Object–Oriented Programming

LAB #12_2. Design Patterns : Singleton

Design Pattern

- 소프트웨어를 설계할 때 자주 발생하는 문제들에 재사용할 수 있는 훌륭한 해결책

- 가이드로 사용할 수 있는 모델 혹은 설계

- 즉, 특정 상황에서 일반적인 문제에 대한 입증된 솔루션

Design Pattern

- 소프트웨어 개발자에서 이미 해결된 문제를 처리하기 위한 툴킷을 제공

- 소프트웨어 문제를 어떻게 해결할지 생각하는데 도움을 줌

- 23개의 디자인 패턴을 정리하고 다음 3가지로 분류
  - Creational(생성) : 객체 생성과 관련된 패턴
  - Structural(구조) : 클래스나 객체를 조합해 더 큰 구조를 만드는 패턴
  - Behavioral(행위) :
    객체나 클래스 사이의 알고리즘이나 책임 분쟁과 관련된 패턴

Design Pattern

- 종류

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	**Decorator**	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	**Observer**
		State
		Strategy
		Template Method
		Visitor

Singleton

- 전역 변수를 사용하지 않고 객체를 하나만 생성하도록 하며,
  생성된 객체를 어디에서든지 참조할 수 있도록 하는 패턴
- 단 하나의 인스턴스를 생성해 사용하는 디자인 패턴
- 사용 예시
  - 로그인
  - 캐시
  - 레지스트리 처리 객체
  - 외부 자원
    - 프린터
    - 디바이스 드라이버
    - 데이터베이스

Singleton

- 장점
  - 메모리 낭비를 방지할 수 있다.
  - 다른 클래스와의 데이터 공유를 편하게 할 수 있다.

- 단점
  - 과도하게 사용하면 다른 클래스의 인스턴스들 간에 결합도가 높아져 기능 검증 및 수정이 어렵다.
  - MultiThreading 환경에서 동기화 처리를 하지 않으면 문제가 발생할 수 있다.

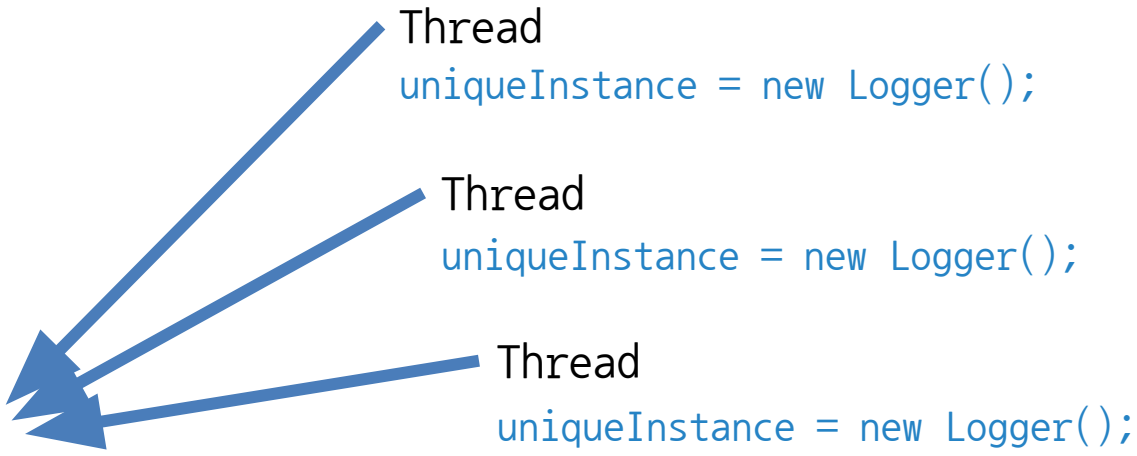
Singleton Example

```
public class Logger
{
    private Logger() {}

    private static Logger uniqueInstance;

    public static Logger getInstance()
    {
        if(uniqueInstance == null)
            uniqueInstance = new Logger();

        return uniqueInstance;
    }
}
```

- 생성자를 private으로 선언했기 때문에 Logger 클래스에서만 클래스의 인스턴스를 만들 수 있다.
- Logger 클래스의 유일한 인스턴스를 저장하기 위한 정적 변수 uniqueInstance
- Lazy Instantiation = 객체가 필요한 상황이 되기 전까지 객체를 생성하지 않음

Singleton Example

```
public class Logger
{
     private Logger() {}

     private static Logger uniqueInstance;

     public static Logger getInstance()
     {
          if(uniqueInstance == null)
               uniqueInstance = new Logger();

          return uniqueInstance;
     }
}
```



- 위의 코드는 여러 스레드가 동시에 접근할 때 문제가 생긴다.

Threading 문제 해결법

1. Simple Locking
   - synchronized 를 사용하여 간단히 `getInstance()` Method를 Lock하는 방법
     ref. [Java] 혼동되는 synchronized 동기화 정리 - 제리 devlog
2. Double-Checked Locking
   - `volatile` 를 사용하여 Lock하는 방법
3. Eager Initialization
   - 클래스 참조 시, 객체 생성하여 충돌을 방지하는 방법

1. Simple Locking

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

1. Simple Locking

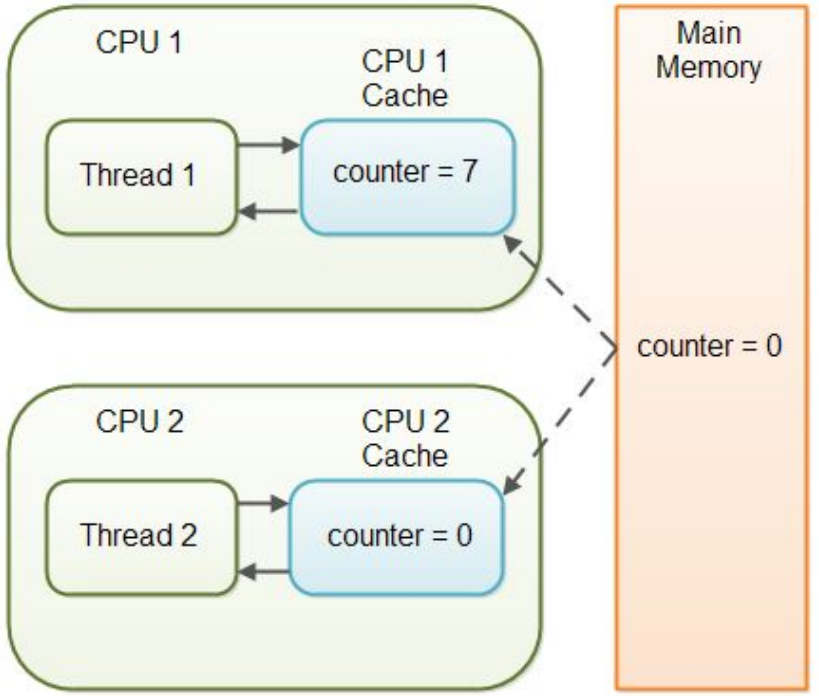
```
public class Singleton
{
    private Singleton() {}
    private static Singleton uniqueInstance;
    public static Singleton getInstance()
    {
        if (uniqueInstance == null) {
            synchronized(Singleton.class){
                uniqueInstance = new Singleton();
            }
      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

        return uniqueInstance;
    }
}
```

volatile Variable

- 멀티스레드 어플리케이션에서는 Task를 수행하는 동안 성능향상을 위해 Main Memory에서 읽은 변수 값을 CPU Cache에 저장하게 된다.

- 만약 멀티스레드 환경에서 스레드가 변수 값을 읽어올 때 각각의 CPU Cache에 저장된 값이 다르기 때문에 다음과 같이 변수 값 불일치 문제가 발생한다.

- Thread 1은 counter값을 증가시키고 있지만 CPU Cache에만 반영되어 있고 실제로 Main Memory에는 반영이 되지 않았을 때, Thread 2가 counter값을 0으로 읽는다.



2. Double-Checked Locking

```
public class Singleton
{
    private Singleton() {}
    private volatile static Singleton uniqueInstance;
    public static Singleton getInstance()
    {
        if (uniqueInstance == null) {       // single checked
            synchronized(Singleton.class) {
                if(uniqueInstance == null)  // double checked
                    uniqueInstance = new Singleton();
            }
        }
        return uniqueInstance;
    }
}
```

volatile Variable

- “Java 변수를 Main Memory에 저장하겠다”라 명시하기 위해 사용

- 매번 변수의 값을 Read할 때마다 CPU cache에 저장된 값이 아닌 Main Memory에서 읽는다.

- 변수의 값을 Write할 때마다 Main Memory에 작성한다.

3. Eager Initialization

```
public class Singleton
{
    private Singleton() {}

    private static Singleton uniqueInstance = new Singleton ();

    public static Singleton getInstance()
    {
        return uniqueInstance;
    }
}
```

- Class가 참조될 때, 바로 Singleton 객체를 생성하기 때문에 오버헤드가 적다.

Further reading

- https://blogs.oracle.com/javamagazine/post/java-thread-synchronization-volatile-final-atomic-deadlocks