# Singleton Pattern

Using DCL

## Key Features

- **Intent**: Have only one of an object, ensure that all entities are using the same instance of this object, without passing a reference
- **Problem**: Several different client objects need to refer to the same thing, and you want to ensure that you do not have more than one of them

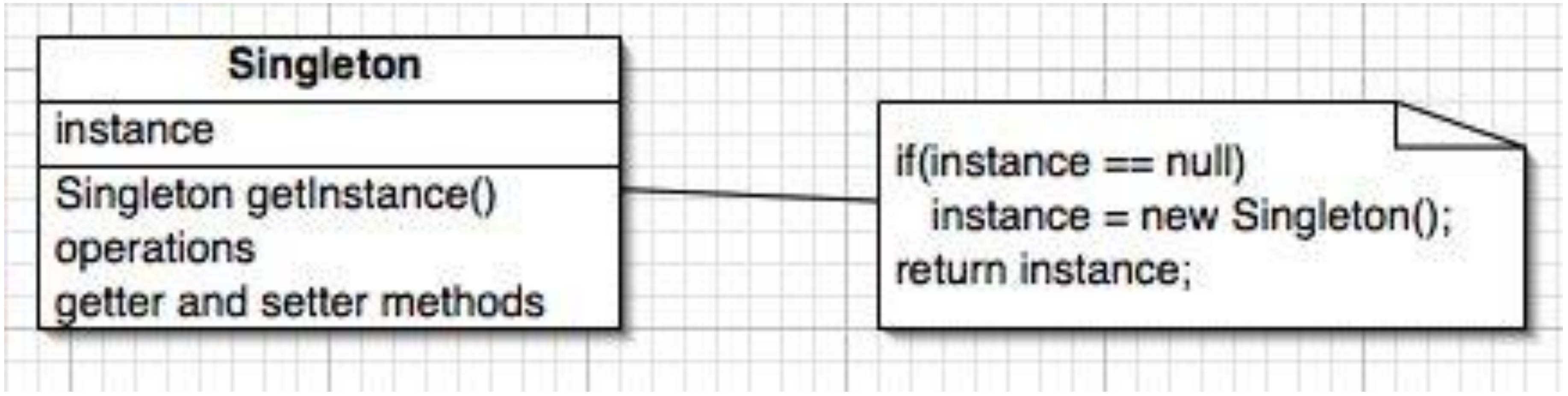
## Key Features

- **Solution**: Guarantee one instance
- **Participants and collaborators**: Clients create an instance of the Singleton solely through the getInstance method
- **Consequences**: Clients need not concern themselves whether an instance of the Singleton exists. This can be controlled from within the Singleton

## Implementation

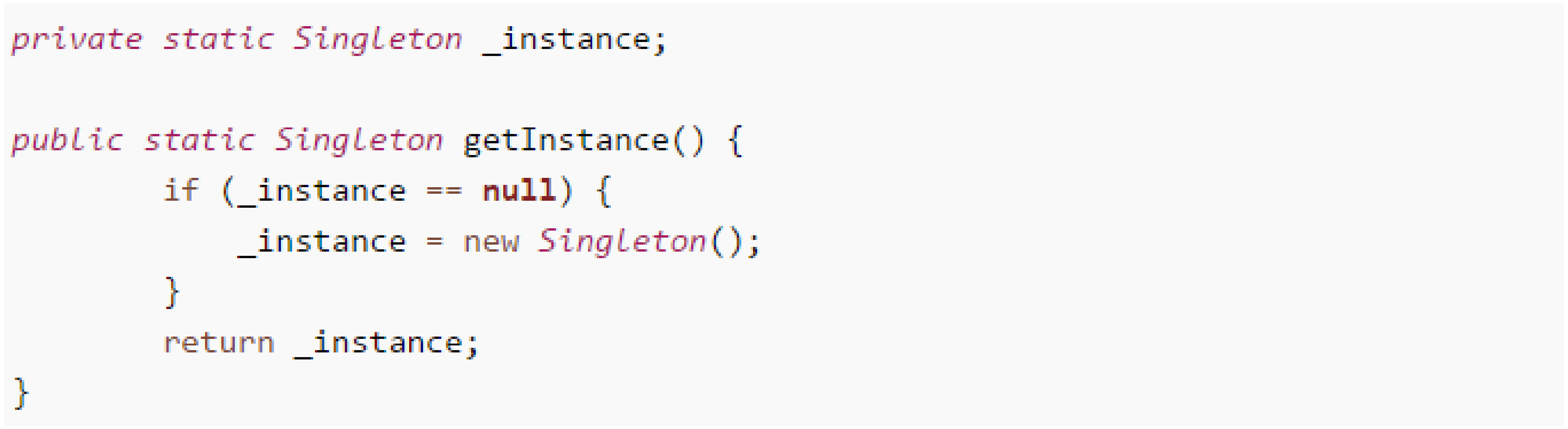
- Add a private static member of the class that refers to the desired object (initially it is null)
- Add a public static method that instantiates this class if this member is null (and sets this member’s value) and then returns the value of this member.
- Set the constructor’s status to protected or private so that no one can directly instantiate this class and bypass the static constructor mechanism

## Generic Structure



```
if(instance == null)
   instance = new Singleton();
return instance;
```

## Implementation



```
private static Singleton _instance;

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

## Problem

- Might lead to multiple instances of the Singleton object when getInstance() called at the same time by multiple parallel threads
- Thereby we lose the purpose of Singleton

## Problem

- When two calls to getInstace() are made by two threads at the same time:
  - First thread checks if instance exists. It does not, so it will create the first instance
  - Before it is done, second thread also checks if instance exists, it does not, so it will create the second instance

# Problem

- If the class is stateless, then not a problem
- Otherwise, changes in one object will not be reflected in all the other objects
  - If there is a connection, there will be two connections
  - If there is a counter, there will be two counters
- A solution would be to make the entire method getInstance() synchronized

```
public static synchronized Singleton getInstanceTS() {
    if (_instance == null) {
        _instance = new Singleton();
    }
    return _instance;
}
```

## synchronized method

- Though it’s a thread-safe and solves issue of multiple instance, it's not very efficient.

- All the threads will have to wait for the check on whether the object already exists

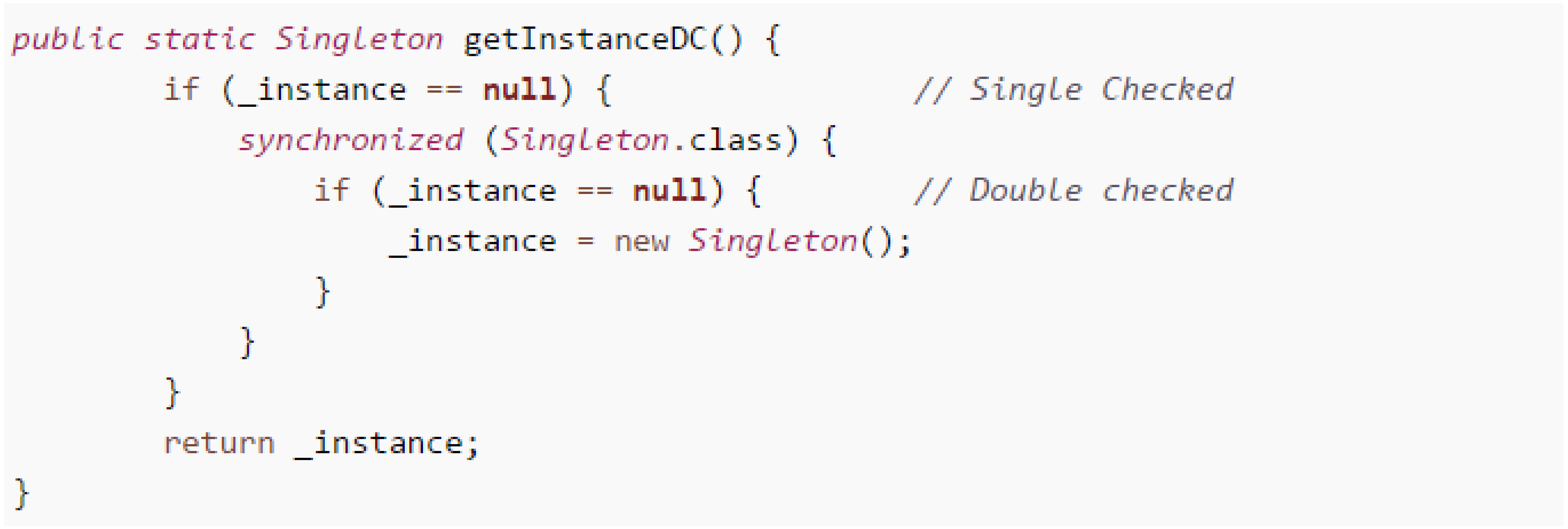
- You need to bear cost of synchronization every time you call this method, while synchronization is only needed on first class, when Singleton instance is created.

## Double Checked Locking (DCL)

- Only applies to multithreaded applications
- Optimizes unnecessary locking
- Synchronization happens utmost one time, so not a bottleneck

# DCL

- **Feature**: Unnecessary locking is avoided by adding another test before creating the object



```java
public static Singleton getInstanceDC() {
        if (_instance == null) {                // Single Checked
            synchronized (Singleton.class) {
                if (_instance == null) {        // Double checked
                    _instance = new Singleton();
                }
            }
        }
        return _instance;
}
```

## DCL

- On surface this method looks perfect, as you only need to pay price for synchronized block one time, but it still broken, until you make _instance variable volatile.
- Without volatile modifier it's possible for another thread in Java to see half initialized state of _instance variable, but with volatile variable guaranteeing happens-before relationship, all the write will happen on volatile _instance before any read of _instance variable.

# DCL Implementation

```
   private volatile static Singleton _instance;
    public static Singleton getInstanceDC() {
        if (_instance == null) {
            synchronized (Singleton.class) {
                if (_instance == null) {
                    _instance = new Singleton();
                }
            }
        }
        return _instance;
    }
}
```