Singleton Pattern

This pattern solves the problem of ensuring that only one instance of a class is produced. For example, one may want to have only one system logger class or print spooler class available. Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

The singleton pattern involves just one class, which provides its own (that is, private) instantiation mechanism to ensure that only one instance is created, and provides a global access point to that instance. In this way, the same instance can be used everywhere, but no other object can create a new instance of it (using the new() keyword) because of the private constructor.

```
class Logger{
        private static Logger instance;
        private Logger(){
        System.out.println("Initializing logger...");
        …
        }
        public static synchronized Logger getInstance(){          Global access point
           if (instance == null)
                instance = new Logger();
           return instance;
        }
        public void doLogging(...){
                ...
        }
}
```

A client using this Logger will do something like: Logger.getInstance().doLogging(…).
The *synchronized* keyword is used in Java to make sure that multi-threaded applications accessing the same resources (such as I/O ports, loggers) share the resources without producing conflict.

Early instantiation using static field (to avoid costly synchronization)

In the following code, the Logger is instantiated when the class is loaded, not when it is accessed:

```
class Logger{
        private static Logger instance = new Logger();
        private Logger(){
        System.out.println("Initializing logger...");
        …
        }
        public static Logger getInstance(){
           return instance;
        }
        public void doLogging(...){
                ...
        }
}
```