1️⃣

싱글턴(Singleton Pattern)

- 특정 클래스에 객체 인스턴스가 하나만 만들어지도록 해주는 패턴
- 전역 변수를 사용할 때와 마찬가지로 객체 인스턴스를 어디서든지 액세스 할 수 있게 만들 수 있음

public static final 필드

- 외부에서 객체를 생성할 수 없도록 생성자를 private로 접근 범위를 제한
- 이미 할당했는데 싱글턴 내부에서 다시 객체를 할당하는 실수가 없도록 final로 선언

```
public class Singleton{
  public static final Singleton INSTANCE = new Singleton();

  private Singleton(){}

  ...
}

//또는

public class Singleton{
  public static final Singleyon INSTANCE = new Singleton();

  private Singleton(){}

  public static Singleton getInstance(){
    return INSTANCE;
  }

  ...
}

//정적 필드로 접근
Singleton.INSTACE.service();

//또는
Singleton obj = Singleton.getInstance();
obj.service();
```

지연 초기화(lazy initialization)

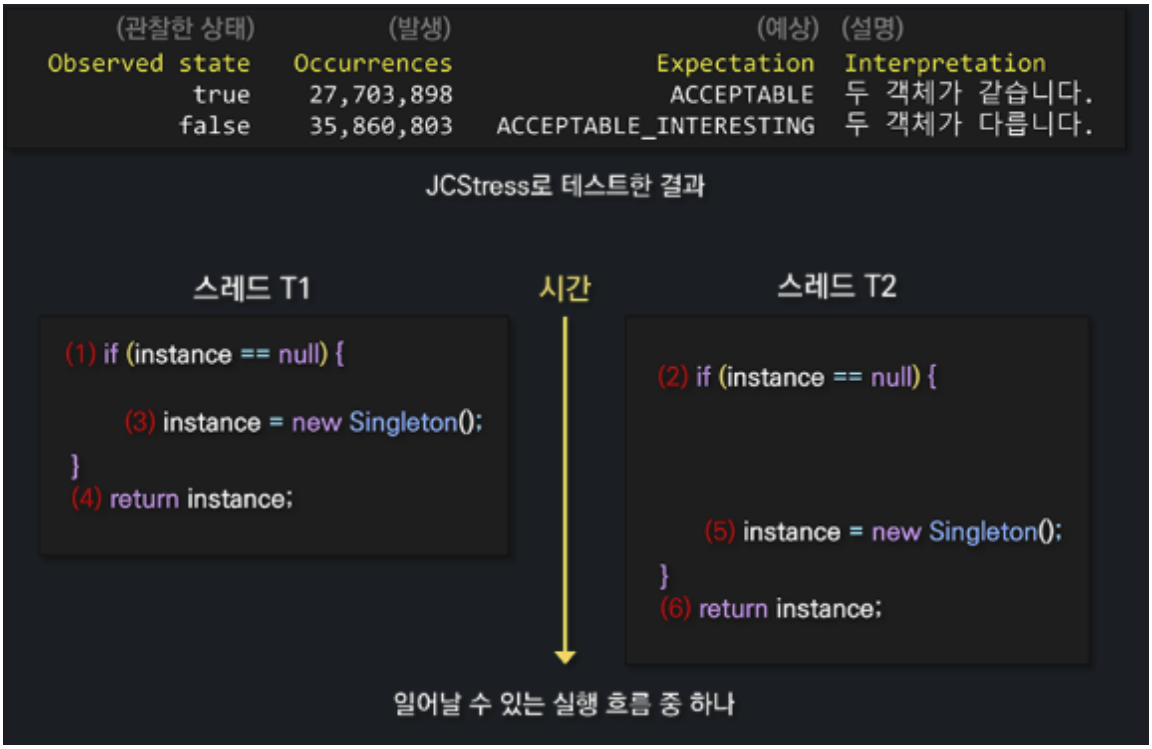
- Singleton.getInstance() 메서드 호출 시점에서 정적 필드가 초기화되지 않았으면 객체를 생성 후 return

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

  private Singleton(){}

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

  ...
}
```

쓰레드 안전(Thread Safe)

- 멀티 쓰레드 환경에서 위와 같이 초기화 방법을 사용하면 안전하지 않다는 문제가 발생

  ex) 두개 이상의 쓰레드각 Singleton.getInstance()를 동시에 호출하면???



- 객체가 유일함을 보장하려 했으나 위와 같이 쓰레드가 서로 다른 객체의 참조를 가지는 상황이 발생할 수 있음
- 해당 문제는 동기화(synchronized)를 해줌으로서 해결이 가능
- 이 방법은 정적필드 초기화 후 싱글턴 객체를 얻을 때 불필요한 동기화 작업이 일어나므로 성능이 저하됨

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

  private Singleton() { }

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

  ...
}
```

더블 체크 락킹(Double-Checked Locking)

- 위의 불필요한 동기화작업으로 성능저하가 되는것을 방지

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

  private Singleton(){}

  public static Singleton getInstance(){
    if(instance == ull){   //(1)
      //하번에 하나의 쓰레드만 접근 할 수 있음
      sunchronized(Singleton.class){   //(2)
        if(instance == null){   //(3)
          instance = new Singleton();    //(4)
        }
      }
    }
    return instance;
  }

  ...
}
```

- 위 과정 중 4번에서 내부 상태 초기화가 이루어지고 있다 가정 했을 때의 문제점
  - 초기화 도중 1번으로 다른 쓰레드가 들어와서 null이 아님을 확인하고 초기화 중인 객체 참조를 그대로 반환 할수도 있음
  - 그렇기 때문에 객체 초기화가 완전히 끝난 다음에야 그 객체에 대한 참조가 저장되게 하기 위해 volatile이란 예약어를 선언
  - 물론 volatile은 그 이상의 일을 함
- 따라서 최종적으로 다음과 같이 작성

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

  private Singleton(){}

  public static Singleton getInstance(){
    if(instance == ull){
      //하번에 하나의 쓰레드만 접근 할 수 있음
      sunchronized(Singleton.class){
        if(instance == null){
          instance = new Singleton();
        }
      }
    }
    return instance;
  }

  ...
}
```

요청 시 초기화 홀더 패턴(Initialization-On-Demand Holder Pattern)

- 홀더 클래스를 사용해서 지연 초기화를 구현
- 더블 체크 락킹보다 더 단순하며 안전함
- **초기화가 언제 일어나는지 그 시점에 대해 알필요가 있음**
  - 클래스나 인터페이스 타입 T는 다음 중 하나가 처음 일어나기 직전에 초기화 됨
    - T는 클래스이며T의 인스턴스가 생성된다.
    - T에 선언된 정적 메소드가 호출된다.
    - T에 선언된 정적 필드가 할당된다.
    - T에 선언된 정적 필드가 사용되며 이때 이 필드는 상수 변수가 아니다.
- 하지만, 굳이 중첩 클래스를 사용할 필요는 없음
  - getInstance() 이외의 공개된 정적 메서드가 있다면 중첩 클래스를 만드는게 적절
  - But!! 그런 사례가 많지 않음

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

  private static final class Holder{
    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance(){
    return Holder.INSTANCE;
  }

  ...
}
```

열거형을 사용하는 방법

- 추가 적인 노력없이 직렬화 가능
- 열거형에는 열거형 상수를 통해 정의된 인스턴스 이외의 인스턴스는 없음
  - 명시적으로 인스턴스화하려고 시도 시 컴파일 타임 에러 발생

- final clone() 메서드를 통해 열거 상수를 복제 할 수 없음을 보장 (CloneNotSupportedEceoption)
- 직렬화 메커니즘을 통한 특수 처리로 역직렬화의 결과로 중복 인스턴스가 생성되지 않음을 보장
- 리플렉션을 통해 열거형을 인스턴스화 하는것은 금지된다.

```
public enum Singleton{
  INSTANCE;

  ...
}
```

- 열거형은 상속받을 수 없음 → 상속이 필요하면 enum은 사용 불가

```
//이 코드는 작동하지 않음
final class Singleton extends Enum<Singleton>{
  public static final INSTANCE = new Singleton();

  ...
}
```

싱글톤의 문제점

- 보통 객체의 유일성이 아닌 전역 접근, 즉 어디에서나 접근 할 수 있다는 점에서 과도하게 오용할 때 문제발생

테스트하기가 어렵다

- 정적 필드는 한번 할당 시 프로그램이 종료되기기 전까지 살아있음
- 각 테스트는 독립적으로 다른 테스트에 영향을 미치지 않아야하는데 싱글턴 객체를 만들면 다른테스트에서 이를 확인할 수 있음
- 일반적으로 인터페이스가 아닌 클래스를 통해 구현되는 싱글턴은 목(mock)으로 대체할 수 없으므로 단위 테스트를 매우 까다롭게 만듬
- 이를 해결하기 위해 주로 의존관계주입(dependency injection, DI)을 사용할 수 있으며 이때는 보통 DI프레임워크가 싱글턴 객체의 생성을 제어

데이터 경쟁이 일어나기 쉽다

- 싱글턴이 상태를 가지고 있다면 상황이 더 복잡해지며, 멀티 쓰레드 환경에서는 공유 변수 접근 시 적절하게 동기화가 이루어지지 않았다면 경쟁 상태(race condition)가 문제가 될 수 있으므로 주의해야 함

변경에 취약해진다

- 어디서에나 접근할 수 있으므로 싱글턴의 구조나 동작에 변경이 일어나면 싱글턴에 의존하고 있는 클래스에서도 변경내용이 적용되기 때문에 문제가 발생함
- 이렇게 긴밀하게 연결된 관계는 테스트가 어렵다는 문제점으로도 이어짐