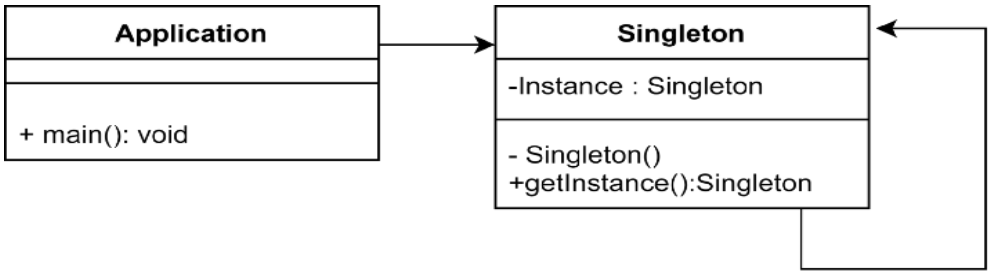
13. 패턴

패턴(디자인 패턴)이란

프로그램 개발 시 자주 나타나는 구조나 방식, 패턴을 구체적이고 체계적으로 나누어 정리한 것이다. 여러 다른 프로그램을 작성시 비슷한 로직이 나타나기 때문에 패턴화 해 효율적으로 다른 여러 작업을 하는데 용이하게 해주는 작업 프레임으로 정리할 수 있다.

1) 싱글톤(singleton) 패턴

보통 new명령어를 통해 새로운 인스턴트를 생성하여 사용하는데 일반적인 클래스의 인스턴트 생성은 제한이 없다. 그러나 필요에 따라 하나의 클래스의 하나의 인스턴트 만을 생성하여 공유해서 사용이 요구될 때 사용하는 패턴이다. 싱글톤 패턴으로 생성된 객체는 생성자가 여러 차례 호출되더라도 실제로 생성되는 객체는 하나이고 이 객체에 접근할 수 있는 전역적인 접촉점을 제공한다.



Ex)예제

```java
public class SingletonClass {
    private int i = 10;//테스트를 위한 변수 선언

    private static SingletonClass INSTANCE;
    //자신을 자료형으로 하는 private 변수 선언 //일반적으로 INSTANCE를 대문자로 선언함
    //자신의 인스턴트객체를 대입할 변수

    //private static SingletonClass INSTANCE = new SingletonClass();
    // 위 같은 식으로도 선언가능. 이렇게 생성하면 힙메모리에 생성자가 바로 할당되기 때문에
    // 사용하지 않는 순간에 데이터를 차지할수 있음;

    public static SingletonClass getInstance() {
        if (INSTANCE == null) { //INSTANCE값이 초기화 되지않았으면 null값이다 이는
            INSTANCE = new SingletonClass(); //아직 한번도 객체가 생성되지않았다는
                                             //뜻이므로 객체를 생성해줌
        }
        return INSTANCE; // null값이 아니면 이미 생성되있다는 뜻이므로 기존 INSTANCE를 리턴
    }

    private SingletonClass() {
    }//기본 생성자를 private로 선언 //외부에서 new 클래스명으로 인스턴스 생성 막기위해

    public int getI() {
        return i;
    }

    public void setI(int i) {
        this.i = i;
    }

```

Ex)

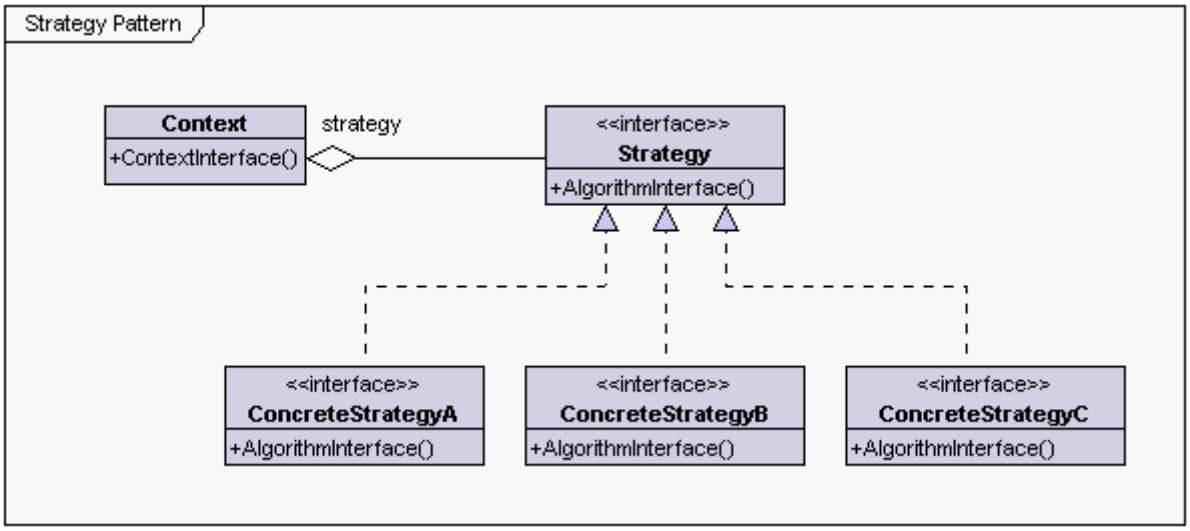
```
public class TestMain {

    public static void main(String[] args) {
        SingletonClass obj1 = SingletonClass.getInstanec();
        SingletonClass obj2 = SingletonClass.getInstanec();
        obj1.setI(99);
        System.out.println("obj1의 i : "+ obj1.getI());
        System.out.println("obj2의 i : "+ obj2.getI());
        obj2.setI(10);
        System.out.println("obj1의 i : "+ obj1.getI());
        System.out.println("obj2의 i : "+ obj2.getI());

      //실행결과 obj1의 i : 99       //다른 두개의 객체가 생성되었지만 같은 값을 공유하고
      //        obj2의 i : 99
      //        obj1의 i : 10        //다른 객체에서 변수를 바꿔도 공유된다
      //        obj2의 i : 10
    }
}
```

2) 전략 패턴 (Strategy Pattern)

여러 로직의 알고리즘 군을 정의하고 추상적인 접근점(인터페이스)을 만들고 각각의 기능을 부품처럼 캡슐화하여 사용자가 자신의 필요에 맞게 전략적(Strategy)으로 취사하여 사용할 수 있도록 하는 패턴이다. 전략패턴은 새로운 요구사항이나 전략패턴으로 생성된 객체의 추가변경사항이 있을 때 수정, 보수가 용이하다는 장점이 있다.



- 공통되는 로직을 하나의 인터페이스로 캡슐화

Ex)예제

```java
package strategy2.interfaces;
//엔진이라는 공통된 로직을 하나의 인터페이스로 묶어 접점을 만듬
public interface IEngine {
    public void engine();
}

public class EngineHigh implements IEngine {

    @Override
    public void engine() {
        System.out.println("고급 엔진 입니다.");
    }
}

public class EngineMid implements IEngine {

    @Override
    public void engine() {
        System.out.println("중급 엔진 입니다..");
    }
}

public class EngineLow implements IEngine {

    @Override
    public void engine() {
        System.out.println("저급 엔진 입니다.");
    }
}
```

Ex)

```java
import strategy2.interfaces.IFuel;

public abstract class Car { //공통적인 항목을 구현할 Car클래스 생성
    private IEngine engine;

    public void engine() {
        engine.engine();
    }
    public void setEngine(IEngine engine) {
        this.engine = engine;
    }

import strategy2.interfaces.EngineLow;

public class Accent extends Car { //Car클래스를 상속받음
                    //필요한 엔진 부품을 가져와 Accent라는 차종을 만듬
    public Accent() {
        setEngine(new EngineLow()); //낮은 엔진 부품(클래스) 장착
    }

```