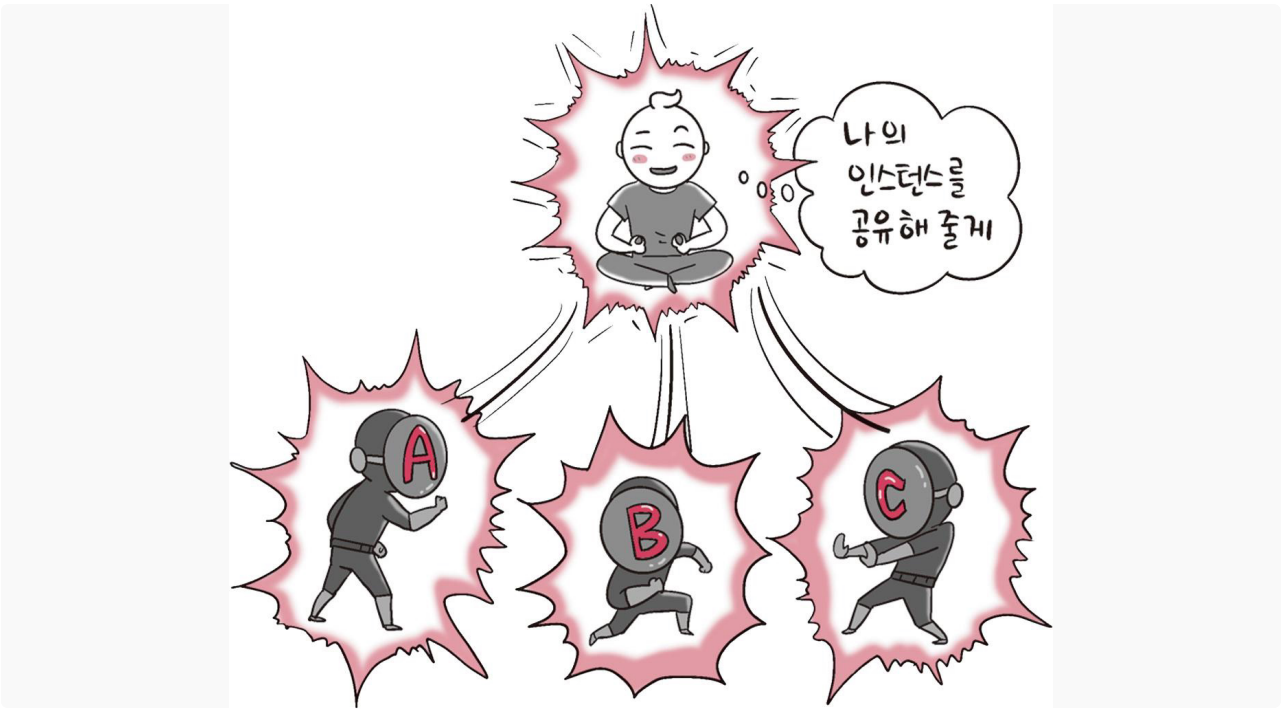
싱글톤 패턴(Singleton Pattern)



싱글톤 패턴의 개념과 동작 방식

싱글톤 패턴(Singleton Pattern)은 객체 생성 패턴 중 하나로, 어떤 클래스에 대해서 인스턴스를 하나만 생성하도록 보장하는 패턴입니다. 이는 전역 변수(global variable)를 사용하지 않고, 클래스 내부에서 유일한 인스턴스를 생성하고, 이를 공유하여 사용하는 방식으로 구현됩니다.

싱글톤 패턴은 일반적으로 다음과 같은 방식으로 구현됩니다.

1. 클래스 내부에 유일한 인스턴스를 저장하기 위한 정적 변수를 생성합니다.
2. 클래스 외부에서는 객체를 생성할 수 없도록 생성자를 **private**으로 선언합니다.
3. 유일한 인스턴스를 반환하는 정적 메서드를 제공합니다.

이렇게 구현된 싱글톤 패턴은 다음과 같은 동작 방식을 가집니다.

1. 최초에 싱글톤 객체를 요청하면, 해당 클래스의 유일한 인스턴스를 생성합니다.
2. 이후에는 이미 생성된 유일한 인스턴스를 반환합니다.
3. 프로그램이 종료될 때까지 유일한 인스턴스가 유지됩니다.

싱글톤 패턴은 자원을 효율적으로 관리하고, 객체 생성에 따른 부하를 최소화할 수 있어서 유용합니다.

예를 들어, 데이터베이스 연결, 로깅, 캐싱 등과 같이 여러 곳에서 공유해야 하는 자원을 관리하기 위해 사용됩니다.

싱글톤 패턴의 장단점과 적용 예시

장점

- 유일한 인스턴스를 제공하므로, 메모리 공간을 절약할 수 있습니다.
- 한 번의 객체 생성으로 재사용성이 증가합니다.
- 전역 인스턴스이기 때문에 다른 객체들이 쉽게 접근할 수 있습니다.
- 인스턴스를 하나만 생성하므로, 자원의 과도한 사용을 막을 수 있습니다.

단점

- 싱글톤 패턴이 잘못 구현되면, 멀티스레드 환경에서 동시성 문제가 발생할 수 있습니다.
- 다른 객체들과의 결합도가 높아질 수 있습니다.
- 객체 생성 시기가 지연될 수 있으므로, 애플리케이션 시작 시간이 늦어질 수 있습니다.

적용 예시

- 로그 처리기, DB 연결 객체, 설정 관리자 등 유일한 인스턴스가 필요한 경우에 사용됩니다.
- 대표적인 예시로 자바의 `java.lang.Runtime` 클래스가 있습니다. 이 클래스는 하나의 인스턴스만 존재하며, 모든 JVM에서 공유됩니다.
- 싱글톤 패턴을 사용하지 않을 경우, 매번 객체를 생성해야 하기 때문에 자원의 낭비가 발생할 수 있습니다. 하지만, 싱글톤 패턴은 객체 생성을 제한하기 때문에 자원의 낭비를 줄일 수 있습니다.

싱글톤 패턴의 구현 방법과 주의할 점

Eager initialization (이른 초기화)

이 방법은 클래스가 로드될 때 인스턴스를 즉시 생성하는 방법입니다. 이를 위해 클래스 내부에 정적 멤버 변수를 선언하고, 이를 클래스의 인스턴스로 초기화합니다.

```
public class Singleton {
    private static final Singleton instance = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return instance;
    }
}
```

Lazy initialization (늦은 초기화)

이 방법은 인스턴스가 처음 요청될 때 인스턴스를 생성하는 방법입니다. 이를 위해 `getInstance()` 메서드에서 인스턴스가 생성되어 있지 않은 경우에만 인스턴스를 생성합니다. 이 방법은 인스턴스를 사용하지 않을 때는 메모리를 절약할 수 있으며, 처음 인스턴스를 생성하는 시점을 미룰 수 있습니다.

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Thread-safe lazy initialization (스레드 안전한 늦은 초기화)

위의 Lazy initialization 방법은 멀티스레드 환경에서 문제가 될 수 있습니다. 동시에 `getInstance()` 메서드를 호출하면 인스턴스를 중복해서 생성할 수 있기 때문입니다. 이를 해결하기 위해서는 인스턴스 생성을 동기화해야 합니다. 하지만 모든 `getInstance()` 메서드를 동기화하면 성능 저하가 발생할 수 있기 때문에, 인스턴스가 생성되어 있는지 확인하는 부분만 동기화합니다.

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

싱글톤 패턴을 구현할 때 주의점

1. 싱글톤 객체는 반드시 단 하나의 인스턴스만 생성되어야 합니다.
2. 인스턴스는 전역 변수로 선언되어야 합니다.
3. 생성자는 반드시 private으로 선언되어야 합니다.
4. 인스턴스는 외부에서 직접 접근할 수 없으며, `getInstance()` 메서드를 통해서만 접근할 수 있어야 합니다.
5. 멀티스레드 환경에서 안전하도록 구현해야 합니다. 동기화를 이용하거나, 미리 인스턴스를 생성하는 등의 방법을 사용해야 합니다.

싱글톤 패턴의 대안 패턴과 비교

의존성 주입(Dependency Injection) 패턴

객체 생성과 의존성 주입을 분리하여, 클래스 간 결합도를 낮추고 유연한 구조를 만들어줍니다. 객체를 생성하고 관리하는 책임은 DI 컨테이너에게 넘기며, 객체간의 의존성은 생성자 주입(Constructor Injection), setter 주입(Setter Injection), 인터페이스 주입(Interface Injection) 등을 통해 주입합니다. 이를 통해 객체의 생성과 의존성 주입이 복잡한 코드를 줄이고 테스트 용이성을 높입니다.

서비스 로케이터(Service Locator) 패턴

객체의 실제 구현체를 찾기 위한 검색 로직을 추상화하여, 애플리케이션에서 필요한 객체를 검색할 수 있는 서비스 로케이터를 제공합니다. 이를 통해 객체의 위치를 중앙 집중적으로 관리하며, 모듈 간의 결합도를 낮춥니다.

액터 패턴(Actor Pattern)

병렬 처리를 위한 모델로, 객체가 비동기 메시지를 주고받으며 상태를 변경하고 결과를 반환합니다. 액터는 고유한 상태(state)를 가지고 있으며, 이 상태는 다른 액터와 공유되지 않습니다. 이를 통해 병렬 처리를 보다 안전하게 처리할 수 있습니다.

싱글톤 패턴과 비교하면, 의존성 주입 패턴과 서비스 로케이터 패턴은 더 유연하고 확장성이 높은 구조를 제공하지만, 코드의 복잡도가 증가하고 초기 설정 비용이 높을 수 있습니다. 액터 패턴은 병렬 처리를 위해 적합하지만, 모든 문제에 적용하기에는 적절하지 않을 수 있습니다. 이러한 패턴들은 적용할 문제에 따라 다르며, 적절한 패턴을 선택하여 사용해야 합니다.