With Java 8 lambda expressions, we can create Thread in java like below too because Runnable is a functional interface.

Thread t = new Thread(() -> {System.out.println("My Runnable");});

t.start();

**Thread Safety in Java**

Thread safety in java is the process to make our program safe to use in multithreaded environment, there are different ways through which we can make our program thread safe.

* Synchronization is the easiest and most widely used tool for thread safety in java.
* Use of Atomic Wrapper classes from *java.util.concurrent.atomic* package. For example AtomicInteger
* Use of locks from *java.util.concurrent.locks* package.
* Using thread safe collection classes, check this post for usage of [ConcurrentHashMap](http://www.journaldev.com/122/java-concurrenthashmap-example-iterator) for thread safety.
* Using volatile keyword with variables to make every thread read the data from memory, not read from thread cache.

When a method is synchronized, it locks the **Object**, if method is static it locks the **Class**, so it's always best practice to use *synchronized block* to lock the only sections of method that needs synchronization.

You should use the *lowest level of locking*, for example if there are multiple synchronized block in a class and one of them is locking the Object, then other synchronized blocks will also be not available for execution by other threads. When we lock an Object, it acquires lock on all the fields of the Object.

Java Synchronization works only in the same JVM, so if you need to lock some resource in multiple JVM environment, it will not work and you might have to look after some global locking mechanism.

Java synchronized keyword cannot be used for constructors and variables.

It is preferable to create a dummy private Object to use for synchronized block, so that it's reference can't be changed by any other code. For example if you have a setter method for Object on which you are synchronizing, it's reference can be changed by some other code leads to parallel execution of the synchronized block.

We should not use any object that is maintained in a constant pool, for example String should not be used for synchronization because if any other code is also locking on same String, it will try to acquire lock on the same reference object from [String pool](http://www.journaldev.com/797/what-is-java-string-pool) and even though both the codes are unrelated, they will lock each other.

To detect a deadlock in java, we need to look at the [java thread dump](http://www.journaldev.com/1053/java-thread-dump-visualvm-jstack-kill-3-jcmd) of the application, in last post I explained how we can generate thread dump using **VisualVM profiler or using jstack utility.**

For analyzing deadlock, we need to look out for the threads with state as **BLOCKED** and then the resources it’s **waiting to lock**. Every resource has a unique ID using which we can find which thread is already holding the lock on the object. For example Thread “t3” is waiting to lock 0x000000013df2f658 but it’s already locked by thread “t1”.

Once we analyze the deadlock situation and found out the threads which are causing deadlock, we need to make code changes to avoid deadlock situation.

### How to avoid deadlock in java

These are some of the guidelines using which we can avoid most of the deadlock situations.

* **Avoid Nested Locks**: This is the most common reason for deadlocks, avoid locking another resource if you already hold one. It’s almost impossible to get deadlock situation if you are working with only one object lock. For example, here is the another implementation of run() method without nested lock and program runs successfully without deadlock situation.
* public void run() {
* String name = Thread.currentThread().getName();
* System.out.println(name + " acquiring lock on " + obj1);
* synchronized (obj1) {
* System.out.println(name + " acquired lock on " + obj1);
* work();
* }
* System.out.println(name + " released lock on " + obj1);
* System.out.println(name + " acquiring lock on " + obj2);
* synchronized (obj2) {
* System.out.println(name + " acquired lock on " + obj2);
* work();
* }
* System.out.println(name + " released lock on " + obj2);
* System.out.println(name + " finished execution.");
* }
* **Lock Only What is Required**: You should acquire lock only on the resources you have to work on, for example in above program I am locking the complete Object resource but if we are only interested in one of it’s fields, then we should lock only that specific field not complete object.
* **Avoid waiting indefinitely**: You can get deadlock if two threads are waiting for each other to finish indefinitely using [thread join](http://www.journaldev.com/1024/java-thread-join-example). If your thread has to wait for another thread to finish, it’s always best to use join with maximum time you want to wait for thread to finish.

In general we follow below steps to create a singleton class:

1. Override the private constructor to avoid any new object creation with new operator.
2. Declare a private static instance of the same class
3. Provide a public static method that will return the singleton class instance variable. If the variable is not initialized then initialize it or else simply return the instance variable.

There are three ways through which we can achieve thread safety.

1. **Create the instance variable at the time of class loading.**

**Pros**:

* + Thread safety without synchronization
  + Easy to implement

**Cons**:

* + Early creation of resource that might not be used in the application.
  + The client application can’t pass any argument, so we can’t reuse it. For example, having a generic singleton class for database connection where client application supplies database server properties.

1. **Synchronize the getInstance() method**

**Pros**:

* + Thread safety is guaranteed.
  + Client application can pass parameters
  + Lazy initialization achieved

**Cons**:

* + Slow performance because of locking overhead.
  + Unnecessary synchronization that is not required once the instance variable is initialized.

1. **Use synchronized block inside the if loop**

**Pros**:

* + Thread safety is guaranteed
  + Client application can pass arguments
  + Lazy initialization achieved
  + Synchronization overhead is minimal and applicable only for first few threads when the variable is null.

**Cons**:

* + Extra if condition

Looking at all the three ways to achieve thread safety, I think third one is the best option