Contents

[1. Introduction 2](#_Toc453585733)

[1.1 Thread States 2](#_Toc453585734)

[1.2 Monitors 4](#_Toc453585735)

[1.3 Timer 4](#_Toc453585736)

[1.4 RMI 4](#_Toc453585737)

[2. Fundamentals 4](#_Toc453585738)

[2.1 Thread Safety 4](#_Toc453585739)

[2.2 Sharing Objects 6](#_Toc453585740)

[2.3 Composing Objects 7](#_Toc453585741)

[2.4 Building Blocks 8](#_Toc453585742)

[3. Structuring Concurrent Applications 8](#_Toc453585743)

[3.1 Executor Service 9](#_Toc453585744)

[3.2 Lock Interface 9](#_Toc453585745)

[3.3 Condition Interface 10](#_Toc453585746)

[4. JVM Memory Model 12](#_Toc453585747)

[4.1 Difference between Java Heap Space and Stack Memory 13](#_Toc453585748)

[5. Performance and Scalability 13](#_Toc453585749)

[5.1 Amdahl’s Law 14](#_Toc453585750)

[6. Advanced Topics 15](#_Toc453585751)

[6.1 Lock 15](#_Toc453585752)

[6.2 ReadWriteLock 15](#_Toc453585753)

[6.3 Condition Queues 16](#_Toc453585754)

[6.4 Condition Object 16](#_Toc453585755)

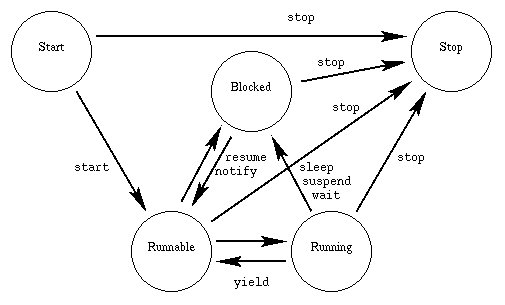
# Introduction

## Thread States

|  |  |
| --- | --- |
| Method Signature | Description |
| String getName() | Retrieves the name of running thread in current context in String format |
| void start() | This method will start new thread of execution by calling run() method of Thread/runnable object. |
| void run() | This method is entry point of thread. Execution of thread starts from this method. |
| void sleep(int sleeptime) | This method suspend thread for mentioned time duration in argument (sleeptime in ms) |
| void yield() | By invoking this method the current thread pause its execution temporarily and allow other threads to execute. |
| void join() | This method used to queue up thread in execution. Once called on thread, current thread will wait till calling thread completes its execution |
| boolean isAlive() | This method will check if thread is alive or dead |

**Thread States**

The thread scheduler's job is to move threads in and out of the running state. While the thread scheduler can move a thread from the running stateback to runnable, other factors can cause a thread to move out of running, but notback to runnable. One of these is when the thread's run()method completes, inwhich case the thread moves from the running state directly to the dead state.



**New/Start:**

This is the state the thread is in after the Thread instance has beencreated, but the start() method has not been invoked on the thread. It isa live Thread object, but not yet a thread of execution. At this point, thethread is considered not alive.

**Runnable:**

This means that a thread can be run when the time-slicing mechanism has CPU cycles available for the thread. Thus, the thread might or might not be running at any moment, but there’s nothing to prevent it from being run if the scheduler can arrange it. That is, it’s not dead or blocked.

**Running:**

This state is important statewhere the action is. This is the state athread is in when the thread scheduler selects it (from the runnable pool) tobe the currently executing process. A thread can transition out of a runningstate for several reasons, including because "the thread scheduler felt like it". There areseveral ways to get to the runnable state, but only one way to get to the runningstate: the scheduler chooses a thread from the runnable pool of thread.

**Blocked:**

The thread can be run, but something prevents it. While a thread is in the blocked state, the scheduler will simply skip it and not give it any CPU time. Until a thread reenters the runnable state, it won’t perform any operations. Blocked state has some sub states as below,

* **Blocked on I/O :** The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.
* **Blocked for join completion:** The thread can come on this state because of waiting the completion of another thread.
* **Blocked for lock acquisition:** The thread can come on this state because of waiting to acquire the lock of an object.

**Dead:**

A thread in the dead or terminated state is no longer schedulable and will not receive any CPU time. Its task is completed, and it is no longer runnable. One way for a task to die is by returning from its run( ) method, but a task’s thread can also be interrupted, as you’ll see shortly.

Lets take an example of Java program to demonstrate various thread state and methods of thread class.

## Monitors

Java's monitor supports two kinds of thread synchronization: mutual exclusion and cooperation. Mutual exclusion, which is supported in the Java virtual machine via object locks, enables multiple threads to independently work on shared data without interfering with each other. Cooperation, which is supported in the Java virtual machine via the wait and notify methods of class Object, enables threads to work together towards a common goal.

## Timer

A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

Timer. Timer is a convenience mechanism for scheduling tasks to run at a later time, either once or periodically. The introduction of a Timer can complicate an otherwise sequential program, because TimerTasks are executed in a thread managed by the Timer, not the application.

## RMI

RMI lets you invoke methods on objects running in another JVM. When you call a remote method with RMI, the method arguments are packaged (marshaled) into a byte stream and shipped over the network to the remote JVM, where they are unpacked (unmarshaled) and passed to the remote method.

# Fundamentals

## Thread Safety

Whether an object needs to be thread-safe depends on whether it will be accessed from multiple threads. This is a property of how the object is used in a program, not what it does.

If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken. There are three ways to fix it: Don't share the state variable across threads; Make the state variable immutable; or Use synchronization whenever accessing the state variable.

Thread safety may be a term that is applied to code, but it is about state, and it can only be applied to the entire body of code that encapsulates its state, which may be an object or an entire program.

**A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.**

No set of operations performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state.

**Class invariants** are methods which check the validity of an object's state (its data). The idea is to define validation methods for fields, and to perform these validations whenever the fields change.

A **race condition** occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing.

The java.util.concurrent.atomic package contains atomic variable classes for effecting atomic state transitions on numbers and object references. E.g. AtomicLong.

Where practical, use existing thread-safe objects, like AtomicLong, to manage your class's state. It is simpler to reason about the possible states and state transitions for existing thread-safe objects than it is for arbitrary state variables, and this makes it easier to maintain and verify thread safety.

When a thread requests a lock that is already held by another thread, the requesting thread blocks. But because intrinsic locks are **reentrant**, if a thread tries to acquire a lock that it already holds, the request succeeds. Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis.

Reentrancy is implemented by associating with each lock an acquisition count and an owning thread. When the count is zero, the lock is considered unheld. When a thread acquires a previously unheld lock, the JVM records the owner and sets the acquisition count to one. If that same thread acquires the lock again, the count is incremented, and when the owning thread exits the synchronized block, the count is decremented. When the count reaches zero, the lock is released.

if(!vector.contains(element)) {

vector.add(element);

}

This attempt at a put-if-absent operation has a race condition, even though both contains and add are atomic. While synchronized methods can make individual operations atomic, additional locking is required when multiple operations are combined into a compound action.

Atomic variables are useful for effecting atomic operations on a single variable, but since we are already using synchronized blocks to construct atomic operations, using two different synchronization mechanisms would be confusing and would offer no performance or safety benefit.

Whenever you use locking, you should be aware of what the code in the block is doing and how likely it is to take a long time to execute. Holding a lock for a long time, either because you are doing something compute-intensive or because you execute a potentially blocking operation, introduces the risk of liveness or performance problems.

## Sharing Objects

That chapter was about using synchronization to prevent multiple threads from accessing the same data at the same time; this chapter examines techniques for sharing and publishing objects so they can be safely accessed by multiple threads. Together, they lay the foundation for building thread-safe classes and safely structuring concurrent applications using the java.util.concurrent library classes.

*Reading data without synchronization is analogous to using the READ\_UNCOMMITTED isolation level in a database, where you are willing to trade accuracy for performance. However, in the case of unsynchronized reads, you are trading away a greater degree of accuracy, since the visible value for a shared variable can be arbitrarily stale.*

Locking is not just about mutual exclusion; it is also about memory visibility. To ensure that all threads see the most up-to-date values of shared mutable variables, the reading and writing threads must synchronize on a common lock.

**Volatile variables**

The Java language also provides an alternative, weaker form of synchronization, volatile variables, to ensure that updates to a variable are propagated predictably to other threads. When a field is declared volatile, the compiler and runtime are put on notice that this variable is shared and that operations on it should not be reordered with other memory operations. Volatile variables are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread.

Volatile reads are only slightly more expensive than non-volatile reads on most current processor architectures.

Use volatile variables only when they simplify implementing and verifying your synchronization policy; avoid using volatile variables when verifying correctness would require subtle reasoning about visibility. Good uses of volatile variables include ensuring the visibility of their own state, that of the object they refer to, or indicating that an important lifecycle event (such as initialization or shutdown) has occurred.

You can use volatile variables only when all the following criteria are met:

* Writes to the variable do not depend on its current value, or you can ensure that only a single thread ever updates the value;
* The variable does not participate in invariants with other state variables; and
* Locking is not required for any other reason while the variable is being accessed.

Do not allow the **this** reference to escape during construction.

Stack thread confinement: Local variables with object can only be reached via other methods. No reference is exposed outside the class.

ThreadLocal: A more formal means of maintaining thread confinement.

Immutability: An object whose state cannot be changed after construction. Are inherently thread safe.

An object is immutable if:

* Its state cannot be modified after construction;
* All its fields are final;

There is a difference between an object being immutable and the reference to it being immutable. Program state stored in immutable objects can still be updated by “replacing” immutable objects with a new instance holding new state;

Immutable objects can be used safely by any thread without additional synchronization, even when synchronization is not used to publish them.

***public static Holder holder = new Holder(42);***

Static initializers are executed by the JVM at class initialization time; because of internal synchronization in the JVM, this mechanism is guaranteed to safely publish any objects initialized in this way

The publication requirements for an object depend on its mutability:

* Immutable objects can be published through any mechanism;
* Effectively immutable objects must be safely published;
* Mutable objects must be safely published, and must be either threadsafe or guarded by a lock.

The most useful policies for using and sharing objects in a concurrent program are:

* **Thread-confined.** A thread-confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread.
* **Shared read-only.** A shared read-only object can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by any thread. Shared read-only objects include immutable and effectively immutable objects.
* **Shared thread-safe.** A thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization.
* **Guarded.** A guarded object can be accessed only with a specific lock held. Guarded objects include those that are encapsulated within other thread-safe objects and published objects that are known to be guarded by a specific lock.

## Composing Objects

The design process for a thread-safe class should include these three basic elements:

* Identify the variables that form the object's state;
* Identify the invariants that constrain the state variables;
* Establish a policy for managing concurrent access to the object's state.

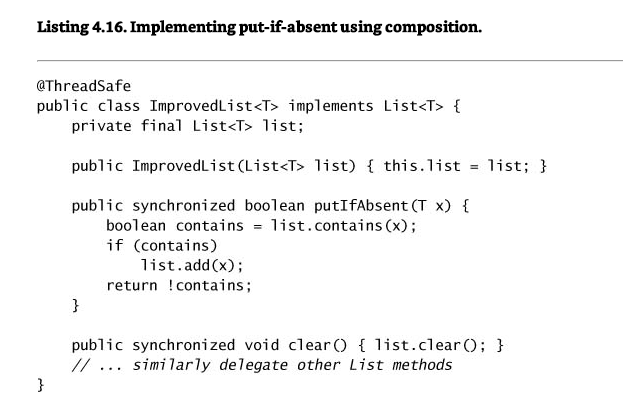
Confinement makes it easier to build thread-safe classes because a class that confines its state can be analyzed for thread safety without having to examine the whole program.

**Java Monitor Pattern**

An object following the Java monitor pattern encapsulates all its mutable state and guards it with the object's own intrinsic lock. E.g. Vector, Hashtable.

Private final Object mylock = new Object();

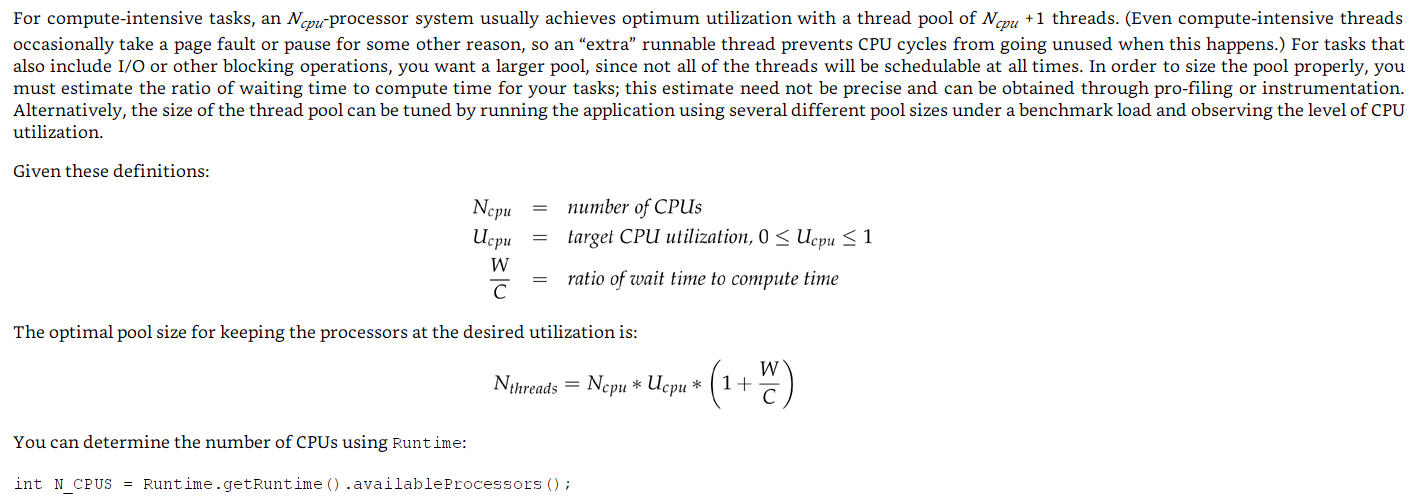
There are advantages to using a private lock object instead of an object's intrinsic lock (or any other publicly accessible lock). Making the lock object private encapsulates the lock so that client code cannot acquire it, whereas a publicly accessible lock allows client code to participate in its synchronization policy— correctly or incorrectly. Clients that improperly acquire another object's lock could cause liveness problems, and verifying that a publicly accessible lock is properly used requires examining the entire program rather than a single class.



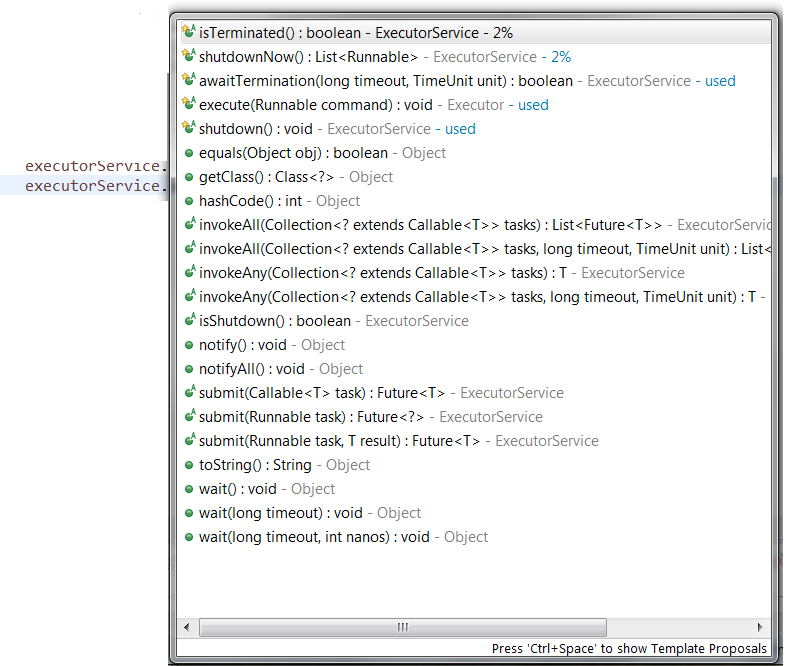
ImprovedList adds an additional level of locking using its own intrinsic lock. It does not care whether the underlying List is thread-safe, because it provides its own consistent locking that provides thread safety even if the List is not thread-safe or changes its locking implementation. While the extra layer of synchronization may add some small performance penalty, [ 7] the implementation in ImprovedList is less fragile than attempting to mimic the locking strategy of another object. In effect, we've used the Java monitor pattern to encapsulate an existing List, and this is guaranteed to provide thread safety so long as our class holds the only outstanding reference to the underlying List.

## Building Blocks

# Structuring Concurrent Applications



## Executor Service

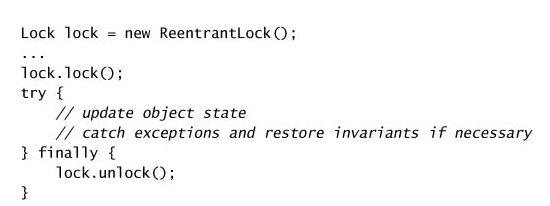


## Lock Interface

Additional tryLock functionality. More control.

Intrinsic locking works fine in most situations but has some functional limitations— it is not possible to interrupt a thread waiting to acquire a lock, or to attempt to acquire a lock without being willing to wait for it forever. Intrinsic locks also must be released in the same block of code in which they are acquired; this simplifies coding and interacts nicely with exception handling, but makes non-blockstructured locking disciplines impossible. None of these are reasons to abandon synchronized, but in some cases a more flexible locking mechanism offers better liveness or performance.

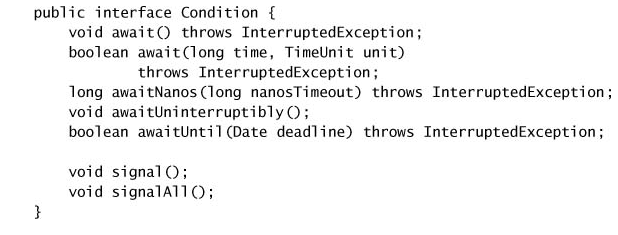
Using timed or polled lock acquisition (tryLock) lets you regain control if you cannot acquire all the required locks, release the ones you did acquire, and try again (or at least log the failure and do something else). Listing 13.3 shows an alternate way of addressing the dynamic ordering deadlock from Section 10.1.2: use tryLock to attempt to acquire both locks, but back off and retry if they cannot both be acquired.



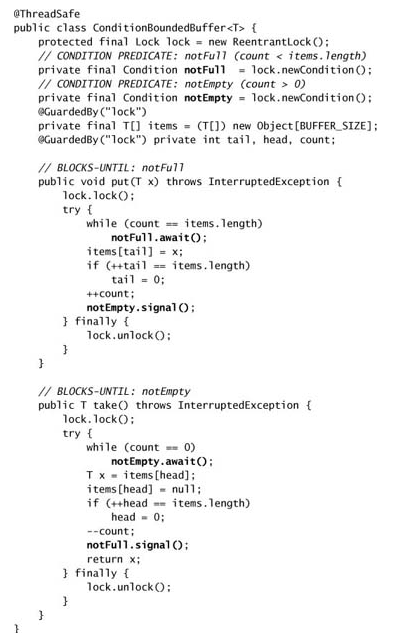
## Condition Interface

Counting Semaphore type behaviour: Producer consumer problem

A Condition is associated with a single Lock, just as a condition queue is associated with a single intrinsic lock; to create a Condition, call Lock.newCondition on the associated lock. And just as Lock offers a richer feature set than intrinsic locking, Condition offers a richer feature set than intrinsic condition queues: multiple wait sets per lock, interruptible and uninterruptible condition waits, deadline-based waiting, and a choice of fair or nonfair queueing.

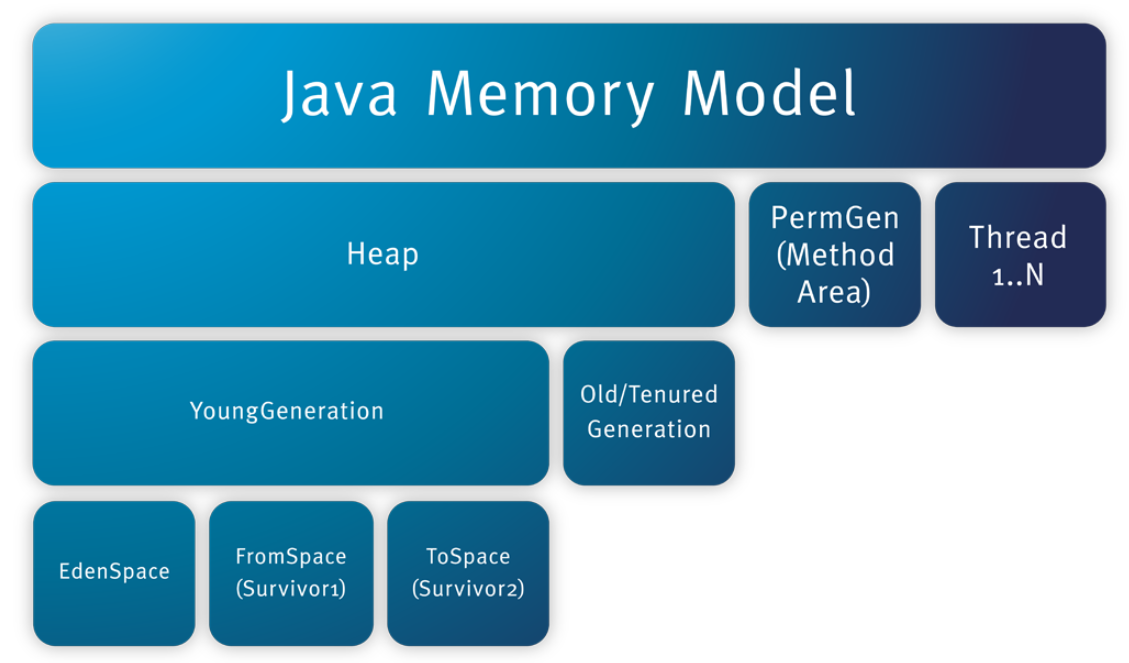


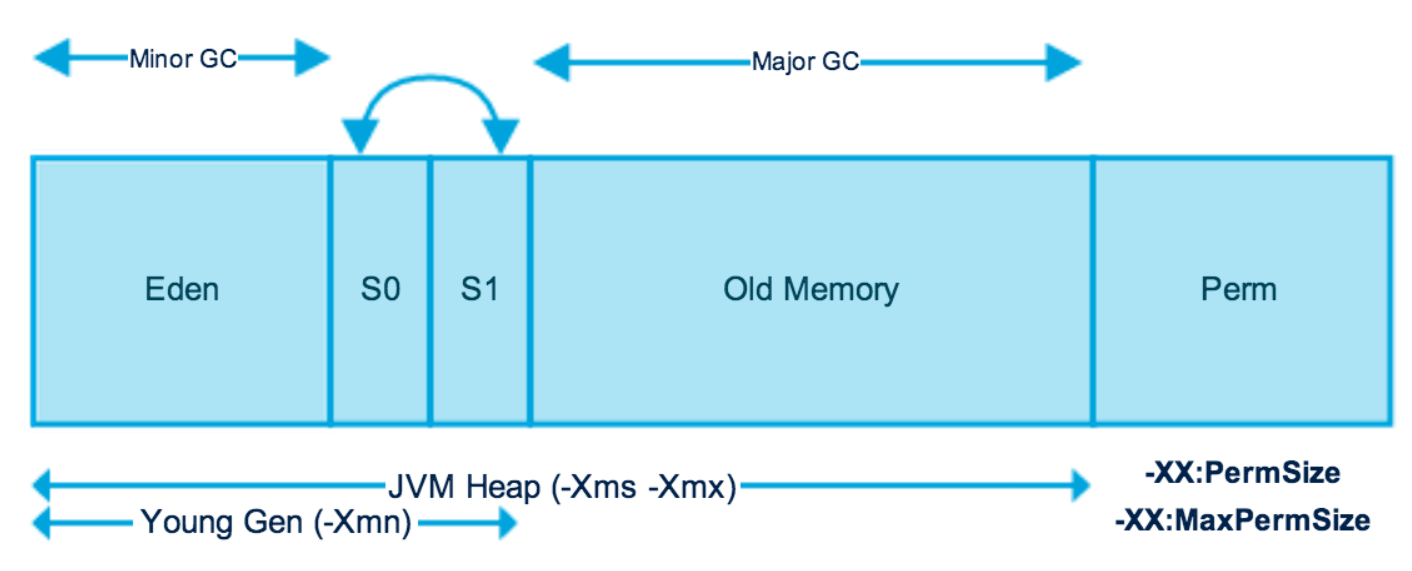
Example:



# JVM Memory Model

The Java memory model specifies how the Java virtual machine works with the computer's memory (RAM). The Java virtual machine is a model of a whole computer so this model naturally includes a memory model - AKA the Java memory model.





* Young generation is the place where all the new objects are created.
* Most of the newly created objects are located in the Eden memory space.
* When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.
* Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it’s full. Old Generation Garbage Collection is called **Major GC** and usually takes longer time.
* Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.
* Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

## Difference between Java Heap Space and Stack Memory

Based on the above explanations, we can easily conclude following differences between Heap and Stack memory.

1. Stack memory only contains local primitive variables and reference variables to objects in heap space.
2. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
3. Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it.
4. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
5. Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at [Java Garbage Collection](http://www.journaldev.com/2856/java-jvm-memory-model-and-garbage-collection-monitoring-tuning).
6. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
7. We can use **-Xms** and **-Xmx** JVM option to define the startup size and maximum size of heap memory. We can use **-Xss** to define the stack memory size.
8. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
9. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

# Performance and Scalability

Most performance decisions involve multiple variables and are highly situational.

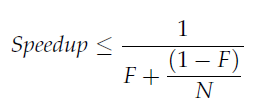
Before deciding that one approach is “faster” than another, ask yourself some questions:

* What do you mean by “faster”?
* Under what conditions will this approach *actually* be faster? Under light or heavy load? With large or small data sets? Can you support your answer with measurements?
* How often are these conditions likely to arise in your situation? Can you support your answer with measurements?
* Is this code likely to be used in other situations where the conditions maybe different?
* What hidden costs, such as increased development or maintenance risk, are you trading for this improved performance? Is this a good trade-off?

***The quest for performance is probably the single greatest source of concurrency bugs.***

## Amdahl’s Law

Amdahl’s law describes how much a program can theoretically be sped up by additional computing resources, based on the proportion of parallelizable and serial components. If F is the fraction of the calculation that must be executed serially, then Amdahl’s law says that on a machine with N processors, we can achieve a speedup of at most:



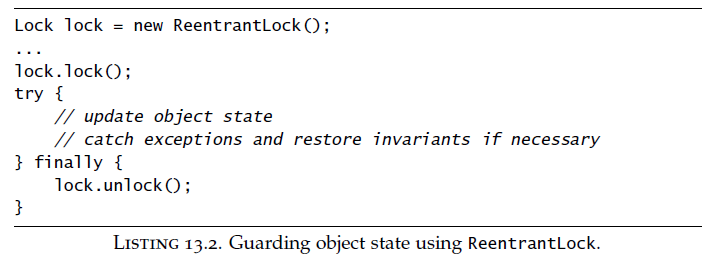
As N approaches infinity, the maximum speedup converges to 1/F, meaning that a program in which fifty percent of the processing must be executed serially can be sped up only by a factor of two, regardless of how many processors are available, and a program in which ten percent must be executed serially can be sped up by at most a factor of ten.

# Advanced Topics

Lock = synchronized;

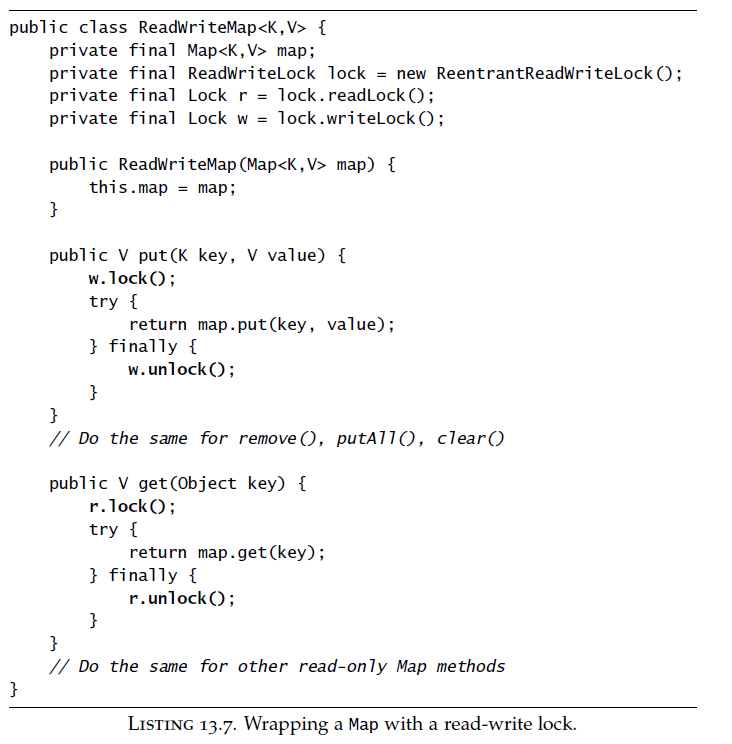
Condition = wait/notify;

## Lock



## ReadWriteLock

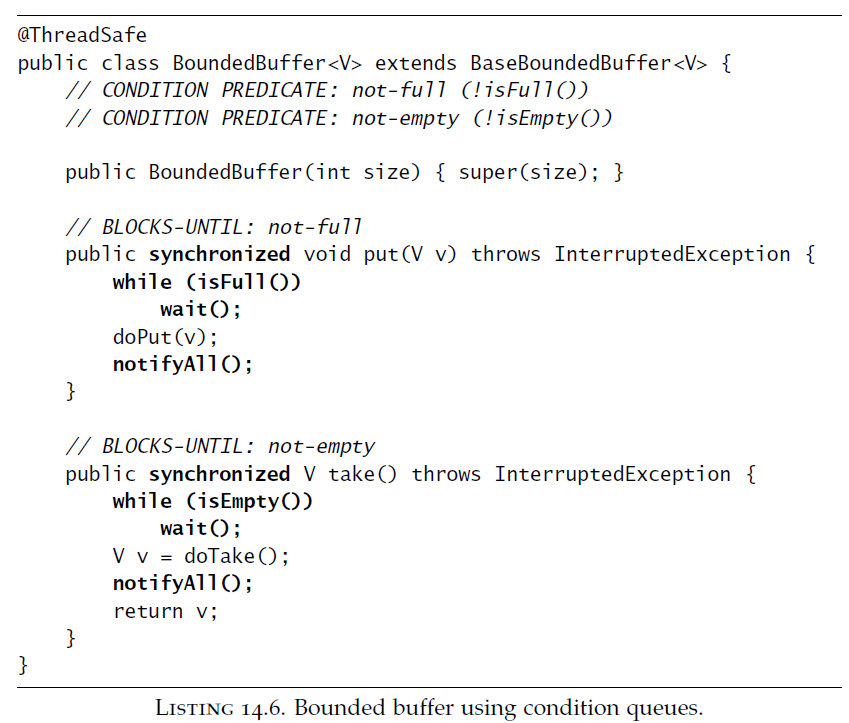
The locking strategy implemented by read-write locks allows multiple simultaneous readers but only a single writer. Like Lock, ReadWriteLock admits multiple implementations that can vary in performance, scheduling guarantees, acquisition preference, fairness, or locking semantics.



## Condition Queues

Just as each Java object can act as a lock, each object can also act as a condition queue, and the **wait**, **notify**, and **notifyAll** methods in **Object** constitute the API for intrinsic condition queues.

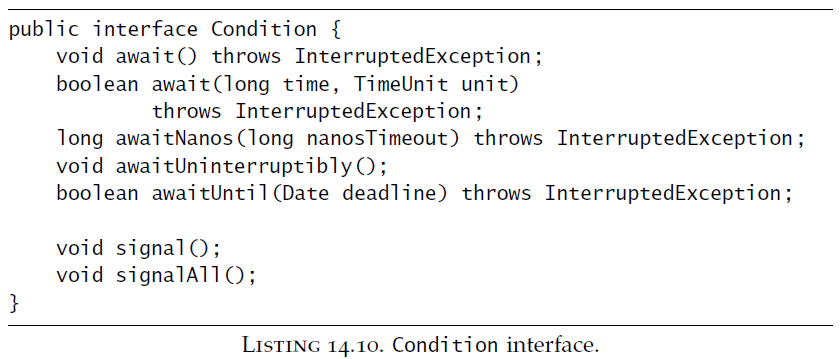
**Object.wait** atomically releases the lock and asks the OS to suspend the current thread, allowing other threads to acquire the lock and therefore modify the object state. Upon waking, it reacquires the lock before returning. Intuitively, calling wait means “I want to go to sleep, but wake me when something interesting happens”, and calling the notification methods means “something interesting happened”.



## Condition Object

Intrinsic condition queues have several drawbacks. Each intrinsic lock can have only one associated condition queue, which means that in classes like BoundedBuffer multiple threads might wait on the same condition queue for different condition predicates, and the most common pattern for locking involves exposing the condition queue object. Both of these factors make it impossible to enforce the uniform waiter requirement for using notify. If you want to write a concurrent object with multiple condition predicates, or you want to exercise more control over the visibility of the condition queue, the explicit Lock and Condition classes offer a more flexible alternative to intrinsic locks and condition queues.

A Condition is associated with a single Lock, just as a condition queue is associated with a single intrinsic lock; to create a Condition, call Lock.newCondition on the associated lock. And just as Lock offers a richer feature set than intrinsic locking, Condition offers a richer feature set than intrinsic condition queues: multiple wait sets per lock, interruptible and uninterruptible condition waits, deadline-based waiting, and a choice of fair or nonfair queueing.



Unlike intrinsic condition queues, you can have as many Condition objects per Lock as you want. Condition objects inherit the fairness setting of their associated Lock; for fair locks, threads are released from Condition.await in FIFO order.

