싱글턴 패턴

이관우

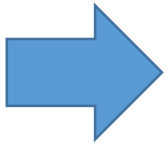
kwlee@hansung.ac.kr

학습 목표

- 인스턴스가 하나뿐인 객체가 필요한 상황을 이해한다.
- 싱글턴 패턴을 이해한다.
- 싱글턴 패턴의 멀티 스레딩 문제 해결 방안을 살펴본다.

인스턴스가 하나뿐이어야 하는 객체..

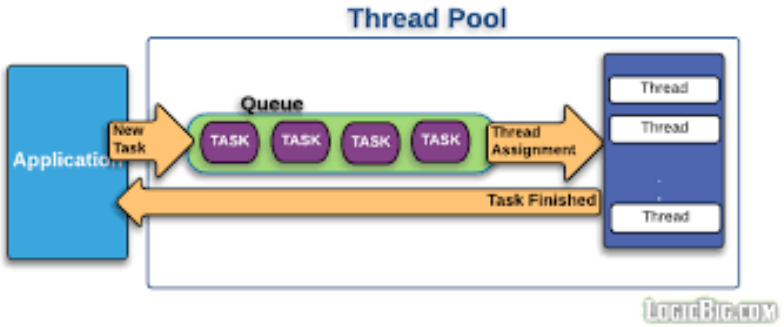
- 스레드 풀
- 캐시
- 대화상자
- 사용자 설정
- 레지스트리 설정
- 로그 기록 용 객체
- 디바이스 드라이버



두 개 이상의 인스턴스는
프로그램을 이상하게 돌아가게 하던지
불필요한 자원을 잡아먹는다.

스레드 풀 (Thread Pool)

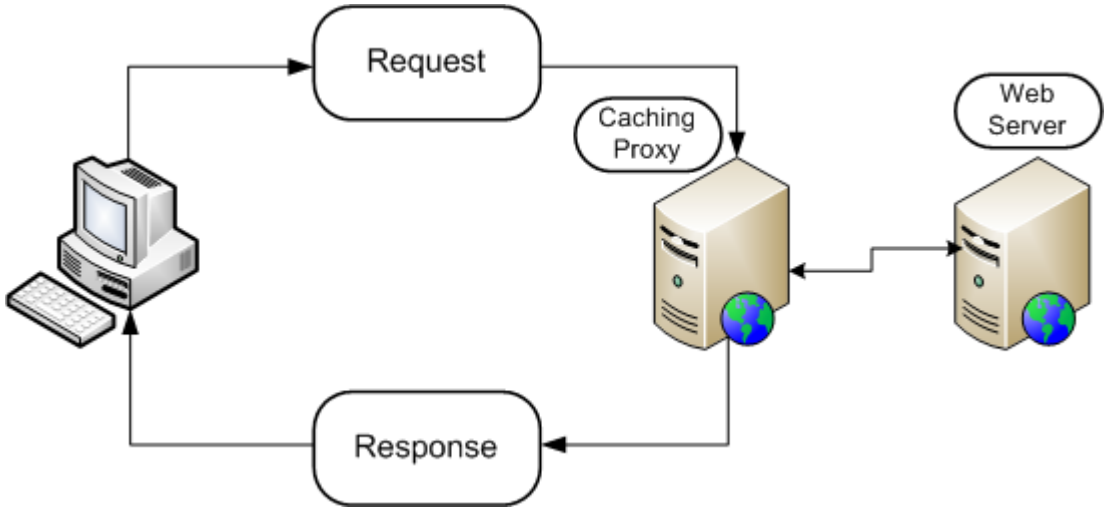
- 스레드 (Thread)란?
  - 프로그램 내에서 실행되는 하나의 실행 흐름 단위
- 멀티 스레딩 (Multi-Threading)
  - 동시에 실행시킬 수 있는 코드 단위를 독립적으로 생성된 스레드에 할당하여 실행하는 기능
- 스레드 풀
  - 스레드를 제한된 개수만큼 정해놓고 작업 큐에 들어오는 작업들을 하나씩 스레드에 할당하는 모듈



캐시 (Cache)

- 캐시란? (Wikipedia 정의)
  - 컴퓨터 과학에서 데이터나 값을 미리 복사해 놓는 임시 장소
  - 캐시의 접근 시간에 비해 원래 데이터를 접근하는 시간이 오래 걸리는 경우나 값을 다시 계산하는 시간을 절약하고 싶은 경우에 사용한다.

- 캐시 종류
  - CPU 캐시
  - 디스크 캐시
  - 웹 캐시

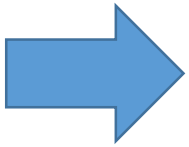


과제 1

- 앞서 설명한 스레드 풀과 캐시 예에서처럼 나머지 예에 대해서도 문제 상황을 설명해 보세요
  - 대화상자
  - 사용자 설정
  - 레지스트리 설정
  - 로그 기록 용 객체
  - 디바이스 드라이버

객체 생성

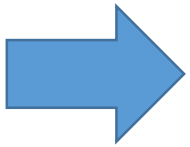
```
class MyClass {
    public MyClass() {}
}
```



```
new MyClass();
new MyClass();
new MyClass();
```

객체 생성

```
class MyClass {
    private MyClass() {}
}
```



```
new MyClass();
```

가능하나요??

객체 생성

```
class MyClass {
    private MyClass() {}

    public static MyClass getInstance() {
      return new MyClass();
    }
}
```

고전적인 싱글턴 패턴 구현

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

        private Singleton() {}

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

        // other useful methods here
        public String getDescription() {
                return "I'm a classic Singleton!";
        }
}
```

초콜릿 공장

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
        System.out.println("Creating unique instance of Chocolate Boiler");
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
                uniqueInstance = new ChocolateBoiler();
        } else
                System.out.println("Returning instance of Chocolate Boiler");
        return uniqueInstance;
    }
```

https://github.com/kwanulee/DesignPattern/blob/master/singleton/chocolate/src/hansung/designpatterns/singleton/chocolate/ChocolateBoiler.java

초콜릿 공장

```
    public void fill() {
        if (isEmpty()) {
                // fill the boiler with a milk/chocolate mixture
                empty = false;
                boiled = false;
                        }
    }
    public void drain() {
        if (!isEmpty() && isBoiled()) {
                // drain the boiled milk and chocolate
                empty = true;
        }
    }
    public void boil() {
        if (!isEmpty() && !isBoiled()) {
                // bring the contents to a boil
                boiled = true;
        }
    }
    public boolean isEmpty() { return empty; }
    public boolean isBoiled() { return boiled; }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/singleton/chocolate/src/hansung/designpatterns/singleton/chocolate/ChocolateBoiler.java

초콜릿 공장 예제 실행

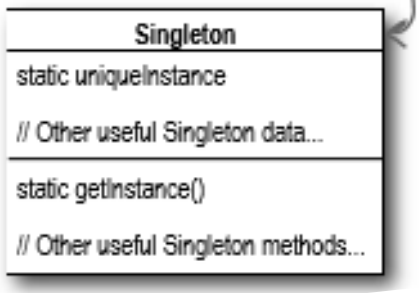
```
    public void fill() {
        if (isEmpty()) {
                // fill the boiler with a milk/chocolate mixture
                empty = false;
                boiled = false;
                        }
    }
    public void drain() {
        if (!isEmpty() && isBoiled()) {
                // drain the boiled milk and chocolate
                empty = true;
        }
    }
    public void boil() {
        if (!isEmpty() && !isBoiled()) {
                // bring the contents to a boil
                boiled = true;
        }
    }
    public boolean isEmpty() { return empty; }
    public boolean isBoiled() { return boiled; }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/singleton/chocolate/src/hansung/designpatterns/singleton/chocolate/ChocolateBoiler.java

싱글턴 패턴

- 정의
  - 싱글턴 패턴은 해당 클래스의 인스턴스가 하나만 만들어지고, 어디서든지 그 인스턴스에 접근할 수 있도록 하기 위한 패턴입니다.

getIntance() 메소드는 정적 메소드 (클래스 메소드)로서, 언제 어디서든 이 메소드를 호출 (Singleton.getInstance())할 수 있고, 게으른 인스턴스 생성을 활용할 수 있는 장점을 제공



uniqueInstance 클래스 변수에 싱글턴의 유일무이한 인스턴스가 저장됨

이 클래스는 일반적인 클래스를 만들 때와 마찬가지로 다양한 데이터와 메소드를 사용할 수 있음.

멀티 스레드 문제

```
public class ChocolateController {
    public static void main(String args[]) {

        Thread thread1 = new Thread() {
            public void run() {
                ChocolateBoiler boiler = ChocolateBoiler.getInstance();
                …
            }
        };

        Thread thread2 = new Thread() {
            public void run() {
                ChocolateBoiler boiler = ChocolateBoiler.getInstance();
                …
            }
        };

        thread1.start();
        thread2.start();
    }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/singleton/chocolate/src/hansung/designpatterns/singleton/chocolate/ChocolateController.java

문제가 발생하는 상황은?

```
public static ChocolateBoiler getInstance() {
1.          if (uniqueInstance == null) {
2.                  uniqueInstance = new ChocolateBoiler();
3.          } else
4.                  System.out.println("Returning instance ...");
5.          return uniqueInstance;
    }
```

	thread1	thread2
Step1		
Step2		
Step3		
Step4		
Step5		
Step6		
Step7		
Step8		
Step9		
Step10		

한 step에 오직 하나의 스레드만 코드를 수행할 수 있다.

문제가 발생하는 상황은?

```
public static ChocolateBoiler getInstance() {
1.          if (uniqueInstance == null) {
2.                      uniqueInstance = new ChocolateBoiler();
3.          } else
4.                  System.out.println("Returning instance ...");
5.          return uniqueInstance;
    }
```

	thread1	thread2
Step1	1	
Step2		1
Step3	2	
Step4	3	
Step5		2
Step6		3
Step7		5
Step8	5	
Step9		
Step10		

한 step에 오직 하나의 스레드만 코드를 수행할 수 있다.

해결방안 1 (Synchronized)

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
        System.out.println("Creating unique instance of Chocolate Boiler");
    }

    public static synchronized ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
                uniqueInstance = new ChocolateBoiler();
        } else
                System.out.println("Returning instance of Chocolate Boiler");
        return uniqueInstance;
    }
```

동기화가 필요한 시점은 메소드가 시작되는 때일 뿐임. 이외에는 동기화는 불필요한 오버헤드 (성능 100배 저하) 를 유발함

https://github.com/kwanulee/DesignPattern/blob/master/singleton/chocolate_threadsafe/src/hansung/designpatterns/singleton/chocolate/ChocolateBoiler.java

해결방안 1 (Synchronized) 실행하기

- 해결방안 1 (Synchronized) 예제 프로젝트 링크
  - https://github.com/kwanulee/DesignPattern/tree/master/singleton/chocolate_threadsafe

- 실행결과

```
Creating unique instance of Chocolate Boiler
elapse =1
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
```

해결방안 2 (클래스 로딩시 생성)

인스턴스가 사용되기 전부터 리소스를 차지.

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    private static ChocolateBoiler uniqueInstance = new ChocolateBoiler();

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
        System.out.println("Creating unique instance of Chocolate Boiler");
    }

    public static ChocolateBoiler getInstance() {
        System.out.println("Returning instance of Chocolate Boiler");
        return uniqueInstance;
    }
```

해결방안 2 (클래스 로딩시 생성) 실행하기

- 해결방안 2 (클래스 로딩시 생성) 예제 프로젝트 링크
  - https://github.com/kwanulee/DesignPattern/tree/master/singleton/chocolate_static

- 실행결과

```
Creating unique instance of Chocolate Boiler
elapse =1
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
elapse =0
```

해결방안 3 (Double-Checking Locking)

uniqueInstance의 변경이
다중 쓰레드에게 올바로 보이도록 보장

처음에만 동기화를 해서
동기화 오버헤드 줄임.

```
public class ChocolateBoiler {
    …
    private volatile static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        …
        System.out.println("Creating unique instance of Chocolate Boiler");
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            synchronized (ChocolateBoiler.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new ChocolateBoiler();
                } else
                    System.out.println("Returning instance..");
            }
        } else
            System.out.println("Returning instance …");
        return uniqueInstance;
    }
```

Volatile 의미: http://thswave.github.io/java/2015/03/08/java-volatile.html

https://github.com/kwanulee/DesignPattern/blob/master/singleton/chocolate_dcl/src/hansung/designpatterns/singleton/chocolate/ChocolateBoiler.java

해결방안 3 (DCL)실행하기

- 해결방안 3 (DCL) 예제 프로젝트 링크
  - https://github.com/kwanulee/DesignPattern/tree/master/singleton/chocolate_dcl

- 실행결과

```
Creating unique instance of Chocolate Boiler
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =1
elapse =3
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
elapse =0
Returning instance of Chocolate Boiler
Returning instance of Chocolate Boiler
elapse =1
elapse =0
Returning instance of Chocolate Boiler
Returning instance of Chocolate Boiler
Returning instance of Chocolate Boiler
elapse =0
```

핵심 정리

- 어떤 클래스에 싱글턴 패턴을 적용하면 애플리케이션에 그 클래스의 인스턴스가 최대 한 개까지만 있도록 할 수 있습니다.
- 싱글턴 패턴을 이용하면 유일한 인스턴스를 어디서든지 접근할 수 있도록 할 수 있습니다.
- 자바에서 싱글턴 패턴을 구현할 때는 private 생성자와 정적 메소드, 정적 변수를 사용합니다.
- 다중 스레드를 사용하는 애플리케이션에서는 속도와 자원 문제를 파악해보고 적절한 구현법을 사용해야 합니다.