    public delete() {}
}
```

solution

```
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.update();
```

thank you