Lecture 06: The Singleton Pattern

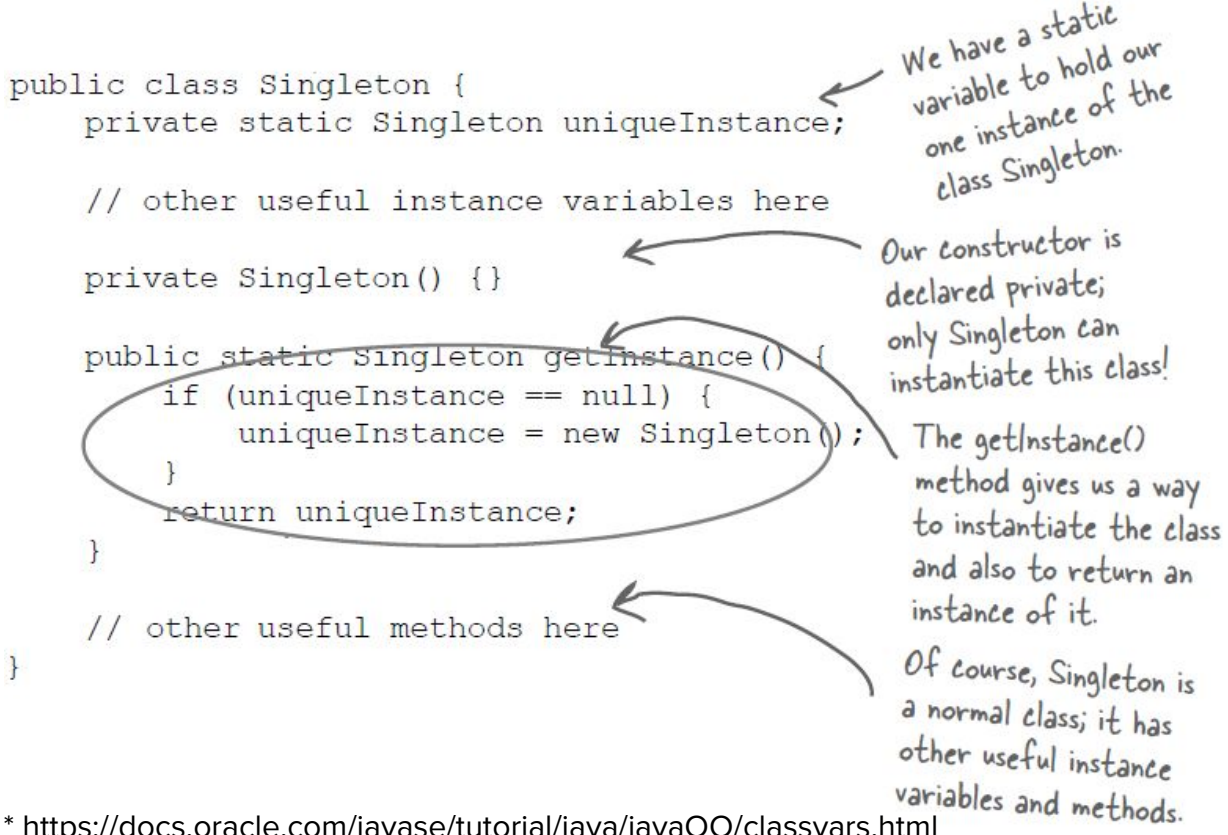
SE313, Software Design and Architecture

Damla Oguz

Chapter 5: The Singleton Pattern

- There are many objects we only need one of: 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, etc.
- In this lecture, we will learn to create one-of-a-kind objects for which there is only one instance.
- The Singleton Pattern ensures we have at most one instance of a class in our application.
- The Singleton Pattern also provides a global access point to that instance.

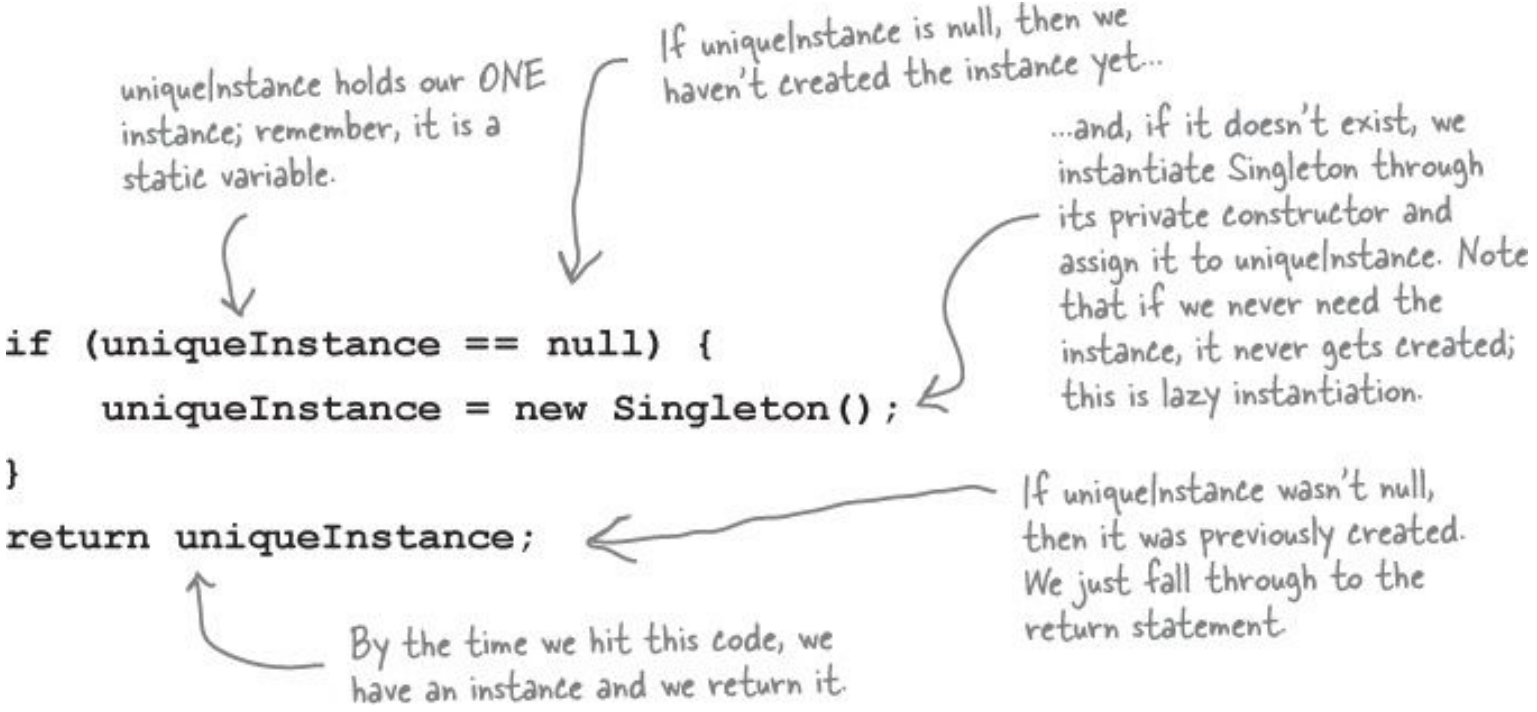
Classic Singleton Pattern implementation



- Sometimes, you want to have variables that are common to all objects. This is accomplished with the *static* modifier.
- Fields that have the static modifier in their declaration are called *static fields* or *class variables*. They are associated with the class, rather than with any object. *

* https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

Code Up Close



The singleton instance is only created when needed.

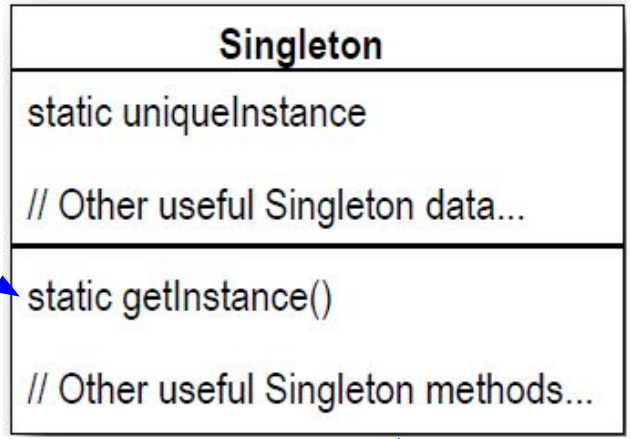
Singleton Pattern defined

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

- 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.

Let's check out the class diagram

The uniqueInstance class variable holds our one and only instance of Singleton.



The getInstance() method is static, which means it's a class method, so you can conveniently access this method from 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.

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

The Chocolate Boiler

- 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.
- 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!

Without using the Singleton Pattern

```
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
      }
    }
```

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.

Without using the Singleton Pattern (cont.)

```
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;
    }
}
```

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.

ChocolateBoiler class turned into a Singleton

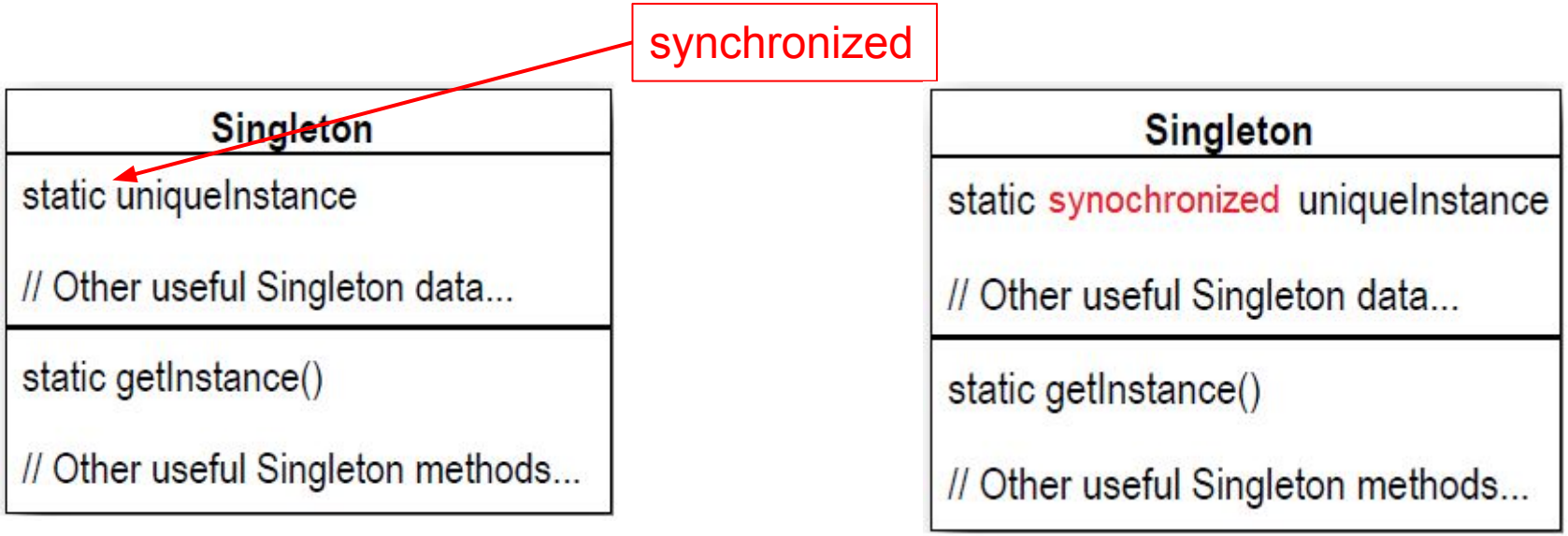
```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
      empty = true;
      boiled = false;
    }

    public static ChocolateBoiler getInstance() {
      if (uniqueInstance == null) {
           System.out.println("Creating unique instance of Chocolate Boiler");
           uniqueInstance = new ChocolateBoiler();
      }
      System.out.println("Returning instance of Chocolate Boiler");
      return uniqueInstance;
    }

// rest of ChocolateBoiler code...
```

Thread Safe? *

- The Java code just shown is not thread safe.
- This means that it is possible for two threads to attempt to create the singleton for the first time simultaneously.
- If both threads check to see if the static variable is empty at the same time, they will both proceed to creating an instance and you will end up with two instances of the singleton object (not good!).

* https://www.cs.colorado.edu/~kena/classes/5448/f09/lectures/22-singletoncommandadapter.pdf

Thread Safe Singleton



Thread Safe Singleton (cont.)

- 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.
- However, synchronization is expensive.
- The only time synchronization is relevant is the first time through this method.
- After the first time through, synchronization is totally unneeded overhead.
- There are other methods to improve multithreading...

How can we improve multithreading?

1. **Do nothing if the performance of getInstance() isn’t critical to your application**
2. **Move to an eagerly created instance rather than a lazily created one**
3. **Use “double-checked locking” to reduce the use of synchronization in getInstance()**

How can we improve multithreading? (cont.)

1. **Do nothing if the performance of getInstance() isn't critical to your application**
    - That's right; if calling the getInstance() method isn't causing substantial overhead for your application, forget about it.
    - 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 getInstance(), you may have to reconsider.

How can we improve multithreading? (cont.)

2. **Move to an eagerly created instance rather than a lazily created one**
    - If your application always creates and uses an instance of the Singleton or the overhead of creation and runtime aspects of the Singleton are not onerous, you may want to create your Singleton eagerly, like this:

```
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.

How can we improve multithreading? (cont.)

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

```
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.

How can we improve multithreading? (cont.)

3. **Use “double-checked locking” to reduce the use of synchronization in getInstance()**
    - With double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize.
    - This way, we only synchronize the first time through, just what we want.
    - Let’s check out the code…

```
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;
    }
}
```

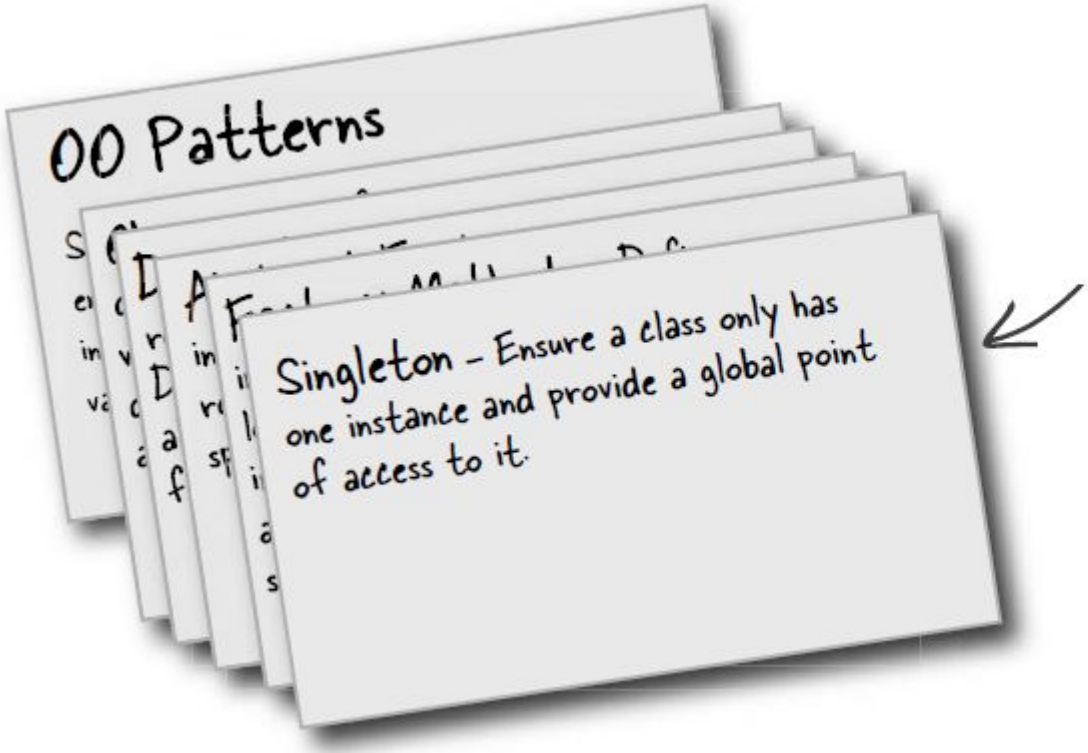
Check for an instance and if there isn't one, enter a synchronized block.

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.

Tools for your Design Toolbox



When you need to ensure you only have one instance of a class running around your application, turn to the Singleton.

Processes and Threads *

- In concurrent programming, there are two basic units of execution: *processes* and *threads*.
- In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.
- A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
- Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

* https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2004.