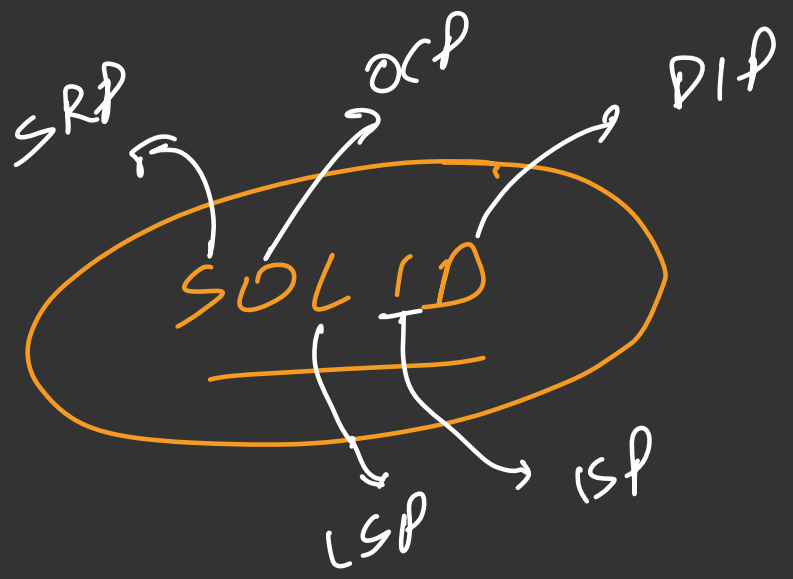

SRP
OCP
DIP
SOLID
LSP
ISP

Aim ↓

1. Abstraction.
2. Varying behaviour → Encapsulate.
3. Loose coupling → modularity

Aim ↑

→ $Class_1 \leftrightarrow Class_2$

→ impact should be min.

Modularity :

Inheritance Interface.

More Coupling ?

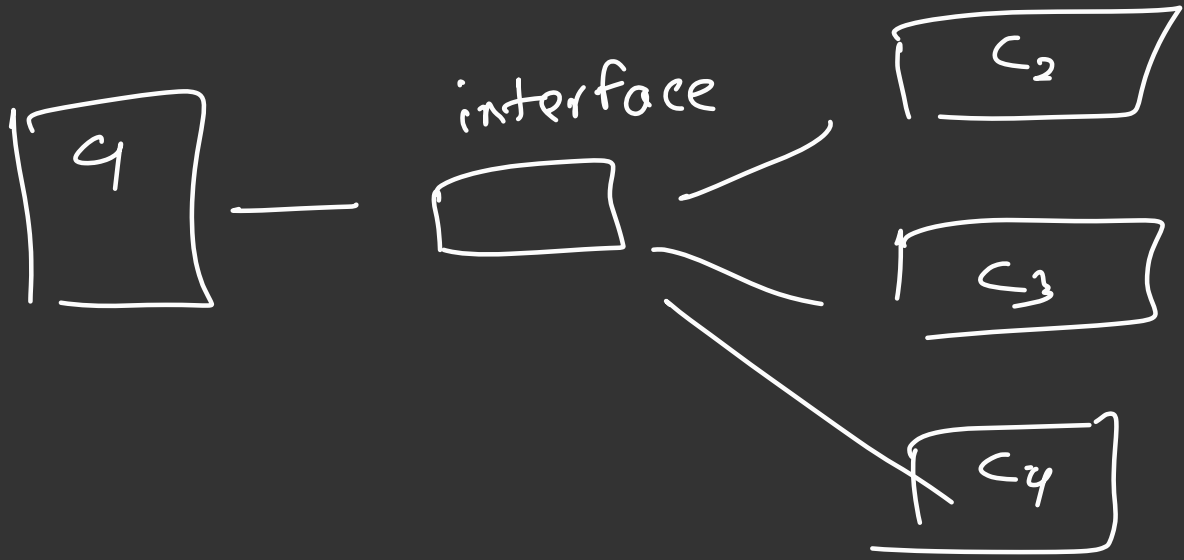
Ans: Inheritance. Why ?

↓

P - C relation (is-a)

Why Interface is less coupled?

- Classes are hidden behind interface or abstraction, interface



Liskov Substitution Principle (LSP)

- Subtypes must be substitutable for their base types without altering the correctness of the program.

- A derived class should be able to replace its base class without breaking functionality.

- Is the subclass truly a "is-a" relationship?

LSP:

Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should also be true for objects y of type S where S is a subtype of T

In simple words,

Objects of a superclass should be able to be replaced with objects of a subclass without affecting the program.

Object of subclass should be able to access the all the methods and properties of the superclass.

LSP: There should be true is-a relationship

- Inheritance
- Child should replace Parent

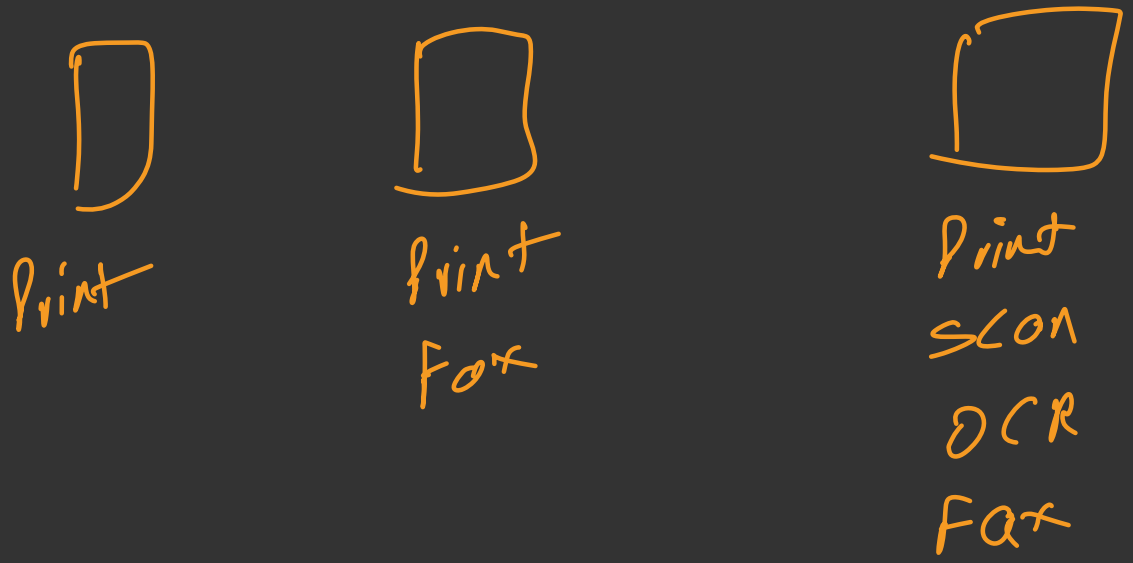
```
class MediaPlayer {
    playAudio(): void {
      console.log("Playing audio");
    }

    playVideo(): void {
      console.log("Playing video");
    }
  }

  class AudioPlayer extends MediaPlayer {
    playVideo(): void {
      throw new Error("Audio player can't play video");
    }
  }
```

```
class Rectangle {
    width: number = 0;
    height: number = 0;

    setWidth(width: number) { this.width = width; }
    setHeight(height: number) { this.height = height; }

    getArea() { return this.width * this.height; }
}

class Square extends Rectangle {
  setWidth(width: number) {
    this.width = width;
    this.height = width;
  }
  setHeight(height: number) {
    this.width = height;
    this.height = height;
  }
}
```

```
function printArea(rect: Rectangle) {
  rect.setWidth(5);
  rect.setHeight(10);
  console.log(rect.getArea());
}

printArea(new Rectangle());
printArea(new Square());
```

Interface Segregation Principle (ISP)

Do not force a class to implement interfaces it does not use.



Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules.
- Both should depend on abstractions (e.g., interfaces).

OR

Abstractions should not depend on details; details should depend on abstractions.

Types of Inheritance in typescript

1. Single
2. Multi-level
3. Hierarchical

Not supported: Multiple Inheritance, hybrid Inheritance

- Diamond Problem

Types of interface
Single Interface
i1
i2

Class implementing multiple interfaces
class ABC implements i1, i2{}

Multiple Interface Inheritance
i3 extends i2, i1
class PQRS implements i3{}

Best Practices:

Prefer Abstraction over Inheritance

Encapsulate what varies