BEST PRACTICES FOR MULTITHREADING IN JAVA

This article lays down some of the best practices which you can use during your design or code reviews, and assumes that you are familiar with the basics of mutithreading in java available at articles/multithreading. I have learned many of these best practices from Effective Java by Joshua Bloch, which tells how to use java to best effect.

Best Practices for Multithreading in Java

1. **If you have shared mutable data, synchronize access to it. Synchronization** makes sure that an object is seen only in a consistent state by any thread, as it **ensures mutual exclusion**. **It also guarantees reliable communication between threads** as it ensures that each thread entering a synchronized region sees the effect of all previous modifications that were guarded by the same lock.
2. For reliable communication between threads, value written by one thread on a variable should be visible to the other immediately. Even though reading or writing a variable other than log or double is atomic, **java doesn't give a guarantee that a value written by one thread on a variable will be visible to the other immediately, unless both read and write to it is guarded by synchronization or the variable is a volatile variable.**
3. Volatile variable guarantees that a value written by one thread will be visible to the other immediately, but does not provide any mutual exclusion. Therefore, **when you need only reliable communication** by making variables visible to other threads, **but no mutual exclusion, use volatile**, as it has slight performance advantage over synchronization. However **for compound statements** like 'myVar++', you should **use synchronization or consider using atomic variables** in java.util.concurrent.atomic package like AtimicLong, which has a getAndIncrement() method, which is atomic.
4. **Alternate approaches to synchronization** are:
   a. **Share** only **immutable data**.
   b. **Do not share** at all.
   c. **Confine mutable data to a single thread**.
   d. **Safely publish effectively mutable data**. These are objects where one thread modifies the data for sometime and then share it with other threads, so that synchronization is only required for the act of sharing.
5. **Inside a synchronized region, never call a method designed for inheritance or one provided by client** like a method of an object passed in, as these will allow client to take control and may result in deadlock. We should use alien methods outside synchronized region.
6. **Over use of synchronization should be avoided** as it might result in deadlock, reduced performance due to improper parallelization and even reduce JVM's code optimization scope. You should do only very less work as required inside synchronized regions. **If a code can be moved outside a synchronized region, it should be moved out**.

7. **Should be careful while calling other methods holding the same lock** and making use of Java's reentrant locking, **as there is a chance to call an unrelated operation in progress guarded by the same lock**. If we call alien methods they might also do so and we won't have any control.
8. **You must synchronize access to mutable static fields** as there can be no guarantee that unrelated clients might do it externally. However **for other** variables and methods, in general, or when in doubt whether to synchronize, **we can decide not to synchronize, but should document** that it is not thread safe.
9. **You should prefer using internal locks to entire object locking**, by not synchronizing on the 'this' object lock. **You can create some private lock objects** (private Object lock = new Object();) and use them for related operations. When you synchronize internally you can make use of many techniques cuch as lock splitting, lock stripping, and non-blocking concurrency control to achieve high concurrency.
10. When writing new code or refactoring code, **use concurrency utilities** such as executor framework, concurrent collections, and synchronizers **in preference to wait and notify**.
11. When writing new code or refactoring code, **do not use threads directly**, but **use the executer framework** that consists of executors and tasks, that splits the duties of a thread. A task is the actual unit of work, and the executor service executes them. A task can be a Runnable or a Callable. Callable is like Runnable except that it returns value. **Executor service can do many things such as wait for tasks to complete, wait for graceful termination, retrieve the results of tasks etc**.
12. **Use the new concurrent collections such as a ConcurrentHashMap in preference to synchronized collections like Collections.synchronizedMap or Hashtable**. ConcurrentHashMap extends Map and include methods cuch as putIfAbsent. Refer to java.util.concurrent package javadoc to see if you already have a concurrent collection for your requirement.
13. **Decide the best thread pool implementation** while working with executors based on requirement. **For example, Executors.newCashedThreadPool** can be used for short lived asynchronous tasks on a lightly loaded server, whereas for long running threads on heavily loaded production servers use**Executors.newFixedThreadPool.** Refer to java doc when in doubt.
14. **Use Synchronizers for coordinating thread activities like waiting for another thread.** Examples for synchronizers are CountDownLatch, Semaphore, CyclicBarrier and Exchanger.
15. The **wait method**, if still used in code, **must be invoked inside a synchronized region** that locks on the object on which it is invoked **and** also, **we should use the wait-loop idiom** always, where you invoke wait inside a loop. The loop will test the condition before and after waiting. Testing the condition before is required to ensure liveness because if the condition already holds and notify or notifyAll has been already invoked, there is no guarantee that thread will ever wake up from wait. Testing the condition after is necessary because if the thread proceeds with the action when the condition does not hold, the results can be unexpected. A thread might wake up when the condition does not hold due to many reasons such as another thread could have changed the condition between the notify invocation and wakeup, another thread could have invoked notify accidentally, a notifyAll was invoked even if condition is not satisfied for some threads, or a thread could rarely wakeup in the absence of notify, known as a spurious wakeup.

16. **Among notify and notifyAll**, if still used, **it is better to use notifyAll along with wait-loop idiom,**because notifyAll guarantees that it will wake all the threads that need to be awakened, and the wait-loop idiom will make any threads wait again until their condition is met.
17. We need to **document clearly how a class behaves during concurrent use**. The synchronized modifier in a method declaration is an implementation detail not part of its exported API and normally, javadoc does not include the synchronized modifier in its output.
18. A class must **clearly document what level of thread safety it supports like immutable** and**unconditionally thread-safe** where external synchronization is not required; **conditionally thread-safe** and**not thread-safe**, where external synchronization is required; **or thread-hostile**, where it is not thread safe even with external synchronization. Conditionally thread safe classes must document which method invocation sequences require external synchronization, and which lock to acquire when executing these sequences.
19. **If more than one thread shares a lazily initialized field**, such as a singleton object reference, **there should be some form of synchronization**. **If the variable is a static variable, a holder class idiom is the preferred** approach and **if the variable is non-static, a double-check idiom is preferred**. In double-check idiom there won't be any locking if the field is already initialized and hence you have to use volatile keyword. The behavior of volatile was not properly defined until java 5 and hence it will work well only from java 5. **Lazy initialization should be used only if needed**, and without lazy initialization, the best way to implement singleton is to use enums. *Refer to the article on singleton for details on all these approaches.*
20. **Program should not rely on thread scheduler dependent features such as thread priority, Thread.yield etc**, or your program will not be portable, as the behavior of these features varies from JVM to JVM.
21. **Use Thread.sleep(1) instead of Thread.yield()** as Thread.yield can, according to java spec, simply return control to the caller without doing anything; and even Thread.sleep(0) can return immediately.
22. You should **design the size of the thread pool such that the average number of runnable threads is not significantly greater than the number of processors**. You should also keep tasks reasonable small and independent of one another.
23. **You should avoid the use of thread groups** because they are obsolete, much functionality has been deprecated and others are not used much. The API also suffers from issues like thread safety, but are not fixed as they are obsolete and alternatives exist.
24. From Java 5, you can **use Thread's setUncaughtException method instead of ThreadGroup's uncaughtException method**, to do something with the uncaught exception, for example to direct stack traces to a log file.

Source : http://javajee.com/best-practices-for-multithreading-in-java