

Part I
05. The Singleton Pattern: One of a Kind Objects

The Little Singleton

How would you create a single object?

```
new MyObject();
```

And, what if another object wanted to create a MyObject? Could it call new on MyObject again?

Yes, of course.

So as long as we have a class, can we always instantiate it one or more times?

Yes. Well, only if it's a public class.

And if not?

Well, if it's not a public class, only classes in the same package can instantiate it. But they can still instantiate it more than once.

Hmm, interesting.

Did you know you could do this?

No, I'd never thought of it, but I guess it makes sense because it is a legal definition.

```
public MyClass {

   private MyClass() {}

}
```

The Little Singleton (Cont.)

What does it mean?

I suppose it is a class that can't be instantiated because it has a private constructor.

Well, is there ANY object that could use the private constructor?

Hmm, I think the code in MyClass is the only code that could call it. But that doesn't make much sense.

Why not ?

Because I'd have to have an instance of the class to call it, but I can't have an instance because no other class can instantiate it. It's a chicken and egg problem: I can use the constructor from an object of type MyClass, but I can never instantiate that object because no other object can use "new MyClass()".

Okay. It was just a thought.

What does this mean?

MyClass is a class with a static method. We can call the static method like this:

```
MyClass.getInstance();
```

```
public MyClass {

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

The Little Singleton (Cont.)

Why did you use MyClass, instead of some object name?

Well, getInstance() is a static method; in other words, it is a CLASS method. You need to use the class name to reference a static method.

Very interesting. What if we put things together.

Now can I instantiate a MyClass?

Wow, you sure can.

```
public MyClass {

    private MyClass() {}

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

So, now can you think of a second way to instantiate an object?

```
MyClass.getInstance();
```

Can you finish the code so that only ONE instance of MyClass is ever created?

Yes, I think so...

Dissecting the Classic Singleton

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.

Dissecting the Classic Singleton (Cont.)

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

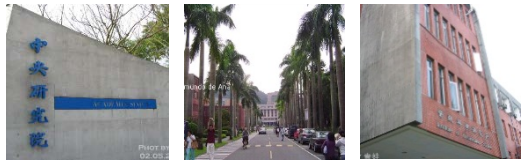
If uniqueInstance is null, then we haven't created the instance yet...

...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 MyClass();
}
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 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.

Everyone knows that all modern chocolate factories have computer controlled chocolate boilers. The job of the boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars. Here's the controller class for Choc-O-Holic, Inc.'s industrial strength Chocolate Boiler. Check out the code; you'll notice they've tried to be very careful to ensure that bad things don't happen, like draining 500 gallons of unboiled mixture, or filling the boiler when it's already full, or boiling an empty boiler!

Turning It into Singleton

```
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;
    }

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

- How might things go wrong if more than one instance of ChocolateBoiler is created in an application?

Singleton Pattern Defined

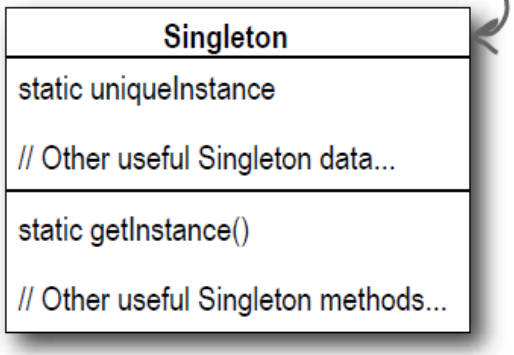
The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

Singleton:
1. One instance
2. Providing global access

- What's really going on here?
  - We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.
  - We're also providing a global access point to the instance:
    - Whenever you need an instance, just query the class and it will hand you back the single instance. As you've seen, we can implement this so that the Singleton is created in a lazy manner, which is especially important for resource intensive objects.

Singleton Pattern Defined (Cont.)

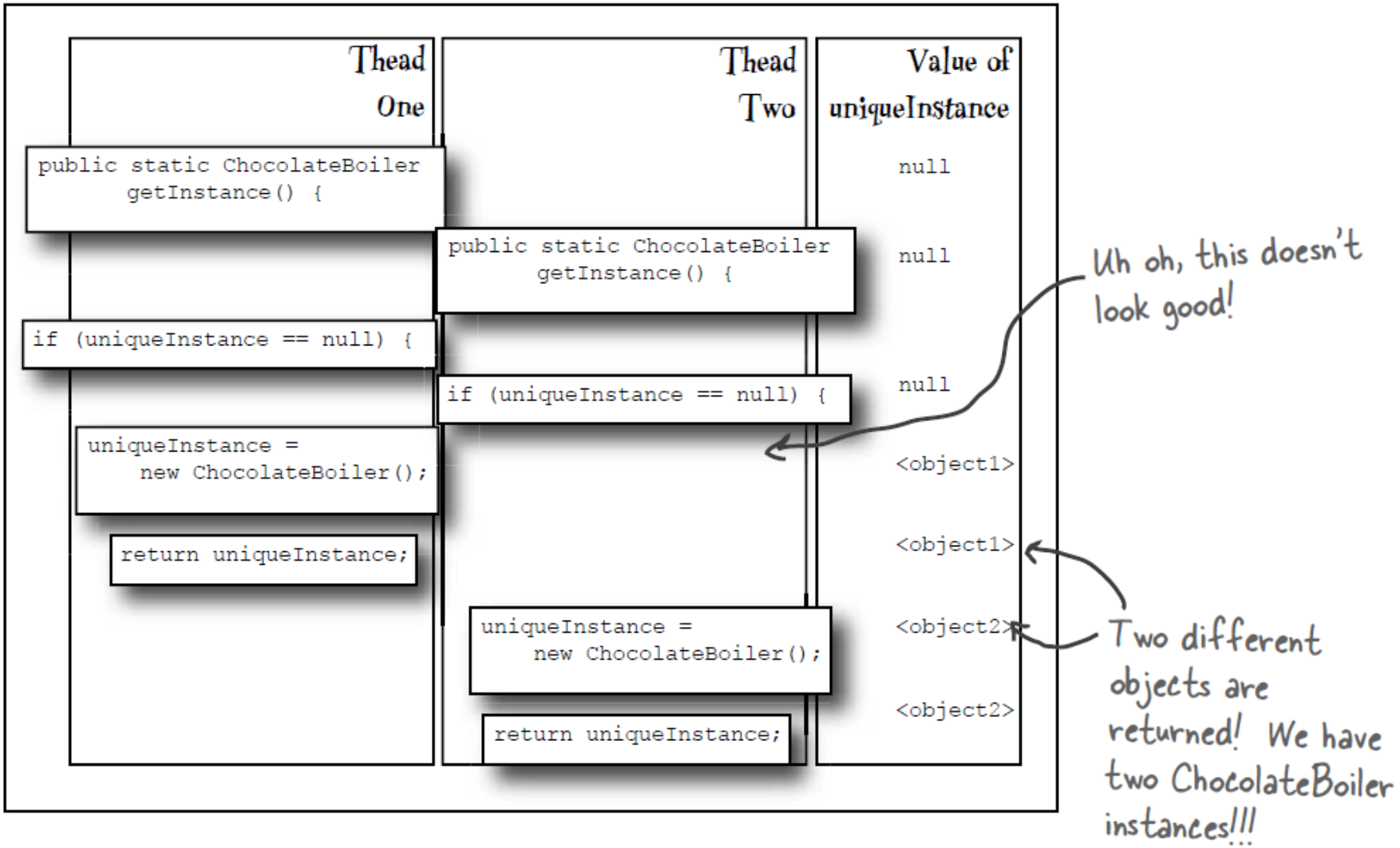
The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



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.

Problems in Multi-Threading



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.

Good point, and it's actually a little worse than you make out: the only time synchronization is relevant to the first time through this method. In other words, once we've set the ***uniqueInstance*** variable to an instance of Singleton, we have no further need to synchronize this method. After the first time through, synchronization is totally unneeded overhead!

Improving Multithreading

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

That's right; if calling the getInstance() method isn't causing substantial overhead for your application, forget about it. Synchronizing getInstance() is straightforward and effective. Just keep in mind that synchronizing a method can decrease performance by a factor of 100, so if a high traffic part of your code begins using getInstance(), you may have to reconsider.

2. Move to an eagerly created instance rather than a lazily created one

If your application always creates and uses an instance of the Singleton or the overhead of creation and runtime aspects of the Singleton are not onerous, you may want to create your Singleton eagerly, like this:

```
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.



Improving Multithreading (Cont.)

3. Use "double-checked locking" to reduce the use of synchronization in getInstance()

With double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize. This way, we only synchronize the first time through, just what we want.

Let's check out the code:

```
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.

Describing the Applicability

Synchronize the getInstance() method:

A straightforward technique that is guaranteed to work. We don't seem to have any performance concerns with the chocolate boiler, so this would be a good choice.

Use eager instantiation:

We are always going to instantiate the chocolate boiler in our code, so statically inializing the instance would cause no concerns. This solution would work as well as the synchronized method, although perhaps be less obvious to a developer familar with the standard pattern.

Double checked locking:

Given we have no performance concerns, double-checked locking seems like overkill. In addition, we'd have to ensure that we are running at least Java 5.

Issues about Singleton

- Can't I just create a class in which all methods and variables are defined as static?
  - Yes, if your class is self-contained and doesn't depend on complex initialization. However, because of the way static initializations are handled in Java, this can get very messy
- I wanted to subclass my Singleton code, but I ran into problems. Is it okay to subclass a Singleton?
  - One problem with subclassing Singleton is that the constructor is private. You can't extend a class with a private constructor. So, the first thing you'll have to do is change your constructor so that it's public or protected. But then, it's not *really* a Singleton anymore, because other classes can instantiate it.

Conclusion

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