Singleton Pattern

It is all about how do we instantiate only one object for a given class. In a sense, it's a convention

Classic Singleton Pattern Implementation

```
public class Singleton {
    private static Singleton uniqueInstance;
    private Singleton() { }
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // other useful methods
}
```

holds only one instance of class singleton

this ensures only Singleton class can instantiate this class

Gives a way to create and return only one of a kind instance

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

→ On examining closely, we notice that this instance is only created when getInstance is called. This is known as lazy instantiation.

Where it is useful
- ↳ registry settings
- ↳ connection and thread pools

Taking an example: Chocolate Factory

Chocolate Boiler has conditions which can only be performed when they meet a certain criteria.

```
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
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
```

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.

To drain the boiler, it must be full (non-empty) and also boiled. Once it is drained we set empty back to true.

Q: what can go wrong if we create two objects of ChocolateBoiler?

→ Let's think.

ChocolateBoiler v1 = new ChocolateBoiler();
ChocolateBoiler v2 = new ChocolateBoiler();

```
        // bring the contents to a boil
        boiled = true;
    }
}

public boolean isEmpty() {
    return empty;
}

public boolean isBoiled() {
    return boiled;
}
```

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.

Here, virtually we are using 2 Boiler's but in reality only one physical boiler exists. This means we might do things like filling the boiler even thought it isn't empty

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

    private static ChocolateBoiler myInstance = null;

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

    public static ChocolateBoiler getInstance() {
        if (myInstance == null) {
            myInstance = new ChocolateBoiler();
        }
        return myInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```

The above code breaks during multithreading. But how?

Lets figure that out.

overlap. Use the code magnets to help you study how the code might interleave to create two boiler objects.

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
boiler.fill();
boiler.boil();
boiler.drain();
```

Make sure you check your answer in BE the JVM Solution before continuing!

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

Thread One	Thread Two	Value of uniqueInstance
		null
1		null
	1	null
2		null
	2	null
3		uq 1
	3	uq 2

As we can

see, multiple instances are created

Dealing with multithreading

```
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
}
```

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.

synchronized keyword ensures that only ONE thread can call the given function. It will not be invoked by two seperate threads at any given time.

Q: But synchronization is expensive. Can we fix that issue?

→ We can do following things

1) Do nothing
   - If performance not an issue we can choose to do nothing.
   - But this will reduce the performance of method by a factor of 100.

2) Move to eagerly created instance
   - If application always creates and use an instance, we can create it eagerly

Eagerly created instance →

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

3) Use "double-checked locking" to reduce use of synchronization

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

This method will become

```
    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.

more clear after understanding threads in Java.