Object Oriented Analysis and Design Patterns

A Report On

Singleton
&
Double Checked Locking

Design Patterns

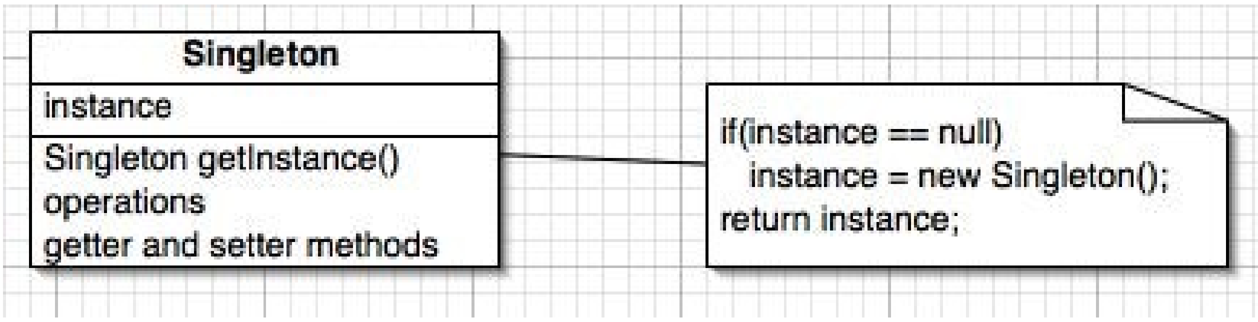
By:
Abhilash K (1MS14IS135)
Salman R (1MS14IS092)

Singleton Design Pattern

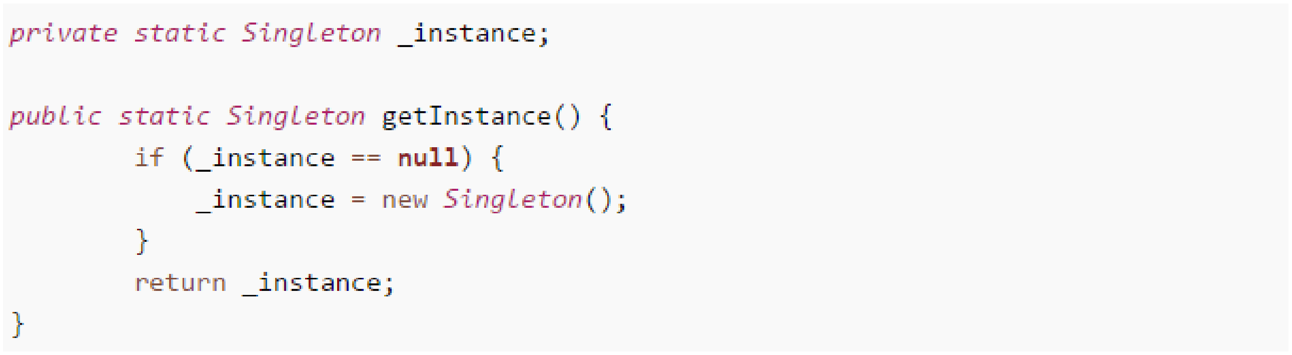
Key Features:

- **Intent:**
    - Ensure a class has one instance and provide a global point of access to it.
- **Problem:**
    - Several different client objects need to refer to the same thing, and we want to ensure that we do not have more than one of them
- **Solution:**
    - Guarantee one instance
- **Participants and Collaborator:**
    - 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 the member is null and then returns the value this member
    - Set the constructor's status to protected or private so that no one can directly instantiate this class an bypass the static constructor mechanism

Generic Structure:



Implementation of Singleton’s getInstance() method:

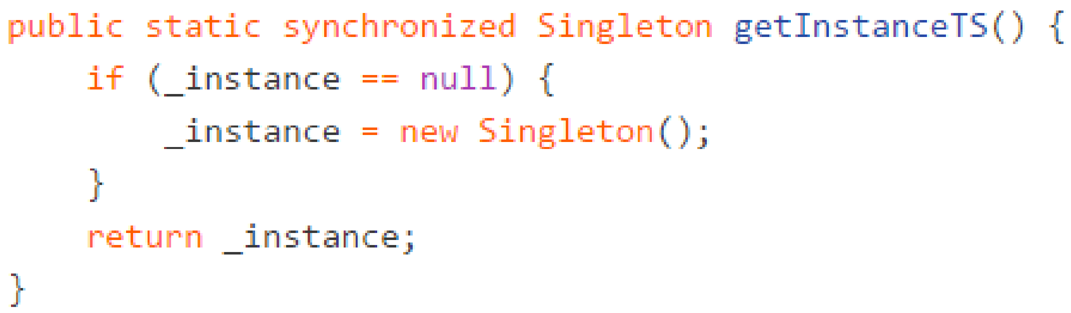


```
private static Singleton _instance;

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

Problem with this approach:

- Might lead to multiple instances of the Singleton object when **getInstance()** is called at the same time by multiple parallel threads
- 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
- Clients begin to use multiple instances individually as though they all refer to the same instance.
- 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
- Thereby we lose the purpose of Singleton design pattern

A solution would be to make the entire method **getInstance() synchronized**



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

- 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.
- Thus, we avoid this approach and use an improvised method, called as **Double Checked Locking (DCL)**

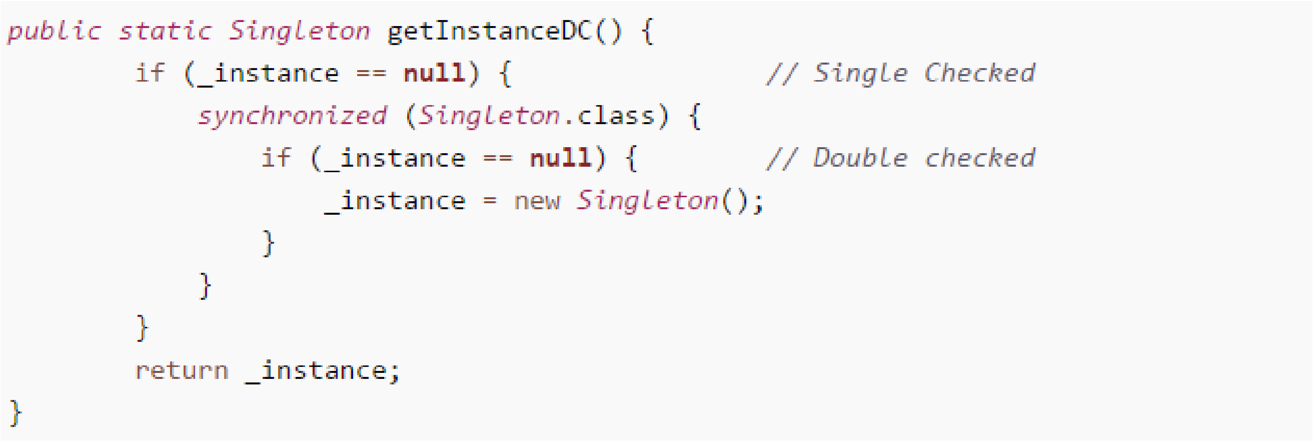
Double Checked Locking (DCL)

- Only applies to multi-threaded applications
- Optimizes unnecessary locking
- Synchronization happens utmost one time, so it is not a bottleneck

Feature:

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

Implementation of DCL's getInstance() method:



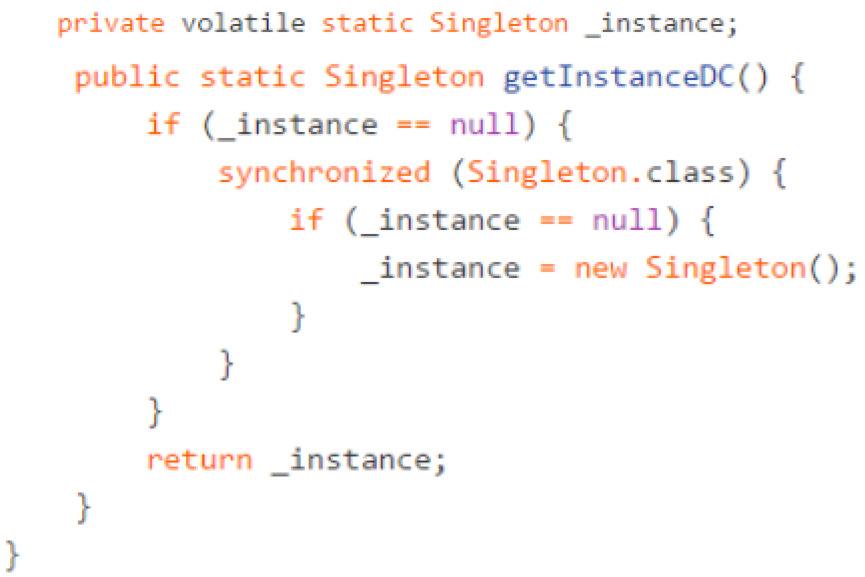
```
public static Singleton getInstanceDC() {
        if (_instance == null) {                // Single Checked
            synchronized (Singleton.class) {
                if (_instance == null) {        // Double checked
                    _instance = new Singleton();
                }
          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

The solution is to do a sync after the test for null and then check again to make sure the instance member has not yet been created. Thus, it is called as **Double Checked Locking**

- Unnecessary locking is avoided by adding another test before creating the object
- 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’s Actual Implementation:

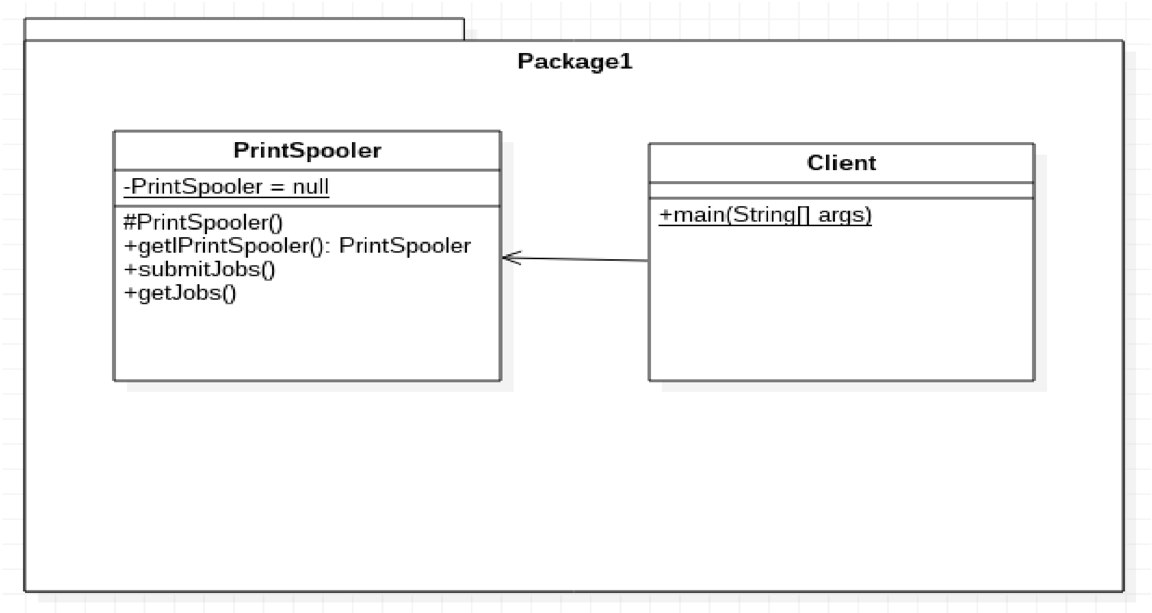


```
          private volatile static Singleton _instance;
           public static Singleton getInstanceDC() {
               if (_instance == null) {
                   synchronized (Singleton.class) {
                       if (_instance == null) {
                           _instance = new Singleton();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

Example:

A **print spooler** is a software program responsible for managing all print jobs currently being sent to the printer or print server.

- It is obvious here that we do want want to maintain multiple print spooler and queues, which leads to duplicate entries and a lot of confusion.
- Thereby, we restrict ourselves to using only one instance of the **print spooler.**
- This is done by using Singleton and Double Checked Locking, since multiple users may try to access the print spooler at the same time



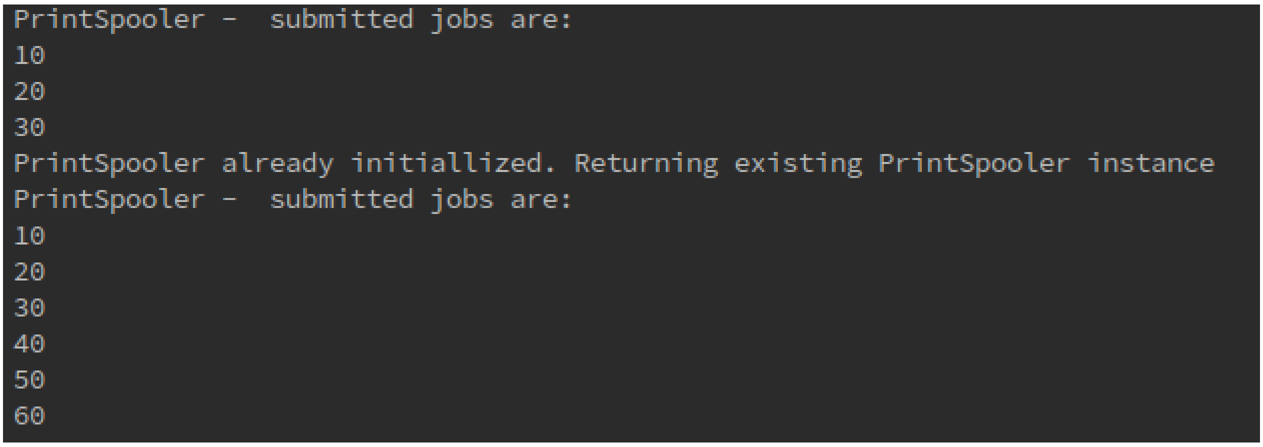
PrintSpooler.java

```
public class PrintSpooler {
    private volatile static PrintSpooler b = null;
    private ArrayList<Integer> jobs = new ArrayList();

    private PrintSpooler(){
        // empty constructor
    }

    public static PrintSpooler getPrintSpooler(){
        if(b == null){
            synchronized(PrintSpooler.class){
                if(b == null)
                    b = new PrintSpooler();
            }
        }else{
            System.out.println("PrintSpooler already initiallized. Returning existing instance");
        }
        return b;
    }


    public void submitJob(int jobId){
        if(jobs != null){
            jobs.add(jobId);
        }
    }
    public void getjobs(){
        System.out.println("PrintSpooler -  submitted jobs are: ");
        for (int job:jobs) {
            System.out.println(job);
        }
    }

}
```

Client.java

```
public class Client {
    public static void main(String[] args){
        PrintSpooler spooler = PrintSpooler.getPrintSpooler();
        spooler.submitJob(10);
        spooler.submitJob(20);
        spooler.submitJob(30);
        spooler.getjobs();
        PrintSpooler spooler2 = PrintSpooler.getPrintSpooler();
        spooler2.submitJob(40);
        spooler2.submitJob(50);
        spooler2.submitJob(60);
        spooler2.getjobs();
    }
}
```

Output:



```
PrintSpooler -  submitted jobs are:
10
20
30
PrintSpooler already initiallized. Returning existing PrintSpooler instance
PrintSpooler -  submitted jobs are:
10
20
30
40
50
60
```

We see that, when we tried to obtain the spooler for the second time, the existing one was returned instead of re instantiating it. **spooler2.getJobs()** returned all the jobs that were submitted to **spooler1** as well since both of them essentially refer to same single instance that was created the first time

Conclusion:

Thus, we can ensure that even in multi-threaded applications, all the dependents will have only one instance and all of them will have a global point of access to read or modify it.