Feel free to contact **santhosh_arena@yahoo.co.in**

SCJP java Programmer certification Exam And Training. This site is entirely independent of Sun Microsystems Inc, The Sun Certified Java Programmers Exam (SCJP) is the internationally recognized java certification exam. I am offering free mock Exam preparation questions, practice tests, tutorials, certification faq and sample code. This page contains a very detailed tutorial organized by topic. At the end of the tutorial you can learn from your mistakes and apply your concepts on the free mock exams java certification practice tests.

Chapter 8 : Locks, Monitors and Synchronization

- Every object has a lock (for every synchronized code block). At any moment, this lock is controlled by at most one thread.
- A thread that wants to execute an object's synchronized code must acquire the lock of the object. If it cannot acquire the lock, the thread goes into blocked state and comes to ready only when the object's lock is available.
- When a thread, which owns a lock, finishes executing the synchronized code, it gives up the lock.
- Monitor (a.k.a **Semaphore)** is an object that can block and revive threads, an object that controls client threads. Asks the client threads to wait and notifies them when the time is right to continue, based on its state. In strict Java terminology, any object that has some synchronized code is a monitor.
- 2 ways to synchronize:

    1. Synchronize the entire method
        - Declare the method to be synchronized - very common practice.
        - Thread should obtain the object's lock.
    2. Synchronize part of the method
        - Have to pass an arbitrary object which lock is to be obtained to execute the synchronized code block (part of a method).
        - We can specify "this" in place object, to obtain very brief locking - not very common.

- wait - points to remember
    - § calling thread gives up CPU
    - § calling thread gives up the lock

- § calling thread goes to monitor's waiting pool
- § wait also has a version with timeout in milliseconds. Use this if you're not sure when the current thread will get notified, this avoids the thread being stuck in wait state forever.

· notify - points to remember

- § one thread gets moved out of monitor's waiting pool to ready state
- § notifyAll moves all the threads to ready state
- § Thread gets to execute must re-acquire the lock of the monitor before it can proceed.

· Note the differences between blocked and waiting.

Blocked	Waiting
Thread is waiting to get a lock on the monitor. (or waiting for a blocking i/o method)	Thread has been asked to wait. (by means of wait method)
Caused by the thread tried to execute some synchronized code. (or a blocking i/o method)	The thread already acquired the lock and executed some synchronized code before coming across a wait call.
Can move to ready only when the lock is available. (or the i/o operation is complete)	Can move to ready only when it gets notified (by means of notify or notifyAll)

· Points for complex models:

1. Always check monitor's state in a while loop, rather than in an if statement.
2. Always call notifyAll, instead of notify.

· **Class locks** control the static methods.

· **wait** and **sleep** must be enclosed in a **try/catch** for **InterruptedException**.

· A single thread can obtain multiple locks on multiple objects (or on the same object)

· A thread owning the lock of an object can call other synchronous methods on the same object. (this is another lock) Other threads can't do that. They should wait to get the lock.

- Non-synchronous methods can be called at any time by any thread.
- Synchronous methods are **re-entrant**. So they can be called recursively.
- Synchronized methods **can be overrided** to be non-synchronous. Synchronized behavior affects only the original class.
- ***Locks on inner/outer objects are independent***. Getting a lock on outer object doesn't mean getting the lock on an inner object as well, that lock should be obtained separately.
- wait and notify should be called from synchronized code. This ensures that while calling these methods the thread always has the lock on the object. If you have wait/notify in non-synchronized code compiler won't catch this. At runtime, if the thread doesn't have the lock while calling these methods, an **IllegalMonitorStateException** is thrown.
- Deadlocks can occur easily. e.g, Thread A locked Object A and waiting to get a lock on Object B, but Thread B locked Object B and waiting to get a lock on Object A. They'll be in this state forever.
- It's the programmer's responsibility to avoid the deadlock. Always get the locks in the same order.
- While 'suspended', the thread keeps the locks it obtained - so suspend is deprecated in 1.2
- Use of **stop** is also deprecated; instead use a flag in run method. Compiler won't warn you, if you have statements after a call to stop, even though they are not reachable.