Singleton Pattern

2023.05.21

싱글톤 패턴이란

- 여러 개의 인스턴스를 만들지 않고,

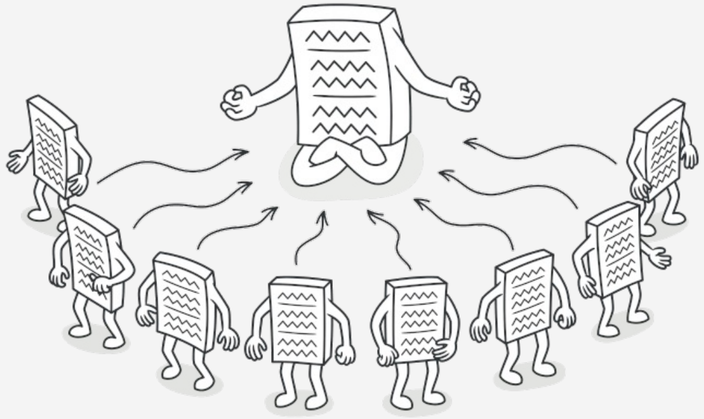
  하나의 인스턴스를 기반으로 공유하는 구현 방법

  - 하나의 클래스를 기반으로 여러 개의 개별적인 인스턴스를 만들 수 있지만,

    그렇게 하지 않고 하나의 클래스를 기반으로 단 하나의 인스턴스를 만들어

    이를 기반으로 로직을 만드는 데 사용

- 인스턴스 생성에 장점이 있으나, 의존성이 높아지고 TDD에 단점



싱글톤 패턴이란

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}

const a = new Rectangle(1, 2);
const b = new Rectangle(1, 2);
console.log(a === b); // False
```

```
class Singleton {
    constructor() {
        if (!Singleton.instance) {
            Singleton.instance = this
        }
        return Singleton.instance
    }
    getInstance() {
        return this.getInstance
    }
}

const a = new Rectangle();
const b = new Rectangle();
console.log(a === b); // True
```

싱글톤 구현

- DB 연결 모듈

```
const URL = 'mongodb://localhost:27017/app'
const createConnection = url => ({"url" : url})

class DB {
  constructor(url) {
    if (!DB.instance) {
      DB.instance = createConnection(url)
    }
    return DB.instance
  }
  connect() {
    return this.instance
  }
}

const a = new DB(URL)
const b = new DB(URL)

console.log(a === b) // true
```

장점

- 하나의 인스턴스를 기반으로 해당 인스턴스를 다른 모듈들이 공유하여 사용

  → 인스턴스를 생성할 때 드는 비용이 줄어듦

  인스턴스 생성에 많은 비용이 소요되는 I/O 바운드 작업에 많이 사용

  * I/O 바운드 : 디스크 연결, 네트워크 통신, 데이터베이스 연결

단점

- 의존성이 높아짐
- TDD(Test Driven Development)를 할 때 걸림돌이 됨
  - TDD시 단위 테스트를 주로 사용

    단위 테스트는 서로 독립적이어야 하고, 테스트를 어떤 순서로든 실행할 수 있어야 함
  - 싱글톤 패턴은 미리 생성된 하나의 인스턴스를 기반으로 구현하기 때문에

    각 테스트마다 독립적인 인스턴스를 만들기 어려움

싱글톤 구현

- 단순한 메서드 호출
  - 그러나 메서드의 원자성이 결여되어 있음
  - 멀티스레드 환경에서는 싱글톤 인스턴스를 2개 이상 만들 수 있음

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {

    }

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

싱글톤 구현

- Synchronized
  - 인스턴스를 반환하기 전까지 격리 공간에 놓기 위해 synchronized 키워드로 잠금
  - 최초로 접근한 스레드가 해당 메서드 호출시에 다른 스레드가 접근하지 못하도록 잠금(lock)을 걸어둠
  - 하지만 getInstance() 메서드를 호출할 때마다 lock이 걸려 성능 저하

```
public class Singleton {

    private static Singleton instance;

    private Singleton() {
    }

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

싱글톤 구현

- 정적 멤버
  - 정적 멤버나 블록은 런타임이 아닌 최초 JVM이 클래스 로딩 때 모든 클래스들을 로드할 때 미리 인스턴스를 생성
  - 클래스 로딩과 동시에 싱글톤 인스턴스를 만드는데, 모듈들이 싱글톤 인스턴스를 요청할 때 만들어진 인스턴스를 반환하면 됨
  - 다만 불필요한 자원낭비라는 문제가 존재 → 싱글톤 인스턴스가 필요없는 경우에도 무조건 인스턴스를 만들어야 함

```
public class Singleton {

    private final static Singleton instance = new Singleton();

    private Singleton() {

    }

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

싱글톤 구현

- 정적 블록

```
public class Singleton {

    private static Singleton instance = null;

    static {
        instance = new Singleton();
    }

    private Singleton() {

    }

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

싱글톤 구현

- 정적 멤버와 Lazy Holder(중첩 클래스)
  - singleInstanceHolder라는 내부 클래스를 하나 더 만들어서,

    Singleton 클래스가 최초에 로딩되더라도 함께 초기화 되지 않고,

    getInstance()가 호출될 때 singleInstanceHolder 클래스가 로딩되어 인스턴스를 생성

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

싱글톤 구현

- 이중 확인 잠금(DCL, Double Checked Locking)
  - 인스턴스 생성 여부를 싱글톤 패턴 잠금 전에 한번, 객체를 생성하기 전에 한번

    2번 체크하면 인스턴스가 존재하지 않을 때만 잠금을 걸 수 있기 때문에 앞선 문제를 해결 가능

```
public class Singleton {

    private volatile Singleton instance;

    private Singleton() {

    }

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

싱글톤 구현

- enum
  - enum의 인스턴스는 기본적으로 스레드 세이프(thread safe)한 점이 보장되기 때문에 이를 통해 생성할 수 있음

    * enum은 자동으로 synchronized하게 생성됨

```
public enum SingletonEnum {
    INSTANCE;
    public void oortCloud() {

    }
}
```

Thank You