**DOUBLE CHECK LOCKING PATTERN**

**What is DCL:-**

The DCL idiom was designed to support lazy initialization, which occurs when a class defers initialization of an owned object until it is actually needed:

**class** SomeClass

{

**private** Resource resource = **null**;

**public** Resource getResource()

{

**if** (resource == **null**)

resource = **new** Resource();

**return** resource;

}

}

Why would you want to defer initialization?

Perhaps creating a Resource is an expensive operation, and users of *SomeClass* might not actually call *getResource()* in any given run. In that case, you can avoid creating the Resource entirely. Regardless, the SomeClass object can be created faster if it doesn't have to also create a Resource at construction time. Delaying some initialization operations until a user actually needs their results can help programs start up faster.

What if you try to use *SomeClass* in a multithreaded application? Then a race condition results: two threads could simultaneously execute the test to see if resource is null and, as a result, initialize resource twice. In a multithreaded environment, you should declare *getResource()* to be synchronized.

Unfortunately, synchronized methods run much slower -- as much as 100 times slower -- than ordinary unsynchronized methods. One of the motivations for lazy initialization is efficiency, but it appears that in order to achieve faster program startup, you have to accept slower execution time once the program starts. That doesn't sound like a great trade-off.

DCL purports to give us the best of both worlds. Using DCL, the *getResource()*method would look like this:

**class** SomeClass

{

**private** Resource resource = **null**;

**public** Resource getResource()

{

**if** (resource == **null**)

{

**synchronized**(SomeClass.**class**)

{

**if** (resource == **null**)

resource = **new** Resource();

}

}

**return** resource;

}

After the first call to *getResource(),* resource is already initialized, which avoids the synchronization hit in the most common code path. DCL also averts the race condition by checking resource a second time inside the synchronized block; that ensures that only one thread will try to initialize resource. DCL seems like a clever optimization -- but it doesn't work.

**Another Example of DCL:-**

**public** **class** DCLSingleton

{

**private** **static** **volatile** DCLSingleton *\_instance* = **null**;

**private** DCLSingleton() { }

**public** **static** DCLSingleton instance()

{

**if** (*\_instance* == **null**)

{

// 1st check

**synchronized** (DCLSingleton.**class**)

{

**if** (*\_instance* == **null**)

// 2nd check

{

*\_instance* = **new** DCLSingleton();

}

}

}

**return** *\_instance*;

}

}