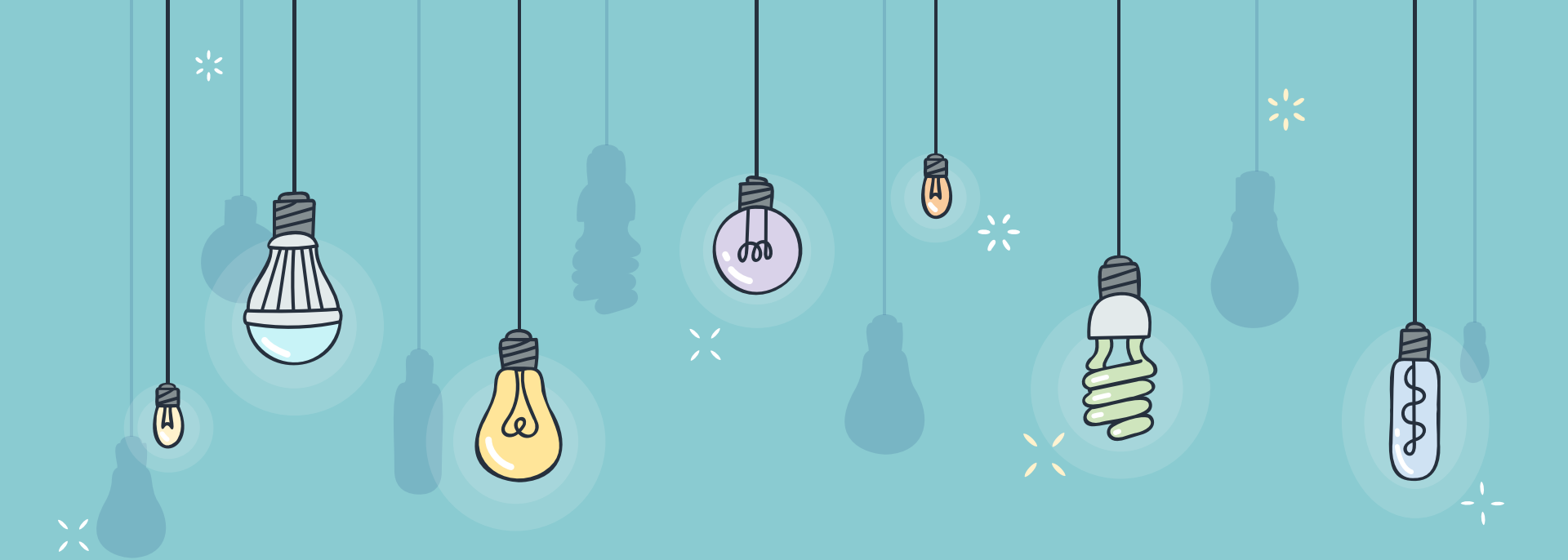
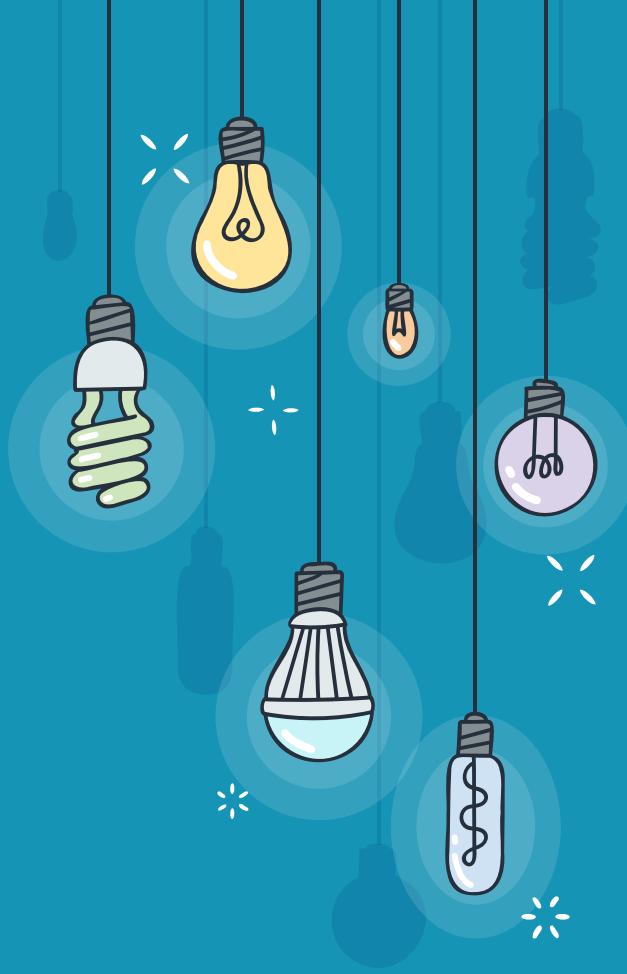


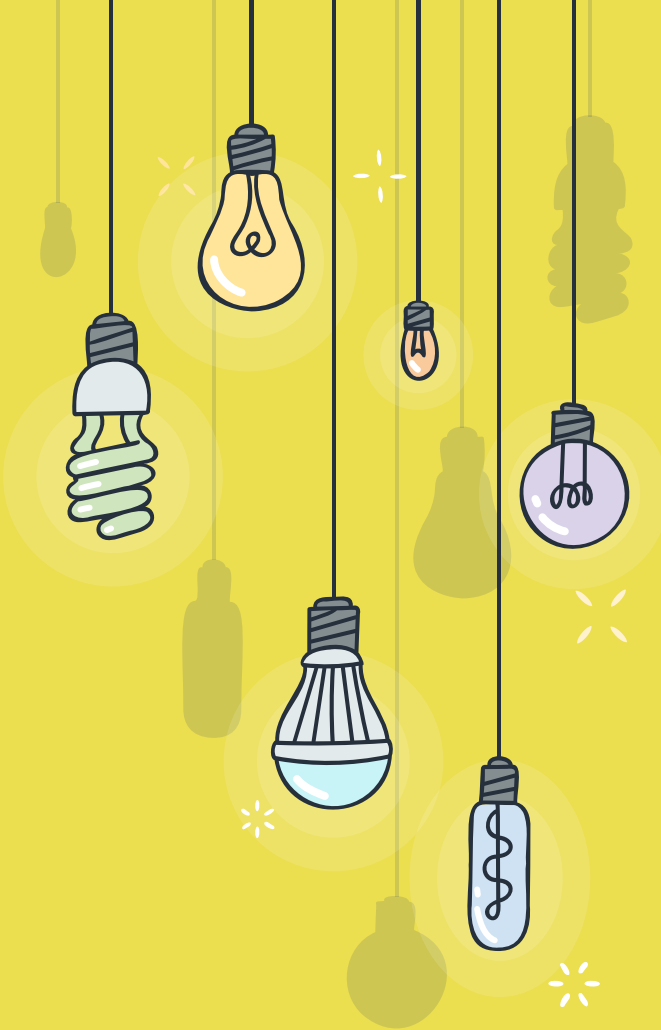
객체지향 설계 원칙

김 지원

1. SOLID 개념
2. SRP 개념
3. OCP 개념
4. LSP 개념
5. ISP 개념
6. DIP 개념



1 SOLID의 개념



객체지향 설계 원칙

- 객체지향의 설계 원칙
  - 변경이나 확장에 유연한 시스템을 설계하기 위해 지켜져야 할 원칙
  - 5가지 원칙 : 원칙의 앞 글자를 따서 SOLID원칙이라 부름
    - SRP
    - OCP
    - LSP
    - ISP
    - DIP

SOLID

객체지향의 설계 원칙의 종류

종류	내용
단일 책임 원칙(SRP)	객체는 단 하나의 책임만 가져야 한다는 원칙
개방-폐쇄 원칙(OCP)	기존의 코드를 변경하지 않고 기능을 추가할 수 있도록 설계해야 한다는 원칙
리스코프 치환 원칙(LSP)	자식 클래스는 최소한 부모 클래스의 기능은 수행할 수 있어야 한다는 원칙
인터페이스 분리 원칙(ISP)	자신이 사용하지 않는 인터페이스와 의존 관계를 맺거나 영향을 받지 않아야 한다는 원칙
의존 역전 원칙(DIP)	의존 관계 성립 시 추상성이 높은 클래스와 의존 관계를 맺어야 한다는 원칙

2 SRP의 개념

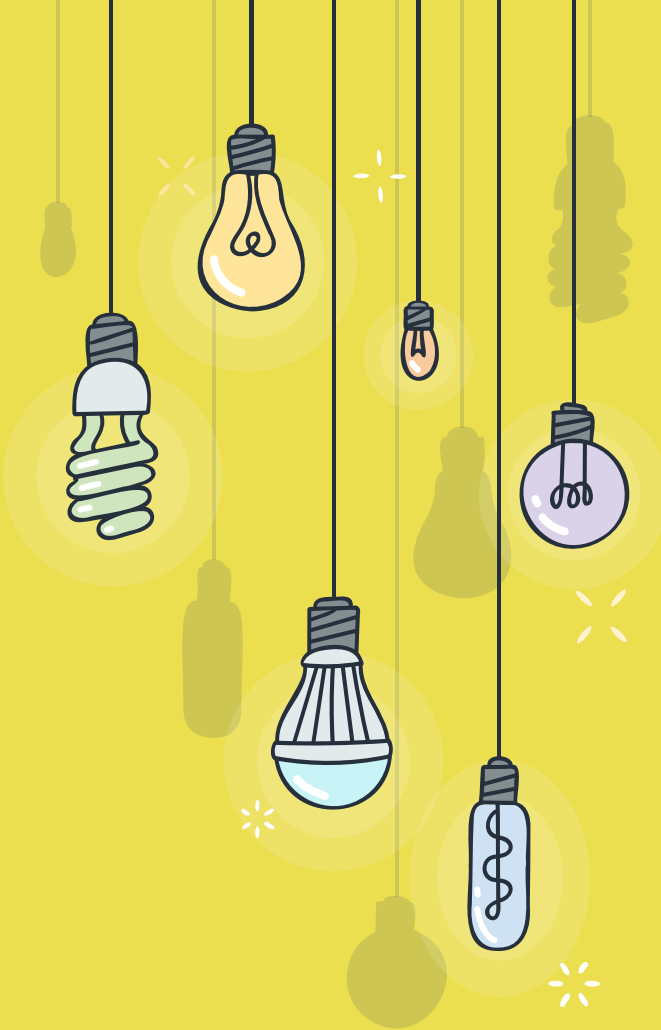
SRP

- 단일 책임의 원칙(SRP, Single Responsibility Principle)
  - 객체는 단 하나의 책임만을 가져야 한다

  - 책임은 변경이유
    - 책임이 많다는 것은 변경될 여지가 많다는 의미이다
    - 책임을 많이 질수록 클래스 내부에서 서로 다른 역할을 수행하는 코드끼리 강하게 결합될 가능성이 높아진다.
    - 책임 분리 필요

  - 장점
    - 변경이 필요할 때 수정할 대상이 명확
    - 시스템이 커지면서 서로 많은 의존성을 갖게 되는 상황에서 서로 영향을 주지 않도록 해 줄 수 있다

3 OCP의 개념



OCP

- 개방 폐쇄 원칙(OCP, Open-Closed Principle)
  - 확장에 대해 열려 있고 수정에 대해서는 닫혀 있어야 한다는 원칙
    - 확장에 대해 열려있다 : 요구사항이 변경될 때 새로운 동작을 추가하여 애플리케이션의 기능을 확장할 수 있다.
    - 수정에 대해 닫혀있다 : 기존의 코드를 수정하지 않고 애플리케이션의 동작을 추가하거나 변경할 수 있다.
  - 해결방법 : 추상화에 의존

OCP

SoundPlayer 클래스는 음악을 재생해주는 클래스로 wav파일을 재생할 수 있다. 그러나 다른 포맷의 파일, 예를 들어 Mp3 파일을 재생하도록 요구사항이 변경 되었다. 개발-폐쇄의 원칙에 맞게 아래 코드 수정하기

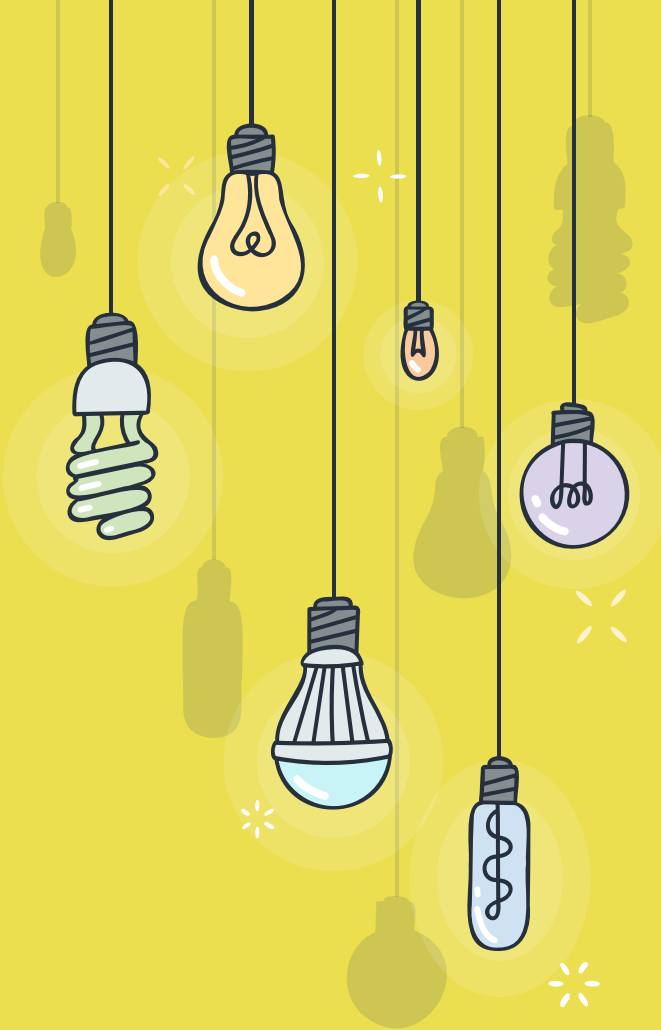
```
class SoundPlayer {
    void play() {
        System.out.println("play wav");
    }
}

public class OcpRun {
    public static void main(String[] args) {
        SoundPlayer sp = new SoundPlayer();
        sp.play();
    }
}
```

OCP

```java
public interface PlayInterface {
    void play();
}

public class Wav implements PlayInterface{
    @Override
    public void play() {
        System.out.println("play wav");
    }
}

public class Mp3 implements PlayInterface{
    @Override
    public void play() {
        System.out.println("play mp3");
    }
}
```

```java
public class SoundPlayer {
    private PlayInterface playInter;
}

void setPlayInter(PlayInterface playInter) {
    this.playInter = playInter;
    void play() {
        playInter.play();
    }
}

public class T01_OCP_answer {
    public static void main(String[] args) {
        SoundPlayer sp = new SoundPlayer();
        sp.setPlayInter(new Wav());
        System.out.print("wav파일 재생 : ");
        sp.play();
        sp.setPlayInter(new Mp3());
        System.out.print("mp3파일 재생 : ");
        sp.play();
    }
}
```

4 LSP의 개념



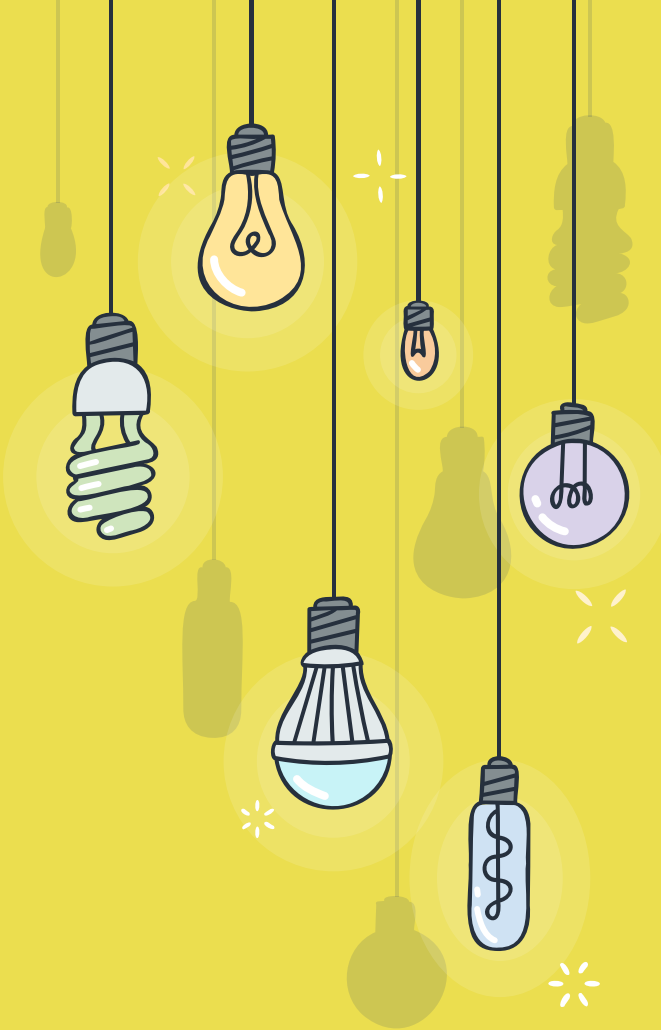
LSP

- 리스코프 치환 원칙 (LSP, Liskov Substitution Principle)
  - 자식 클래스는 최소한 부모 클래스의 기능은 수행할 수 있어야 한다는 원칙
    - 즉, 상위 타입의 객체를 하위 타입의 객체로 치환해도 동작에 문제가 없어야 한다
  - 가장 단순한 해결 방법 : 재정의를 하지 않는 것

LSP

```
class Cat {
    void speak() {
        System.out.println("야옹");
    }
}

class WhiteCat extends Cat {
    String color = "white";
}

class BlackCat extends Cat {
    String color = "black";
}

public class LiskovRun {
    public static void main(String[] args) {
        Cat cat = new WhiteCat();
        WhiteCat whiteCat = (WhiteCat)cat;
        Cat cat2 = new BlackCat();
        BlackCat blackCat = (BlackCat)cat2;
        cat.speak();
        whiteCat.speak();
        blackCat.speak();
    }
}
```

5 ISP의 개념



LSP

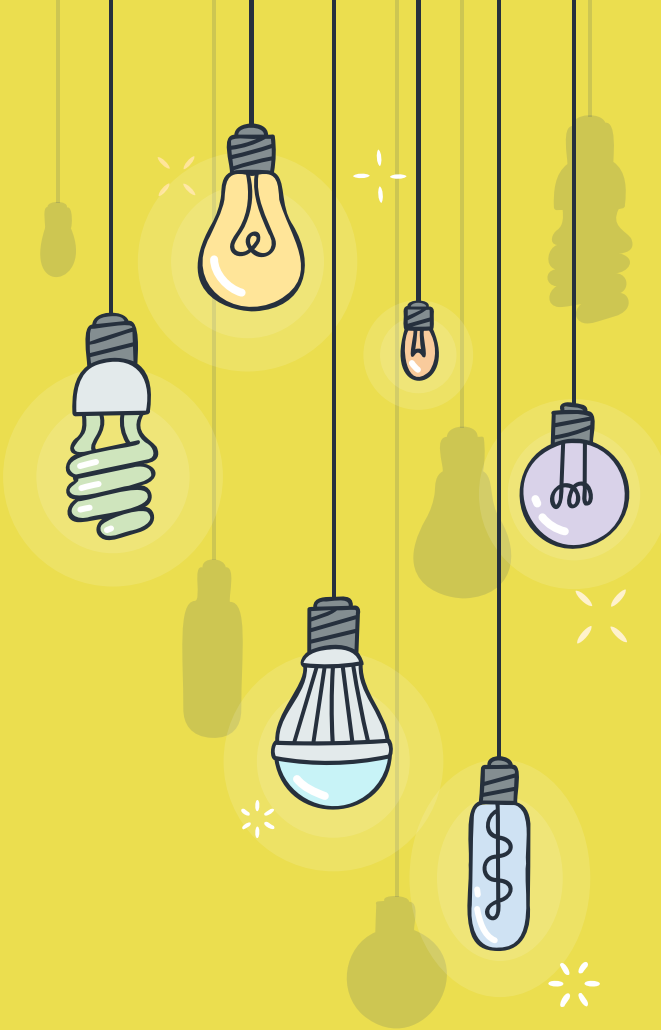
- 인터페이스 분리 원칙 (ISP, Interface segregation principle)
  - 자신이 사용하지 않는 인터페이스와 의존 관계를 맺거나 영향을 받지 않아야 한다는 원칙
    - 즉, 클라이언트의 목적과 용도에 적합한 인터페이스 만을 제공하는 것

LSP EX

```java
public interface Car {
    void drive();
    void turnLeft();
    void turnRight();
    void stop();
    void water();
    void patientTransport();
}
```

```java
class Sedan implements Car{
    public void drive() {}
    public void turnLeft() {}
    public void turnRight () {}
    public void stop() {}
    // public void water() {}
    // public void patientTransport() {}
}
class FireEngine implements Car{
    public void drive() {}
    public void turnLeft() {}
    public void turnRight () {}
    public void stop() {}
    public void water() {}
    // public void patientTransport() {}
}
class Ambulance implements Car{
    public void drive() {}
    public void turnLeft() {}
    public void turnRight () {}
    public void stop() {}
    public void patientTransport() {}
    // public void water() {}
}
```

6 DIP의 개념



DIP

- 의존 역전 원칙 (DIP, Dependency Inversion Principle)
  - 고수준 모듈은 저수준 모듈의 구현에 의존해서는 안 된다
    - 즉, interface(or 추상클래스)에 의존 해야지 구현 클래스(변경이 빈번하게 이루어짐)에 의존해서는 안된다
    - 고수준 모듈 : 추상화된 모듈 (interface, 추상 클래스)
    - 저수준 모듈 : interface(추상클래스)의 기능을 구현한 모듈

DIP EX

```
class Dog {
    @Override
    public String toString() {
        return "강아지";
    }
}
class Cat {
    @Override
    public String toString() {
        return " 고양이";
    }
}
```

```
class Pet {
    Dog dog;
    Cat cat;
    public void setDog(Dog dog) {
        this.dog = dog;
    }
    public void setCat(Cat cat) {
        this.cat = cat;
    }
}
```

- Pet은 Dog와 Cat에 의존하고 있다.
- 여기에 애완동물 햄스터를 더 키운다면?

Thanks!

- Any questions?