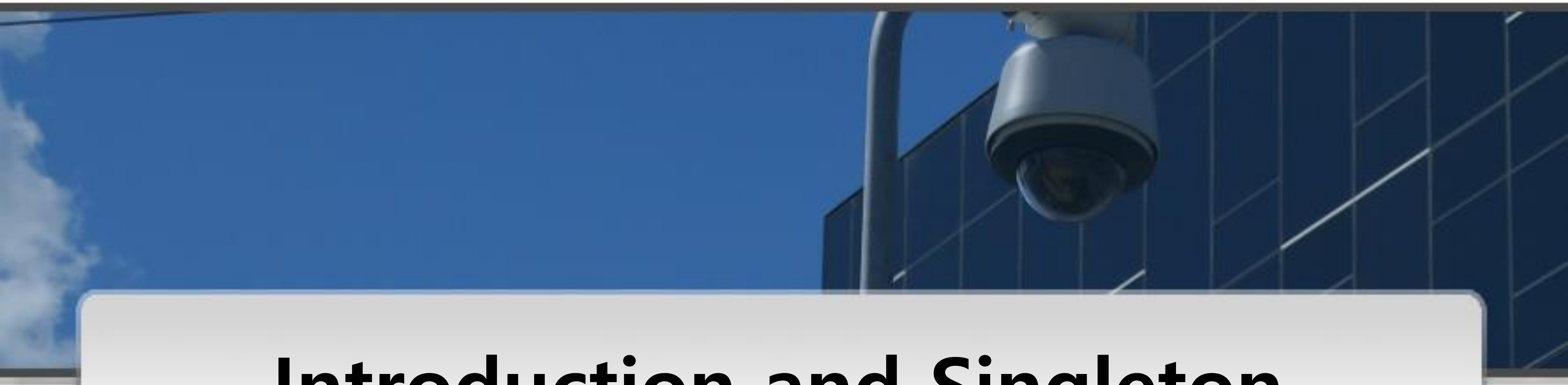


Introduction and Singleton

Lab #13

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

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

Logger – as a Singleton

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

    private static Logger uniqueInstance();

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

            return uniqueInstance;
    }
}
```

Lazy Instantiation

- 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

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:

```
public static Logger MyGlobalLogger = new Logger();
```

- 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;
        }
}
```

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

Threading (Contd)

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

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

                return uniqueInstance;
        }
}
```

Thread 1

```
public class Singleton
{
        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;
        }
}
```

Option #1: Simple Locking 2

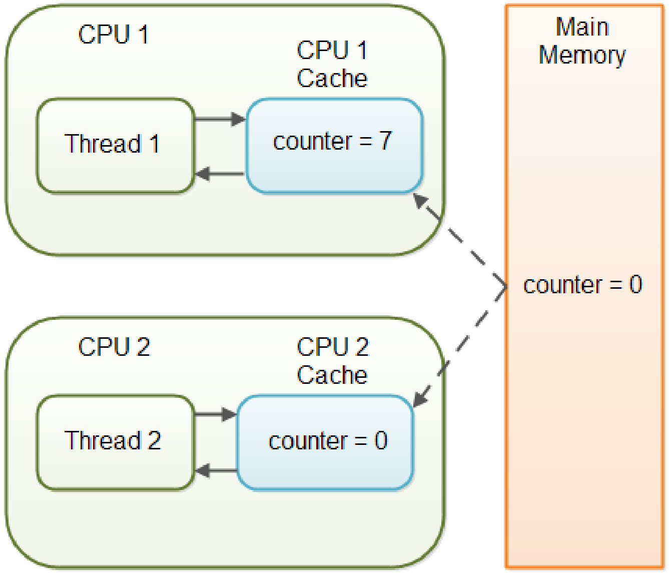
```
public class Singleton
{
        private Singleton() {}
        private static Singleton uniqueInstance;
        public static synchronized Singleton getInstance()
        {
                if (uniqueInstance == null) {
                        uniqueInstance = new Singleton();
                }
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

volatile Variable

- 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



Option #3: “Eager” Initialization

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

        private static Singleton uniqueInstance = new Singleton()

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

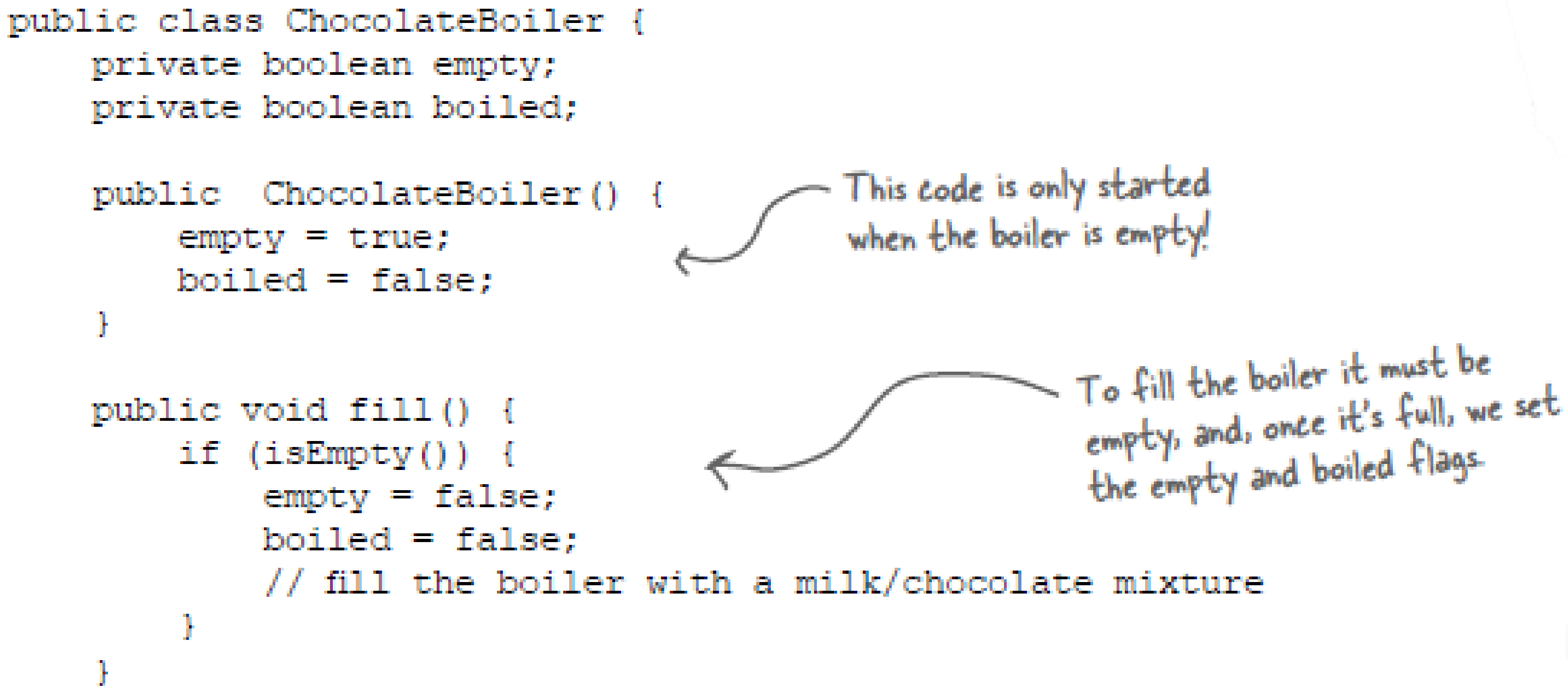
Runtime guarantees that this is thread-safe

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 나와 있다. 다음 코드는 원활한 초콜릿 보일러 가동을 위해 세심한 주의를 기울인 코드이다.
- 하지만, 해당 클래스의 인스턴스가 2개 이상 생성되는 순간 세심한 주의를 기울였음에도 여러 가지 문제가 발생할 수 있다.
- 다음 클래스를 인스턴스를 2개 이상 생성할 수 없도록 Singleton 클래스로 변경해야 한다.

Self-Test (1) (Contd)

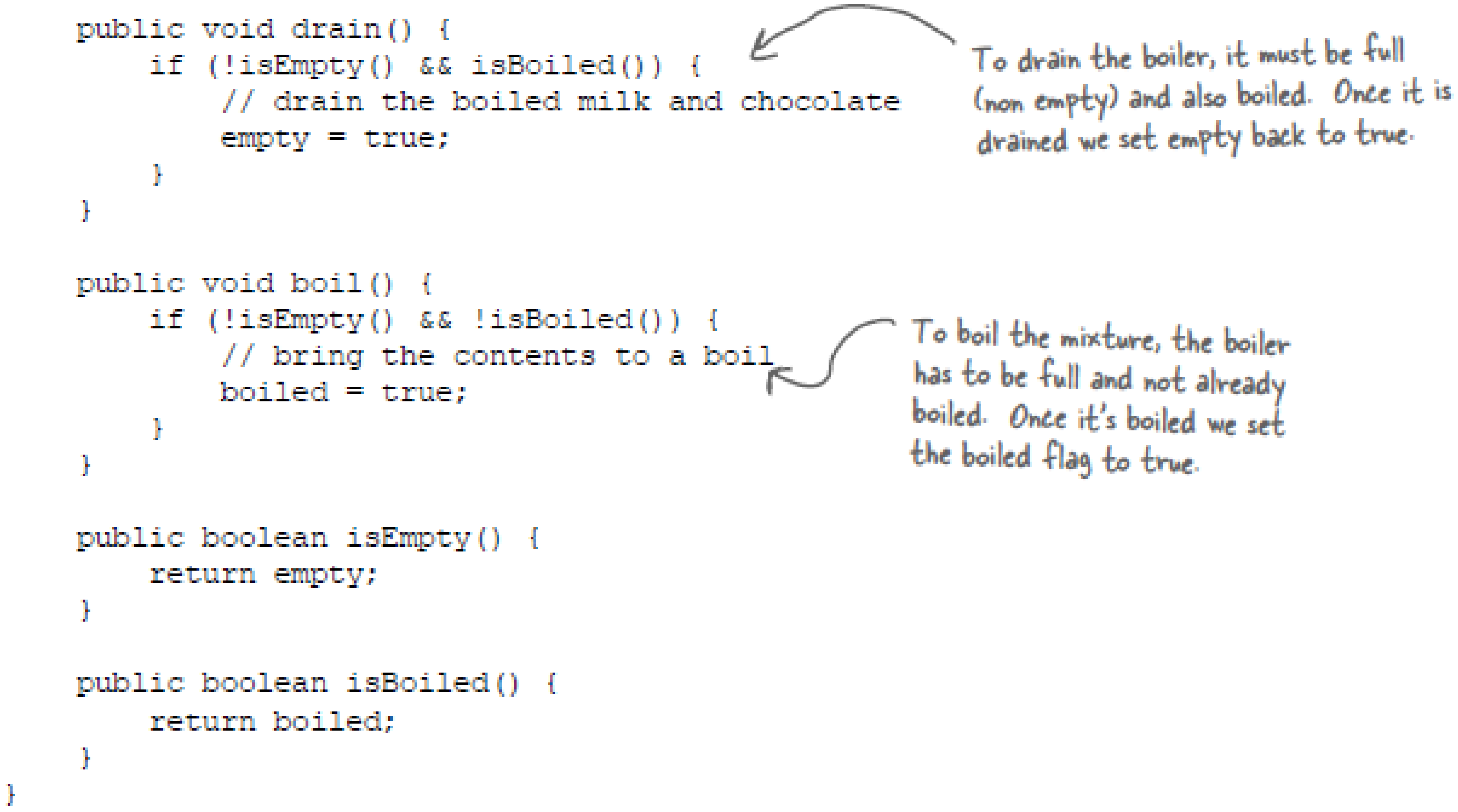


```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    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)

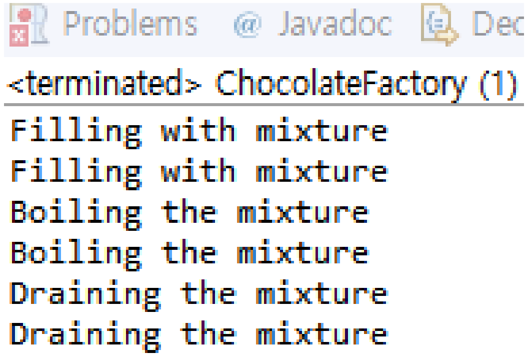


```
    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

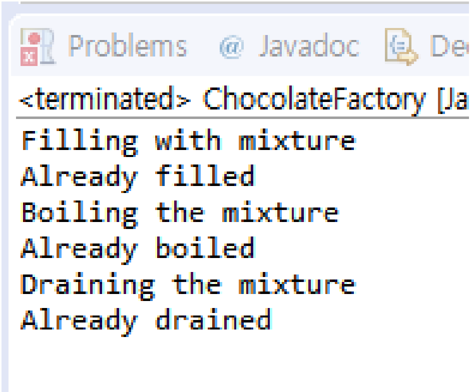
To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

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)



```
Filling with mixture
Filling with mixture
Boiling the mixture
Boiling the mixture
Draining the mixture
Draining the mixture
```



```
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