Head First Design Patterns

by Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates

ch05. Singleton Pattern

The Singleton Pattern

- **Purpose**
  - Ensures that only one instance of a class is allowed within a system.

- **Use When**
  - Exactly one instance of a class is required.
  - Controlled access to a single object is necessary.

One of a Kind Objects

- **One and only one**
  - Window Manager, Printer Spooler, Thread Pools, Caches, Logging, Factory
  - We want to instantiate them only once
- **Why is it difficult after all?**
  - How about global variables?
  - Multi-threading issue

Usage of Singleton

```
public class AnyClientProgram {
    // Singleton s = new Singleton();
    Singleton s = Singleton.getInstance();
}
```

Clients are not allowed to use new operator!

The Skeleton of Singleton

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

    // other useful instance variables

    private Singleton() {}

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

    // other useful methods
}
```

Beware of Concurrency!!

Design Points

- **Make the constructor be private**
  - `private Singleton() {}`
  - Private constructor ?
  - Otherwise …
- **Provide a getInstance() method**
  - `public static Singleton getInstance()`
  - Should it be static?
  - Why public ?
- **Remember the instance once you have created it**
  - private static Singleton uniqueInstance

  -
    ```
    if (uniqueInstance == null)
        uniqueInstance = new Singleton();
    ```

Exercise: Applying Singleton

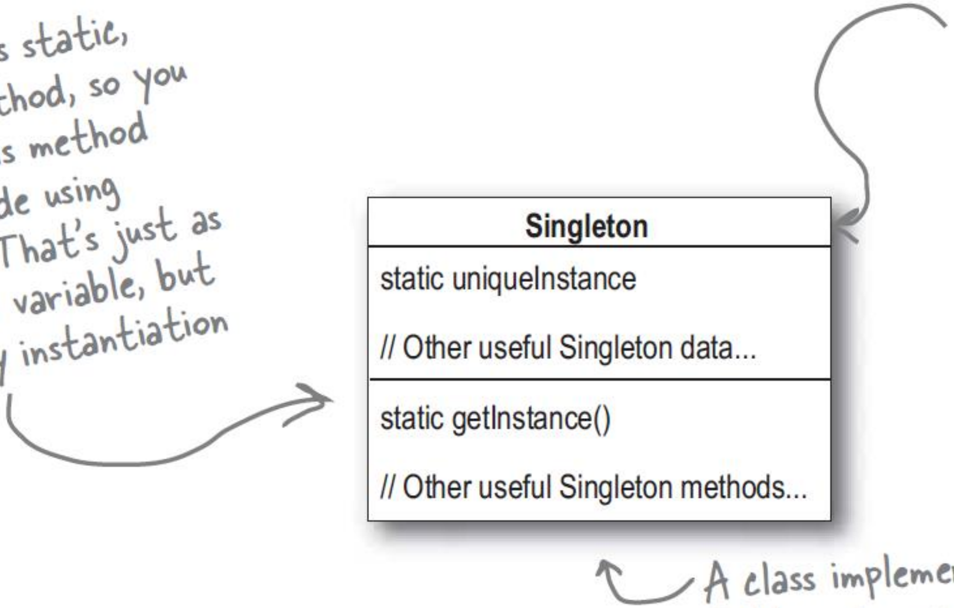
```
public class ChocolateBoiler {
  private boolean empty;
  private boolean boiled;


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







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

Solution

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

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

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

}
```

Class Diagram

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



Singleton

static uniqueInstance

// Other useful Singleton data...

static getInstance()

// Other useful Singleton methods...

The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Using Singleton on Multi-threads

- **Thread One and Thread Two executes**

```
ChocolateBoiler boiler = ChocolateBoiler.getInstance();
boiler.fill();
boiler.boil();

boiler.drain();
```

What happened?

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

Make sure you check your answer on page 188 before turning the page!

Thread One	Thread Two	Value of uniqueInstance

BE the JVM

Thead One	Thead Two	Value of uniqueInstance
`public static ChocolateBoiler getInstance() {`		null
	`public static ChocolateBoiler getInstance() {`	null
`if (uniqueInstance == null) {`		
	`if (uniqueInstance == null) {`	null
`uniqueInstance = new ChocolateBoiler();`		<object1>
`return uniqueInstance;`		<object1>
	`uniqueInstance = new ChocolateBoiler();`	<object2>
	`return uniqueInstance;`	<object2>

Uh oh, this doesn't look good!

Two different objects are returned! We have two ChocolateBoiler instances!!!

Solving the Problem (Option 1)

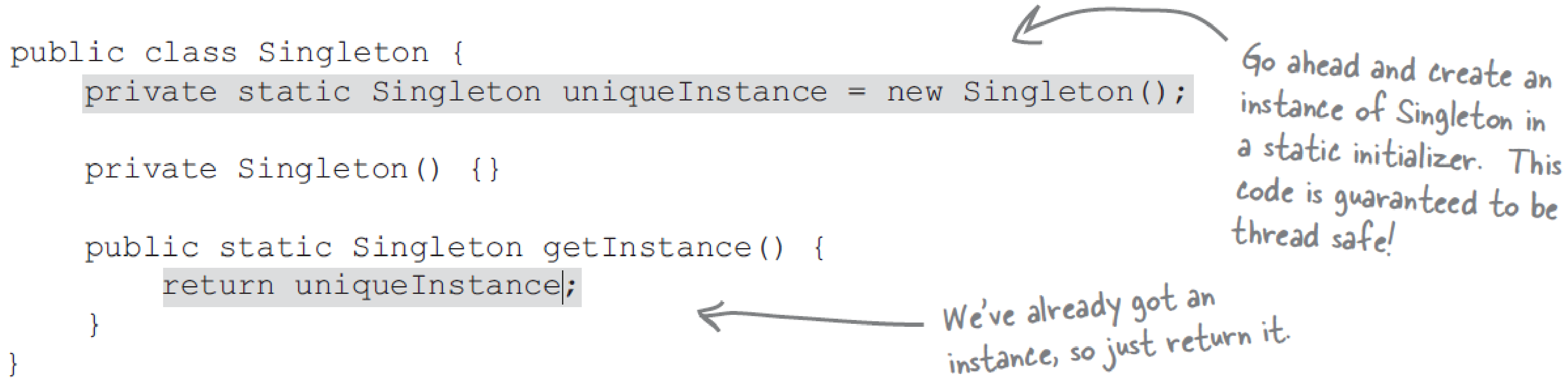
```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

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

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

Isn't this <u>too much locking</u>?

Can we improve multithreading?

Solving the Problem (Option2)



```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

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

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

Remind the <u>overhead</u> of creating Singleton instance always

Developing Idea

1. Check that the variable is initialized (without obtaining the lock). If it is initialized, return it immediately.
2. Obtain the lock.
3. **Double-check** whether the variable has already been initialized: if another thread acquired the lock first, it may have already done the initialization. If so, return the initialized variable.
4. Otherwise, initialize and return the variable.

Then, how about this?

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

    // other useful instance variables

    private Singleton() {}

        public static Singleton getInstance() {
            if (uniqueInstance == null) {
                synchronized(Singleton.class) {
                    if (uniqueInstance == null)
                        uniqueInstance = new Singleton();
                }
            }
            return uniqueInstance;
    }
    // other useful methods
}
```

Double-Checked Locking

- 이전 알고리즘은 효율적인 해결책으로 보임. 하지만 민감한 문제를 가지고 있으며 이를 피해야 함.

- 문제점
  - 위 시나리오에서 스레드 A는 uniqueInstance 변수의 값이 null 인지 검사한 후 lock을 얻어 uniqueInstance 의 값을 초기화하기 시작함(new Singleton( ))
  - 어떤 프로그래밍 언어 환경에서는 컴파일러가 uniqueInstance 변수의 초기화가 완료되지 않은 상황(즉, Singleton 객체 생성이 완료되지 않은 상황)에서 다음을 진행하도록 코드 최적화를 함
  - CPU를 얻은 스레드 B는 uniqueInstance의 값이 초기화 된 것으로 생각하고 uniqueInstance의 값을 얻어서 자신의 일을 진행함
  - 결국 스레드 B는 초기화가 완료되지 않은 객체를 얻어서 일을 하게 되어 프로그램이 crash 될 수 있음
  - 이 상황을 방지해야 함
  - 자바에서는 volatile이라는 키워드를 이용함

Corrected Double-Checked Locking (Option 3)

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

    private Singleton() {}

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

Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

* The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

Using volatile keyword

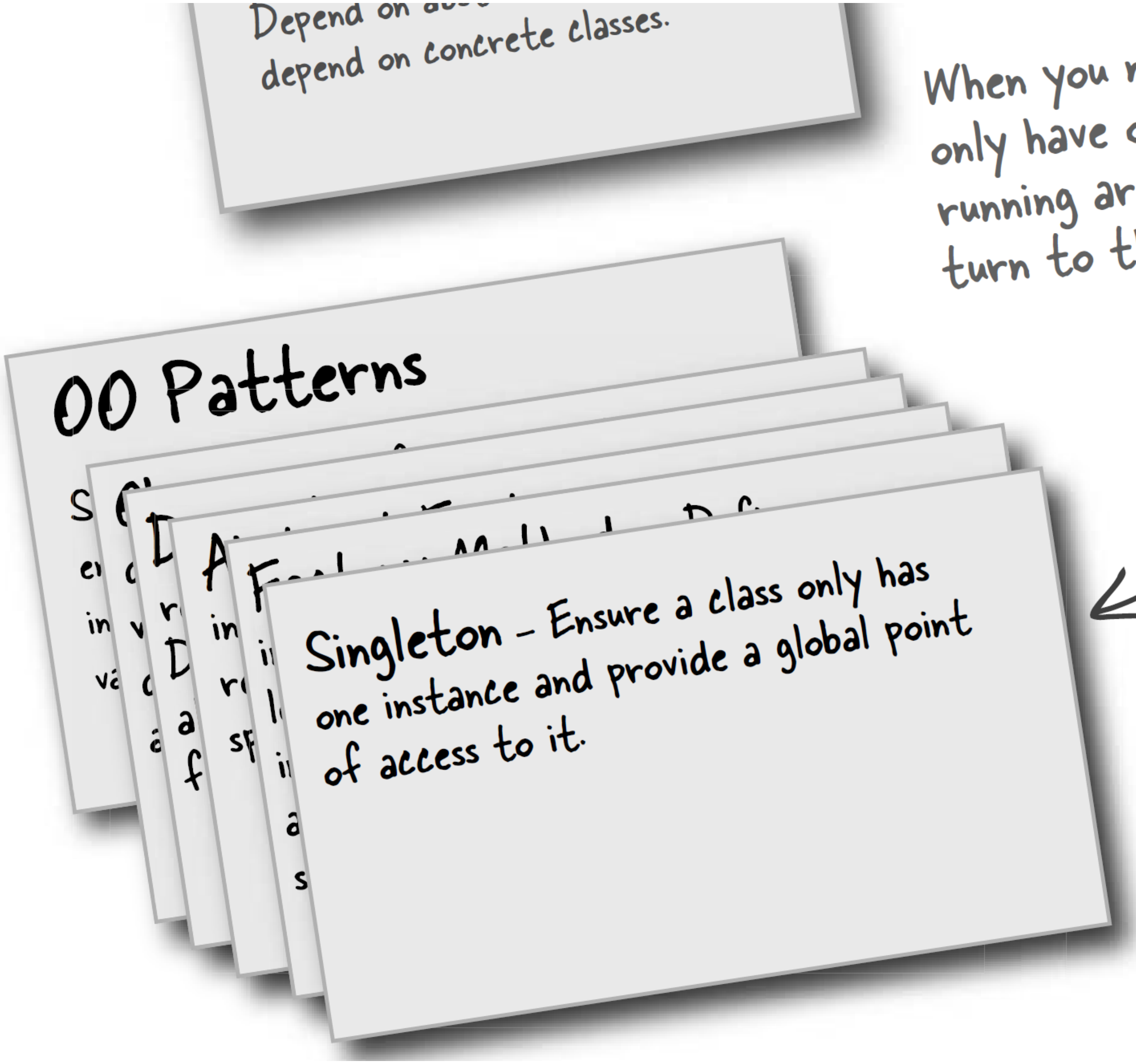
스레드 A는 uniqueInstance 변수가 null 임을 확인하고 lock을 얻은 후 초기화를 진행함

초기화 시 객체(Singleton 객체) 생성이 완료된 후에야 uniqueInstance 변수가 이 객체를 가리키도록 함

이후, 스레드 B는 uniqueInstance 변수가 null이 아니므로 lock을 얻지 않고 바로 uniqueInstance 변수에 담긴 Singleton 객체(완변하게 생성을 마친 Singleton 객체)를 사용함

Reviewing the Options

- Synchronize the getInstance() method (Option 1):
  - A straightforward technique that is guaranteed to work. It causes small impact on run-time performance due to frequent locking.
- Use eager instantiation (Option 2):
  - In case we are always going to instantiate the class, then statically initializing the instance would cause no concerns.
- Double checked locking (Option 3):
  - A perfect solution w.r.t performance. However, double-checked locking may be overkill in case we have no performance concerns. In addition, we'd have to ensure that we are running at least Java 5



Singleton – Ensure a class only has one instance and provide a global point of access to it.

When you need to ensure you only have one instance of a class running around your application, turn to the Singleton.

Related Patterns

- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Facade objects are often Singletons because only one Facade object is required.
- State objects are often Singletons.

Discussion

- The Singleton design pattern is one of the most inappropriately used patterns. Designers frequently use Singletons in a misguided attempt to replace global variables. The Singleton does not do away with the global, it merely renames it.