Agenda

- Singleton Pattern

Singleton Pattern

Private Constructor

```
public MyClass {

    private MyClass() {}

}
```

Static Method

```
public MyClass {

    public static MyClass getInstance() {
    }
}
```

Static Method Creating an Instance each time it is called

```
public MyClass {

    private MyClass() {}

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

Singleton Class

Let's rename MyClass to Singleton.

We have a static variable to hold our one instance of the class Singleton.

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

    // other useful instance variables here

    private Singleton() {}

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

    // other useful methods here
}
```

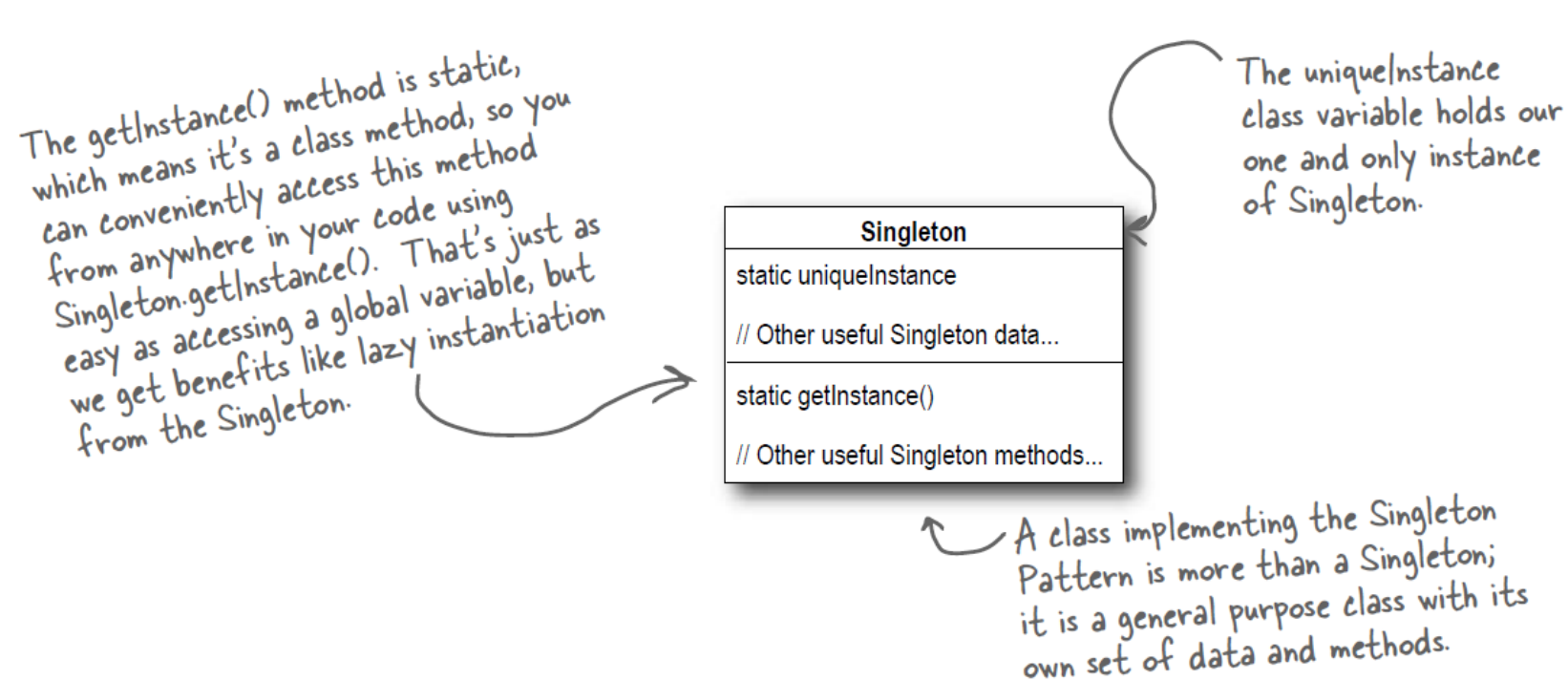
Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

The Singleton Pattern

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

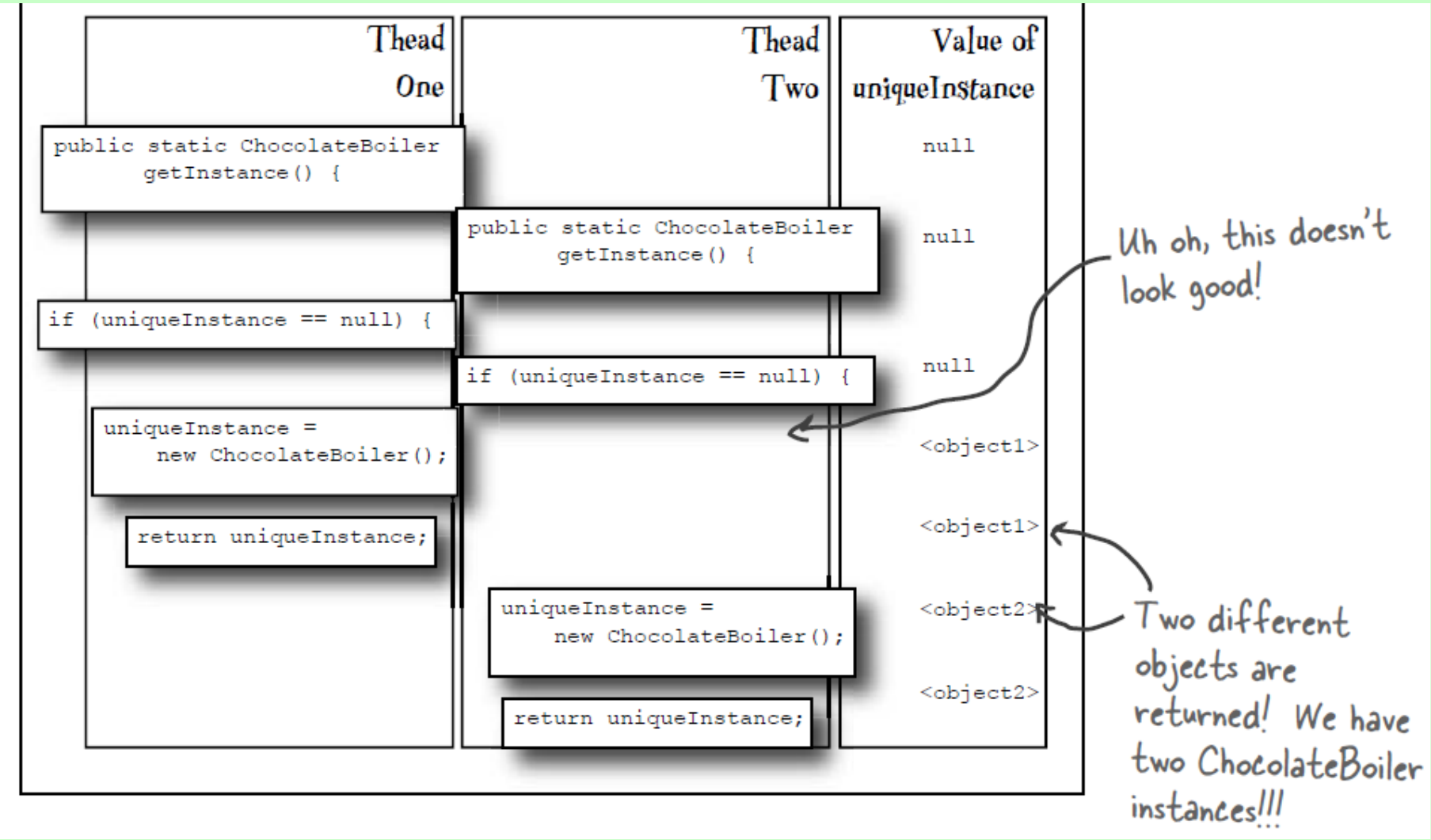


we have a problem...

- We have two threads, each executing this code. Your job is to play the JVM and determine whether there is a case in which two threads might get ahold of different boiler objects.

```
public static ChocolateBoiler
        getInstance() {

    if (uniqueInstance == null) {

        uniqueInstance =
            new ChocolateBoiler();

    }

    return uniqueInstance;

}
```

we have a problem...



Dealing with multithreading

```
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.

An eagerly created instance rather than a lazily created one

```
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.

double-checked locking

```
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.

References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
  First Edition October 2004
  ISBN 10: 0-596-00712-4
- http://www.csee.wvu.edu/classes/cs210/Fall_2006/CS_210_Sept_26.ppt