Introduction and Singleton

Lab #13

Contents

- **Introduction to Design Patterns**
  - What is a Design Pattern
  - Why Study Design Patterns
  - The Gang of Four
- **The Singleton Pattern**
  - Logger Example
  - Lazy instantiation
  - Singleton vs. Static Variables
  - Threading: Simple, Double-Checked, Eager Initialization
  - Lab
  - Inheritance
  - Add Registry to Singleton
  - Dependencies
  - Unit Testing
  - Articles

What is a Design Pattern

- **A problem that someone has already solved**
- **A model or design to use as a guide**

- **More formally: "A proven solution to a common problem in a specific context"**

- **Real World Examples**
  - Blueprint for a house
  - Manufacturing

Why Study Design Patterns?

- **Provides software developers a toolkit for handling problems that have already been solved**
- **Provides a vocabulary that can be used amongst software developers**
  - The Pattern Name itself helps establish a vocabulary
- **Helps you *think* about how to solve a software problem**

The Gang of Four

- **"Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides**
- **Defines a Catalog of different design patterns**
- **Three different types**
  - Creational: "Creating objects in a manner suitable for the situation"
  - Structural: "Ease the design by identifying a simple way to realize relationship between entities
  - Behavioral: "Common communication patterns between objects"

The Gang of Four: Pattern Catalog

Creational
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

How Design Patterns Solve Design

- **Finding Appropriate Objects**
- **Determine Object Granularity**
- **Specify Object Interfaces**
- **Specifying Object Implementations**
  - Programming to an interface, not an implementation

    interface에 맞춰 implements 하는것.
- **Encourage Reusability**
  - Inheritance vs. Composition
- **Support Extensibility**
  - Frameworks

Reality

- **Problems with design early on**
  - It is sometimes very hard to "see" a design pattern
  - Not all requirements are known
  - A design that is applicable early on becomes obsolete
  - "Paralysis by Analysis"

    ↳ 개발X 분석만 O
- **Due to these realities, refactoring is inevitable!**
- **Question: How do you mitigate the fact that you won't have all of the design figure out?**

Common Pitfall

- "I just learned about Design Pattern XYZ, Let's use it!"
- Reality: If you are going to use a Design Pattern, you should have a reason to do so
- The *software requirements* should really drive why you are going to use (or not use) a Design Pattern

Example: Logger

- **What is wrong with this code?**

```
public class Logger {

    public Logger() {}

    public void LogMessage() {
        //Open File "log.txt"
        //Write Message
        //Close File
    }
}
```

Example: Logger (Contd)

- Since there is an external Shared Resource ("log.txt"), we want to closely control how we communicate with it
- We shouldn't create an object of the Logger class every time we want to access this Shared Resource. Is there any reason for that?
- We need ONE (단일 객체 만이 필요로 한다.)

thread가 동시에 한자원을 접근할때 문제가 생김.

Singleton

- **GoF Definition: "The Singleton Pattern ensures a class has only *one instance*, and provides a global point of access to it."**

- **Best Uses**
  - Logging
  - Caches
  - Registry Settings
  - Access External Resources
    - Printer
    - Device Driver
    - Database

shared => 동시에 접근하는 경우 설계 수
=> 싱글톤으로 단일객체 생성

Logger – as a Singleton (고전적 방식)

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

    private static Logger uniqueInstance;

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

        return uniqueInstance;
    }
}
```

Lazy Instantiation

객체보다

필요할 때만 구체화 생성

- Objects are only created, when it is needed
- Helps control that we've created the Singleton just once
- If it is resource intensive to set up, we want to do it once

* 문제
1. 객체가 무조건 한번 생성됨 (쓰든, 안쓰든)
2. multi thread 문제

Singleton vs. Static Variables

- What if we had *not* created a Singleton for the Logger class?

- Let's pretend the *Logger()* constructor did a lot of setup
- In our main program file, we had this code:

Logger MyGlobalLogger = new Logger();

문제 1.

- All of the Logger setup will occur regardless if we ever need to log or not

Threading

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

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

                return uniqueInstance;
        }
}
```

문제2

What would happen if two different threads accessed this line at the same time?

객체 생성중 다른 thread가 들어와 null로 인식해, 또 객체 생성

Threading (Contd)

```
private Singleton() {}

private static Singleton uniqueInstance;

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

Thread 1

```
private Singleton() {}

private static Singleton uniqueInstance;

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

Thread 2

Option #1: Simple Locking

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

이미 객체가 생성되어 있는데 null인지 확인하려고 계속 lock,

오버헤드↑

Option #1: Simple Locking 2

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

Option #2: DCL (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;
        }
}
```

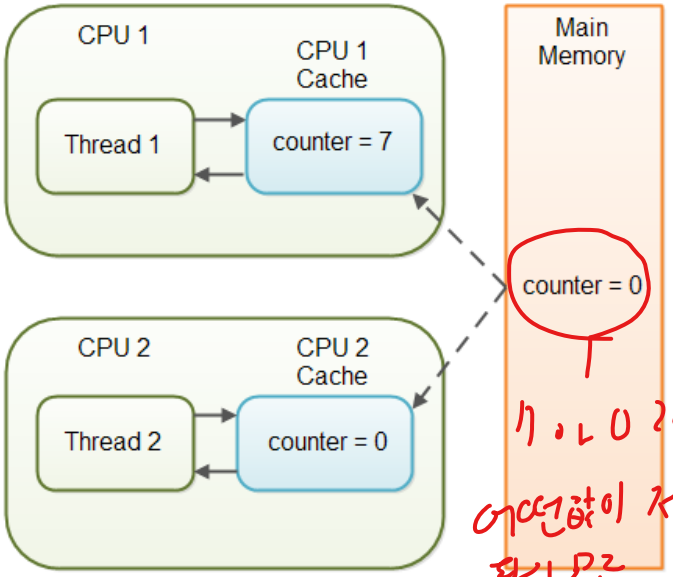
처음으로 null 인지 판단

그 다음 lock

volatile Variable (main 메모리에 cache 값 갱신)

↳ 나중에 실행되는 값이 main에 저장됨.

- Used to mark a Java variable as "being stored in main memory"
- Every read/write of a volatile variable is directly from/to main memory, not from/to the cache
- Guarantees visibility of changes to variables across threads



7 or 0 ??
어떤값이 저장될지 모름
=> 나중에 쓴 쓰레드 값으로 설정

Option #3: "Eager" Initialization

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

        private static Singleton uniqueInstance = new Singleton()

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

Runtime guarantees that this is thread-safe

1. Instance is created the first time any member of the class is referenced.
2. Good to use if the application always creates; and if little overhead to create.

Self-Test (1)

- 초콜릿 공장의 최신형 초콜릿 보일러를 제어하기 위한 클래스가 있다. 해당 클래스는 원활한 초콜릿 보일러 가동을 위해 세심한 주의를 기울여 작성되었다.
- 그럼에도 해당 클래스의 인스턴스가 2개 이상 생성되는 순간 여러 가지 문제가 발생할 수 있다.
- 다음 클래스를 인스턴스를 2개 이상 생성할 수 없도록 Singleton 클래스로 변경해야 한다.

Self-Test (1) (Contd)

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

    public  ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
```

This code is only started when the boiler is empty!

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags.

Self-Test (1) (Contd)

```
    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;
    }
}
```

To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled we set the boiled flag to true.

Self-Test (1) (Contd)

```
Problems  @ Javadoc  Dec
<terminated> ChocolateFactory (1)
Filling with mixture
Filling with mixture
Boiling the mixture
Boiling the mixture
Draining the mixture
Draining the mixture
```

```
Problems  @ Javadoc  De
<terminated> ChocolateFactory [Ja
Filling with mixture
Already filled
Boiling the mixture
Already boiled
Draining the mixture
Already drained
```

- Singleton 패턴이 적용되지 않은 코드는 한 객체가 이미 수행한 동작을 다른 객체가 그대로 수행하는 것을 볼 수 있다.
- Singleton 패턴이 적용된 코드는 한 객체가 이미 수행한 동작을 다른 객체가 수행하지 않는 것을 볼 수 있다.

Self-Test (2)

- Self-Test (1)에서 작성했던 Singleton 디자인 패턴을 적용한 초콜릿 보일러가 멀티쓰레딩 최적화 적용 시 문제가 발생할 수 있음을 확인했다.
- 멀티쓰레딩 최적화를 적용해도 문제가 발생하지 않도록 초콜릿 보일러 클래스를 다음과 같은 DCL 방식으로 수정할 것