

Spring에 자주 사용되는 세 가지 주요 Design Pattern - 이수빈

전체 개요



Spring Framework를 활용하려면 내부 구성에 대한 이해가 필수적입니다. Spring은 내부 로직에 있어 객체의 생성과 관리를 위해 다양한 디자인 패턴을 사용합니다. 그 중에서 가장 많이 사용되는 세가지 Design Pattern에 대해 소개하겠습니다.

1. **템플릿 메서드 패턴** - 스프링의 `JdbcTemplate`, `HibernateTemplate` 등의 템플릿 클래스들은 템플릿 메서드 패턴을 활용하여 중복 코드를 제거하고, 변동 가능성이 있는 부분만을 하위 클래스나 콜백 메서드에서 구현하게 해서 유연성과 재사용성을 높입니다.
2. **팩토리 패턴** - 스프링의 핵심 컨테이너인 `ApplicationContext` 는 빈(bean) 객체의 생성과 관리를 담당하는데, 이 때 팩토리 패턴을 사용하여 객체 생성의 책임을 분리하고, 구체적인 클래스를 직접 명시하지 않고도 빈 객체를 생성합니다.
3. **싱글톤 패턴** - 스프링은 빈 객체의 기본 스코프(scope)를 싱글톤으로 관리합니다. 이를 통해 애플리케이션 전체에서 해당 빈의 인스턴스가 한 개만 생성되고 재사용되는 것을 보장합니다.

1. Template Method

| 하위 클래스에서 구체적으로 처리하기

개요

Template Method 패턴은 알고리즘의 구조를 메서드에 정의하고, 그 중 일부의 동작을 서브 클래스로 확장하여 재정의하게 합니다. 이로써, 알고리즘의 구조는 변경되지 않으면서 서브 클래스 별로 특정 단계의 구현을 변경할 수 있습니다.

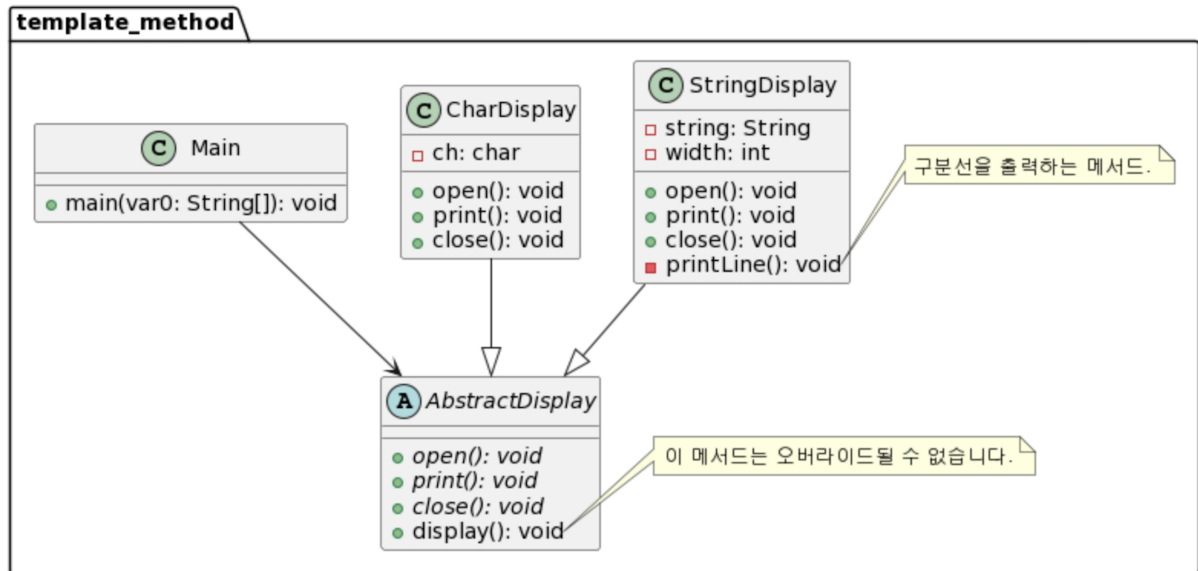
템플릿은 이미 만들어진 틀을 의미합니다.

- 문구적으로 보면 , 문자 모양으로 구멍이 뚫린 판으로 문자를 따라 쓰는 것이 있다
- 펜으로 쓰면 펜으로 쓴 문자 등 도구 상관 없이 문자는 템플릿 구멍과 동일하다

상위 클래스에서 템플릿 역할을 하여 뼈대를 결정하고, 하위 클래스에서 뼈대의 구체적인 내용을 결정하는 디자인 패턴입니다.

예제

- 문자나 문자열을 5회 반복해서 표시하기
- UML



template_method 패키지: **Template Method** 패턴과 관련된 모든 클래스를 포함합니다.

AbstractDisplay 추상 클래스:

- **Role:** Abstract Class (추상 클래스)
- **open(), print(), close():** 이 메서드들은 추상 메서드로 정의되어 있으며, Concrete Class인 **CharDisplay** 와 **StringDisplay** 에서 실제 동작을 구현해야 합니다.
- **display():** 이 메서드가 바로 **Template Method** 입니다. 구체적인 알고리즘의 흐름(순서)를 정의하고 있으며, open() -> print() -> close()의 순서대로 실행됩니다. 이 메서드는 오버라이드될 수 없게 설정되어 있어서 알고리즘의 구조 자체는 변하지 않습니다.

CharDisplay 클래스:

- **Role:** Concrete Class (구체 클래스)
- **ch:** 문자 하나를 저장하는 멤버 변수입니다.
- **open(), print(), close():** **AbstractDisplay** 의 추상 메서드들을 구체적으로 구현하고 있습니다.

StringDisplay 클래스:

- **Role:** Concrete Class (구체 클래스)
- **string, width:** 각각 출력할 문자열과 문자열의 길이(너비)를 저장하는 멤버 변수입니다.
- **open(), print(), close():** **AbstractDisplay** 의 추상 메서드들을 구체적으로 구현하고 있습니다.
- **printLine():** 이 클래스만의 특화된 메서드로, 구분선을 출력하는 기능을 담당합니다.

Main 클래스:

- **Role:** Client (클라이언트)
- 이 클래스는 **Template Method** 패턴을 실제로 어떻게 사용하는지를 시연하는 역할을 합니다.

관계:

- **CharDisplay --|> AbstractDisplay, StringDisplay --|> AbstractDisplay:** CharDisplay와 StringDisplay는 AbstractDisplay를 상속받습니다. 이 관계를 통해 AbstractDisplay에 정의된 알고리즘의 구조를 그대로 이어받으면서 필요한 부분만 재정의하여 사용합니다.
- **Main --> AbstractDisplay:** Main 클래스는 AbstractDisplay 혹은 그를 상속받는 클래스의 인스턴스를 통해 **Template Method**를 실행합니다.

Abstract 클래스 (AbstractDisplay 클래스)

- 상위 클래스에서 템플릿에 해당하는 메서드를 정의
- 실체가 없는 추상 메서드로 구성

Concrete 클래스 (CharDisplay 클래스, StringDisplay 클래스)

- 하위 클래스에서 추상 메서드의 구현
- 코드

▼ AbstractDisplay.java

```
package template_method;
/**
 * AbstractDisplay 역할의 추상 클래스.
 * 일부 메서드를 구현하고 있지만, 실제 구현 내용은 하위 클래스에 위임합니다.
 *
 * @author Template_Method
 * @version 1.0
 */
public abstract class AbstractDisplay {

    public AbstractDisplay() {}

    /**
     * 서브클래스에서 구현할 open 메서드.
     */
    public abstract void open();

    /**
     * 서브클래스에서 구현할 print 메서드.
     */
    public abstract void print();

    /**
     * 서브클래스에서 구현할 close 메서드.
     */
    public abstract void close();

    /**
     * 5번의 print를 실행하는 display 메서드.
     * 이 메서드는 오버라이드될 수 없습니다.
     */
    public final void display() {
        open();
    }
}
```

```

        for (int i = 0; i < 5; ++i) {
            print();
        }
        close();
    }
}

```

▼ CharDisplay.java

```

package template_method;

/**
 * CharDisplay 클래스는 AbstractDisplay의 구현을 제공합니다.
 *
 * @author Template_Method
 * @version 1.0
 */
public class CharDisplay extends AbstractDisplay {
    private char ch;

    /**
     * CharDisplay의 생성자입니다.
     *
     * @param var1 출력할 문자
     */
    public CharDisplay(char var1) {
        this.ch = var1;
    }

    @Override
    public void open() {
        System.out.print("<<");
    }

    @Override
    public void print() {
        System.out.print(ch);
    }

    @Override
    public void close() {
        System.out.println(">>");
    }
}

```

▼ StringDisplay.java

```

package template_method;

/**
 * StringDisplay 클래스는 AbstractDisplay의 구현을 제공합니다.
 *
 * @author Template_Method
 * @version 1.0
 */
public class StringDisplay extends AbstractDisplay {
    private String string;
    private int width;

    /**
     * StringDisplay의 생성자입니다.
     *
     * @param var1 출력할 문자열
     */
    public StringDisplay(String var1) {
        this.string = var1;
        this.width = var1.getBytes().length;
    }
}

```

```

    }

    @Override
    public void open() {
        this.printLine();
    }

    @Override
    public void print() {
        System.out.println("|" + this.string + "|");
    }

    @Override
    public void close() {
        this.printLine();
    }

    /**
     * 구분선을 출력하는 메서드.
     */
    private void printLine() {
        System.out.print("+");
        for (int i = 0; i < this.width; ++i) {
            System.out.print("-");
        }
        System.out.println("+");
    }
}

```

▼ Main.java

```

package template_method;

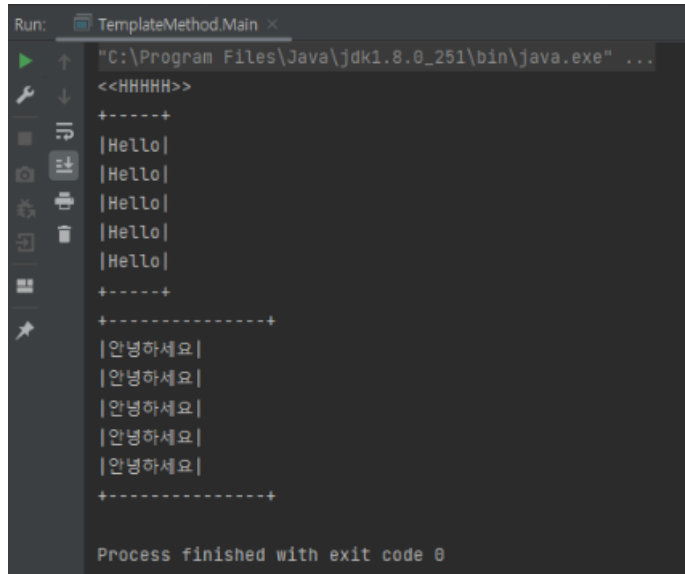
/**
 * Template Method 패턴을 테스트하는 Main 클래스.
 *
 * @author Template_Method
 * @version 1.0
 */
public class Main {

    public static void main(String[] var0) {
        // 'H'를 가진 CharDisplay 인스턴스를 1개 만든다
        AbstractDisplay d1 = new CharDisplay('H');
        AbstractDisplay d2 = new StringDisplay("Hello");
        AbstractDisplay d3 = new StringDisplay("안녕하세요");

        // d1, d2, d3 모두 AbstractDisplay의 하위 클래스의 인스턴스이므로 상속한 display 메서드 호출
        // 실제 동작은 CharDisplay나 StringDisplay에서 결정한다
        d1.display();
        d2.display();
        d3.display();
    }
}

```

▼ 실행 결과



```
Run: TemplateMethod.Main x
"C:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
<<HHHHH>>
+-----+
|Hello|
|Hello|
|Hello|
|Hello|
|Hello|
+-----+
+-----+
|안녕하세요|
|안녕하세요|
|안녕하세요|
|안녕하세요|
|안녕하세요|
+-----+
Process finished with exit code 0
```

장점

로직을 공통화할 수 있습니다

- 상위 클래스의 메소드를 사용하므로 하위 클래스에서 알고리즘을 기술할 필요가 없습니다
- 템플릿 메소드에서 에러가 나면 템플릿 메소드만 수정하면 됩니다

특징

하위 클래스를 상위 클래스와 동일시한다

- CharDisplay 인스턴스와 StringDisplay 인스턴스를 AbstractDisplay형의 변수에 대입하고 있습니다.
- 상위 클래스의 변수가 있고, 그 변수에 클래스형의 인스턴스 대입하고 있다

상위 클래스형의 변수에 하위 클래스의 어떠한 인스턴스를 대입해도 제대로 작동할 수 있도록 한다

: LSP 법칙(상속의 일반적 원칙임)

2. Factory Method

| 하위 클래스에서 인스턴스 생성하기

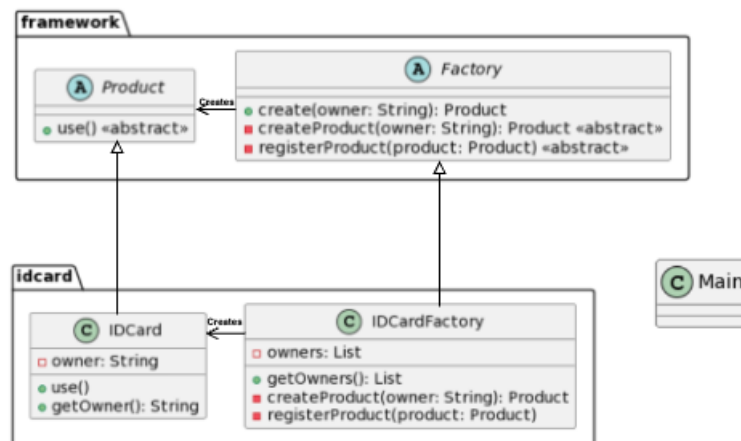
개요

인스턴스를 생성하는 공장을 Template Method 패턴으로 구성한 패턴입니다.

예제

- ID 카드를 만드는 공장
- UML

Factory Method Pattern



1. **factory.framework 패키지:** 핵심 구조를 정의하는 두 개의 추상 클래스를 포함하고 있습니다.
 - **Factory**: 제품 생성에 관련된 메서드를 가집니다. 인스턴스 생성 시 Template Method를 사용합니다.
 - 구체적인 내용은 **IDCardFactory** 에서 구현합니다.
 - **create** 메서드는 public이며, **createProduct** 와 **registerProduct** 는 protected이고 추상 메서드로 되어 있습니다.
 - **Product**: 제품을 나타내며 **use** 라는 추상 메서드를 포함하고 있어 무엇이든 **use** 할 수 있습니다.
2. **factory.idcard 패키지:** **factory_method** 패턴의 구체적인 구현을 담고 있습니다.
 - **IDCardFactory**: **Factory** 클래스를 상속받아 구현한 클래스입니다. **owners** 리스트를 통해 생성된 ID 카드의 소유자 정보를 관리합니다. **IDCard** 인스턴스를 생성하여 **IDCard** 만드는 일을 구현합니다.
 - **IDCard**: **Product** 클래스를 상속받아 **use** 메소드를 구현한 클래스입니다. 각 ID 카드는 **owner** 라는 속성을 가지고 있습니다.
3. **관계:**
 - **IDCardFactory** 는 **Factory** 를 상속받습니다.
 - **IDCard** 는 **Product** 를 상속받습니다.
 - **Main** 클래스는 **IDCardFactory** 와 **Product** 와 사용하는 연관 관계를 가집니다.

- 코드

- ▼ framework/Product.java

```
package factory.framework;

/**
 * 모든 제품이 공통으로 가져야할 기능을 정의하는 추상 클래스.
 */
public abstract class Product {

    /**
     * 제품 사용을 위한 추상 메서드. 하위 클래스에서 구체적인 동작을 구현해야 합니다.
     */
    public abstract void use();
}
```

- ▼ framework/Factory.java

```
package factory.framework;

/**
 * 제품 생성 및 등록을 위한 추상 팩토리 클래스.
 * 구체적인 제품 생성은 하위 클래스에서 구현되어야 합니다.
 */
public abstract class Factory {

    /**
     * 제품을 생성하고 등록한 후, 생성된 제품을 반환합니다.
     * @param owner 제품의 소유주 정보
     * @return 생성된 Product 객체
     */
    public final Product create(String owner) {
        Product p = createProduct(owner);
        registerProduct(p);
        return p;
    }

    /**
     * 제품을 생성하는 추상 메서드. 하위 클래스에서 구현되어야 합니다.
     * @param owner 제품의 소유주 정보
     * @return Product 객체
     */
    protected abstract Product createProduct(String owner);

    /**
     * 생성된 제품을 등록하는 추상 메서드. 하위 클래스에서 구현되어야 합니다.
     * @param product 생성된 Product 객체
     */
    protected abstract void registerProduct(Product product);
}
```

- ▼ idcard/IDCard.java

```
package factory.idcard;

import factory.framework.*;

/**
 * ID 카드를 표현하는 클래스.
 */
public class IDCard extends Product{
    private String owner;
```



```

/**
 * 주어진 소유주 정보로 IDCard 객체를 생성하고 초기화합니다.
 * @param owner 카드의 소유주 이름
 */
IDCard(String owner) {
    System.out.println(owner + "의 카드를 만듭니다.");
    this.owner = owner;
}

@Override
public void use() {
    System.out.println(owner + "의 카드를 사용합니다.");
}

/**
 * 카드의 소유주 정보를 반환합니다.
 * @return 카드의 소유주 이름
 */
public String getOwner() {
    return this.owner;
}
}

```

▼ idcard/IDCardFactory.java

```

package factory.idcard;

import factory.framework.*;
import java.util.ArrayList;
import java.util.List;

/**
 * ID 카드를 생성하고 관리하는 팩토리 클래스.
 */
public class IDCardFactory extends Factory{
    private List owners = new ArrayList();

    @Override
    protected Product createProduct(String owner) {
        return new IDCard(owner);
    }

    @Override
    protected void registerProduct(Product product) {
        owners.add(((IDCard)product).getOwner());
    }

    /**
     * 지금까지 생성된 모든 카드 소유주의 리스트를 반환합니다.
     * @return 소유주 리스트
     */
    public List getOwners() {
        return this.owners;
    }
}

```

▼ Main.java

```

package factory;

import factory.framework.Product;
import factory.idcard.IDCardFactory;

/**
 * Factory 패턴을 테스트하는 메인 클래스.
 */

```

```

public class Main {
    public Main() {}

    public static void main(String[] var0) {
        IDCardFactory var1 = new IDCardFactory();
        Product var2 = var1.create("홍길동");
        Product var3 = var1.create("이순신");
        Product var4 = var1.create("강감찬");
        var2.use();
        var3.use();
        var4.use();
    }
}

```

▼ 실행 결과

```

Run: FactoryMethod.Main x
"C:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
홍길동의 카드를 만듭니다.
이순신의 카드를 만듭니다.
강감찬의 카드를 만듭니다.
홍길동의 카드를 사용합니다.
이순신의 카드를 사용합니다.
강감찬의 카드를 사용합니다.
Process finished with exit code 0

```

특징

전혀 다른 '제품'과 '공장'을 만들 때, framework 패키지를 import한 다른 factory 패키지를 만들면 됩니다. 즉, framework 패키지의 내용을 수정할 필요가 없습니다(framework 패키지는 idcard 패키지에 의존하고 있지 않다)

그 외

▼ 인스턴스 생성

1. 추상메소드로 한다(예제 프로그램에서 사용하는 방법)

```

abstract class Factory{
    public abstract Product createProduct(String name);
}

```

2. 디폴트의 구현을 준비해둔다

하위 클래스에서 구현하지 않았을 때 사용

```

class Factory{
    public Product createProduct(String name){
        return new Product(name); // 이때 Product는 추상 클래스가 아니어야 함
    }
}

```

3. 예러를 이용한다

디폴트의 구현 내용을 에러처리하여 하위 클래스에서 구현하지 않았을 경우 에러 발생

```
class Factory{
    public Product createProduct(String name){
        throw new FactoryMethodRuntimeExeption(name); // FactoryMethodRuntimeExeption는 구현되어 있다고 가정
    }
}
```

3. Singleton

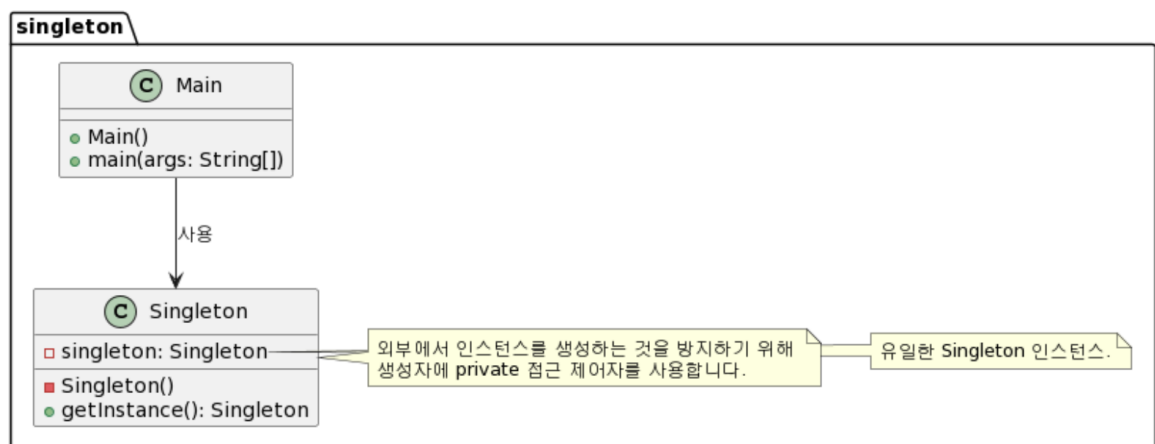
| 인스턴스를 한 개만 만들기

개요

클래스의 인스턴스가 단 하나만 필요한 경우 1개만 존재한다는 것을 ‘보증’한다.

예제

- UML



1. **Singleton 패키지:** Singleton 패턴의 클래스들이 포함되어 있습니다.
2. **Singleton 클래스:**
 - `singleton: Singleton` : 이는 private 정적 멤버로, Singleton 클래스의 유일한 인스턴스를 저장합니다.
 - `Singleton()` : private 생성자로, 이를 통해 외부에서 Singleton 클래스의 인스턴스를 직접 생성할 수 없게 만듭니다.
 - `+ getInstance(): Singleton` : public 정적 메서드로, Singleton 클래스의 유일한 인스턴스를 반환합니다.
3. **Main 클래스:** Singleton 패턴을 시연하는 클래스입니다.

1. 관계:

- **Main --> Singleton**: Main 클래스에서 Singleton 클래스를 사용하고 있음을 나타냅니다.

• 코드

▼ Singleton.java

```
package singleton;

/**
 * 싱글턴 패턴을 구현한 클래스. 이 클래스의 인스턴스는 한 개만 생성됩니다.
 */
public class Singleton {

    /**
     * 유일한 Singleton 인스턴스.
     */
    private static Singleton singleton = new Singleton();

    /**
     * Singleton 생성자.
     * <p>
     * 외부에서 인스턴스를 생성하는 것을 방지하기 위해 private 접근 제어자를 사용합니다.
     */
    private Singleton(){
        System.out.println("인스턴스를 생성했습니다");
    }

    /**
     * 유일한 Singleton 인스턴스를 반환하는 메서드.
     *
     * @return Singleton의 유일한 인스턴스
     */
    public static Singleton getInstance(){
        return singleton;
    }
}
```

▼ Main.java

```
package singleton;

/**
 * Singleton 패턴을 테스트하는 메인 클래스.
 */
public class Main {

    /**
     * 메인 생성자.
     */
    public Main(){
    }

    /**
     * 프로그램의 진입점.
     * @param args 명령행 인수
     */
    public static void main(String[] args){
        System.out.println("Start");
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();
        if (obj1 == obj2){
            System.out.println("obj1과 obj2는 같은 인스턴스입니다.");
        }
    }
}
```

```

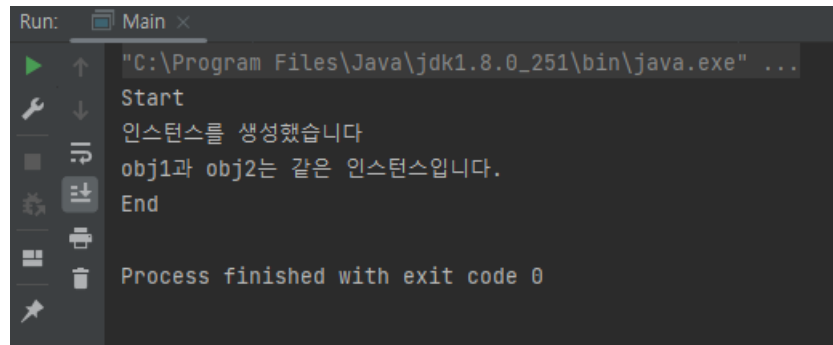
        System.out.println("End");
    }
}

```

최초로 getInstance 메소드를 호출했을 때 Singleton 클래스가 초기화됩니다

그리고 이때 static 필드의 초기화가 이루어지고 유일한 인스턴스가 만들어집니다

▼ 실행 결과



특징

- 외부에서 생성할 수 없습니다.
 - 복수의 인스턴스가 있으면 인스턴스들이 서로 영향을 미치고, 뜻하지 않은 버그가 발생할 가능성이 있습니다
- static을 사용하여 고정된 영역에 할당하므로 메모리 낭비가 없습니다.