The Singleton Pattern

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

Toni Sellarès
Universitat de Girona

Singleton: motivation

- Singleton: How to instantiate just one object - one and only one!
- Why?
  - Many objects we need only one of: dialog boxes, objects that handle preferences and registry settings, etc.
  - If more than one instantiated:
    - Incorrect program behavior, overuse of resources, inconsistent results.
- Alternatives:
  - Use a *global variable*
    - Downside: assign an object to a global variable then that object might be created when application begins. If application never ends up using it and object is resource intensive: waste!
  - Use a *static variable*
    - Downside: how do you prevent creation of more than one class object?

Singleton: solution (1)

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.
Can we always instantiate a class one or more times?	Yes. Caveat: Only if it is public class
And if not?	Only classes in the same package can instantiate it - but they can instantiate it more than once.

Is this possible?
public MyClass {
    private MyClass ( ) { }
}
	Yes. It is a legal definition

What does it mean?	A class that can't be instantiated because it has a private constructor

Singleton: solution (2)

- Is there any class that could use a private constructor?
- What's the meaning of the following?

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

- Instantiating a class with a private constructor:

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

The Singleton Pattern

Constructor is declared private; only singleton can instantiate this class!

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

```
public class Singleton {
    private static Singleton uniqueInstance;
     // other useful instance variables

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

     // other useful methods
}
```

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

Of course, Singleton is a regular class so it has other useful instances and methods.

Singleton: 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…

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

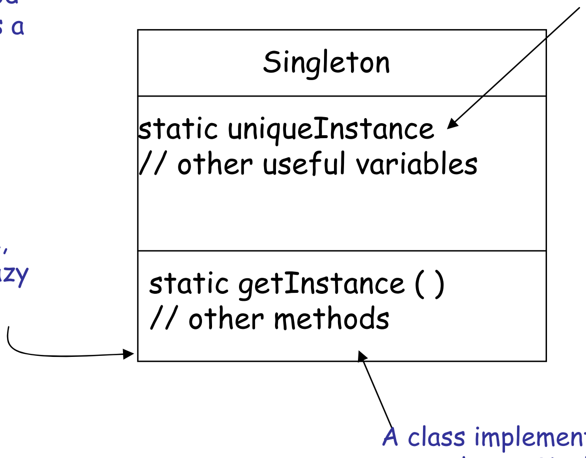
If uniqueInstance wasn't null, then it was previously created. We just fall through to the return statement. In either case, we have an instance and we return it.

…and if it doesn't exist, we instantiate Singleton through its private constructor and assign it to the uniqueInstance. Note that if we never need the uniqueInstance, it never gets created --> **lazy instantiation.**

Singleton Pattern Defined

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

The getInstance ( ) method is static, which means it is a class method, so you can conveniently access this method 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 one instance of Singleton.

A class implementing a Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Singleton: Example with alternative code

```
class iSpooler {
   //this is a prototype for a printer-spooler class such that only one instance can ever exist
   static boolean instance_flag = false; //true if 1 instance
   //the constructor is privatized, but need not have any content
   private iSpooler() { }
   //static Instance method returns one instance or null
   static public iSpooler Instance() {
     if (! instance_flag) {
          instance_flag = true;
          return new iSpooler(); //only callable from within
     }
     else
        return null; //return no further instances
     }
     public void finalize() {
         instance_flag = false;
     }
}
```

```
public class iSpooler{
        private static iSpooler uniqueInstance;
        private iSpooler ( ) {  }
        public static iSoopler getInstance( ) {
                if (uniqueInstance == null) {
                       uniqueInstance = new
iSpooler( );
                }
                return uniqueInstance;
        }
}
```

Dealing with Multi-threading

The Singleton class is not thread-safe: if two threads – we will call them Thread 1 and Thread 2, call *Singleton.getInstance()* at the same time, two *Singleton* instances can be created if Thread 1 is preempted just after it enters the if block and control is subsequently given to Thread 2.

Easy fix: make `getInstance` ( ) a synchronized method

```
public class Singleton {
      private static Singleton uniqueInstance;
       // other useful instance variables

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

       // other useful methods
}
```

By adding the synchronized keyword to getInstance ( ) method, 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.

This fixes the problem, but synchronization is expensive; is this really an issue? -- synchronization is really only needed the first time through this method. Once we have created the first Singleton instance, we have no further need to synchronize this method. So after the first time, synchronization is totally unneeded overhead.

Can we improve multithreading?

1. Do nothing if the performance of `getInstance` ( ) isn't critical to your application. [*remember that synchronizing can decrease performance by a factor of 100*]
2. Move to an eagerly created instance rather than a lazily created one.

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

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

  private Singleton ( ) { }

  public static Singleton getInstance  ( ) {
         return uniqueInstance;
   }
}
```

We've already got an instance, so just return it.

3. Use "double-locking" to reduce the use of synchronization in getInstance ()
   - First check to see is instance is created, THEN synchronize.

```
public class Singleton {
    private volatile static Singleton uniqueInstance;
     // other useful instance variables

    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 the synchronized block.

The **volatile** keyword ensures that multiple threads handle uniqueInstance variable correctly when it is being initialized to the Singleton instance. Only JDK 1.5!

If performance is an issue then this method can drastically reduce overhead!

Note we only synchronize the first time through.

Once in the block, check again if null. If so create instance.

Summary

- 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 multi-threaded applications.
- Beware of double-checked locking implementation: it is not thread-safe pre JDK 1.5
- Be careful if you are using multiple class loaders: this can defeat the purpose of the Singleton implementation
- If you are using a JVM earlier than 1.2, you'll need to create a registry of Singletons to defeat the garbage collector.