The Notion of a Singleton

- There are many objects we only need one of:
  - Thread pools, caches, dialog boxes, logging objects, device drivers, etc.
  - In many cases, instantiating more than one of such objects creates all kinds of problems (e.g., incorrect program behavior, resource overuse, inconsistent results)
- We could just use global (static) variables
  - The Singleton pattern gives all of the upsides without the downsides
    - E.g., object isn't forced to be created when the application starts
- Basically, the Singleton is used anytime you want every object in the application to use the same global resource

Towards a Singleton

- In Java, how do you create a single object?
  - `new myObject();`
- And if you call that a second time?
  - You get a second, distinct object
- Can you always instantiate a class this way?
- How could you prevent such instantiation?
  - ```
    public class MyClass {
      private MyClass() {}
    }
    ```
- Who can use such a private constructor?
  - Only code within **`MyClass`**
- How can you get access to code within **`MyClass`** if you can't instantiate it?
- What does this do:
  - ```
    public class MyClass {
      public static MyClass getInstance() {
        …
      }
    }
    ```
- How would you call that?
  - `MyClass.getInstance();`
- How would you fill out the implementation to make sure that only a single instance of **`MyClass`** is ever created?

The Classic 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!

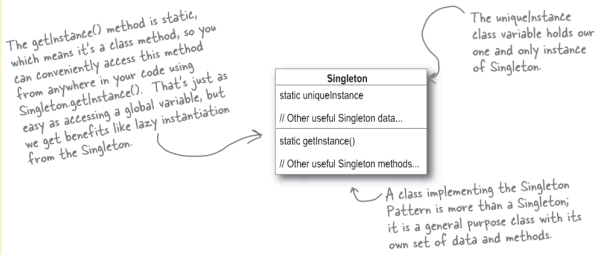
```
public class Singleton {
  private static Singleton uniqueInsta
  // …
  private Singleton() {}
  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // …
}
```

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

The Singleton Pattern

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

The Singleton Class Diagram



We have a problem

- The Singleton pattern, as we have implemented it, is not **thread safe**
- Let's explore what happens when multiple threads access the singleton pattern simultaneously.
- Draw a sequence diagram with three actors
  - The `Singleton` object
  - Two "client" threads
- Show a sequence of legal calls to the `Singleton` object's `getInstance()` method (including its internals) that results in the instantiation of two objects

An example

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

Easy Fix

```
public class Singleton {
  private static Singleton uniqueInstance;
  // …
  private Singleton() {}
  public static synchronized S
    if (uniqueInstance == null
      uniqueInstance = new Sin
    }
    return uniqueInstance;
  }
  // …
}
```

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

This turns out to not be a great idea. Why?

Synchronization is expensive. But that's not all…

When is synchronization *really* necessary?

What are the options

- Just roll with synchronized `getInstance()` anyway. Maybe the performance of `getInstance()` isn't a big deal for you.

- Use **eager instantiation** instead of **lazy instantiation**

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

饿汉式

One more option

One more option…

- **Double checked locking** (Java 5

```
public class Singleton {
   private volatile static S
   private Singleton () {}
   public static Singleton g
     if (uniqueInstance == n
       synchronized (Singlet
         if (uniqueInstance
           uniqueInstance =
         }
       }
     }
     return uniqueInstance;
   }
}
```

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

Check for an instance; if there isn't one, enter the `synchronized` block

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

Notice that we only synchronize the first time through!

This approach can drastically reduce the overhead of synchronization