The Singleton Pattern

ACS-3913 LECTURE 12

The Singleton Pattern

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

Singleton uses

- Connection and thread pools
- Logging facilities
- Preference and registry objects
- Device, printer and graphic drivers
- UI dialog and other modal controls

- Anywhere you want to ensure a resource exists only once

Creating ~~one~~ two instances

```
new MySingleObject();
new MySingleObject();
```

Prevent instantiation

1. Create a class with a private constructor

```
public class Singleton{
   private Singleton(){}
}
```

Prevent instantiation

2. Add a method that instantiates a class singleton and returns the instance

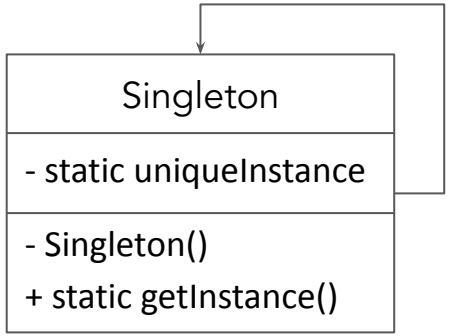
```
public class Singleton{
   private Singleton(){}

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

Prevent instantiation

3. Add a private Singleton field that determines and assign the new instance of Singleton the first time it is created.

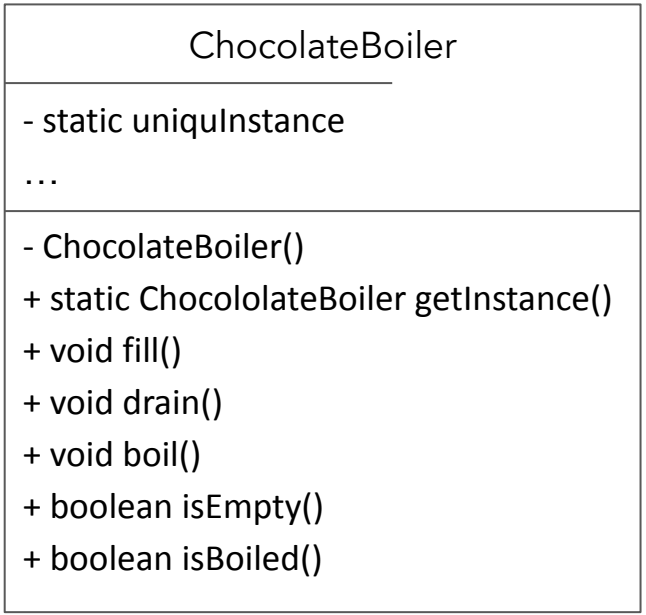
```
public class Singleton{
    private static Singleton unique;
    private Singleton(){}

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

The Singleton Pattern



+ other useful fields and methods

Example: Chocolate Boiler



Dealing with multithreading

- If there's only ever one thread, the previous implementation will work.
- However, if there's more than one thread we could end up with multiple "singletons"
  - Each thread has its own "copy" of variables.
  - The value of a variable could be "out of sync" with the main copy.
  - We could have `getInstance` started by several threads.
    - Each could test the value of `uniqueInstance` and proceed to instantiate the singleton.
- To work correctly, we need a technique to synchronize the actions of the threads.
  - With Java we could use synchronization of methods or variables.

Multi-threading and Singletons

The text presents synchronized static methods as one solution.

- Results in locking and unlocking of the class

  → only one synchronized method will execute at a time.

```
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
}
```

Synchronized methods

- When a static synchronized method is invoked, the thread must first acquire the intrinsic lock for the Class object associated with the class
  - this is done automatically for you.
- Only one thread can hold a lock at one time
  - other threads that request a lock are placed in a wait queue.

Synchronized methods

Synchronized methods introduce overhead

- we can avoid them.

Some alternatives:

a. double checked locking
    - This scheme uses a synchronized statement and a volatile variable.
b. eager instantiation

Synchronized methods

If a variable is not declared as volatile (i.e. it is non volatile) then thread A, when accessing the variable, may not see the most recent value that was written by some other thread, say thread B.

If a variable is declared as volatile then it is guaranteed that any thread which reads the field will see the most recently written value.

1. Double-checked locking

Check first to see if the instance exists or not. If not, then lock up a block of code.

```
// Danger! This implementation of Singleton not guaranteed to work prior to Java 5
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;
          }
     }
}
```

A thread's copy of a volatile attribute is reconciled with the "master" copy each time it is referenced.

Note these two checks on uniqueInstance

Second check is necessary to verify uniqueInstance is still null

A synchronized block of code. Only one thread at a time will execute this.

Results in very little overhead compared to synchronizing a whole method/class.

2. Eager instantiation

If your system always instantiates the singleton, then create it in advance

- very simple
- done by class loader prior to the class being used.

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

- When the class is loaded, the instance is created and available.
- `getInstance()` is always realized the same way.