Singleton Pattern

Ensures a class has one and only one instance, and provides a global point of access to it

Where does it fit?

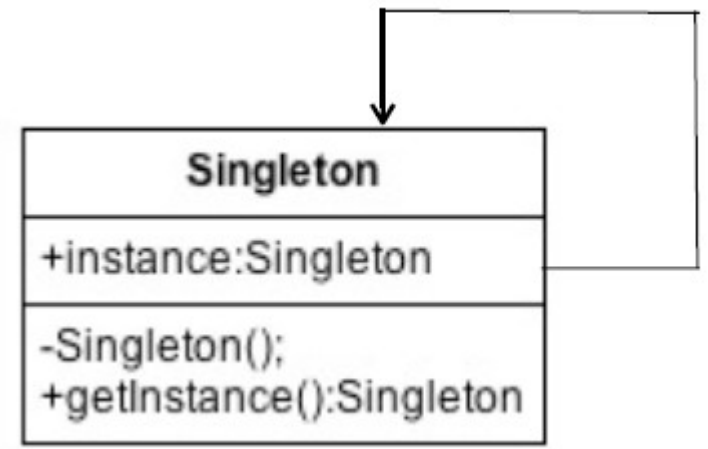
- ★ Structural Patterns
  - » strategy
- ★ Behavioral Patterns
  - » observer
  - » decorator
- ★ Creational Patterns
  - » factory method
  - » abstract factory
  - » singleton

Problem

Some classes only need to be instantiated once, otherwise you get incorrect behavior, overuse of resources, and inconsistent results

- Manage a pool of resources
  - Connections
  - Threads
- Registry access

Class Diagram



instance and getInstance are static

Classic Singleton Solution

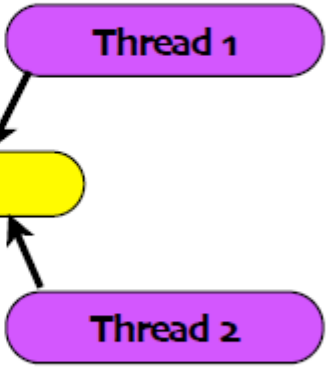
```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

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

The Solution is NOT Thread Safe: Why Not?

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

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

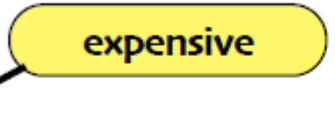
Answer!

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

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

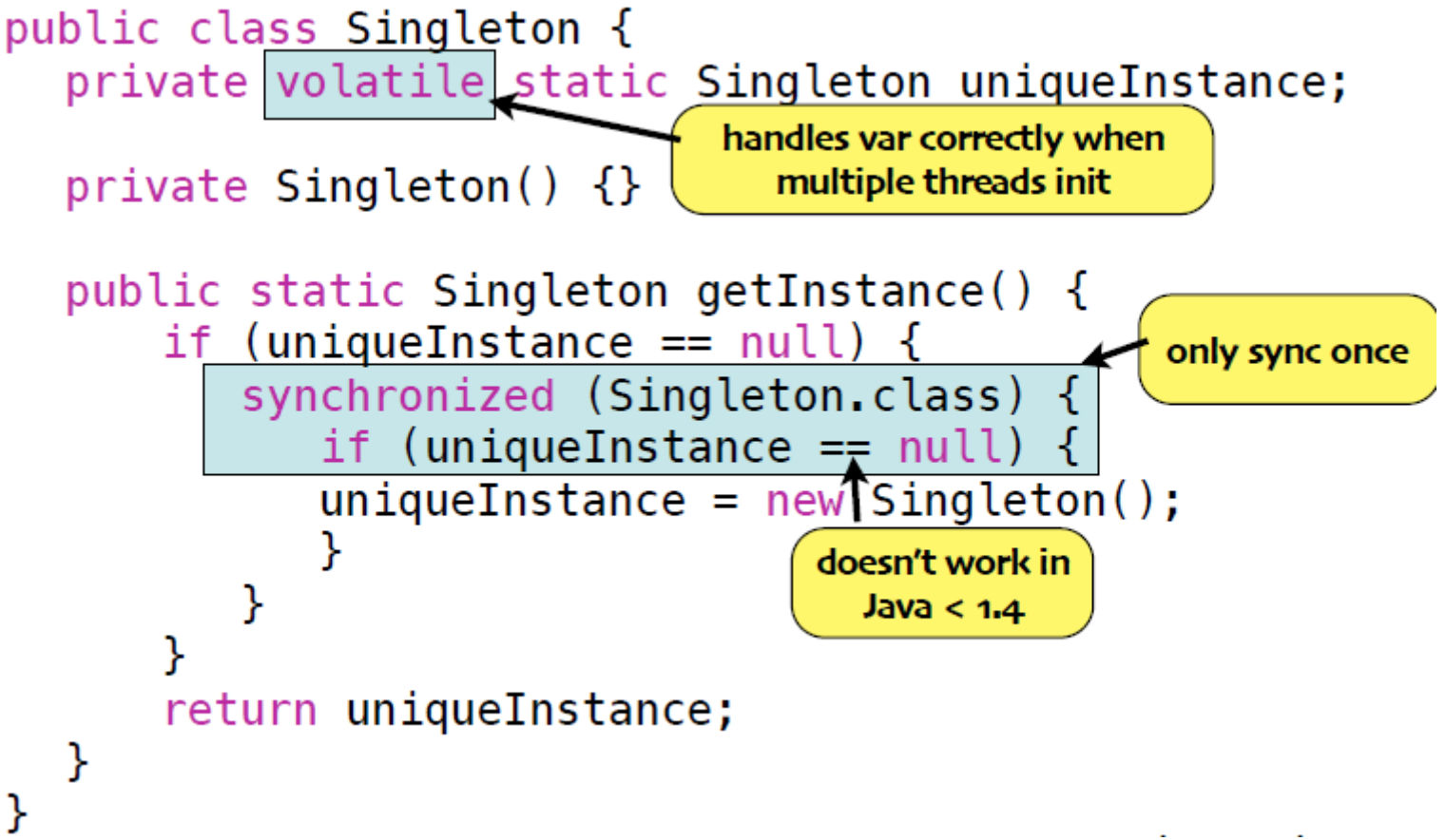


A possible solution to the threading problem: can slow down app

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

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



Second solution: double checked locking (requires Java 1.4 or greater)



Third solution: 'eager' instance

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

   private Singleton() {}

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

JVM handles instance creation before threads have access

Benefits of Singleton

- ★ Ensures you only at most have one instance
- ★ Provides global access point
- ★ Examine your performance requirements to determine which implementation of the singleton works best.
- ★ Some implementations don't work with older version of Java (see book)

Example: Main App

```
public class LoggerTester {
    public static void main(String[] args) {
        Test test = new Test();
        System.out.println("Testing my logger");
        Logger.getInstance().log("testing one two three");
        test.run();
    }
}

class Test {
    public Test() {
    }
    void run() {
        Logger.getInstance().log("testing from within Test");
    }
}
```

Logger class

```
public class Logger {
    private static Logger uniqueInstance = new Logger();
    // eagerly created instance
    private int loggedmessages;

    private Logger() {
        loggedmessages=0;
    }
    public static Logger getInstance() {
        return uniqueInstance;
    }
    public void log( String msg )
    {
     loggedmessages++;
     System.out.println("LOG:"+ msg);
    }
}
```

What will be printed!?

Testing my logger

...

What is printed

LOG: testing one two three
LOG: testing from within Test

...side note, clearly loggedmessages is 2

Example 2: Two Patterns used to solve problem

A one of a kind chocolate factory!

Main App

```
public class SingletonfactoryTest {
    public static void main(String[] args) {
        BarProducer t1 = new BarProducer(1);
        BarProducer t2 = new BarProducer(2);
        t1.start();
        t2.start();
    }
}
```

BarProducer (Thread)

```
public class BarProducer extends Thread{
     Factory wonka;
     public int count=0; // each producer creates 50 bars
     public int id;

     public BarProducer(int identifier) {
          id=identifier;
          System.out.println("creating new Bar Producer with ID:"+id);
     }
     public void run()
        {
          while (count<50) {
               try {
                    sleep(100);
               } catch (InterruptedException e) {}
               count++;
               WonkaBarFactory.getInstance().create();
          }
       }}
```

Factory and Bar

```
public interface Factory {
    Bar create();
}
public abstract class Bar {
   public int id;
}
public class WonkaBar extends Bar {

    public WonkaBar(int identifier) {
          id = identifier;
    }
}
```

Concrete Singleton Factory

```
public class WonkaBarFactory implements Factory {
     private int counter=0;
     private static WonkaBarFactory uniqueInstance; // = new WonkaBarFactory();

     private WonkaBarFactory() { }

     public synchronized Bar create() {
          Bar bar = new WonkaBar(counter++);
          System.out.println("new Wonka bar created with id:" + counter);
          return bar;
     }

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

Summary

- Singleton ensures at most one instance of a class in an application
- Singleton provides global access point to that instance
- The way you ensure your single instance can have performance issues, so customize based on your app
- Multiple class loaders can defeat Singleton – watch out for this
- Singleton class does something useful AND enforces single instance (two different things which is against OO principles, but acceptable based on singleton need)