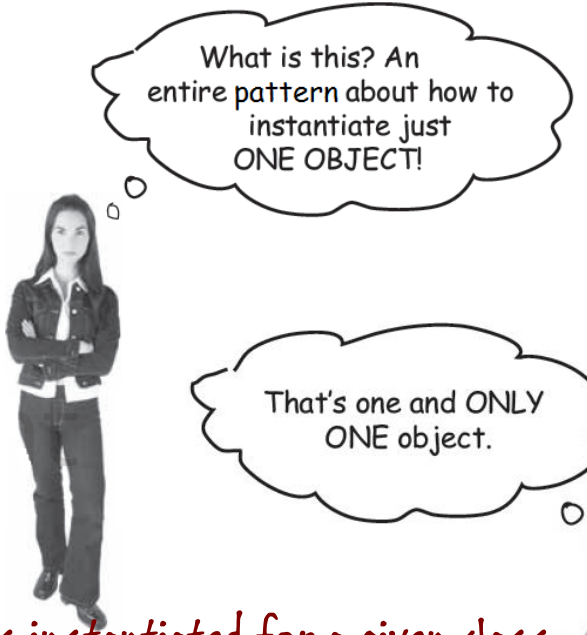
BIM492 Design Patterns

05. Singleton Pattern

One of a Kind Objects

Developer: What use is that?

Guru: There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards. In fact, for many of these types of objects, if we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.



Developer: Okay, so maybe there are classes that should only be instantiated once, but do I need a whole class for this? Can't I just do this by convention or by global variables? You know, like in Java, I could do it with a static variable.

Guru: In many ways, the Singleton Pattern is a convention for ensuring one and only one object is instantiated for a given class. If you've got a better one, the world would like to hear about it; but remember, like all patterns, the Singleton Pattern is a time-tested method for ensuring only one object gets created. The Singleton Pattern also gives us a global point of access, just like a global variable, but without the downsides.

Developer: What downsides?

Guru: Well, here's one example: if you assign an object to a global variable, then you have to create that object when your application begins. Right? What if this object is resource intensive and your application never ends up using it? As you will see, with the Singleton Pattern, we can create our objects only when they are needed.

Developer: This still doesn't seem like it should be so difficult.

Guru: If you've got a good handle on static class variables and methods as well as access modifiers, it's not. But, in either case, it is interesting to see how a Singleton works, and, as simple as it sounds, Singleton code is hard to get right. Just ask yourself: how do I prevent more than one object from being instantiated? It's not so obvious, is it?

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?

```
public MyClass {
   private MyClass() {}
}
```

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

The Little Singleton

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()`".

The Little Singleton

Okay, it was just a thought.
What does this mean?

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

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

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

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`?

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

Wow; you sure can.

The Little Singleton

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...

OK, LET'S FINISH THE CODE!

Dissecting the classic Singleton Pattern implementation

Let's rename MyClass to Singleton.

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

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.

```
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
}
```

Code Up Close

Code Up Close

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

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.

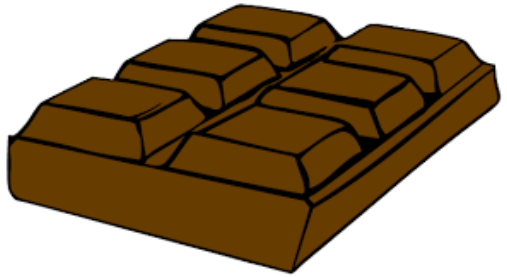
```
public class ChocolateBoiler {
  private boolean empty;
  private boolean boiled;
  public ChocolateBoiler() {
    empty = true;
    boiled = false;
  }
  public void fill() {
    if (isEmpty()) {
      empty = false;
      boiled = false;
      // fill the boiler with a milk/chocolate mixture
    }
  }
  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.



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!

Sharpen your pencil

Choc-O-Holic has done a decent job of ensuring bad things don't happen, don't ya think? Then again, you probably suspect that if two ChocolateBoiler instances get loose, some very bad things can happen.

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

Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a 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...
}
```

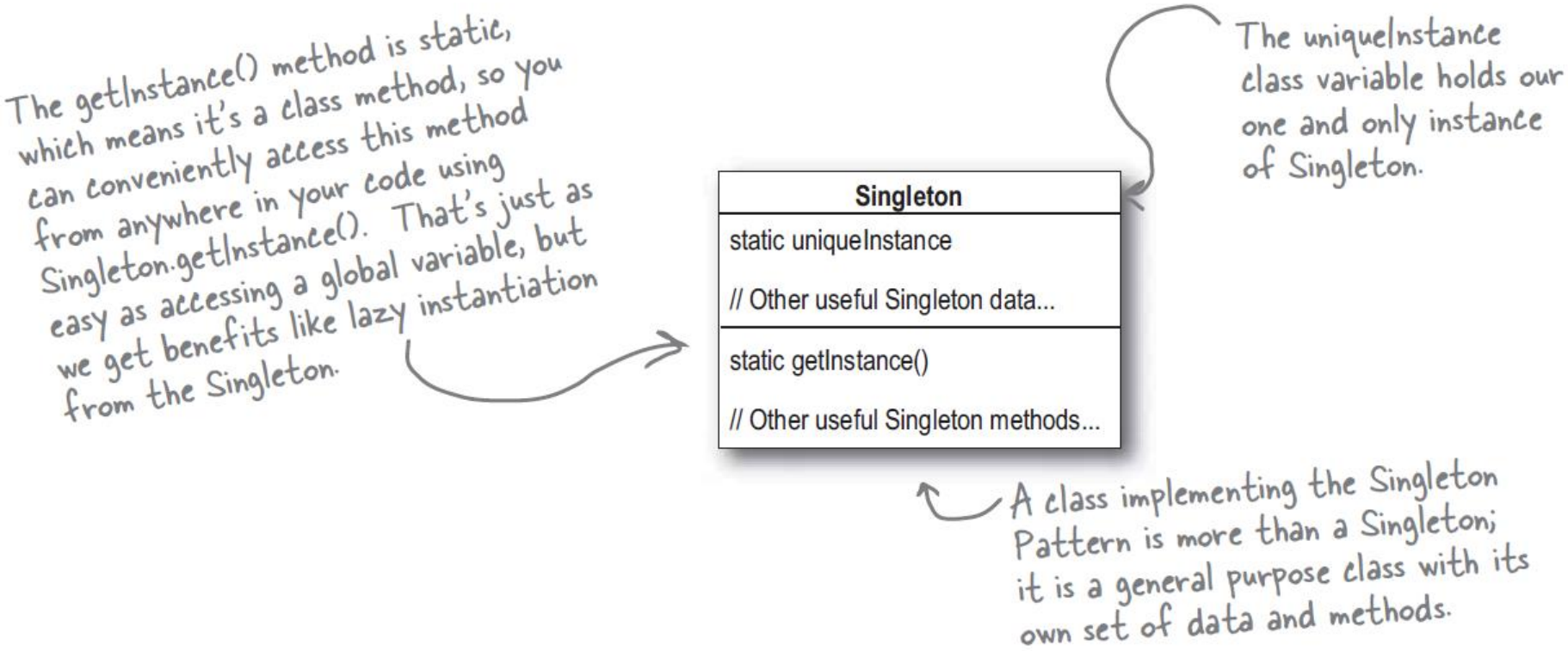
Singleton Pattern defined

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

No big surprises there.

But, let's break it down a bit more:

- 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.



Hershey, PA
~~Houston~~, we have a problem...

ChocolateBoiler's `fill()` method was able to start filling the boiler even though a batch of milk and chocolate was already boiling! That's 500 gallons of spilled milk (and chocolate)! What happened!?

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? Right?

Be the JVM

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.

Hint: you really just need to look at the sequence of operations in the `getInstance()` method and the value of `uniqueInstance` to see how they might overlap. Use the code Magnets to help, you study how the code might interleave to create two boiler objects.

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

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

Thread One	Thread Two	Value of uniqueInstance

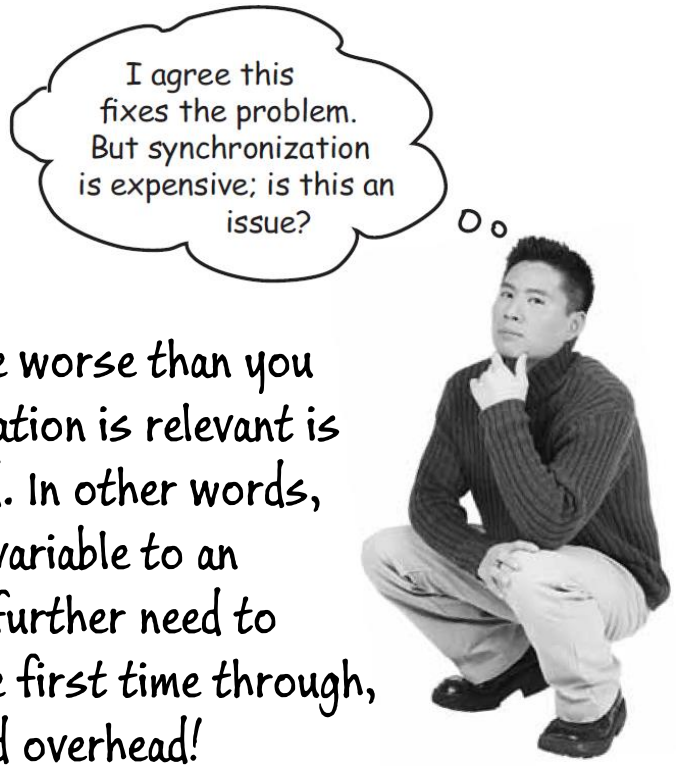
Dealing with multithreading

```
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 is 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!

Can we improve multithreading?

1. Do nothing if the performance of `getInstance()` isn't critical to your application
   - 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 the method, you may have to reconsider.

2. Move to an eagerly created instance rather than a lazily created one
   - Using this approach, 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.

```
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.

Can we improve multithreading? continued...

3. Use "double-checked locking" to reduce the use of synchronization in `getInstance()`
   - first check to see if an instance is created
   - if not THEN synchronize

```
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.

! If performance is an issue in use of `getInstance()` method then this method can drastically reduce the overhead.

Meanwhile, back at the Chocolate Factory

Sharpen your pencil

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code

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 initializing the instance would cause no concerns. This solution would work as well as the synchronized method, although perhaps be less obvious to a developer familiar 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.

there are no Dumb Questions

Q: Such a simple pattern consisting of only one class, Singletons sure seem to have some problems.

A: Well, you're warned up front! But don't let the problems discourage you; while implementing Singletons correctly can be tricky, after this class you are now well informed on the techniques for creating Singletons and should use them wherever you need to control the number of instances you are creating.

Q: I just create a class in which all methods and variables are defined as static? Wouldn't that be the same as a Singleton?

A: 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, especially if multiple classes are involved. Often this scenario can result in subtle, hard to find bugs involving order of initialization. Unless there is a compelling need to implement your "singleton" this way, it is far better to stay in the object world.

Q: What about class loaders? I heard there is a chance that two class loaders could each end up with their own instance of Singleton.

A: Yes, that is true as each class loader defines a namespace. If you have two or more classloaders, you can load the same class multiple times (once in each classloader). Now, if that class happens to be a Singleton, then since we have more than one version of the class, we also have more than one instance of the Singleton. So, if you are using multiple classloaders and Singletons, be careful. One way around this problem is to specify the classloader yourself.

there are no Dumb Questions

Q: I've always been taught that a class should do one thing and one thing only. For a class to do two things is considered bad OO design. Isn't a Singleton violating this?

A: You would be referring to the "One Class, One Responsibility" principle, and yes, you are correct, the Singleton is not only responsible for managing its one instance (and providing global access), it is also responsible for whatever its main role is in your application. So, certainly it can be argued it is taking on two responsibilities. Nevertheless, it isn't hard to see that there is utility in a class managing its own instance; it certainly makes the overall design simpler. In addition, many developers are familiar with the Singleton pattern as it is in wide use. That said, some developers do feel the need to abstract out the Singleton functionality.

Q: I wanted to subclass my Singleton code, but I ran into problems. Is it okay to subclass a Singleton?

A: 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. If you do change your constructor, there's another issue. The implementation of Singleton is based on a static variable, so if you do a straightforward subclass, all of your derived classes will share the same instance variable. This is probably not what you had in mind. So, for subclassing to work, implementing registry of sorts is required in the base class. Before implementing such a scheme, you should ask yourself what you are really gaining from subclassing a Singleton. Like most patterns, the Singleton is not necessarily meant to be a solution that can fit into a library. In addition, the Singleton code is trivial to add to any existing class.

there are no Dumb Questions

Q: I still don't totally understand why global variables are worse than a Singleton.

A: In Java, global variables are basically static references to objects. There are a couple of disadvantages to using global variables in this manner. We've already mentioned one: the issue of lazy versus eager instantiation. But we need to keep in mind the intent of the pattern: to ensure only one instance of a class exists and to provide global access. A global variable can provide the latter, but not the former. Global variables also tend to encourage developers to pollute the namespace with lots of global references to small objects. Singletons don't encourage this in the same way, but can be abused nonetheless.

Tools for your Design Toolbox

Bullet Points

- 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.
- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.
- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multithreaded applications (and we should consider all applications multithreaded!).
- Beware of the double-checked locking implementation; it is not thread-safe in versions before Java 2, version 5.
- Be careful, if you're using multiple classloaders.