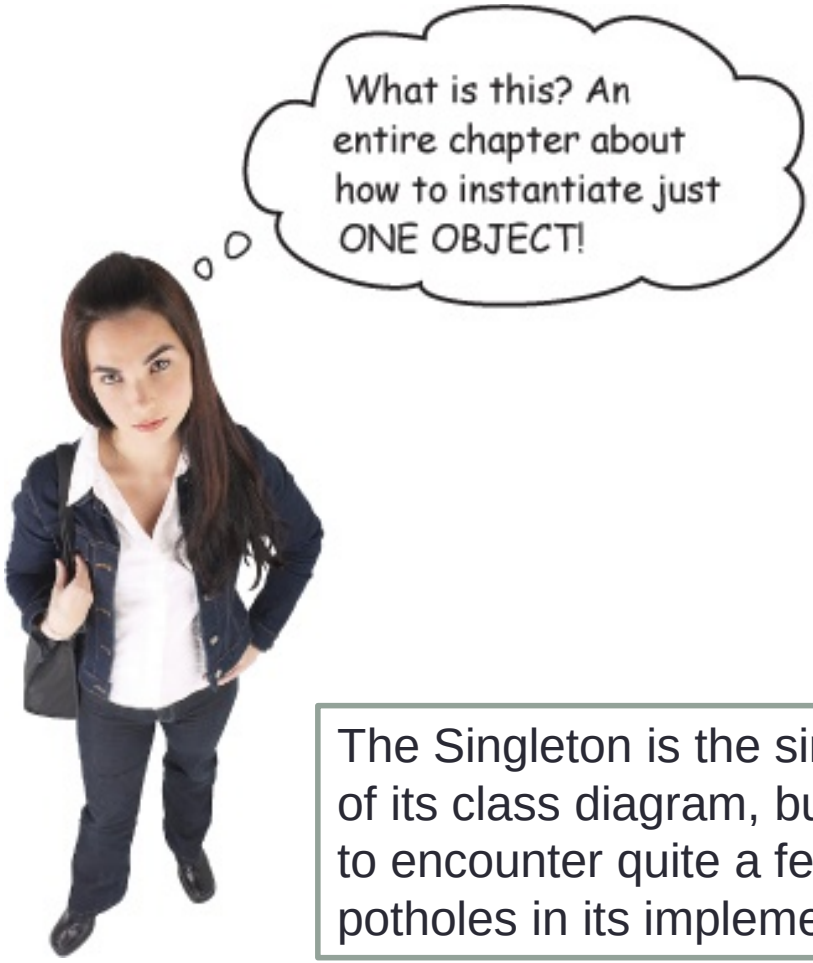
THE SINGLETON PATTERN

Week 4-1

Chandan R. Rupakheti

One of a kind objects



The Singleton is the simplest in terms of its class diagram, but we are going to encounter quite a few bumps and potholes in its implementation!

The Classic Singleton Pattern

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.

Lazy Initialization

uniqueInstance holds our ONE instance; remember, it is a static variable.

If uniqueInstance is null, then we haven't created the instance yet...

...and, if it doesn't exist, we instantiate Singleton through its private constructor and assign it to uniqueInstance. Note that if we never need the instance, it never gets created; this is lazy instantiation.

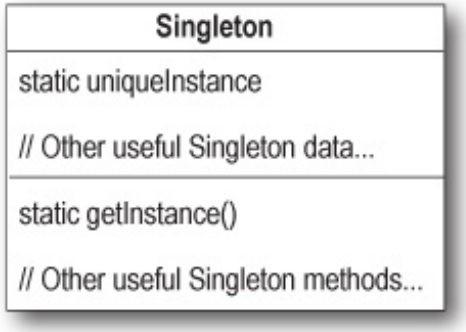
```
if (uniqueInstance == null) {
    uniqueInstance = new Singleton();
}
return uniqueInstance;
```

If uniqueInstance wasn't null, then it was previously created. We just fall through to the return statement.

By the time we hit this code, we have an instance and we return it.

The Singleton Pattern Defined

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.



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.

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

The Chocolate Factory

```
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
        }
    }

    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;
    }
}
```

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.

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.

Q2

Let's code a white-board version of singleton ChocolateBoiler and see if there are any problems with the implementation?

Hershey, PA we have a problem …

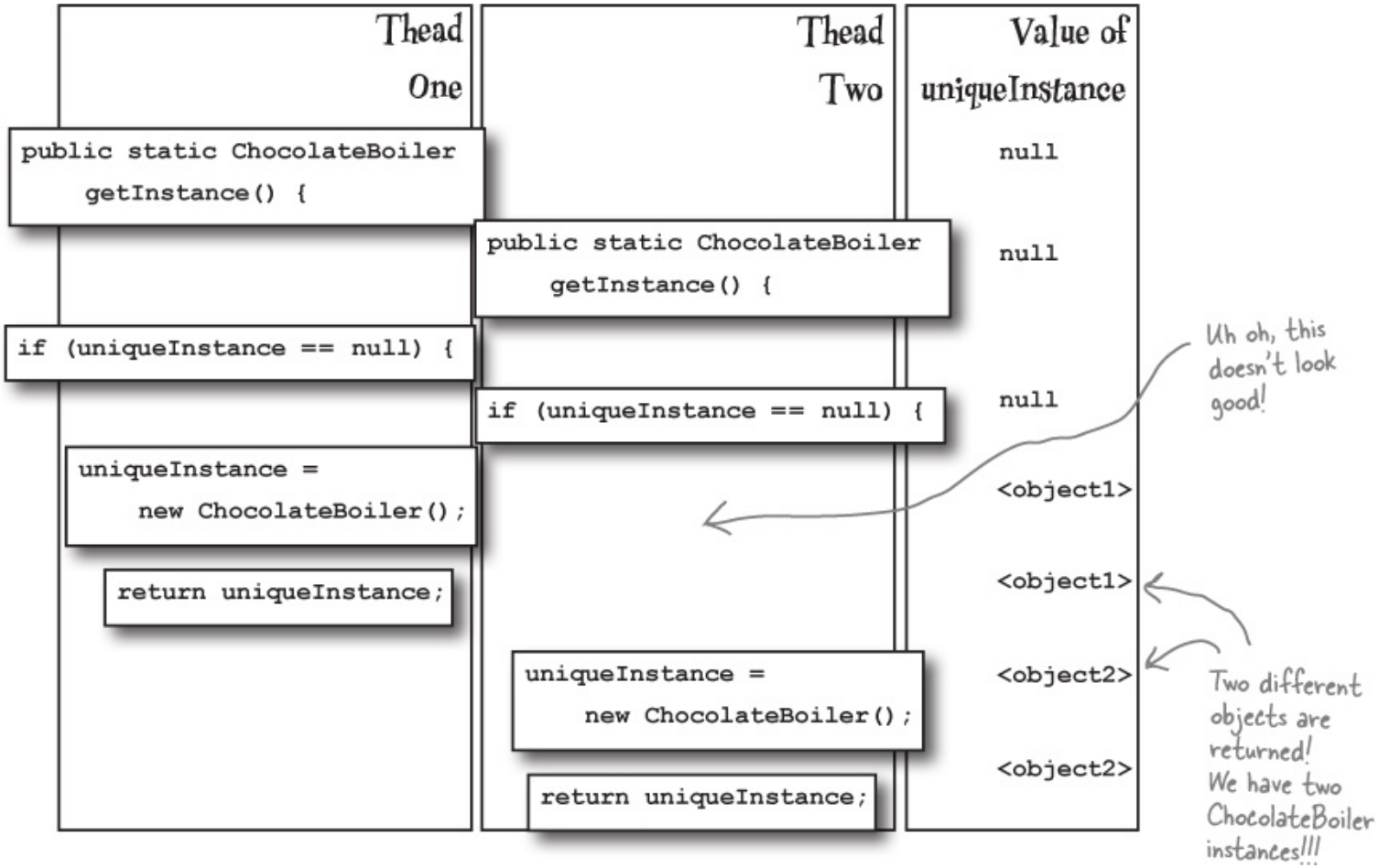
We don't know what happened! The new Singleton code was running fine. The only thing we can think of is that we just added some optimizations to the Chocolate Boiler Controller that makes use of multiple threads.

Could the addition of threads have caused this?

Isn't it the case that once we've set the uniqueInstance variable to the sole instance of ChocolateBoiler, all calls to getInstance() should return the same instance?

Be the JVM

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



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.

Can we improve multithreading? 1/2

1

Do nothing if the performance of getInstance() isn't critical to your application.

Keep in mind that synchronizing a method can decrease performance by a factor of 100, so revaluate this decision if needed.

2

Use eager static instantiation

```
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.

Here, we rely on the JVM to create the unique instance of the Singleton when the class is loaded. The JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable.

Can we improve multithreading? 2/2

3 Use 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.

With double-checked locking, we first check to see if an instance is created, and if not, then we synchronize.

Recap

The Singleton Pattern ensures you have at most one instance of a class in your application.

The Singleton Pattern also provides a global access point to that instance.

Be careful if you are using multiple threads or multiple class loaders; this could defeat the Singleton implementation and result in multiple instances.