What is Concurrency?

Answer:- Concurrency is the ability to run several programs or several parts of a program in parallel. Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware. e.g. modern computers has several CPU’s or several cores within one CPU, program can utilize all cores for some part of processing; thus completing task much before in time in comparison to sequential processing.

The backbone of java concurrency are threads. A thread is a lightweight process which has its own call stack, but can access shared data of other threads in the same process. A Java application runs by default in one process. Within a Java application you can work with many threads to achieve parallel processing or concurrency.

What makes java application concurrent?

Answer:- The very first class, you will need to make a java class concurrent, is **java.lang.Thread class**. This class is the basis of all concurrency concepts in java. Then you have **java.lang.Runnable interface** to abstract the thread behavior out of thread class. Other classes you will need to build advanced applications can be found at **java.util.concurrent package** added in Java 1.5.

Is java concurrency really that simple?

Answer:- Some way yes and some way no, Concurrent applications usually have more complex design in comparison to single threaded application. Code executed by multiple threads accessing shared data need special attention. Errors arising from incorrect thread synchronization are very hard to detect, reproduce and fix. They usually shows up in higher environments like production, and replicating the error is sometimes not possible in lower environments.Apart from complex defects, concurrency requires more resources to run the application. So make sure, you have sufficient resources in your kitty.

What are the Multi Threadings Benefits?

Answer:- Better resource utilization. Simpler program design in some situations.More responsive programs.

What are the Multithreading Costs?

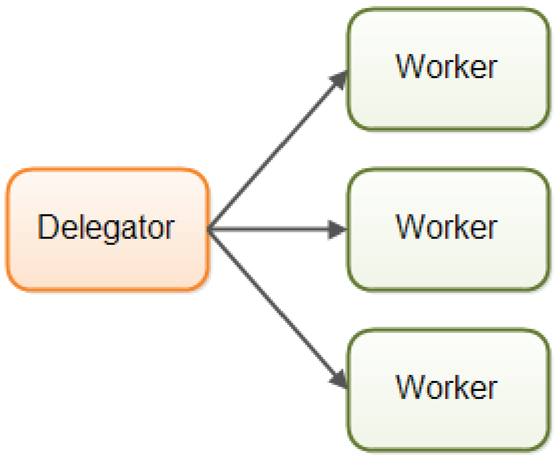
Answer:- More complex design, Context Switching Overhead,Increased Resource Consumption.

Concurrency Models:-

Concurrent systems can be implemented using different concurrency models. A concurrency model specifies how threads in the the system collaborate to complete the jobs they are are given. Different concurrency models split the jobs in different ways, and the threads may communicate and collaborate in different ways. There are following Popular Concurrency Models:-

- **Parallel Workers.**
- **Assembly Lines.**
- **Functional Parallelism**

Parallel Workers



In the parallel worker concurrency model a delegator distributes the incoming jobs to different workers. Each worker completes the full job. The workers work in parallel, running in different threads, and possibly on different CPUs.

If the parallel worker model was implemented in a car factory, each car would be produced by one worker. The worker would get the specification of the car to build, and would build everything from start to end.

The parallel worker concurrency model is the most commonly used concurrency model in Java applications (although that is changing). Many of the concurrency utilities in the **java.util.concurrent Java package** are designed for use with this model. You can also see traces of this model in the design of the Java Enterprise Edition application servers.

Parallel Workers Advantages:-

The advantage of the parallel worker concurrency model is that it is easy to understand. To increase the parallelization of the application you just add more workers.

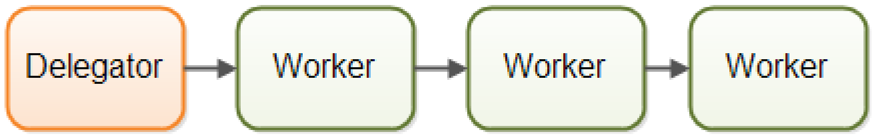
Parallel Workers Disadvantages:-

- **Shared State Can Get Complex.** In reality the parallel worker concurrency model is a bit more complex.The shared workers often need access to some kind of shared data, either in memory or in a shared database. Some of this shared state is in communication mechanisms like job queues. But some of this shared state is business data, data caches, connection pools to the database etc.As soon as shared state sneaks into the parallel worker concurrency model it starts getting complicated. The threads need to access the shared data in a way that makes sure that changes by one thread are visible to the others (pushed to main memory and not just stuck in the CPU cache of the CPU executing the thread). Threads need to avoid race conditions, deadlock and many other shared state concurrency problems.

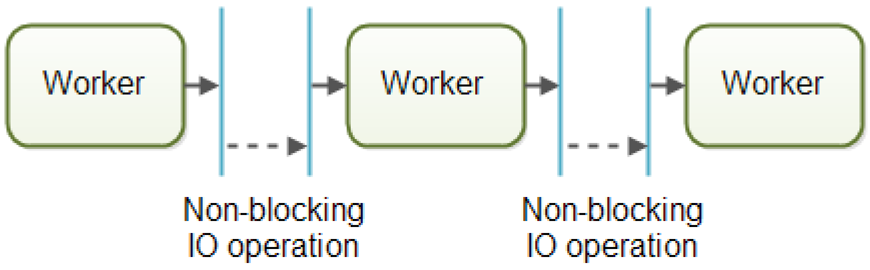
Additionally, part of the parallelization is lost when threads are waiting for each other when accessing the shared data structures. Many concurrent data structures are blocking, meaning one or a limited set of threads can access them at any given time. This may lead to contention on these shared data structures. High contention will essentially lead to a degree of serialization of execution of the part of the code that access the shared data structures.

- **Stateless Workers:-** Shared state can be modified by other threads in the system. Therefore workers must re-read the state every time it needs it, to make sure it is working on the latest copy. This is true no matter whether the shared state is kept in memory or in an external database. A worker that does not keep state internally (but re-reads it every time it is needed) is called stateless .Re-reading data every time you need it can get slow. Especially if the state is stored in an external database.
- **Job Ordering is Nondeterministic:-**There is no way to guarantee which jobs are executed first or last. Job A may be given to a worker before job B, yet job B may be executed before job A.The nondeterministic nature of the parallel worker model makes it hard to reason about the state of the system at any given point in time. It also makes it harder (if not impossible) to guarantee that one jobs happens before another.

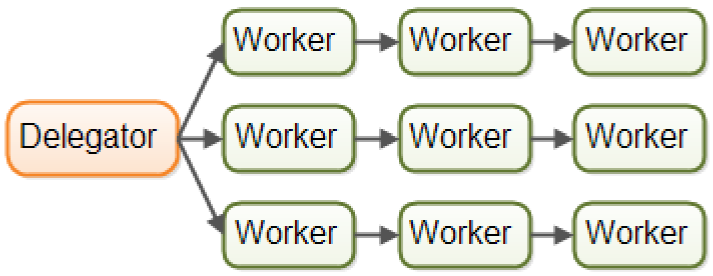
Assembly Line



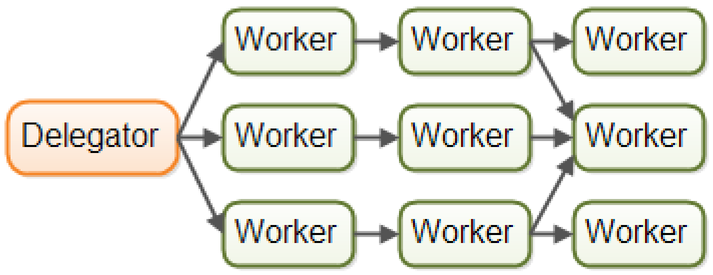
The workers are organized like workers at an assembly line in a factory. Each worker only performs a part of the full job. When that part is finished the worker forwards the job to the next worker.
Each worker is running in its own thread, and shares no state with other workers. This is also sometimes referred to as a *shared nothing* concurrency model.
Systems using the assembly line concurrency model are usually designed to use non-blocking IO. Non-blocking IO means that when a worker starts an IO operation (e.g. reading a file or data from a network connection) the worker does not wait for the IO call to finish. IO operations are slow, so waiting for IO operations to complete is a waste of CPU time. The CPU could be doing something else in the meanwhile. When the IO operation finishes, the result of the IO operation (e.g. data read or status of data written) is passed on to another worker.
With non-blocking IO, the IO operations determine the boundary between workers. A worker does as much as it can until it has to start an IO operation. Then it gives up control over the job. When the IO operation finishes, the next worker in the assembly line continues working on the job, until that too has to start an IO operation etc.



In reality, the jobs may not flow along a single assembly line. Since most systems can perform more than one job, jobs flows from worker to worker depending on the job that needs to be done. In reality there could be multiple different virtual assembly lines going on at the same time. This is how job flow through assembly line system might look in reality:



Jobs may even be forwarded to more than one worker for concurrent processing. For instance, a job may be forwarded to both a job executor and a job logger. This diagram illustrates how all three assembly lines finish off by forwarding their jobs to the same worker (the last worker in the middle assembly line):



The assembly lines can get even more complex than this.

Reactive, Event Driven Systems

Systems using an assembly line concurrency model are also sometimes called *reactive systems*, or *event driven systems*. The system's workers react to events occurring in the system, either received from the outside world or emitted by other workers. Examples of events could be an incoming HTTP request, or that a certain file finished loading into memory etc.
At the time of writing, there are a number of interesting reactive / event driven platforms available, and more will come in the future. Some of the more popular ones seems to be:

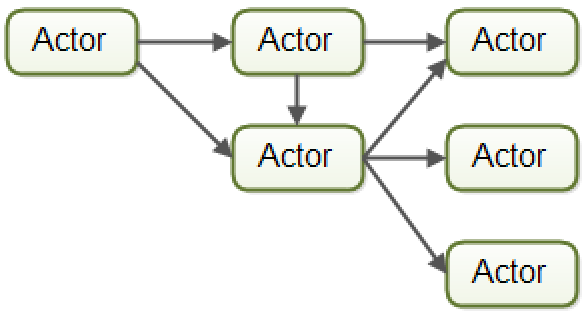
- Vert.x
- Akka
- Node.JS (JavaScript)

Personally I find Vert.x to be quite interesting (especially for a Java / JVM dinosaur like me).

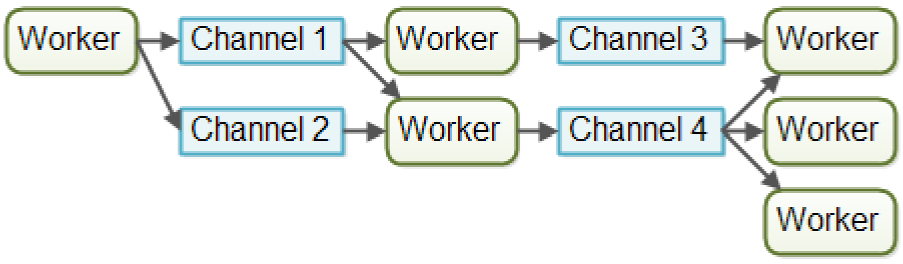
Actors vs. Channels

Actors and channels are two similar examples of assembly line (or reactive / event driven) models.

In the actor model each worker is called an *actor*. Actors can send messages directly to each other. Messages are sent and processed asynchronously. Actors can be used to implement one or more job processing assembly lines, as described earlier. Here is a diagram illustrating the actor model:



In the channel model, workers do not communicate directly with each other. Instead they publish their messages (events) on different channels. Other workers can then listen for messages on these channels without the sender knowing who is listening. Here is a diagram illustrating the channel model:



At the time of writing, the channel model seems more flexible to me. A worker does not need to know about what workers will process the job later in the assembly line. It just needs to know what channel to forward the job to (or send the message to etc.). Listeners on channels can subscribe and unsubscribe without affecting the workers writing to the channels. This allows for a somewhat looser coupling between workers.

Assembly Line Advantages

The assembly line concurrency model has several advantages compared to the parallel worker model. I will cover the biggest advantages in the following sections.

No Shared State

The fact that workers share no state with other workers means that they can be implemented without having to think about all the concurrency problems that may arise from concurrent access to shared state. This makes it much easier to implement workers. You implement a worker as if it was the only thread performing that work - essentially a singlethreaded implementation.

Stateful Workers

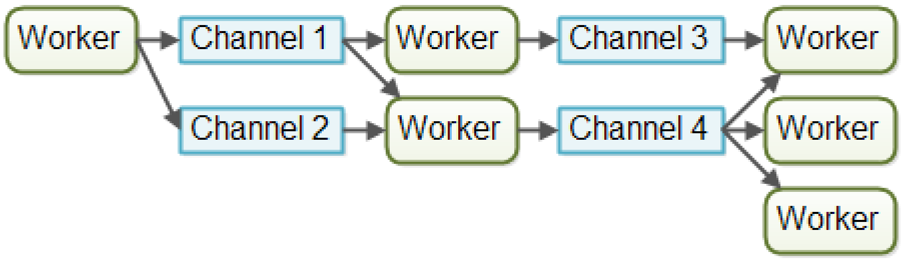
Since workers know that no other threads modify their data, the workers can be stateful. By stateful I mean that they can keep the data they need to operate in memory, only writing changes back the eventual external storage systems. A stateful worker can therefore often be faster than a stateless worker.

Better Hardware Conformity

Singlethreaded code has the advantage that it often conforms better with how the underlying hardware works. First of all, you can usually create more optimized data structures and algorithms when you can assume the code is executed in single threaded mode.
Second, singlethreaded stateful workers can cache data in memory as mentioned above. When data is cached in memory there is also a higher probability that this data is also cached in the CPU cache of the CPU executing the thread. This makes accessing cached data even faster.
I refer to it as *hardware conformity* when code is written in a way that naturally benefits from how the underlying hardware works. Some developers call this *mechanical sympathy*. I prefer the term hardware conformity because computers have very few mechanical parts, and the word "sympathy" in this context is used as a metaphor for "matching better" which I believe the word "conform" conveys reasonably well. Anyways, this is nitpicking. Use whatever term you prefer.

Job Ordering is Possible

It is possible to implement a concurrent system according to the assembly line concurrency model in a way that guarantees job ordering. Job ordering makes it much easier to reason about the state of a system at any given point in time. Furthermore, you could write all incoming jobs to a log. This log could then be used to rebuild the state of the system from scratch in case any part of the system fails. The jobs are written to the log in a certain order, and this order becomes the guaranteed job order. Here is how such a design could look:



Implementing a guaranteed job order is not necessarily easy, but it is often possible. If you can, it greatly simplifies tasks like backup, restoring data, replicating data etc. as this can all be done via the log file(s).

Assembly Line Disadvantages

The main disadvantage of the assembly line concurrency model is that the execution of a job is often spread out over multiple workers, and thus over multiple classes in your project. Thus it becomes harder to see exactly what code is being executed for a given job.
It may also be harder to write the code. Worker code is sometimes written as callback handlers. Having code with many nested callback handlers may result in what some developer call *callback hell*. Callback hell simply means that it gets hard to track what the code is really doing across all the callbacks, as well as making sure that each callback has access to the data it needs.
With the parallel worker concurrency model this tends to be easier. You can open the worker code and read the code executed pretty much from start to finish. Of course parallel worker code may also be spread over many different classes, but the execution sequence is often easier to read from the code.

Functional Parallelism

The basic idea of functional parallelism is that you implement your program using function calls. Functions can be seen as "agents" or "actors" that send messages to each other, just like in the assembly line concurrency model (AKA reactive or event driven systems). When one function calls another, that is similar to sending a message.
All parameters passed to the function are copied, so no entity outside the receiving function can manipulate the data. This copying is essential to avoiding race conditions on the shared data. This makes the function execution similar to an atomic operation. Each function call can be executed independently of any other function call.
When each function call can be executed independently, each function call can be executed on separate CPUs. That means, that an algorithm implemented functionally can be executed in parallel, on multiple CPUs.
With Java 7 we got the java.util.concurrent package contains the **ForkAndJoinPool** which can help you implement something similar to functional parallelism. With Java 8 we got parallel **streams** which can help you parallelize the iteration of large collections. Keep in mind that there are developers who are critical of the ForkAndJoinPool (you can find a link to criticism in my ForkAndJoinPool tutorial).
The hard part about functional parallelism is knowing which function calls to parallelize. Coordinating function calls across CPUs comes with an overhead. The unit of work completed by a function needs to be of a certain size to be worth this overhead. If the function calls are very small, attempting to parallelize them may actually be slower than a singlethreaded, single CPU execution.
From my understanding (which is not perfect at all) you can implement an algorithm using an reactive, event driven model and achieve a breakdown of the work which is similar to that achieved by functional parallelism. With an even driven model you just get more control of exactly what and how much to parallelize (in my opinion).

Additionally, splitting a task over multiple CPUs with the overhead the coordination of that incurs, only makes sense if that task is currently the only task being executed by the the program. However, if the system is concurrently executing multiple other tasks (like e.g. web servers, database servers and many other systems do), there is no point in trying to parallelize a single task. The other CPUs in the computer are anyways going to be busy working on other tasks, so there is not reason to try to disturb them with a slower, functionally parallel task. You are most likely better off with an assembly line (reactive) concurrency model, because it has less overhead (executes sequentially in singlethreaded mode) and conforms better with how the underlying hardware works.

Which Concurrency Model is Best?

Answer:- As is often the case, the answer is that it depends on what your system is supposed to do. If your jobs are naturally parallel, independent and with no shared state necessary, you might be able to implement your system using the parallel worker model. Many jobs are not naturally parallel and independent though. For these kinds of systems I believe the assembly line concurrency model has more advantages than disadvantages, and more advantages than the parallel worker model.

Concurrency vs. Parallelism In Detail

Concurrency is related to how an application handles multiple tasks it works on. An application may process one task at at time (sequentially) or work on multiple tasks at the same time (concurrently).

Parallelism on the other hand, is related to how an application handles each individual task. An application may process the task serially from start to end, or split the task up into subtasks which can be completed in parallel.

An application can be concurrent, but not parallel. This means that it processes more than one task at the same time, but the tasks are not broken down into subtasks.

An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel.

Additionally, an application can be neither concurrent or parallel. This means that it works on only one task at a time, and the task is never broken down into subtasks for parallel execution.

Finally, an application can also be both concurrent and parallel, in that it both works on multiple tasks at the same time, and also breaks each task down into subtasks for parallel execution. However, some of the benefits of concurrency and parallelism may be lost in this scenario, as the CPUs in the computer are already kept reasonably busy with either concurrency or parallelism alone. Combining it may lead to only a small performance gain or even performance loss. Make sure you analyze and measure before you adopt a concurrent parallel model blindly.

Subclass or Runnable?

There are no rules about which of the two methods that is the best. Both methods works. Personally though, I prefer implementing Runnable, and handing an instance of the implementation to a Threadinstance. When having the Runnable's executed by a **thread pool** it is easy to queue up the Runnableinstances until a thread from the pool is idle. This is a little harder to do with Thread subclasses.
Sometimes you may have to implement Runnable as well as subclass Thread. For instance, if creating a subclass of Thread that can execute more than one Runnable. This is typically the case when implementing a thread pool.

Race Conditions

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.
In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

The situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section.

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Race conditions can be avoided by **proper thread synchronization** in critical sections. Thread synchronization can be achieved using a synchronized block of Java code. Thread synchronization can also be achieved using other synchronization constructs like locks or atomic variables like java.util.concurrent.atomic.AtomicInteger.

Thread Safety and Shared Resources

Code that is safe to call by multiple threads simultaneously is called *thread safe*. If a piece of code is thread safe, then it contains no **race conditions**. Race condition only occur when multiple threads update shared resources. Therefore it is important to know what resources Java threads share when executing.

Local Variables

Local variables are stored in each thread's own stack. That means that local variables are never shared between threads. That also means that all local primitive variables are thread safe.

```
public void someMethod(){

  long threadSafeInt = 0;

  threadSafeInt++;
}
```

Local Object References

Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in each threads's local stack. All objects are stored in the shared heap.

If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods and objects as long as none of these methods or objects make the passed object available to other threads.

```
public void someMethod(){

  LocalObject localObject = new LocalObject();

  localObject.callMethod();
  method2(localObject);
}

public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

Object Member Variables

Object member variables (fields) are stored on the heap along with the object. Therefore, if two threads call a method on the same object instance and this method updates object member variables, the method is not thread safe.

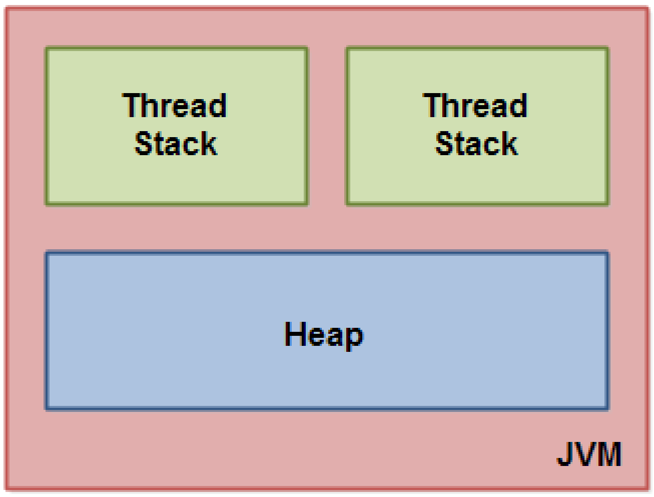
Java Memory Model

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.

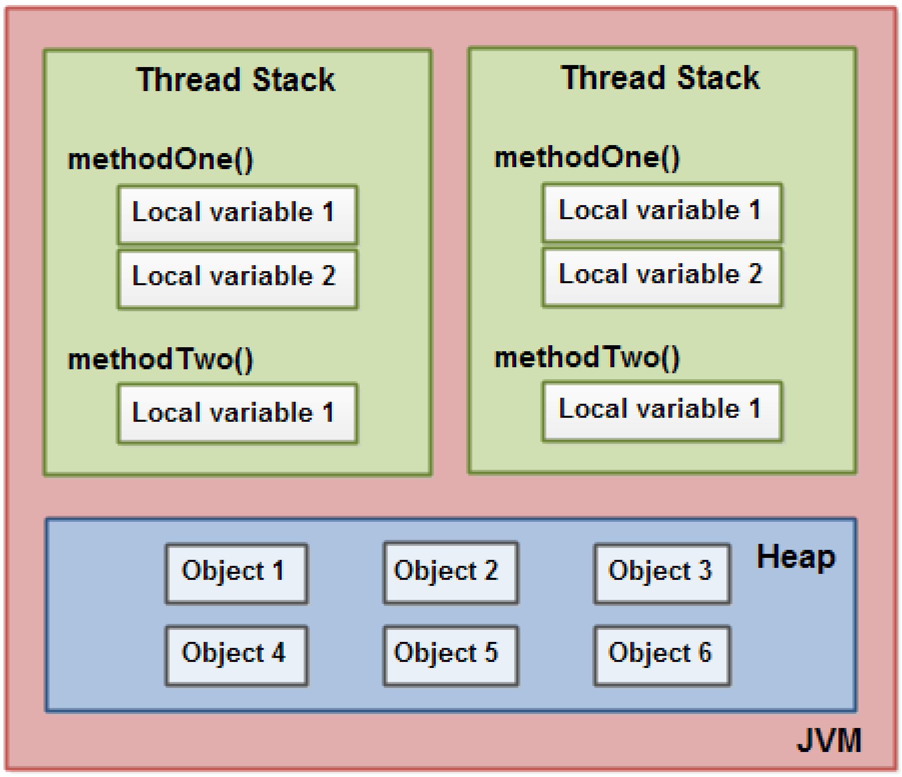
It is very important to understand the Java memory model if you want to design correctly behaving concurrent programs. The Java memory model specifies how and when different threads can see values written to shared variables by other threads, and how to synchronize access to shared variables when necessary.
The original Java memory model was insufficient, so the Java memory model was revised in Java 1.5. This version of the Java memory model is still in use in Java 8.

The Internal Java Memory Model

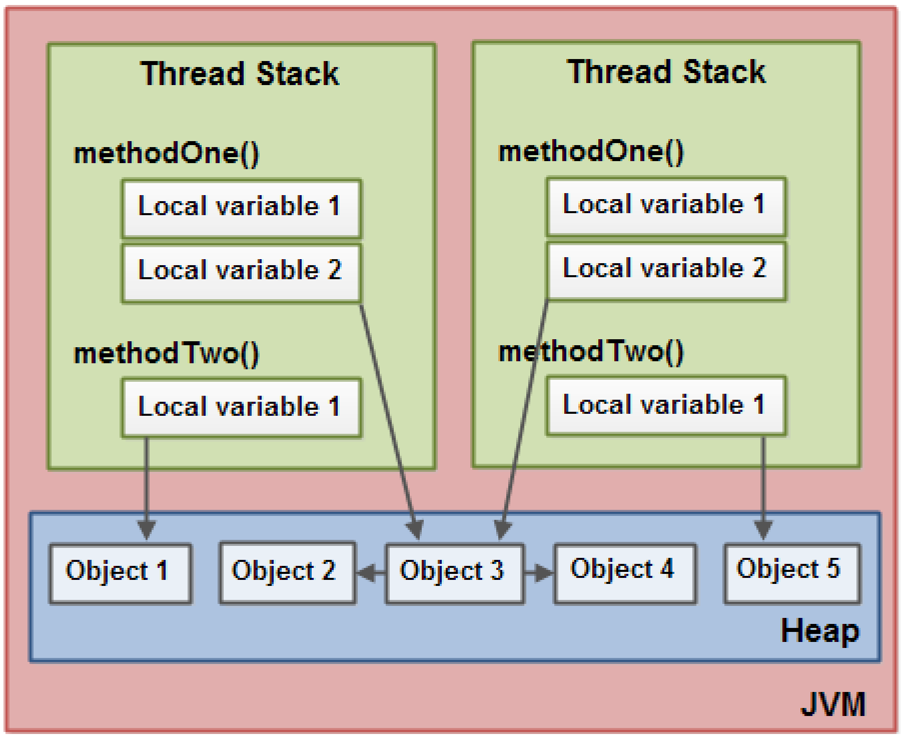
The Java memory model used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:



Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution. I will refer to this as the "call stack". As the thread executes its code, the call stack changes.
The thread stack also contains all local variables for each method being executed (all methods on the call stack). A thread can only access it's own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it. Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, each thread has its own version of each local variable.
All local variables of primitive types (boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a pritimive variable to another thread, but it cannot share the primitive local variable itself.
The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.
Here is a diagram illustrating the call stack and local variables stored on the thread stacks, and objects stored on the heap:



A local variable may be of a primitive type, in which case it is totally kept on the thread stack.
A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself if stored on the heap.
An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.
An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.
Static class variables are also stored on the heap along with the class definition.
Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.
Here is a diagram illustrating the points above:



Two threads have a set of local variables. One of the local variables (Local Variable 2) point to a shared object on the heap (Object 3). The two threads each have a different reference to the same object. Their references are local variables and are thus stored in each thread's thread stack (on each). The two different references point to the same object on the heap, though.
Notice how the shared object (Object 3) has a reference to Object 2 and Object 4 as member variables (illustrated by the arrows from Object 3 to Object 2 and Object 4). Via these member variable references in Object 3 the two threads can access Object 2 and Object 4.
The diagram also shows a local variable which point to two different objects on the heap. In this case the references point to two different objects (Object 1 and Object 5), not the same object. In theory both threads could access both Object 1 and Object 5, if both threads had references to both objects. But in the diagram above each thread only has a reference to one of the two objects.
So, what kind of Java code could lead to the above memory graph? Well, code as simple as the code below:

```
public class MyRunnable implements Runnable() {

   public void run() {
       methodOne();
   }

   public void methodOne() {
       int localVariable1 = 45;
```

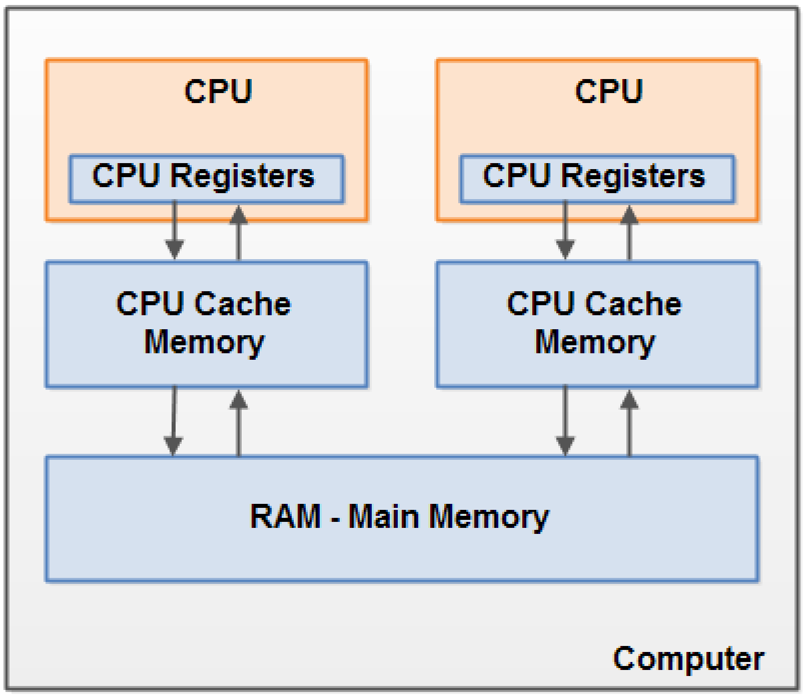
```
        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;

        //... do more with local variables.

        methodTwo();
    }

    public void methodTwo() {
        Integer localVariable1 = new Integer(99);

        //... do more with local variable.
    }
}


public class MySharedObject {

    //static variable pointing to instance of MySharedObject

    public static final MySharedObject sharedInstance =
        new MySharedObject();


    //member variables pointing to two objects on the heap

    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);

    public long member1 = 12345;
    public long member1 = 67890;
}
```

If two threads were executing the run() method then the diagram shown earlier would be the outcome. The run() method calls methodOne() and methodOne() calls methodTwo(). methodOne() declares a primitive local variable (localVariable1 of type int) and an local variable which is an object reference (localVariable2).
Each thread executing methodOne() will create its own copy of localVariable1 and localVariable2 on their respective thread stacks. The localVariable1 variables will be completely separated from each other, only living on each thread's thread stack. One thread cannot see what changes another thread makes to its copy of localVariable1.
Each thread executing methodOne() will also create their own copy of localVariable2. However, the two different copies of localVariable2 both end up pointing to the same object on the heap. The code setslocalVariable2 to point to an object referenced by a static variable. There is only one copy of a static variable and this copy is stored on the heap. Thus, both of

the two copies of localVariable2 end up pointing to the same instance of MySharedObject which the static variable points to. The MySharedObjectinstance is also stored on the heap. It corresponds to Object 3 in the diagram above.
Notice how the MySharedObject class contains two member variables too. The member variables themselves are stored on the heap along with the object. The two member variables point to two otherInteger objects. These Integer objects correspond to Object 2 and Object 4 in the diagram above.
Notice also how methodTwo() creates a local variable named localVariable1. This local variable is an object reference to an Integer object. The method sets the localVariable1 reference to point to a newInteger instance. The localVariable1 reference will be stored in one copy per thread executingmethodTwo(). The two Integer objects instantiated will be stored on the heap, but since the method creates a new Integer object every time the method is executed, two threads executing this method will create separate Integer instances. The Integer objects created inside methodTwo() correspond to Object 1 and Object 5 in the diagram above.
Notice also the two member variables in the class MySharedObject of type long which is a primitive type. Since these variables are member variables, they are still stored on the heap along with the object. Only local variables are stored on the thread stack.

Hardware Memory Architecture

Modern hardware memory architecture is somewhat different from the internal Java memory model. It is important to understand the hardware memory architecture too, to understand how the Java memory model works with it. This section describes the common hardware memory architecture, and a later section will describe how the Java memory model works with it.
Here is a simplified diagram of modern computer hardware architecture:

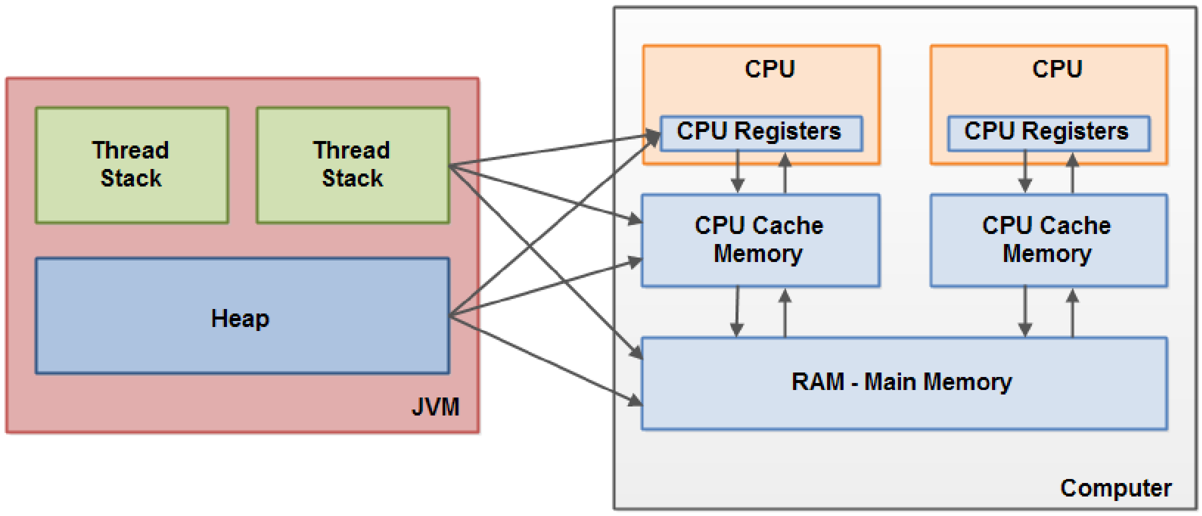


A modern computer often has 2 or more CPUs in it. Some of these CPUs may have multiple cores too. The point is, that on a modern computer with 2 or more CPUs it is possible to have more than one thread running simultaneously. Each CPU is capable of running one thread at any given time. That means that if your Java application is multithreaded, one thread per CPU may be running simultaneously (concurrently) inside your Java application.
Each CPU contains a set of registers which are essentially in-CPU memory. The CPU can perform operations much faster on these registers than it can perform on variables in main memory. That is because the CPU can access these registers much faster than it can access main memory.
Each CPU may also have a CPU cache memory layer. In fact, most modern CPUs have a cache memory layer of some size. The CPU can access its cache memory much faster than main memory, but typically not as fast as it can access its internal registers. So, the CPU cache memory is somewhere in between the speed of the internal registers and main memory. Some CPUs may have multiple cache layers (Level 1 and Level 2), but this is not so important to know to understand how the Java memory model interacts with memory. What matters is to know that CPUs can have a cache memory layer of some sort.
A computer also contains a main memory area (RAM). All CPUs can access the main memory. The main memory area is typically much bigger than the cache memories of the CPUs.
Typically, when a CPU needs to access main memory it will read part of main memory into its CPU cache. It may even read part of the cache into its internal registers and then perform operations on it. When the CPU needs to write the result back to main memory it will flush the value from its internal register to the cache memory, and at some point flush the value back to main memory.
The values stored in the cache memory is typically flushed back to main memory when the CPU needs to store something else in the cache memory. The CPU cache can have data

written to part of its memory at a time, and flush part of its memory at a time. It does not have to read / write the full cache each time it is updated. Typically the cache is updated in smaller memory blocks called "cache lines". One or more cache lines may be read into the cache memory, and one or mor cache lines may be flushed back to main memory again.

Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture

As already mentioned, the Java memory model and the hardware memory architecture are different. The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers. This is illustrated in this diagram:



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

- Visibility of thread updates (writes) to shared variables.
- Race conditions when reading, checking and writing shared variables.

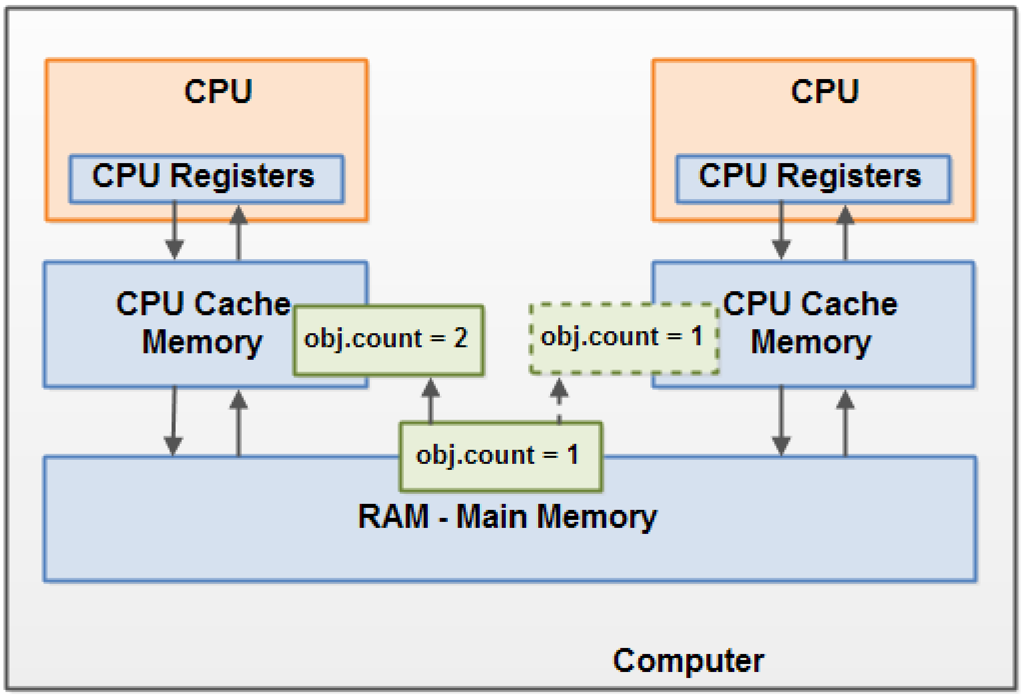
Both of these problems will be explained in the following sections.

Visibility of Shared Objects

If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to the shared object made by one thread may not be visible to other threads.

Imagine that the shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. There it makes a change to the shared object. As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.

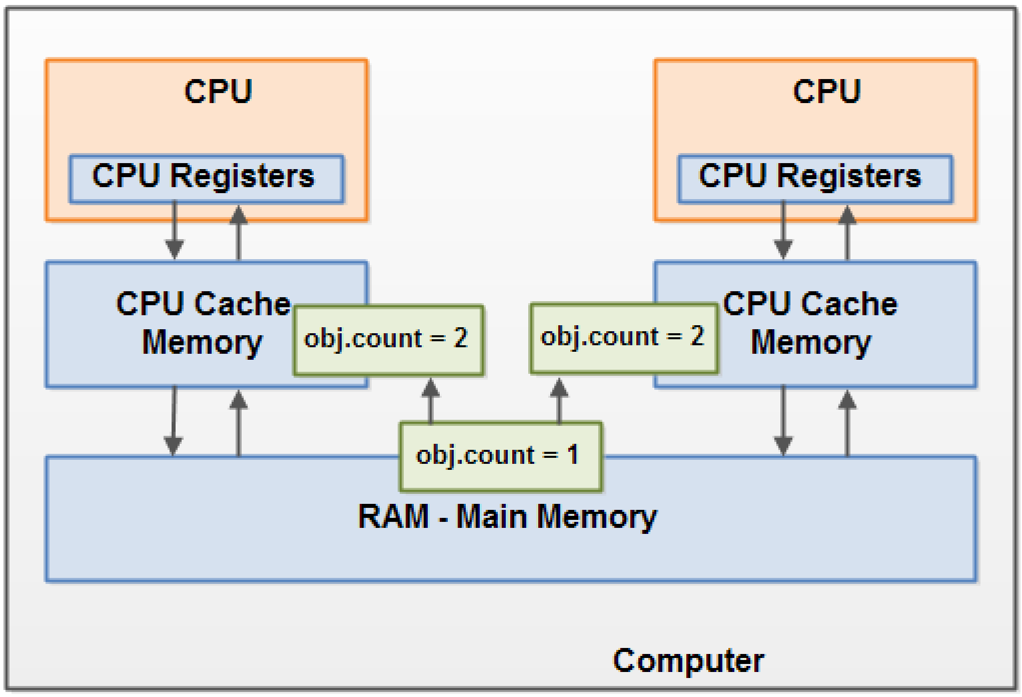
The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2. This change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.



To solve this problem you can use **Java's volatile keyword**. The volatile keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.

Race Conditions

If two or more threads share an object, and more than one thread updates variables in that shared object,**race conditions** may occur.
Imagine if thread A reads the variable count of a shared object into its CPU cache. Imagine too, that thread B does the same, but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.
If these increments had been carried out sequentially, the variable count would be been incremented twice and had the original value + 2 written back to main memory.
However, the two increments have been carried out concurrently without proper synchronization. Regardless of which of thread A and B that writes its updated version of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments.
This diagram illustrates an occurrence of the problem with race conditions as described above:



To solve this problem you can use a **Java synchronized block**. A synchronized block guarantees that only one thread can enter a given critical section of the code at any given time. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

The Java synchronized Keyword

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

The synchronized keyword can be used to mark four different types of blocks:

1. **Instance methods**
2. **Static methods**
3. **Code blocks inside instance methods**
4. **Code blocks inside static methods**

These blocks are synchronized on different objects. Which type of synchronized block you need depends on the concrete situation.

Synchronized instance method:

```
public synchronized void add(int value){
    this.count += value;
```

```
 }
```

Notice the use of the synchronized keyword in the method declaration. This tells Java that the method is synchronized.
A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

Static methods are marked as synchronized just like instance methods using the synchronized keyword. Here is a Java synchronized static method example:

```
 public static synchronized void add(int value){
     count += value;
 }
```

Also here the synchronized keyword tells Java that the method is synchronized.
Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.
If the static synchronized methods are located in different classes, then one thread can execute inside the static synchronized methods of each class. One thread per class regardless of which static synchronized method it calls.

synchronized block of Java code inside an unsynchronized Java method:

```
 public void add(int value){

    synchronized(this){
       this.count += value;
    }
 }
```

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.
Notice how the Java synchronized block construct takes an object in parentheses. In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.
Only one thread can execute inside a Java code block synchronized on the same monitor object.

Synchronized Blocks in Static Methods

Here are the same two examples as static methods. These methods are synchronized on the class object of the class the methods belong to:

```
 public class MyClass {

    public static synchronized void log1(String msg1, String msg2){
       log.writeln(msg1);
       log.writeln(msg2);
    }


    public static void log2(String msg1, String msg2){
       synchronized(MyClass.class){
          log.writeln(msg1);
          log.writeln(msg2);
       }
    }
  }
```

Only one thread can execute inside any of these two methods at the same time.
Had the second synchronized block been synchronized on a different object than MyClass.class, then one thread could execute inside each method at the same time.

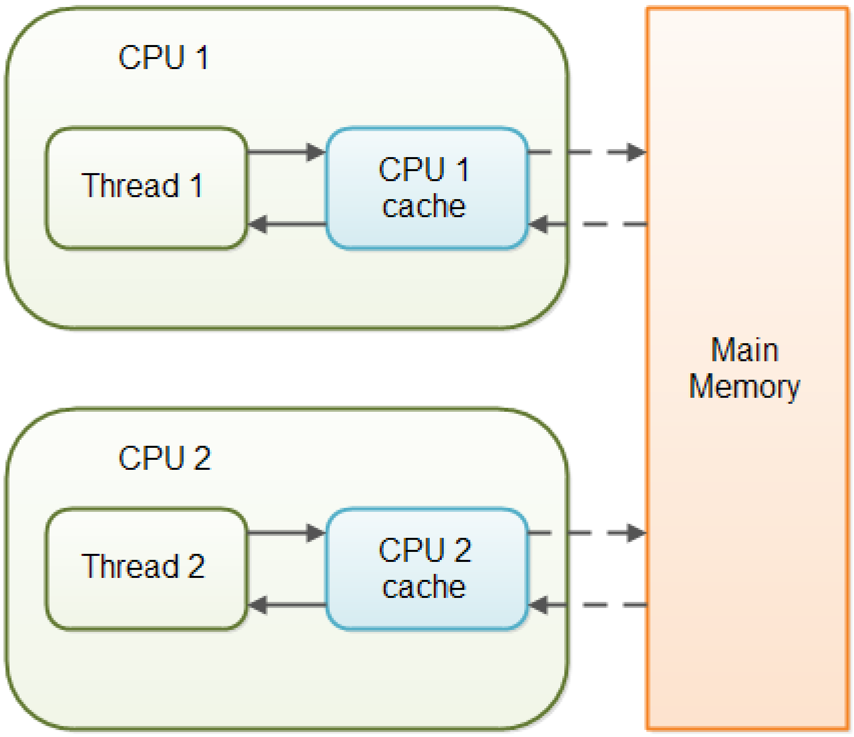
Java Volatile Keyword

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Actually, since Java 5 the volatile keyword guarantees more than just that volatile variables are written to and read from main memory. I will explain that in the following sections.

Variable Visibility Problems

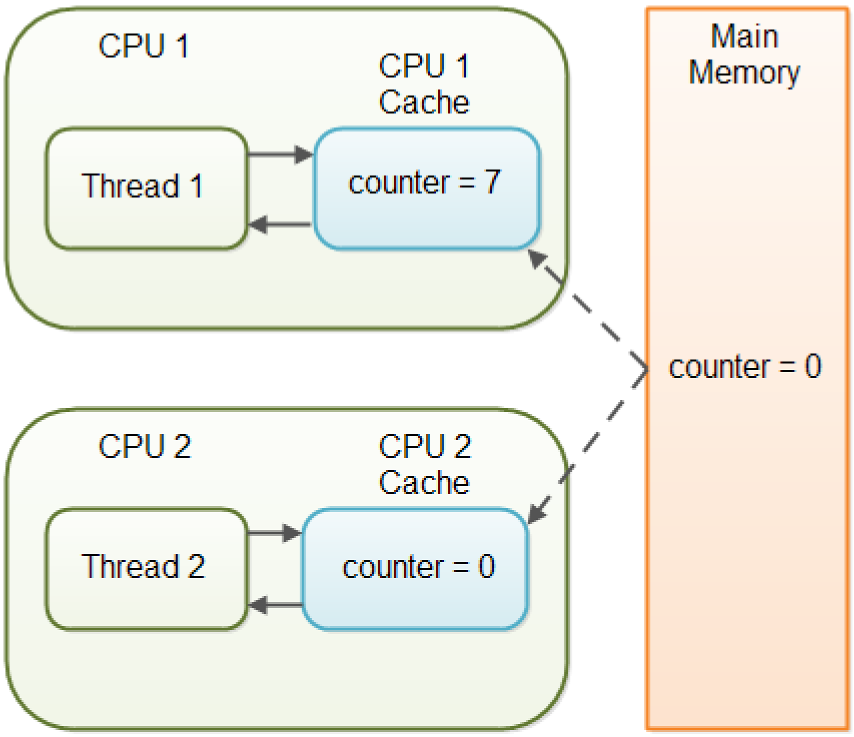
The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.
In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This is illustrated here:



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.
Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

```
public class SharedObject {

    public int counter = 0;

}
```

Imagine too, that only Thread 1 increments the counter variable, but both Thread 1 and Thread 2 may read the counter variable from time to time.
If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

The Java volatile Visibility Guarantee

The Java volatile keyword is intended to address variable visibility problems. By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory.
Here is how the volatile declaration of the counter variable looks:

```
public class SharedObject {

    public volatile int counter = 0;

}
```

Declaring a variable volatile thus *guarantees the visibility* for other threads of writes to that variable.
In the scenario given above, where one thread (T1) modifies the counter, and another thread (T2) reads the counter (but never modifies it), declaring the counter variable volatile is enough to guarantee visibility for T2 of writes to the counter variable.
If, however, both T1 and T2 were incrementing the counter variable, then declaring the counter variable volatile would not have been enough. More on that later.

Full volatile Visibility Guarantee

Actually, the visibility guarantee of Java volatile goes beyond the volatile variable itself. The visibility guarantee is as follows:

- If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable.
- If Thread A reads a volatile variable, then all all variables visible to Thread A when reading the volatile variable will also be re-read from main memory.

Let me illustrate that with a code example:

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;


    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

The udpate() method writes three variables, of which only days is volatile.
The full volatile visibility guarantee means, that when a value is written to days, then all variables visible to the thread are also written to main memory. That means, that when a value is written to days, the values of years and months are also written to main memory.
When reading the values of years, months and days you could do it like this:

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;

    public int totalDays() {
        int total = this.days;
        total += months * 30;
        total += years * 365;
        return total;
    }

    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

Notice the totalDays() method starts by reading the value of days into the total variable. When reading the value of days, the values of months and years are also read into main memory. Therefore you are guaranteed to see the latest values of days, months and years with the above read sequence.

Instruction Reordering Challenges

The Java VM and the CPU are allowed to reorder instructions in the program for performance reasons, as long as the semantic meaning of the instructions remain the same. For instance, look at the following instructions:

```
int a = 1;
int b = 2;

a++;
b++;
```

These instructions could be reordered to the following sequence without losing the semantic meaning of the program:

```
'
int a = 1;
a++;

int b = 2;
b++;
```

However, instruction reordering present a challenge when one of the variables is a volatile variable. Let us look at the MyClass class from the example earlier in this Java volatile tutorial:

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;


    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

Once the update() method writes a value to days, the newly written values to years and months are also written to main memory. But, what if the Java VM reordered the instructions, like this:

```
public void update(int years, int months, int days){
    this.days   = days;
    this.months = months;
    this.years  = years;
}
```

The values of months and years are still written to main memory when the days variable is modified, but this time it happens before the new values have been written to months and years. The new values are thus not properly made visible to other threads. The semantic meaning of the reordered instructions has changed.
Java has a solution for this problem, as we will see in the next section.

The Java volatile Happens-Before Guarantee

To address the instruction reordering challenge, the Java volatile keyword gives a "happens-before" guarantee, in addition to the visibility guarantee. The happens-before guarantee guarantees that:

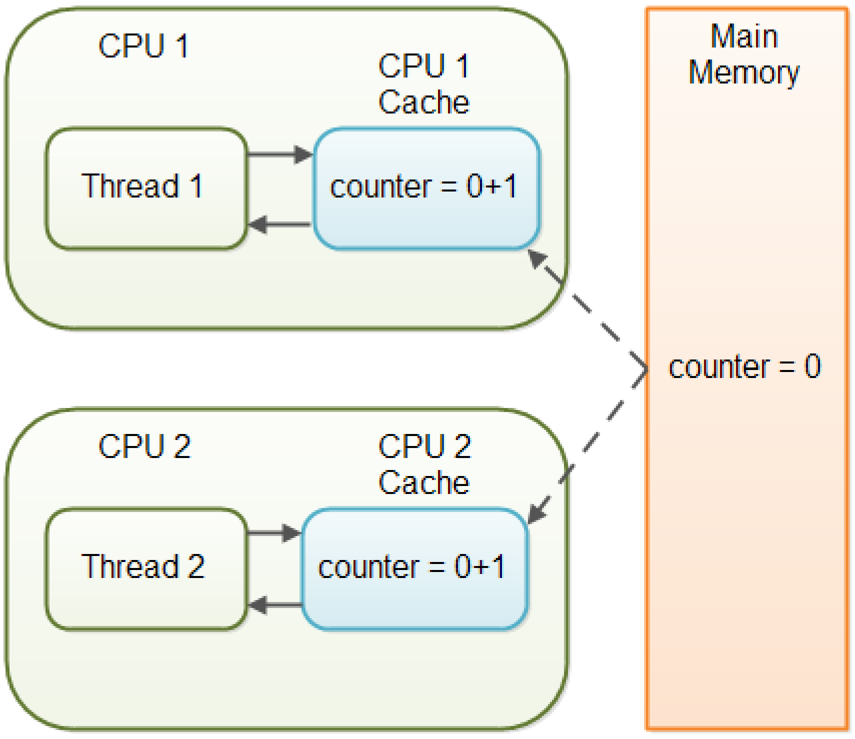
- Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable.
- The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable. Notice that it is still possible for e.g. reads / writes of other variables located after a write to a volatile to be reordered to occur before that write to the volatile. Just not the other way around. From after to before is allowed, but from before to after is not allowed.
- Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable. Notice that it is possible for reads of other variables that occur before the read of a volatile variable can be reordered to occur after the read of the volatile. Just not the other way around. From before to after is allowed, but from after to before is not allowed.

The above happens-before guarantee assures that the visibility guarantee of the volatile keyword are being enforced.

volatile is Not Always Enough

Even if the volatile keyword guarantees that all reads of a volatile variable are read directly from main memory, and all writes to a volatile variable are written directly to main memory, there are still situations where it is not enough to declare a variable volatile.
In the situation explained earlier where only Thread 1 writes to the shared counter variable, declaring the counter variable volatile is enough to make sure that Thread 2 always sees the latest written value.
In fact, multiple threads could even be writing to a shared volatile variable, and still have the correct value stored in main memory, if the new value written to the variable does not depend on its previous value. In other words, if a thread writing a value to the shared volatile variable does not first need to read its value to figure out its next value.

As soon as a thread needs to first read the value of a volatile variable, and based on that value generate a new value for the shared volatile variable, a volatile variable is no longer enough to guarantee correct visibility. The short time gap in between the reading of the volatile variable and the writing of its new value, creates an race condition where multiple threads might read the same value of the volatile variable, generate a new value for the variable, and when writing the value back to main memory - overwrite each other's values. The situation where multiple threads are incrementing the same counter is exactly such a situation where a volatile variable is not enough. The following sections explain this case in more detail.
Imagine if Thread 1 reads a shared counter variable with the value 0 into its CPU cache, increment it to 1 and not write the changed value back into main memory. Thread 2 could then read the same counter variable from main memory where the value of the variable is still 0, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. This situation is illustrated in the diagram below:



Thread 1 and Thread 2 are now practically out of sync. The real value of the shared counter variable should have been 2, but each of the threads has the value 1 for the variable in their CPU caches, and in main memory the value is still 0. It is a mess! Even if the threads eventually write their value for the shared counter variable back to main memory, the value will be wrong.

When is volatile Enough?

As I have mentioned earlier, if two threads are both reading and writing to a shared variable, then using the volatile keyword for that is not enough. You need to use a synchronized in that case to guarantee that the reading and writing of the variable is atomic. Reading or

writing a volatile variable does not block threads reading or writing. For this to happen you must use the synchronized keyword around critical sections.
As an alternative to a synchronized block you could also use one of the many atomic data types found in the java.util.concurrent package. For instance, the AtomicLong or AtomicReference or one of the others.
In case only one thread reads and writes the value of a volatile variable and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.
The volatile keyword is guaranteed to work on 32 bit and 64 variables.

Performance Considerations of volatile

Reading and writing of volatile variables causes the variable to be read or written to main memory. Reading from and writing to main memory is more expensive than accessing the CPU cache. Accessing volatile variables also prevent instruction reordering which is a normal performance enhancement technique. Thus, you should only use volatile variables when you really need to enforce visibility of variables.

Java ThreadLocal

The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to a ThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables.

Creating a ThreadLocal

Here is a code example that shows how to create a ThreadLocal variable:

```
private ThreadLocal myThreadLocal = new ThreadLocal();
```

As you can see, you instantiate a new ThreadLocal object. This only needs to be done once per thread. Even if different threads execute the same code which accesses a ThreadLococal, each thread will see only its own ThreadLocal instance. Even if two different threads set different values on the same ThreadLocal object, they cannot see each other's values.

Accessing a ThreadLocal

Once a ThreadLocal has been created you can set the value to be stored in it like this:

```
myThreadLocal.set("A thread local value");
```

You read the value stored in a ThreadLocal like this:

```
String threadLocalValue = (String) myThreadLocal.get();
```

The get() method returns an Object and the set() method takes an Object as parameter.

Generic ThreadLocal

You can create a generic ThreadLocal so that you do not have to typecast the value returned by get(). Here is a generic ThreadLocal example:

```
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
```

Now you can only store strings in the ThreadLocal instance. Additionally, you do not need to typecast the value obtained from the ThreadLocal:

```
myThreadLocal.set("Hello ThreadLocal");

String threadLocalValue = myThreadLocal.get();
```

Initial ThreadLocal Value

Since values set on a ThreadLocal object only are visible to the thread who set the value, no thread can set an initial value on a ThreadLocal using set() which is visible to all threads. Instead you can specify an initial value for a ThreadLocal object by subclassing ThreadLocal and overriding the initialValue() method. Here is how that looks:

```
private ThreadLocal myThreadLocal = new ThreadLocal<String>() {
  @Override protected String initialValue() {
    return "This is the initial value";
  }
};
```

Now all threads will see the same initial value when calling get() before having called set() .

Full ThreadLocal Example

Here is a fully runnable Java ThreadLocal example:

```
public class ThreadLocalExample {


  public static class MyRunnable implements Runnable {

    private ThreadLocal<Integer> threadLocal =
        new ThreadLocal<Integer>();

    @Override
    public void run() {
      threadLocal.set( (int) (Math.random() * 100D) );
```

```
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
            }

            System.out.println(threadLocal.get());
        }
    }


    public static void main(String[] args) {
        MyRunnable sharedRunnableInstance = new MyRunnable();

        Thread thread1 = new Thread(sharedRunnableInstance);
        Thread thread2 = new Thread(sharedRunnableInstance);

        thread1.start();
        thread2.start();

        thread1.join(); //wait for thread 1 to terminate
        thread2.join(); //wait for thread 2 to terminate
    }

}
```

This example creates a single MyRunnable instance which is passed to two different threads. Both threads execute the run() method, and thus sets different values on the ThreadLocal instance. If the access to the set() call had been synchronized, and it had *not* been a ThreadLocal object, the second thread would have overridden the value set by the first thread.
However, since it *is* a ThreadLocal object then the two threads cannot see each other's values. Thus, they set and get different values.

InheritableThreadLocal

The InheritableThreadLocal class is a subclass of ThreadLocal. Instead of each thread having its own value inside a ThreadLocal, the InheritableThreadLocal grants access to values to a thread and all child threads created by that thread.

JDK release-wise multi-threading concepts

As per **JDK 1.x release**, there were only few classes present in this initial release. To be very specific, there classes/interfaces were:

- **java.lang.Thread**
- **java.lang.ThreadGroup**
- **java.lang.Runnable**
- **java.lang.Process**
- **java.lang.ThreadDeath**

and some exception classes e.g.

- **java.lang.IllegalMonitorStateException**
- **java.lang.IllegalStateException**
- **java.lang.IllegalThreadStateException.**

It also had few synchronized collections e.g. **java.util.Hashtable**.

JDK 1.2 and **JDK 1.3** had no noticeable changes related to Multithreading. **JDK 1.4**, there were few JVM level changes to suspend/resume multiple threads with single call. But no major API changes were present.

JDK 1.5 was first big release after JDK 1.x; The biggest change in java multi-threading applications cloud happened in this release and it had included multiple concurrency utilities.

- **Executor,**
- **Semaphore,**
- **Mutex,**
- **Barrier,**
- **Latches,**
- **Concurrent collections** and
- **Blocking Queues;**

JDK 1.6 was more of platform fixes than API upgrades. So new change was present in JDK 1.6.

JDK 1.7 added support for **ForkJoinPool** which implemented work-stealing technique to maximize the throughput. Also **Phaser class** was added.

JDK 1.8 is largely known for Lambda changes, but it also had few concurrency changes as well. Two new interfaces and four new classes were added in **java.util.concurrent package e.g. CompletableFuture and CompletionException.**The Collections Framework has undergone a major revision in Java 8 to add aggregate operations based on the newly added streams facility and lambda expressions; resulting in large number of methods added in almost all Collection classes, and thus in concurrent collections as well.

What is Thread safety?

Answer:- 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.

A thread-safe class as one that is no more broken in a concurrent environment than in a single-threaded environment. Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.**A good example of thread safe class is java servlets** which have no fields and references, no fields from other classes etc. They are stateless.

```
public class StatelessFactorizer implements Servlet
{
    public void service(ServletRequest req, ServletResponse resp)
    {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

The transient state for a particular computation exists solely in local variables that are stored on the thread's stack and are accessible only to the executing thread. One thread accessing a StatelessFactorizer cannot influence the result of another thread accessing the same StatelessFactorizer; because the two threads do not share state, it is as if they were accessing different instances. Since the actions of a thread accessing a stateless object cannot affect the correctness of operations in other threads, stateless objects are thread-safe.

Object level Locking vs. Class level Locking in Java

Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in synch. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as synchronized; the thread holds the monitor for that method's object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.Synchronization in java is achieved using synchronized keyword. **You can use synchronized keyword in your class on defined methods or blocks. Keyword can not be used with variables or attributes in class definition.**

Object level locking

Object level locking is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe.

Class level locking

Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

Thread Deadlock

A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding. Deadlock can occur when multiple threads need the same locks, at the same time, but obtain them in different order.

Deadlock Prevention

In some situations it is possible to prevent deadlocks. I'll describe three techniques in this text:

1. **Lock Ordering**
2. **Lock Timeout**
3. **Deadlock Detection**

Lock Ordering

Deadlock occurs when multiple threads need the same locks but obtain them in different order. If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur.

Lock Timeout

Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry. The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking.

Deadlock Detection

Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.
Every time a thread **takes** a lock it is noted in a data structure (map, graph etc.) of threads and locks. Additionally, whenever a thread **requests** a lock this is also noted in this data structure.
When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks. For instance, if a Thread A requests lock 7, but lock 7 is held by Thread B, then Thread A can check if Thread B has requested any of the locks Thread A holds (if any). If Thread B has requested so, a deadlock has occurred (Thread A having taken lock 1, requesting lock 7, Thread B having taken lock 7, requesting lock 1).

what do the threads do if a deadlock is detected?
Answer:-One possible action is to release all locks, backup, wait a random amount of time and then retry. This is similar to the simpler lock timeout mechanism except threads only backup when a deadlock has actually occurred. Not just because their lock requests timed out. However, if a lot of threads are competing for the same locks they may repeatedly end up in a deadlock even if they back up and wait.
A better option is to determine or assign a priority of the threads so that only one (or a few) thread backs up. The rest of the threads continue taking the locks they need as if no deadlock had occurred. If the priority assigned to the threads is fixed, the same threads will always be given higher priority. To avoid this you may assign the priority randomly whenever a deadlock is detected.

Starvation and Fairness

If a thread is not granted CPU time because other threads grab it all, it is called "starvation". The thread is "starved to death" because other threads are allowed the CPU time instead of it. The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute.

Causes of Starvation in Java

The following three common causes can lead to starvation of threads in Java:

1. Threads with high priority swallow all CPU time from threads with lower priority.
2.
3.
4. Threads are blocked indefinately waiting to enter a synchronized block, because other threads are constantly allowed access before it.
5.
6.
7. Threads waiting on an object (called wait() on it) remain waiting indefinitely because other threads are constantly awakened instead of it.

Threads with high priority swallow all CPU time from threads with lower priority

You can set the thread priority of each thread individually. The higher the priority the more CPU time the thread is granted. You can set the priority of threads between 1 and 10. Exactly how this is interpreted depends on the operating system your application is running on. For most applications you are better off leaving the priority unchanged.

Threads are blocked indefinitely waiting to enter a synchronized block

Java's synchronized code blocks can be another cause of starvation. Java's synchronized code block makes no guarantee about the sequence in which threads waiting to enter the synchronized block are allowed to enter. This means that there is a theoretical risk that a thread remains blocked forever trying to enter the block, because other threads are constantly granted access before it. This problem is called "starvation", that a thread is "starved to death" by because other threads are allowed the CPU time instead of it.

Threads waiting on an object (called wait() on it) remain waiting indefinitely

The notify() method makes no guarantee about what thread is awakened if multiple thread have called wait() on the object notify() is called on. It could be any of the threads waiting. Therefore there is a risk that a thread waiting on a certain object is never awakened because other waiting threads are always awakened instead of it.

Implementing Fairness in Java

While it is not possible to implement 100% fairness in Java we can still implement our synchronization constructs to increase fairness between threads.
First lets study a simple synchronized code block:

```
public class Synchronizer{

  public synchronized void doSynchronized(){
    //do a lot of work which takes a long time
  }

}
```

If more than one thread call the doSynchronized() method, some of them will be blocked until the first thread granted access has left the method. If more than one thread are blocked waiting for access there is no guarantee about which thread is granted access next.

Using Locks Instead of Synchronized Blocks

To increase the fairness of waiting threads first we will change the code block to be guarded by a lock rather than a synchronized block:

```
public class Synchronizer{
  Lock lock = new Lock();

  public void doSynchronized() throws InterruptedException{
    this.lock.lock();
      //critical section, do a lot of work which takes a long time
    this.lock.unlock();
  }

}
```

Notice how the doSynchronized() method is no longer declared synchronized. Instead the critical section is guarded by the lock.lock() and lock.unlock() calls.

A simple implementation of the Lock class could look like this:

```
public class Lock{
  private boolean isLocked      = false;
  private Thread  lockingThread = null;

  public synchronized void lock() throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked      = true;
    lockingThread = Thread.currentThread();
  }

  public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
      throw new IllegalMonitorStateException(
        "Calling thread has not locked this lock");
    }
    isLocked      = false;
    lockingThread = null;
    notify();
  }
}
```

If you look at the Synchronizer class above and look into this Lock implementation you will notice that threads are now blocked trying to access the lock() method, if more than one thread calls lock() simultanously. Second, if the lock is locked, the threads are blocked in the wait() call inside the while(isLocked) loop in the lock() method. Remember that a thread calling wait() releases the synchronization lock on the Lock instance, so threads waiting to enter lock() can now do so. The result is that multiple threads can end up having called wait() inside lock().

If you look back at the doSynchronized() method you will notice that the comment between lock() and unlock() states, that the code in between these two calls take a "long" time to execute. Let us further assume that this code takes long time to execute compared to entering the lock() method and calling wait() because the lock is locked. This means that the majority of the time waited to be able to lock the lock and enter the critical section is spent waiting in the wait() call inside the lock() method, not being blocked trying to enter the lock() method.

As stated earlier synchronized blocks makes no guarantees about what thread is being granted access if more than one thread is waiting to enter. Nor does wait() make any guarantees about what thread is awakened when notify() is called. So, the current version of the Lock class makes no different guarantees with respect to fairness than synchronized version of doSynchronized(). But we can change that.

The current version of the Lock class calls its own wait() method. If instead each thread calls wait() on a separate object, so that only one thread has called wait() on each object, the

Lock class can decide which of these objects to call notify() on, thereby effectively selecting exactly what thread to awaken.

A Fair Lock

Below is shown the previous Lock class turned into a fair lock called FairLock. You will notice that the implementation has changed a bit with respect to synchronization and wait() / notify() compared to the Lock class shown earlier.

Exactly how I arrived at this design beginning from the previous Lock class is a longer story involving several incremental design steps, each fixing the problem of the previous step: Nested Monitor Lockout, Slipped Conditions, and Missed Signals. That discussion is left out of this text to keep the text short, but each of the steps are discussed in the appropriate texts on the topic (see the links above). What is important is, that every thread calling lock() is now queued, and only the first thread in the queue is allowed to lock the FairLock instance, if it is unlocked. All other threads are parked waiting until they reach the top of the queue.

```
public class FairLock {
    private boolean           isLocked       = false;
    private Thread            lockingThread  = null;
    private List<QueueObject> waitingThreads =
            new ArrayList<QueueObject>();

  public void lock() throws InterruptedException{
    QueueObject queueObject           = new QueueObject();
    boolean     isLockedForThisThread = true;
    synchronized(this){
        waitingThreads.add(queueObject);
    }

    while(isLockedForThisThread){
      synchronized(this){
        isLockedForThisThread =
            isLocked || waitingThreads.get(0) != queueObject;
        if(!isLockedForThisThread){
          isLocked = true;
           waitingThreads.remove(queueObject);
           lockingThread = Thread.currentThread();
           return;
         }
      }
      try{
        queueObject.doWait();
      }catch(InterruptedException e){
        synchronized(this) { waitingThreads.remove(queueObject); }
        throw e;
      }
    }
  }
  public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
      throw new IllegalMonitorStateException(
        "Calling thread has not locked this lock");
    }
    isLocked      = false;
    lockingThread = null;
    if(waitingThreads.size() > 0){
      waitingThreads.get(0).doNotify();
    }
  }
}

public class QueueObject {

  private boolean isNotified = false;

  public synchronized void doWait() throws InterruptedException {
    while(!isNotified){
        this.wait();
    }
    this.isNotified = false;
  }

  public synchronized void doNotify() {
    this.isNotified = true;
    this.notify();
  }

  public boolean equals(Object o) {
    return this == o;
  }
}
```

First you might notice that the lock() method is no longer declared synchronized. Instead only the blocks necessary to synchronize are nested inside synchronized blocks.
FairLock creates a new instance of QueueObject and enqueue it for each thread calling lock(). The thread calling unlock() will take the top QueueObject in the queue and call doNotify() on it, to awaken the thread waiting on that object. This way only one waiting thread is awakened at a time, rather than all waiting threads. This part is what governs the fairness of the FairLock.
Notice how the state of the lock is still tested and set within the same synchronized block to avoid slipped conditions.
Also notice that the QueueObject is really a semaphore. The doWait() and doNotify() methods store the signal internally in the QueueObject. This is done to avoid missed signals

caused by a thread being preempted just before calling queueObject.doWait(), by another thread which calls unlock() and thereby queueObject.doNotify(). The queueObject.doWait() call is placed outside the synchronized(this) block to avoid nested monitor lockout, so another thread can actually call unlock() when no thread is executing inside the synchronized(this) block in lock() method.
Finally, notice how the queueObject.doWait() is called inside a try - catch block. In case an InterruptedException is thrown the thread leaves the lock() method, and we need to dequeue it.

A Note on Performance

If you compare the Lock and FairLock classes you will notice that there is somewhat more going on inside the lock() and unlock() in the FairLock class. This extra code will cause the FairLock to be a sligtly slower synchronization mechanism than Lock. How much impact this will have on your application depends on how long time the code in the critical section guarded by the FairLock takes to execute. The longer this takes to execute, the less significant the added overhead of the synchronizer is. It does of course also depend on how often this code is called.

Nested Monitor Lockout

How Nested Monitor Lockout Occurs

Nested monitor lockout is a problem similar to deadlock. A nested monitor lockout occurs like this:

```
Thread 1 synchronizes on A
Thread 1 synchronizes on B (while synchronized on A)
Thread 1 decides to wait for a signal from another thread before continuing
Thread 1 calls B.wait() thereby releasing the lock on B, but not A.

Thread 2 needs to lock both A and B (in that sequence)
      to send Thread 1 the signal.
Thread 2 cannot lock A, since Thread 1 still holds the lock on A.
Thread 2 remain blocked indefinately waiting for Thread1
      to release the lock on A

Thread 1 remain blocked indefinately waiting for the signal from
      Thread 2, thereby
      never releasing the lock on A, that must be released to make
      it possible for Thread 2 to send the signal to Thread 1, etc.
```

This may sound like a pretty theoretical situation, but look at the naive Lock implemenation below:

```
//lock implementation with nested monitor lockout problem

public class Lock{
  protected MonitorObject monitorObject = new MonitorObject();
  protected boolean isLocked = false;

  public void lock() throws InterruptedException{
    synchronized(this){
      while(isLocked){
        synchronized(this.monitorObject){
            this.monitorObject.wait();
        }
      }
      isLocked = true;
    }
  }

  public void unlock(){
    synchronized(this){
      this.isLocked = false;
      synchronized(this.monitorObject){
        this.monitorObject.notify();
      }
    }
  }
}
```

Notice how the lock() method first synchronizes on "this", then synchronizes on the monitorObject member. If isLocked is false there is no problem. The thread does not call monitorObject.wait(). If isLocked is true however, the thread calling lock() is parked waiting in the monitorObject.wait() call.
The problem with this is, that the call to monitorObject.wait() only releases the synchronization monitor on the monitorObject member, and not the synchronization monitor associated with "this". In other words, the thread that was just parked waiting is still holding the synchronization lock on "this".
When the thread that locked the Lock in the first place tries to unlock it by calling unlock() it will be blocked trying to enter the synchronized(this) block in the unlock() method. It will remain blocked until the thread waiting in lock() leaves the synchronized(this) block. But the thread waiting in the lock() method will not leave that block until the isLocked is set to false, and a monitorObject.notify() is executed, as it happens in unlock().
Put shortly, the thread waiting in lock() needs an unlock() call to execute successfully for it to exit lock() and the synchronized blocks inside it. But, no thread can actually execute unlock() until the thread waiting in lock() leaves the outer synchronized block.
This result is that any thread calling either lock() or unlock() will become blocked indefinately. This is called a nested monitor lockout.

A More Realistic Example

You may claim that you would never implement a lock like the one shown earlier. That you would not call wait() and notify() on an internal monitor object, but rather on the This is probably true. But there are situations in which designs like the one above may arise. For instance, if you were to implement fairness in a Lock. When doing so you want each thread to call wait() on each their own queue object, so that you can notify the threads one at a time.

Look at this naive implementation of a fair lock:

//Fair Lock implementation with nested monitor lockout problem

```
public class FairLock {
  private boolean           isLocked       = false;
  private Thread            lockingThread  = null;
  private List<QueueObject> waitingThreads =
          new ArrayList<QueueObject>();

  public void lock() throws InterruptedException{
    QueueObject queueObject = new QueueObject();

    synchronized(this){
      waitingThreads.add(queueObject);

      while(isLocked || waitingThreads.get(0) != queueObject){

        synchronized(queueObject){
          try{
            queueObject.wait();
          }catch(InterruptedException e){
            waitingThreads.remove(queueObject);
            throw e;
          }
        }
      }
      waitingThreads.remove(queueObject);
      isLocked = true;
      lockingThread = Thread.currentThread();
    }
  }

  public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
      throw new IllegalMonitorStateException(
        "Calling thread has not locked this lock");
    }
    isLocked      = false;
    lockingThread = null;
    if(waitingThreads.size() > 0){
      QueueObject queueObject = waitingThreads.get(0);
      synchronized(queueObject){
        queueObject.notify();
      }
    }
  }
}

public class QueueObject {}
```

At first glance this implementation may look fine, but notice how the lock() method calls queueObject.wait(); from inside two synchronized blocks. One synchronized on "this", and nested inside that, a block synchronized on the queueObject local variable. When a thread calls queueObject.wait()it releases the lock on the QueueObject instance, but not the lock associated with "this".
Notice too, that the unlock() method is declared synchronized which equals a synchronized(this) block. This means, that if a thread is waiting inside lock() the monitor object associated with "this" will be locked by the waiting thread. All threads calling unlock() will remain blocked indefinately, waiting for the waiting thread to release the lock on "this". But this will never happen, since this only happens if a thread succeeds in sending a signal to the waiting thread, and this can only be sent by executing the unlock() method.
And so, the FairLock implementation from above could lead to nested monitor lockout. A better implementation of a fair lock is described in the text Starvation and Fairness.

Nested Monitor Lockout vs. Deadlock

The result of nested monitor lockout and deadlock are pretty much the same: The threads involved end up blocked forever waiting for each other.
The two situations are not equal though. As explained in the text on Deadlock a deadlock occurs when two threads obtain locks in different order. Thread 1 locks A, waits for B. Thread 2 has locked B, and now waits for A. As explained in the text on Deadlock Prevention deadlocks can be avoided by always locking the locks in the same order (Lock Ordering). However, a nested monitor lockout occurs exactly by two threads taking the locks **in the same order**. Thread 1 locks A and B, then releases B and waits for a signal from Thread 2. Thread 2 needs both A and B to send Thread 1 the signal. So, one thread is waiting for a signal, and another for a lock to be released.
The difference is summed up here:
In deadlock, two threads are waiting for each other to release locks.

In nested monitor lockout, Thread 1 is holding a lock A, and waits for a signal from Thread 2. Thread 2 needs the lock A to send the signal to Thread 1.

Slipped Conditions

What is Slipped Conditions?

Slipped conditions means, that from the time a thread has checked a certain condition until it acts upon it, the condition has been changed by another thread so that it is errornous for the first thread to act. Here is a simple example:

```
public class Lock {

  private boolean isLocked = true;

  public void lock(){
    synchronized(this){
      while(isLocked){
        try{
          this.wait();
        } catch(InterruptedException e){
          //do nothing, keep waiting
        }
      }
    }

    synchronized(this){
      isLocked = true;
    }
  }

  public synchronized void unlock(){
    isLocked = false;
    this.notify();
  }

}
```

Notice how the lock() method contains two synchronized blocks. The first block waits until isLocked is false. The second block sets isLocked to true, to lock the Lock instance for other threads.

Imagine that isLocked is false, and two threads call lock() at the same time. If the first thread entering the first synchronized block is preempted right after the first synchronized block, this thread will have checked isLocked and noted it to be false. If the second thread is now allowed to execute, and thus enter the first synchronized block, this thread too will see isLocked as false. Now both threads have read the condition as false. Then both threads will enter the second synchronized block, set isLocked to true, and continue.

This situation is an example of slipped conditions. Both threads test the condition, then exit the synchronized block, thereby allowing other threads to test the condition, before any of the two first threads change the conditions for subsequent threads. In other words, the condition has slipped from the time the condition was checked until the threads change it for subsequent threads.
To avoid slipped conditions the testing and setting of the conditions must be done atomically by the thread doing it, meaning that no other thread can check the condition in between the testing and setting of the condition by the first thread.
The solution in the example above is simple. Just move the line isLocked = true; up into the first synchronized block, right after the while loop. Here is how it looks:

```
public class Lock {

    private boolean isLocked = true;

    public void lock(){
      synchronized(this){
        while(isLocked){
          try{
            this.wait();
          } catch(InterruptedException e){
            //do nothing, keep waiting
          }
        }
        isLocked = true;
      }
    }

    public synchronized void unlock(){
      isLocked = false;
      this.notify();
    }

}
```

Now the testing and setting of the isLocked condition is done atomically from inside the same synchronized block.

A More Realistic Example

You may rightfully argue that you would never implement a Lock like the first implementation shown in this text, and thus claim slipped conditions to be a rather theoretical problem. But the first example was kept rather simple to better convey the notion of slipped conditions.
A more realistic example would be during the implementation of a fair lock, as discussed in the text on Starvation and Fairness. If we look at the naive implementation from the text Nested Monitor Lockout, and try to remove the nested monitor lock problem it, it is easy to

arrive at an implementation that suffers from slipped conditions. First I'll show the example from the nested monitor lockout text:

```
//Fair Lock implementation with nested monitor lockout problem

public class FairLock {
  private boolean           isLocked       = false;
  private Thread            lockingThread  = null;
  private List<QueueObject> waitingThreads =
          new ArrayList<QueueObject>();

  public void lock() throws InterruptedException{
    QueueObject queueObject = new QueueObject();

    synchronized(this){
      waitingThreads.add(queueObject);

      while(isLocked || waitingThreads.get(0) != queueObject){

        synchronized(queueObject){
          try{
            queueObject.wait();
          }catch(InterruptedException e){
            waitingThreads.remove(queueObject);
            throw e;
          }
        }
      }
      waitingThreads.remove(queueObject);
      isLocked = true;
      lockingThread = Thread.currentThread();
    }
  }

  public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
      throw new IllegalMonitorStateException(
        "Calling thread has not locked this lock");
    }
    isLocked      = false;
    lockingThread = null;
    if(waitingThreads.size() > 0){
      QueueObject queueObject = waitingThread.get(0);
      synchronized(queueObject){
        queueObject.notify();
      }
    }
  }
}

public class QueueObject {}
```

Notice how the synchronized(queueObject) with its queueObject.wait() call is nested inside the synchronized(this) block, resulting in the nested monitor lockout problem. To avoid this problem the synchronized(queueObject) block must be moved outside the synchronized(this) block. Here is how that could look:

```
//Fair Lock implementation with slipped conditions problem

public class FairLock {
  private boolean           isLocked       = false;
  private Thread            lockingThread  = null;
  private List<QueueObject> waitingThreads =
          new ArrayList<QueueObject>();

  public void lock() throws InterruptedException{
    QueueObject queueObject = new QueueObject();

    synchronized(this){
      waitingThreads.add(queueObject);
    }

    boolean mustWait = true;
    while(mustWait){

      synchronized(this){
        mustWait = isLocked || waitingThreads.get(0) != queueObject;
      }

      synchronized(queueObject){
        if(mustWait){
          try{
            queueObject.wait();
          }catch(InterruptedException e){
            waitingThreads.remove(queueObject);
            throw e;
          }
        }
      }
    }

    synchronized(this){
      waitingThreads.remove(queueObject);
      isLocked = true;
      lockingThread = Thread.currentThread();
    }
  }
}
```

Note: Only the lock() method is shown, since it is the only method I have changed.
Notice how the lock() method now contains 3 synchronized blocks.
The first synchronized(this) block checks the condition by setting mustWait = isLocked || waitingThreads.get(0) != queueObject.
The second synchronized(queueObject) block checks if the thread is to wait or not. Already at this time another thread may have unlocked the lock, but lets forget that for the time being. Let's assume that the lock was unlocked, so the thread exits the synchronized(queueObject) block right away.
The third synchronized(this) block is only executed if mustWait = false. This sets the condition isLocked back to true etc. and leaves the lock() method.
Imagine what will happen if two threads call lock() at the same time when the lock is unlocked. First thread 1 will check the isLocked condititition and see it false. Then thread 2 will do the same thing. Then neither of them will wait, and both will set the state isLocked to true. This is a prime example of slipped conditions.

Removing the Slipped Conditions Problem

To remove the slipped conditions problem from the example above, the content of the last synchronized(this) block must be moved up into the first block. The code will naturally have to be changed a little bit too, to adapt to this move. Here is how it looks:

```
//Fair Lock implementation without nested monitor lockout problem,
//but with missed signals problem.

public class FairLock {
  private boolean           isLocked       = false;
  private Thread            lockingThread  = null;
  private List<QueueObject> waitingThreads =
          new ArrayList<QueueObject>();

  public void lock() throws InterruptedException{
    QueueObject queueObject = new QueueObject();

    synchronized(this){
      waitingThreads.add(queueObject);
    }

    boolean mustWait = true;
    while(mustWait){


      synchronized(this){
        mustWait = isLocked || waitingThreads.get(0) != queueObject;
      if(!mustWait){
        waitingThreads.remove(queueObject);
        isLocked = true;
        lockingThread = Thread.currentThread();
        return;
      }
    }

   synchronized(queueObject){
     if(mustWait){
       try{
         queueObject.wait();
       }catch(InterruptedException e){
         waitingThreads.remove(queueObject);
         throw e;
       }
     }
   }
  }
 }
}
```

Notice how the local variable mustWait is tested and set within the same synchronized code block now. Also notice, that even if the mustWait local variable is also checked outside the synchronized(this) code block, in the while(mustWait) clause, the value of the mustWait variable is never changed outside the synchronized(this). A thread that evaluates mustWait to false will atomically also set the internal conditions (isLocked) so that any other thread checking the condition will evaluate it to true.

The return; statement in the synchronized(this) block is not necessary. It is just a small optimization. If the thread must not wait (mustWait == false), then there is no reason to enter the synchronized(queueObject) block and execute the if(mustWait) clause.

The observant reader will notice that the above implementation of a fair lock still suffers from a missed signal problem. Imagine that the FairLock instance is locked when a thread calls lock(). After the first synchronized(this) block mustWait is true. Then imagine that the thread calling lock() is preempted, and the thread that locked the lock calls unlock(). If you look at the unlock() implementation shown earlier, you will notice that it calls queueObject.notify(). But, since the thread waiting in lock() has not yet called queueObject.wait(), the call to queueObject.notify() passes into oblivion. The signal is missed. When the thread calling lock() right after calls queueObject.wait() it will remain blocked until some other thread calls unlock(), which may never happen.

The missed signals problems is the reason that the FairLock implementation shown in the text Starvation and Fairness has turned the QueueObject class into a semaphore with two methods: doWait() and doNotify(). These methods store and react the signal internally in the QueueObject. That way the signal is not missed, even if doNotify() is called before doWait().

Locks in Java

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. Locks (and other more advanced synchronization mechanisms) are created using synchronized blocks, so it is not like we can get totally rid of the synchronized keyword.
From Java 5 the package java.util.concurrent.locks contains several lock implementations, so you may not have to implement your own locks. But you will still need to know how to use them, and it can still be useful to know the theory behind their implementation. For more details, see my tutorial on the java.util.concurrent.locks.Lock interface.

A Simple Lock

Let's start out by looking at a synchronized block of Java code:

```
public class Counter{

  private int count = 0;

  public int inc(){
    synchronized(this){
      return ++count;
    }
  }
}
```

Notice the synchronized(this) block in the inc() method. This block makes sure that only one thread can execute the return ++count at a time. The code in the synchronized block could have been more advanced, but the simple ++count suffices to get the point across.
The Counter class could have been written like this instead, using a Lock instead of a synchronized block:

```
public class Counter{

  private Lock lock = new Lock();
  private int count = 0;

  public int inc(){
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }
}
```

The lock() method locks the Lock instance so that all threads calling lock() are blocked until unlock() is executed.
Here is a simple Lock implementation:

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
    isLocked = false;
    notify();
  }
}
```

Notice the while(isLocked) loop, which is also called a "spin lock". Spin locks and the methods wait() and notify() are covered in more detail in the text Thread Signaling. While isLocked is true, the thread calling lock() is parked waiting in the wait() call. In case the thread should return unexpectedly from the wait() call without having received a notify() call (AKA a Spurious Wakeup) the thread re-checks the isLocked condition to see if it is safe to proceed or not, rather than just assume that being awakened means it is safe to proceed. If isLocked is false, the thread exits the while(isLocked) loop, and sets isLocked back to true, to lock the Lock instance for other threads calling lock().
When the thread is done with the code in the critical section (the code between lock() and unlock()), the thread calls unlock(). Executing unlock() sets isLocked back to false, and notifies (awakens) one of the threads waiting in the wait() call in the lock() method, if any.

Lock Reentrance

Synchronized blocks in Java are reentrant. This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object. Here is an example:

```
public class Reentrant{

  public synchronized outer(){
    inner();
  }

  public synchronized inner(){
    //do something
  }
}
```

Notice how both outer() and inner() are declared synchronized, which in Java is equivalent to a synchronized(this) block. If a thread calls outer() there is no problem calling inner() from inside outer(), since both methods (or blocks) are synchronized on the same monitor object ("this"). If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.
The lock implementation shown earlier is not reentrant. If we rewrite the Reentrant class like below, the thread calling outer() will be blocked inside the lock.lock() in the inner() method.

```
public class Reentrant2{

  Lock lock = new Lock();

  public outer(){
    lock.lock();
    inner();
    lock.unlock();
  }

  public synchronized inner(){
    lock.lock();
    //do something
    lock.unlock();
  }
}
```

A thread calling outer() will first lock the Lock instance. Then it will call inner(). Inside the inner() method the thread will again try to lock the Lock instance. This will fail (meaning the thread will be blocked), since the Lock instance was locked already in the outer() method.
The reason the thread will be blocked the second time it calls lock() without having called unlock() in between, is apparent when we look at the lock() implementation:

```
public class Lock{

  boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }
  ...
}
```

It is the condition inside the while loop (spin lock) that determines if a thread is allowed to exit the lock() method or not. Currently the condition is that isLocked must be false for this to be allowed, regardless of what thread locked it.
To make the Lock class reentrant we need to make a small change:

```
public class Lock{

  boolean isLocked = false;
  Thread  lockedBy = null;
  int     lockedCount = 0;

  public synchronized void lock()
  throws InterruptedException{
    Thread callingThread = Thread.currentThread();
    while(isLocked && lockedBy != callingThread){
      wait();
    }
    isLocked = true;
    lockedCount++;
    lockedBy = callingThread;
  }


  public synchronized void unlock(){
    if(Thread.curentThread() == this.lockedBy){
      lockedCount--;

      if(lockedCount == 0){
        isLocked = false;
        notify();
      }
    }
  }

  ...
}
```

Notice how the while loop (spin lock) now also takes the thread that locked the Lock instance into consideration. If either the lock is unlocked (isLocked = false) or the calling thread is the thread that locked the Lock instance, the while loop will not execute, and the thread calling lock() will be allowed to exit the method.
Additionally, we need to count the number of times the lock has been locked by the same thread. Otherwise, a single call to unlock() will unlock the lock, even if the lock has been

locked multiple times. We don't want the lock to be unlocked until the thread that locked it, has executed the same amount of unlock() calls as lock() calls.
The Lock class is now reentrant.

Lock Fairness

Java's synchronized blocks makes no guarantees about the sequence in which threads trying to enter them are granted access. Therefore, if many threads are constantly competing for access to the same synchronized block, there is a risk that one or more of the threads are never granted access - that access is always granted to other threads. This is called starvation. To avoid this a Lock should be fair. Since the Lock implementations shown in this text uses synchronized blocks internally, they do not guarantee fairness. Starvation and fairness are discussed in more detail in the text Starvation and Fairness.

Calling unlock() From a finally-clause

When guarding a critical section with a Lock, and the critical section may throw exceptions, it is important to call the unlock() method from inside a finally-clause. Doing so makes sure that the Lock is unlocked so other threads can lock it. Here is an example:

```
lock.lock();
try{
  //do critical section code, which may throw exception
} finally {
  lock.unlock();
}
```

This little construct makes sure that the Lock is unlocked in case an exception is thrown from the code in the critical section. If unlock() was not called from inside a finally-clause, and an exception was thrown from the critical section, the Lock would remain locked forever, causing all threads calling lock() on that Lock instance to halt indefinately.

Read / Write Locks in Java

A read / write lock is more sophisticated lock than the Lock implementations shown in the text Locks in Java. Imagine you have an application that reads and writes some resource, but writing it is not done as much as reading it is. Two threads reading the same resource does not cause problems for each other, so multiple threads that want to read the resource are granted access at the same time, overlapping. But, if a single thread wants to write to the resource, no other reads nor writes must be in progress at the same time. To solve this problem of allowing multiple readers but only one writer, you will need a read / write lock. Java 5 comes with read / write lock implementations in the java.util.concurrent package. Even so, it may still be useful to know the theory behind their implementation.

Read / Write Lock Java Implementation

First let's summarize the conditions for getting read and write access to the resource:

Read Access	If no threads are writing, and no threads have requested write access.
Write Access	If no threads are reading or writing.

If a thread wants to read the resource, it is okay as long as no threads are writing to it, and no threads have requested write access to the resource. By up-prioritizing write-access requests we assume that write requests are more important than read-requests. Besides, if reads are what happens most often, and we did not up-prioritize writes, starvation could occur. Threads requesting write access would be blocked until all readers had unlocked the ReadWriteLock. If new threads were constantly granted read access the thread waiting for write access would remain blocked indefinately, resulting in starvation. Therefore a thread can only be granted read access if no thread has currently locked the ReadWriteLock for writing, or requested it locked for writing.
A thread that wants write access to the resource can be granted so when no threads are reading nor writing to the resource. It doesn't matter how many threads have requested write access or in what sequence, unless you want to guarantee fairness between threads requesting write access.
With these simple rules in mind we can implement a ReadWriteLock as shown below:

```
public class ReadWriteLock{

  private int readers       = 0;
  private int writers       = 0;
  private int writeRequests = 0;

  public synchronized void lockRead() throws InterruptedException{
    while(writers > 0 || writeRequests > 0){
      wait();
    }
    readers++;
  }

  public synchronized void unlockRead(){
    readers--;
    notifyAll();
  }

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;

    while(readers > 0 || writers > 0){
      wait();
    }
    writeRequests--;
    writers++;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writers--;
    notifyAll();
  }
}
```

The ReadWriteLock has two lock methods and two unlock methods. One lock and unlock method for read access and one lock and unlock for write access.
The rules for read access are implemented in the lockRead() method. All threads get read access unless there is a thread with write access, or one or more threads have requested write access.
The rules for write access are implemented in the lockWrite() method. A thread that wants write access starts out by requesting write access (writeRequests++). Then it will check if it can actually get write access. A thread can get write access if there are no threads with read access to the resource, and no threads with write access to the resource. How many threads have requested write access doesn't matter.
It is worth noting that both unlockRead() and unlockWrite() calls notifyAll() rather than notify(). To explain why that is, imagine the following situation:
Inside the ReadWriteLock there are threads waiting for read access, and threads waiting for write access. If a thread awakened by notify() was a read access thread, it would be put back to waiting because there are threads waiting for write access. However, none of the threads awaiting write access are awakened, so nothing more happens. No threads gain neither read nor write access. By calling noftifyAll() all waiting threads are awakened and check if they can get the desired access.
Calling notifyAll() also has another advantage. If multiple threads are waiting for read access and none for write access, and unlockWrite() is called, all threads waiting for read access are granted read access at once - not one by one.

Read / Write Lock Reentrance

The ReadWriteLock class shown earlier is not reentrant. If a thread that has write access requests it again, it will block because there is already one writer - itself. Furthermore, consider this case:

1. Thread 1 gets read access.
2.
3.
4. Thread 2 requests write access but is blocked because there is one reader.
5.
6.
7. Thread 1 re-requests read access (re-enters the lock), but is blocked because there is a write request

In this situation the previous ReadWriteLock would lock up - a situation similar to deadlock. No threads requesting neither read nor write access would be granted so.
To make the ReadWriteLock reentrant it is necessary to make a few changes. Reentrance for readers and writers will be dealt with separately.

Read Reentrance

To make the ReadWriteLock reentrant for readers we will first establish the rules for read reentrance:

- A thread is granted read reentrance if it can get read access (no writers or write requests), or if it already has read access (regardless of write requests).

To determine if a thread has read access already a reference to each thread granted read access is kept in a Map along with how many times it has acquired read lock. When determing if read access can be granted this Map will be checked for a reference to the calling thread. Here is how the lockRead() and unlockRead() methods looks after that change:

```
public class ReadWriteLock{

  private Map<Thread, Integer> readingThreads =
     new HashMap<Thread, Integer>();

  private int writers        = 0;
  private int writeRequests  = 0;

  public synchronized void lockRead() throws InterruptedException{
    Thread callingThread = Thread.currentThread();
    while(! canGrantReadAccess(callingThread)){
      wait();
    }

    readingThreads.put(callingThread,
       (getAccessCount(callingThread) + 1));
  }


  public synchronized void unlockRead(){
    Thread callingThread = Thread.currentThread();
    int accessCount = getAccessCount(callingThread);
    if(accessCount == 1){ readingThreads.remove(callingThread); }
    else { readingThreads.put(callingThread, (accessCount -1)); }
    notifyAll();
  }


  private boolean canGrantReadAccess(Thread callingThread){
    if(writers > 0)            return false;
    if(isReader(callingThread) return true;
    if(writeRequests > 0)      return false;
    return true;
  }

  private int getReadAccessCount(Thread callingThread){
    Integer accessCount = readingThreads.get(callingThread);
    if(accessCount == null) return 0;
    return accessCount.intValue();
  }

  private boolean isReader(Thread callingThread){
    return readingThreads.get(callingThread) != null;
  }

}
```

As you can see read reentrance is only granted if no threads are currently writing to the resource. Additionally, if the calling thread already has read access this takes precedence over any writeRequests.

Write Reentrance

Write reentrance is granted only if the thread has already write access. Here is how the lockWrite() and unlockWrite() methods look after that change:

```
public class ReadWriteLock{

    private Map<Thread, Integer> readingThreads =
        new HashMap<Thread, Integer>();

    private int writeAccesses    = 0;
    private int writeRequests    = 0;
    private Thread writingThread = null;

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
    Thread callingThread = Thread.currentThread();
    while(! canGrantWriteAccess(callingThread)){
      wait();
    }
    writeRequests--;
    writeAccesses++;
    writingThread = callingThread;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writeAccesses--;
    if(writeAccesses == 0){
      writingThread = null;
    }
    notifyAll();
  }

  private boolean canGrantWriteAccess(Thread callingThread){
    if(hasReaders())             return false;
    if(writingThread == null)    return true;
    if(!isWriter(callingThread)) return false;
    return true;
  }

  private boolean hasReaders(){
    return readingThreads.size() > 0;
  }

  private boolean isWriter(Thread callingThread){
    return writingThread == callingThread;
  }
}
```

Notice how the thread currently holding the write lock is now taken into account when determining if the calling thread can get write access.

Read to Write Reentrance

Sometimes it is necessary for a thread that have read access to also obtain write access. For this to be allowed the thread must be the only reader. To achieve this the writeLock() method should be changed a bit. Here is what it would look like:

```
public class ReadWriteLock{

    private Map<Thread, Integer> readingThreads =
        new HashMap<Thread, Integer>();

    private int writeAccesses    = 0;
    private int writeRequests    = 0;
    private Thread writingThread = null;

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
    Thread callingThread = Thread.currentThread();
    while(! canGrantWriteAccess(callingThread)){
      wait();
    }
    writeRequests--;
    writeAccesses++;
    writingThread = callingThread;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writeAccesses--;
    if(writeAccesses == 0){
      writingThread = null;
    }
    notifyAll();
  }

  private boolean canGrantWriteAccess(Thread callingThread){
    if(isOnlyReader(callingThread))    return true;
    if(hasReaders())                   return false;
    if(writingThread == null)          return true;
    if(!isWriter(callingThread))       return false;
    return true;
  }

  private boolean hasReaders(){
    return readingThreads.size() > 0;
  }

  private boolean isWriter(Thread callingThread){
    return writingThread == callingThread;
  }

  private boolean isOnlyReader(Thread thread){
      return readers == 1 && readingThreads.get(callingThread) != null;
      }

}
```

Now the ReadWriteLock class is read-to-write access reentrant.

Write to Read Reentrance

Sometimes a thread that has write access needs read access too. A writer should always be granted read access if requested. If a thread has write access no other threads can have read nor write access, so it is not dangerous. Here is how the canGrantReadAccess() method will look with that change:

```
public class ReadWriteLock{

    private boolean canGrantReadAccess(Thread callingThread){
```

```
    **if(isWriter(callingThread)) return true;**
    if(writingThread != null)   return false;
    if(isReader(callingThread)  return true;
    if(writeRequests > 0)       return false;
    return true;
  }

}
```

Fully Reentrant ReadWriteLock

Below is the fully reentran ReadWriteLock implementation. I have made a few refactorings to the access conditions to make them easier to read, and thereby easier to convince yourself that they are correct.

```
public class ReadWriteLock{

  private Map<Thread, Integer> readingThreads =
       new HashMap<Thread, Integer>();

   private int writeAccesses    = 0;
   private int writeRequests    = 0;
   private Thread writingThread = null;


  public synchronized void lockRead() throws InterruptedException{
    Thread callingThread = Thread.currentThread();
    while(! canGrantReadAccess(callingThread)){
      wait();
    }

    readingThreads.put(callingThread,
     (getReadAccessCount(callingThread) + 1));
  }

  private boolean canGrantReadAccess(Thread callingThread){
    if( isWriter(callingThread) ) return true;
    if( hasWriter()             ) return false;
    if( isReader(callingThread) ) return true;
    if( hasWriteRequests()      ) return false;
    return true;
  }


  public synchronized void unlockRead(){
    Thread callingThread = Thread.currentThread();
    if(!isReader(callingThread)){
      throw new IllegalMonitorStateException("Calling Thread does not" +
        " hold a read lock on this ReadWriteLock");
    }
    int accessCount = getReadAccessCount(callingThread);
    if(accessCount == 1){ readingThreads.remove(callingThread); }
    else { readingThreads.put(callingThread, (accessCount -1)); }
    notifyAll();
  }

  public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
    Thread callingThread = Thread.currentThread();
    while(! canGrantWriteAccess(callingThread)){
      wait();
    }
    writeRequests--;
    writeAccesses++;
    writingThread = callingThread;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    if(!isWriter(Thread.currentThread()){
      throw new IllegalMonitorStateException("Calling Thread does not" +
        " hold the write lock on this ReadWriteLock");
    }
    writeAccesses--;
    if(writeAccesses == 0){
      writingThread = null;
    }
    notifyAll();
  }

  private boolean canGrantWriteAccess(Thread callingThread){
    if(isOnlyReader(callingThread))    return true;
    if(hasReaders())                   return false;
    if(writingThread == null)          return true;
    if(!isWriter(callingThread))       return false;
    return true;
  }


  private int getReadAccessCount(Thread callingThread){
    Integer accessCount = readingThreads.get(callingThread);
    if(accessCount == null) return 0;
    return accessCount.intValue();
  }


  private boolean hasReaders(){
    return readingThreads.size() > 0;
  }

  private boolean isReader(Thread callingThread){
    return readingThreads.get(callingThread) != null;
  }

  private boolean isOnlyReader(Thread callingThread){
    return readingThreads.size() == 1 &&
           readingThreads.get(callingThread) != null;
  }

  private boolean hasWriter(){
    return writingThread != null;
  }

  private boolean isWriter(Thread callingThread){
    return writingThread == callingThread;
  }

  private boolean hasWriteRequests(){
      return this.writeRequests > 0;
  }

}
```

Calling unlock() From a finally-clause

When guarding a critical section with a ReadWriteLock, and the critical section may throw exceptions, it is important to call the readUnlock() and writeUnlock() methods from inside a finally-clause. Doing so makes sure that the ReadWriteLock is unlocked so other threads can lock it. Here is an example:

```
lock.lockWrite();
try{
  //do critical section code, which may throw exception
} finally {
  lock.unlockWrite();
}
```

This little construct makes sure that the ReadWriteLock is unlocked in case an exception is thrown from the code in the critical section. If unlockWrite() was not called from inside a finally-clause, and an exception was thrown from the critical section, the ReadWriteLock would remain write locked forever, causing all threads calling lockRead() or lockWrite() on that ReadWriteLock instance to halt indefinately. The only thing that could unlock the ReadWriteLockagain would be if the ReadWriteLock is reentrant, and the thread that had it locked when the exception was thrown, later succeeds in locking it, executing the critical section and calling unlockWrite() again afterwards. That would unlock the ReadWriteLock again. But why wait for that to happen, **if** it happens? Calling unlockWrite() from a finally-clause is a much more robust solution.

Reentrance Lockout

Reentrance lockout is a situation similar to deadlock and nested monitor lockout. Reentrance lockout is also covered in part in the texts on Locks and Read / Write Locks. Reentrance lockout may occur if a thread reenters a Lock, ReadWriteLock or some other synchronizer that is not reentrant. Reentrant means that a thread that already holds a lock can retake it. Java's synchronized blocks are reentrant. Therefore the following code will work without problems:

```
public class Reentrant{

  public synchronized outer(){
    inner();
  }

  public synchronized inner(){
    //do something
  }
}
```

Notice how both outer() and inner() are declared synchronized, which in Java is equivalent to a synchronized(this) block. If a thread calls outer() there is no problem calling inner() from inside outer(), since both methods (or blocks) are synchronized on the same monitor object ("this"). If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.
The following Lock implementation is not reentrant:

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
    isLocked = false;
    notify();
  }
}
```

If a thread calls lock() twice without calling unlock() in between, the second call to lock() will block. A reentrance lockout has occurred.
To avoid reentrance lockouts you have two options:

1. Avoid writing code that reenters locks
2. Use reentrant locks

Which of these options suit your project best depends on your concrete situation. Reentrant locks often don't perform as well as non-reentrant locks, and they are harder to implement, but this may not necessary be a problem in your case. Whether or not your code is easier to implement with or without lock reentrance must be determined case by case.

Semaphores

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid missed signals, or to guard a critical section like you would with a lock. Java 5 comes with semaphore implementations in the java.util.concurrent package so you don't have to implement your own semaphores. Still, it can be useful to know the theory behind their implementation and use.
Java 5 comes with a built-in Semaphore so you don't have to implement your own. You can read more about it in the java.util.concurrent.Semaphore text, in my java.util.concurrent tutorial.

Simple Semaphore

Here is a simple Semaphore implementation:

```
public class Semaphore {
  private boolean signal = false;

  public synchronized void take() {
    this.signal = true;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(!this.signal) wait();
    this.signal = false;
  }

}
```

The take() method sends a signal which is stored internally in the Semaphore. The release() method waits for a signal. When received the signal flag is cleared again, and the release() method exited.
Using a semaphore like this you can avoid missed signals. You will call take() instead of notify() and release() instead of wait(). If the call to take() happens before the call to release() the thread calling release() will still know that take() was called, because the signal is stored internally in the signal variable. This is not the case with wait() and notify().
The names take() and release() may seem a bit odd when using a semaphore for signaling. The names origin from the use of semaphores as locks, as explained later in this text. In that case the names make more sense.

Using Semaphores for Signaling

Here is a simplified example of two threads signaling each other using a Semaphore:

```
Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();
sender.start();

public class SendingThread {
  Semaphore semaphore = null;

  public SendingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      //do something, then signal
      this.semaphore.take();

    }
  }
}

public class RecevingThread {
  Semaphore semaphore = null;
  public ReceivingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      this.semaphore.release();
      //receive signal, then do something...
    }
  }
}
```

Counting Semaphore

The Semaphore implementation in the previous section does not count the number of signals sent to it by take() method calls. We can change the Semaphore to do so. This is called a counting semaphore. Here is a simple implementation of a counting semaphore:

```
public class CountingSemaphore {
  private int signals = 0;

  public synchronized void take() {
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }

}
```

Bounded Semaphore

The CoutingSemaphore has no upper bound on how many signals it can store. We can change the semaphore implementation to have an upper bound, like this:

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }
  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

Notice how the take() method now blocks if the number of signals is equal to the upper bound. Not until a thread has called release() will the thread calling take() be allowed to deliver its signal, if the BoundedSemaphore has reached its upper signal limit.

Using Semaphores as Locks

It is possible to use a bounded semaphore as a lock. To do so, set the upper bound to 1, and have the call to take() and release() guard the critical section. Here is an example:

```
BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{
  //critical section
} finally {
  semaphore.release();
}
```

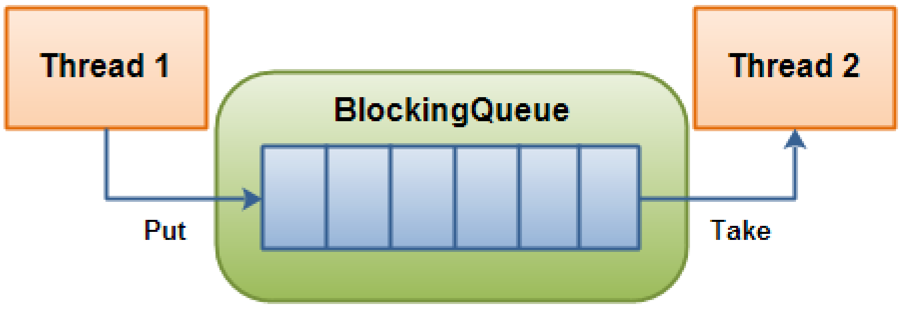
In contrast to the signaling use case the methods take() and release() are now called by the same thread. Since only one thread is allowed to take the semaphore, all other threads calling take() will be blocked until release() is called. The call to release() will never block since there has always been a call to take() first.

You can also use a bounded semaphore to limit the number of threads allowed into a section of code. For instance, in the example above, what would happen if you set the limit of the BoundedSemaphore to 5? 5 threads would be allowed to enter the critical section at a time. You would have to make sure though, that the thread operations do not conflict for these 5 threads, or you application will fail.

The relase() method is called from inside a finally-block to make sure it is called even if an exception is thrown from the critical section.

Blocking Queues

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.
Here is a diagram showing two threads cooperating via a blocking queue:



A BlockingQueue with one thread putting into it, and another thread taking from it.

Java 5 comes with blocking queue implementations in the java.util.concurrent package. You can read about that class in my java.util.concurrent.BlockingQueue tutorial. Even if Java 5 comes with a blocking queue implementation, it can be useful to know the theory behind their implementation.

Blocking Queue Implementation

The implementation of a blocking queue looks similar to a Bounded Semaphore. Here is a simple implementation of a blocking queue:

```
public class BlockingQueue {

  private List queue = new LinkedList();
  private int  limit = 10;

  public BlockingQueue(int limit){
    this.limit = limit;
  }


  public synchronized void enqueue(Object item)
  throws InterruptedException  {
    while(this.queue.size() == this.limit) {
      wait();
    }
    if(this.queue.size() == 0) {
      notifyAll();
    }
    this.queue.add(item);
  }


  public synchronized Object dequeue()
  throws InterruptedException{
    while(this.queue.size() == 0){
      wait();
    }
    if(this.queue.size() == this.limit){
      notifyAll();
    }

    return this.queue.remove(0);
  }

}
```

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items.

Thread Pools

Thread Pools are useful when you need to limit the number of threads running in your application at the same time. There is a performance overhead associated with starting a new thread, and each thread is also allocated some memory for its stack etc.
Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a Blocking Queue which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.
Thread pools are often used in multi threaded servers. Each connection arriving at the server via the network is wrapped as a task and passed on to a thread pool. The threads in the thread pool will process the requests on the connections concurrently. A later trail will get into detail about implementing multithreaded servers in Java.

Java 5 comes with built in thread pools in the java.util.concurrent package, so you don't have to implement your own thread pool. You can read more about it in my text on the java.util.concurrent.ExecutorService. Still it can be useful to know a bit about the implementation of a thread pool anyways.
Here is a simple thread pool implementation. Please note that this implementation uses my own BlockingQueue class as explained in my Blocking Queues tutorial. In a real life implementation you would probably use one of Java's built-in blocking queues instead.

```
public class ThreadPool {

    private BlockingQueue taskQueue = null;
    private List<PoolThread> threads = new ArrayList<PoolThread>();
    private boolean isStopped = false;

    public ThreadPool(int noOfThreads, int maxNoOfTasks){
        taskQueue = new BlockingQueue(maxNoOfTasks);

        for(int i=0; i<noOfThreads; i++){
            threads.add(new PoolThread(taskQueue));
        }
        for(PoolThread thread : threads){
            thread.start();
        }
    }

    public synchronized void  execute(Runnable task) throws Exception{
        if(this.isStopped) throw
            new IllegalStateException("ThreadPool is stopped");

        this.taskQueue.enqueue(task);
    }

    public synchronized void stop(){
        this.isStopped = true;
        for(PoolThread thread : threads){
           thread.doStop();
        }
    }

}

public class PoolThread extends Thread {

    private BlockingQueue taskQueue = null;
    private boolean       isStopped = false;

    public PoolThread(BlockingQueue queue){
        taskQueue = queue;
    }

    public void run(){
        while(!isStopped()){
            try{
                Runnable runnable = (Runnable) taskQueue.dequeue();
                runnable.run();
            } catch(Exception e){
                //log or otherwise report exception,
                //but keep pool thread alive.
            }
        }
    }

    public synchronized void doStop(){
        isStopped = true;
        this.interrupt(); //break pool thread out of dequeue() call.
    }

    public synchronized boolean isStopped(){
        return isStopped;
    }
}
```

The thread pool implementation consists of two parts. A ThreadPool class which is the public interface to the thread pool, and a PoolThread class which implements the threads that execute the tasks.
To execute a task the method ThreadPool.execute(Runnable r) is called with a Runnable implementation as parameter. The Runnable is enqueued in the blocking queue internally, waiting to be dequeued.
The Runnable will be dequeued by an idle PoolThread and executed. You can see this in the PoolThread.run() method. After execution the PoolThread loops and tries to dequeue a task again, until stopped.
To stop the ThreadPool the method ThreadPool.stop() is called. The stop called is noted internally in the isStopped member. Then each thread in the pool is stopped by calling doStop() on each thread. Notice how the execute() method will throw an IllegalStateException if execute() is called after stop() has been called.
The threads will stop after finishing any task they are currently executing. Notice the this.interrupt() call in PoolThread.doStop(). This makes sure that a thread blocked in a wait() call inside the taskQueue.dequeue() call breaks out of the wait() call, and leaves the dequeue() method call with an InterruptedException thrown. This exception is caught in the PoolThread.run() method, reported, and then the isStopped variable is checked. Since isStopped is now true, the PoolThread.run() will exit and the thread dies.

Compare and Swap

Compare and swap is a technique used when designing concurrent algorithms. Basically, compare and swap compares an expected value to the concrete value of a variable, and if the concrete value of the variable is equals to the expected value, swaps the value of the variable for a new variable. Compare and swap may sound a bit complicated but it is actually reasonably simple once you understand it, so let me elaborate a bit further on the topic.

What Situations Compare And Swap is Intended to Support

A very commonly occurring patterns in programs and concurrent algorithms is the "check then act" pattern. The check then act pattern occurs when the code first checks the value of a variable and then acts based on that value. Here is a simple example:

```
class MyLock {

    private boolean locked = false;

    public boolean lock() {
        if(!locked) {
            locked = true;
            return true;
        }
        return false;
    }
}
```

This code has many errors if it was to be used in a multithreaded application, but please ignore that for now.
As you can see, the lock() method first checks if the locked member variable is equal to false (check), and if it is it ses locked to true (then act).
If multiple threads had access to the same MyLock instance, the above lock() function would not be guaranteed to work. If a thread A checks the value of locked and sees that it is false, a thread B may also check the value of locked at exactly the same time. Or, in fact, at any time before thread A sets locked to false. Thus, both thread A and thread B may see locked as being false, and both will then act based on that information.
To work properly in a multithreaded application, "check then act" operations must be atomic. By atomic is meant that both the "check" and "act" actions are executed as an atomic (non-dividable) block of code. Any thread that starts executing the block will finish executing the block without interference from other threads. No other threads can execute the atomic block at the same time.
Here is the code example from earlier with the lock() method turned into an atomic block of code using the synchronized keyword:

```
class MyLock {

    private boolean locked = false;

    public synchronized boolean lock() {
        if(!locked) {
            locked = true;
            return true;
        }
        return false;
    }
}
```

Now the lock() method is synchronized so only one thread can executed it at a time on the same MyLock instance. The lock() method is effectively atomic.
The atomic lock() method is actually an example of "compare and swap". The lock() method *compares* the variable locked to the expected value false and if locked is equal to this expected value, it *swaps* the variable's value to true .

Compare And Swap As Atomic Operation

Modern CPUs have built-in support for atomic compare and swap operations. From Java 5 you can get access to these functions in the CPU via some of the new atomic classes in the java.util.concurrent.atomic package.
Here is an example showing how to implement the lock() method shown earlier using the AtomicBoolean class:

```
public static class MyLock {
    private AtomicBoolean locked = new AtomicBoolean(false);

    public boolean lock() {
        return locked.compareAndSet(false, true);
    }

}
```

Notice how the locked variable is no longer a boolean but an AtomicBoolean. This class has a compareAndSet() function which will compare the value of the AtomicBoolean instance to an expected value, and if has the expected value, it swaps the value with a new value. In this case it compares the value of locked to false and if it is false it sets the new value of the AtomicBoolean to true.
The compareAndSet() method returns true if the value was swapped, and false if not.
The advantage of using the compare and swap features that comes with Java 5+ rather than implementing your own is that the compare and swap features built into Java 5+ lets you utilize the underlying compare and swap features of the CPU your application is running on. This makes your compare and swap code faster.

Anatomy of a Synchronizer

Even if many synchronizers (locks, semaphores, blocking queue etc.) are different in function, they are often not that different in their internal design. In other words, they consist of the same (or similar) basic parts internally. Knowing these basic parts can be a great help when designing synchronizers. It is these parts this text looks closer at.

Note: The content of this text is a part result of a M.Sc. student project at the IT University of Copenhagen in the spring 2004 by Jakob Jenkov, Toke Johansen and Lars Bjørn. During this project we asked Doug Lea if he knew of similar work. Interestingly he had come up with similar conclusions independently of this project during the development of the Java 5 concurrency utilities. Doug Lea's work, I believe, is described in the book "Java Concurrency in Practice". This book also contains a chapter with the title "Anatomy of a Synchronizer" with content similar to this text, though not exactly the same.

The purpose of most (if not all) synchronizers is to guard some area of the code (critical section) from concurrent access by threads. To do this the following parts are often needed in a synchronizer:

1. State
2. Access Condition
3. State Changes
4. Notification Strategy
5. Test and Set Method
6. Set Method

Not all synchronizers have all of these parts, and those that have may not have them exactly as they are described here. Usually you can find one or more of these parts, though.

State

The state of a synchronizer is used by the access condition to determine if a thread can be granted access. In a Lock the state is kept in a boolean saying whether the Lock is locked or not. In a Bounded Semaphore the internal state is kept in a counter (int) and an upper bound (int) which state the current number of "takes" and the maximum number of "takes". In a Blocking Queue the state is kept in the List of elements in the queue and the maximum queue size (int) member (if any).

Here are two code snippets from both Lock and a BoundedSemaphore. The state code is marked in bold.

```
public class Lock{

  //state is kept here
  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  ...
}

public class BoundedSemaphore {

  //state is kept here
      private int signals = 0;
      private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signal++;
    this.notify();
  }
  ...
}
```

Access Condition

The access conditions is what determines if a thread calling a test-and-set-state method can be allowed to set the state or not. The access condition is typically based on the state of the synchronizer. The access condition is typically checked in a while loop to guard against Spurious Wakeups. When the access condition is evaluated it is either true or false.
In a Lock the access condition simply checks the value of the isLocked member variable. In a Bounded Semaphore there are actually two access conditions depending on whether you are trying to "take" or "release" the semaphore. If a thread tries to take the semaphore the signals variable is checked against the upper bound. If a thread tries to release the semaphore the signals variable is checked against 0.
Here are two code snippets of a Lock and a BoundedSemaphore with the access condition marked in bold. Notice how the conditions is always checked inside a while loop.

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    //access condition
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  ...
}

public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    //access condition
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    //access condition
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

State Changes

Once a thread gains access to the critical section it has to change the state of the synchronizer to (possibly) block other threads from entering it. In other words, the state needs to reflect the fact that a thread is now executing inside the critical section. This should affect the access conditions of other threads attempting to gain access.
In a Lock the state change is the code setting isLocked = true. In a semaphore it is either the code signals-- or signals++;
Here are two code snippets with the state change code marked in bold:

```
public class Lock{

  private boolean isLocked = false;
  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    //state change
    isLocked = true;
  }

  public synchronized void unlock(){
    //state change
    isLocked = false;
    notify();
  }
}

public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    //state change
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    //state change
    this.signals--;
    this.notify();
  }
}
```

Notification Strategy

Once a thread has changed the state of a synchronizer it may sometimes need to notify other waiting threads about the state change. Perhaps this state change might turn the access condition true for other threads.
Notification Strategies typically fall into three categories.

1. Notify all waiting threads.
2. Notify 1 random of N waiting threads.
3. Notify 1 specific of N waiting thread.

Notifying all waiting threads is pretty easy. All waiting threads call wait() on the same object. Once a thread want to notify the waiting threads it calls notifyAll() on the object the waiting threads called wait() on.
Notifying a single random waiting thread is also pretty easy. Just have the notifying thread call notify() on the object the waiting threads have called wait() on. Calling notify makes no guarantee about which of the waiting threads will be notified. Hence the term "random waiting thread".
Sometimes you may need to notify a specific rather than a random waiting thread. For instance if you need to guarantee that waiting threads are notified in a specific order, be it the order they called the synchronizer in, or some prioritized order. To achive this each waiting thread must call wait() on its own, separate object. When the notifying thread wants to notify a specific waiting thread it will call notify() on the object this specific thread has called wait() on. An example of this can be found in the text Starvation and Fairness.
Below is a code snippet with the notification strategy (notify 1 random waiting thread) marked in bold:

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      //wait strategy - related to notification strategy
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
    isLocked = false;
    notify(); //notification strategy
  }
}
```

Test and Set Method

Synchronizer most often have two types of methods of which test-and-set is the first type (set is the other). Test-and-set means that the thread calling this method **tests** the internal state of the synchronizer against the access condition. If the condition is met the thread **sets** the internal state of the synchronizer to reflect that the thread has gained access.
The state transition usually results in the access condition turning false for other threads trying to gain access, but may not always do so. For instance, in a Read - Write Lock a

thread gaining read access will update the state of the read-write lock to reflect this, but other threads requesting read access will also be granted access as long as no threads has requested write access.
It is imperative that the test-and-set operations are executed atomically meaning no other threads are allowed to execute in the test-and-set method in between the test and the setting of the state.
The program flow of a test-and-set method is usually something along the lines of:

1. Set state before test if necessary
2. Test state against access condition
3. If access condition is not met, wait
4. If access condition is met, set state, and notify waiting threads if necessary

The lockWrite() method of a ReadWriteLock class shown below is an example of a test-and-set method. Threads calling lockWrite() first sets the state before the test (writeRequests++). Then it tests the internal state against the access condition in the canGrantWriteAccess() method. If the test succeeds the internal state is set again before the method is exited. Notice that this method does not notify waiting threads.

```
public class ReadWriteLock{
  private Map<Thread, Integer> readingThreads =
    new HashMap<Thread, Integer>();

  private int writeAccesses    = 0;
  private int writeRequests    = 0;
  private Thread writingThread = null;

  ...


    public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
    Thread callingThread = Thread.currentThread();
    while(! canGrantWriteAccess(callingThread)){
    wait();
    }
    writeRequests--;
    writeAccesses++;
    writingThread = callingThread;
    }


  ...
}
```

The BoundedSemaphore class shown below has two test-and-set methods: take() and release(). Both methods test and sets the internal state.

```
public class BoundedSemaphore {
 private int signals = 0;
```

```
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }


      public synchronized void take() throws InterruptedException{
      while(this.signals == bound) wait();
      this.signals++;
      this.notify();
      }

      public synchronized void release() throws InterruptedException{
      while(this.signals == 0) wait();
      this.signals--;
      this.notify();
      }

}
```

Set Method

The set method is the second type of method that synchronizers often contain. The set method just sets the internal state of the synchronizer without testing it first. A typical example of a set method is the unlock() method of a Lock class. A thread holding the lock can always unlock it without having to test if the Lock is unlocked.
The program flow of a set method is usually along the lines of:

1. Set internal state
2. Notify waiting threads

Here is an example unlock() method:

```
public class Lock{

  private boolean isLocked = false;

      public synchronized void unlock(){
      isLocked = false;
      notify();
      }

}
```

Non-blocking Algorithms

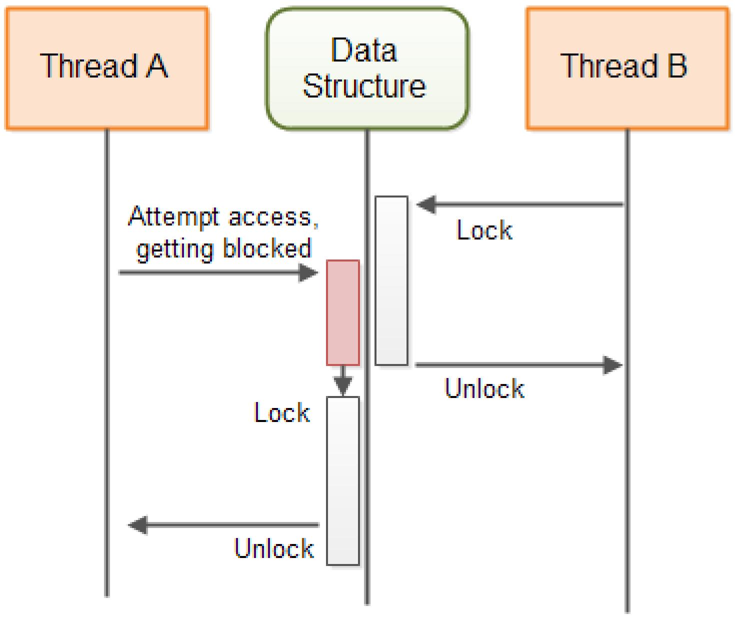
Non-blocking algorithms in the context of concurrency are algorithms that allows threads to access shared state (or otherwise collaborate or communicate) without blocking the threads involved. In more general terms, an algorithm is said to be non-blocking if the suspension of one thread cannot lead to the suspension of other threads involved in the algorithm.
To better understand the difference between blocking and non-blocking concurrency algorithms, I will start by explaining blocking algorithms and then continue with non-blocking algorithms.

Blocking Concurrency Algorithms

A blocking concurrency algorithm is an algorithm which either:

- A: Performs the action requested by the thread - OR
- B: Blocks the thread until the action can be performed safely

Many types of algorithms and concurrent data structures are blocking. For instance, the different implementations of the java.util.concurrent.BlockingQueue interface are all blocking data structures. If a thread attempts to insert an element into a BlockingQueue and the queue does not have space, the inserting thread is blocked (suspended) until the BlockingQueue has space for the new element.
This diagram illustrates the behaviour of a blocking algorithm guarding a shared data structure:

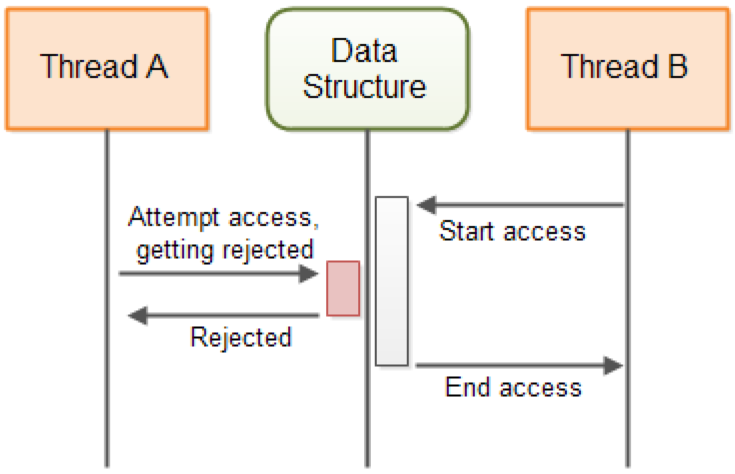


Non-blocking Concurrency Algorithms

A non-blocking concurrency algorithm is an algorithm which either:

- A: Performs the action requested by the thread - OR
- B: Notifies the requesting thread that the action could not be performed

Java contains several non-blocking data structures too. The AtomicBoolean, AtomicInteger, AtomicLong and AtomicReference are all examples of non-blocking data structures.
This diagram illustrates the behaviour of a non-blocking algorithm guarding a shared data structure:



Non-blocking vs Blocking Algorithms

The main difference between blocking and non-blocking algorithms lies in the second step of their behaviour as described in the above two sections. In other words, the difference lies in what the blocking and non-blocking algorithms do when the requested action cannot be performed:
Blocking algorithms block the thread until the requested action can be performed.
Non-blocking algorithms notify the thread requesting the action that the action cannot be performed.
With a blocking algorithm a thread may become blocked until it is possible to perform the requested action. Usually it will be the actions of another thread that makes it possible for the first thread to perform the requested action. If for some reason that other thread is suspended (blocked) somewhere else in the application, and thus cannot perform the action that makes the first thread's requested action possible, the first thread remains blocked - either indefinitely, or until the other thread finally performs the necessary action.
For instance, if a thread tries to insert an element into a full BlockingQueue the thread will block until another thread has taken an element from the BlockingQueue. If for some reason the thread that is supposed to take elements from the BlockingQueue is blocked (suspended) somewhere else in the application, the thread trying to insert the new element remains blocked - either indefinitely, or until the thread taking elements finally takes an element from the BlockingQueue.

Non-blocking Concurrent Data Structures

In a multithreaded system, threads usually communicate via some kind of data structure. Such data structures can be anything from simple variables to more advanced data structures like queues, maps, stacks etc. To facilitate correct, concurrent access to the data structures by multiple threads, the data structures must be guarded by some *concurrent algorithm*. The guarding algorithm is what makes the data structure a *concurrent data structure*.

If the algorithm guarding a concurrent data structure is blocking (uses thread suspension), it is said to be a *blocking algorithm*. The data structure is thus said to be a *blocking, concurrent data structure*.

If the algorithm guarding a concurrent data structure is non-blocking, it is said to be a *non-blocking algorithm*. The data structure is thus said to be a *non-blocking, concurrent data structure*.

Each concurrent data structure is designed to support a certain method of communication. Which concurrent data structure you can use thus depends on your communication needs. I will cover some non-blocking concurrent data structures in the following sections, and explain in what situations they can be used. The explanation of how these non-blocking data structures work should give you an idea about how non-blocking data structures can be designed and implemented.

Volatile Variables

Java volatile variables are variables that are always read directly from main memory. When a new value is assigned to a volatile variable the value is always written immediately to main memory. This guarantees that the latest value of a volatile variable is always visible to other threads running on other CPUs. Other threads will read the value of the volatile from main memory every time, instead of from e.g. the CPU cache of the CPU the threads are running on.

Volatile variables are non-blocking. The writing of a value to a volatile variable is an atomic operation. It cannot be interrupted. However, a read-update-write sequence performed on a volatile variable is not atomic. Thus, this code may still lead to race conditions if performed by more than one thread:

```
volatile myVar = 0;

...
int temp = myVar;
temp++;
myVar = temp;
```

First the value of the volatile variable myVar is read from main memory into a temp variable. Then the temp variable is incremented by 1. Then the value of the temp variable is assigned to the volatile myVar variable which means it will be written back to main memory.

If two threads execute this code and both of them read the value of myVar, add one to it and write the value back to main memory, then you risk that instead of 2 being added to the

myVar variable, only 1 will be added (e.g. both threads read the value 19, increment to 20, and write 20 back).
You might think you won't write code like above, but in practice the above code is equivalent to this:

```
myVar++;
```

When executed, the value of myVar is read into a CPU register or the local CPU cache, one is added, and then the value from the CPU register or CPU cache is written back to main memory.

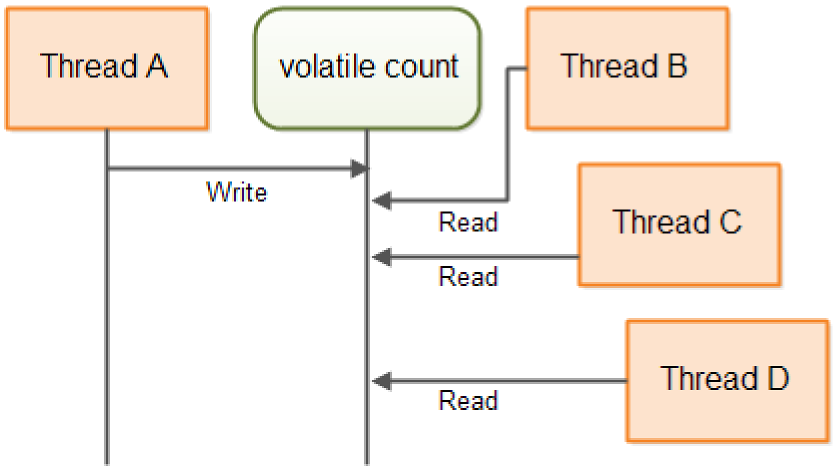
The Single Writer Case

In some cases you only have a single thread writing to a shared variable, and multiple threads reading the value of that variable. No race conditions can occur when only a single thread is updating a variable, no matter how many threads are reading it. Therefore, whenever you have only a single writer of a shared variable you can use a volatile variable.
The race conditions occur when multiple threads perform a read-update-write sequence of operations on a shared variable. If you only have one thread perform a read-update-write sequence of operations, and all other threads only perform a read operation, you have no race conditions.
Here is a single writer counter which does not use synchronization but is still concurrent:

```
public class SingleWriterCounter {

   private volatile long count = 0;

   /**
    * Only one thread may ever call this method,
    * or it will lead to race conditions.
    */
   public void inc() {
      this.count++;
   }


   /**
    * Many reading threads may call this method
    * @return
    */
   public long count() {
      return this.count;
   }
}
```

Multiple threads can access the same instance of this counter, as long as only one thread calls inc(). And I don't mean one thread at a time. I mean, only the same, single thread is ever allowed to call inc(). Multiple threads can call count(). This will not cause any race conditions.

This diagram illustrates how the threads would access the volatile count variable:



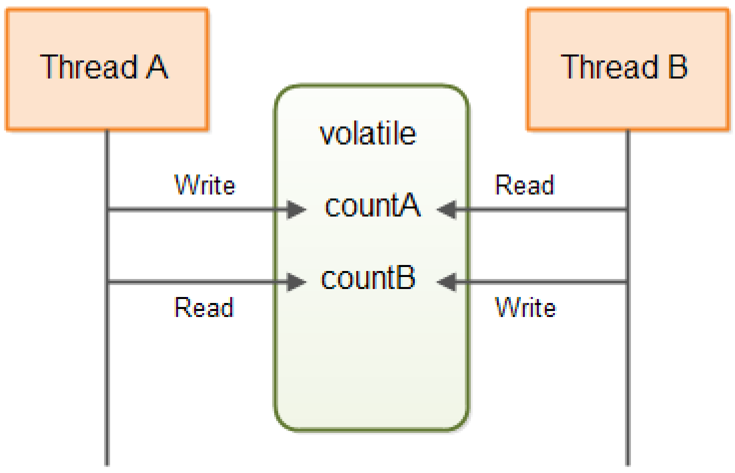
More Advanced Data Structures Based on Volatile Variables

It is possible to create data structures that use combinations of volatile variables, where each volatile variable is only written by a single thread, and read by multiple threads. Each volatile variable may be written by a different thread (but only one thread). Using such a data structure multiple threads may be able to send information to each other in a non-blocking way, using the volatile variables.

Here is a simple double writer counter class that shows how that could look:

```
public class DoubleWriterCounter {

    private volatile long countA = 0;
    private volatile long countB = 0;

    /**
     * Only one (and the same from thereon) thread may ever call this method,
     * or it will lead to race conditions.
     */
    public void incA() { this.countA++;  }


    /**
     * Only one (and the same from thereon) thread may ever call this method,
     * or it will lead to race conditions.
     */
    public void incB() { this.countB++;  }


    /**
     * Many reading threads may call this method
     */
    public long countA() { return this.countA; }


    /**
     * Many reading threads may call this method
     */
    public long countB() { return this.countB; }
}
```

As you can see, the DoubleWriterCounter now contains two volatile variables, and two pairs of incrementation and read methods. Only a single thread may ever call incA(), and only a single thread may ever call incB(). It can be different threads calling incA() and incB() though. Many threads are allowed to call countA() and countB(). This will not cause race conditions.
The DoubleWriterCounter can be used for e.g. two threads communicating. The two counts could be tasks produced and tasks consumed. This diagram shows two thread communicating via a data structure similar to the above:



The smart reader will recognize that you could have achieved the effect of the DoubleWriterCounter by using two SingleWriterCounter instances. You could even have used more threads and SingleWriterCounter instances if you needed to.

Optimistic Locking With Compare and Swap

If you really need more than one thread to write to the same, shared variable, a volatile variable will not be sufficient. You will need some kind of exclusive access to the variable. This is how such exclusive access could look using a synchronized block in Java:

```
public class SynchronizedCounter {
    long count = 0;

    public void inc() {
        synchronized(this) {
            count++;
        }
    }

    public long count() {
        synchronized(this) {
            return this.count;
        }
    }
}
```

Notice how the inc() and count() methods both contain a synchronized block. This is what we want to avoid - synchronized blocks and wait() - notify() calls etc.
Instead of the two synchronized blocks we can use one of Java's atomic variables. In this case the AtomicLong. Here is how the same counter class could look using an AtomicLong instead:

```
import java.util.concurrent.atomic.AtomicLong;

public class AtomicCounter {
    private AtomicLong count = new AtomicLong(0);

    public void inc() {
        boolean updated = false;
        while(!updated){
            long prevCount = this.count.get();
            updated = this.count.compareAndSet(prevCount, prevCount + 1);
        }
    }

    public long count() {
        return this.count.get();
    }
}
```

This version is just as thread-safe as the previous version. What is interesting about this version is the implementation of the inc() method. The inc() method no longer contains a synchronized block. Instead it contains these lines:

```
boolean updated = false;
while(!updated){
    long prevCount = this.count.get();
    updated = this.count.compareAndSet(prevCount, prevCount + 1);
}
```

These lines are not an atomic operation. That means, that it is possible for two different threads to call the inc() method and execute the long prevCount = this.count.get() statement,

and thus both obtain the previous count for the counter. Yet, the above code does not contain any race conditions.
The secret is in the second of the two lines inside the while loop. The compareAndSet() method call is an atomic operation. It compares the internal value of the AtomicLong to an expected value, and if the two values are equal, sets a new internal value for the AtomicLong. The compareAndSet() method is typically supported by compare-and-swap instructions directly in the CPU. Therefore no synchronization is necessary, and no thread suspension is necessary. This saves the thread suspension overhead.
Imagine that the internal value of the AtomicLong is 20. Then two threads read that value, and both tries to call compareAndSet(20, 20 + 1). Since compareAndSet() is an atomic operation, the threads will execute this method sequentially (one at a time).
The first thread will compare the expected value of 20 (the previous value of the counter) to the internal value of the AtomicLong. Since the two values are equal, the AtomicLong will update its internal value to 21 (20 + 1). The updated variable will be set to true and the while loop will stop.
Now the second thread calls compareAndSet(20, 20 + 1). Since the internal value of the AtomicLong is no longer 20, this call will fail. The internal value of the AtomicLong will not be set to 21. The updated variable will be set to false, and the thread will spin one more time around the while loop. This time it will read the value 21 and attempt to update it to 22. If no other thread has called inc() in the meantime, the second iteration will succeed in updating the AtomicLong to 22.

Why is it Called Optimistic Locking?

The code shown in the previous section is called *optimistic locking*. Optimistic locking is different from traditional locking, sometimes also called pessimistic locking. Traditional locking blocks the access to the shared memory with a synchronized block or a lock of some kind. A synchronized block or lock may result in threads being suspended.
Optimistic locking allows all threads to create a copy of the shared memory without any blocking. The threads may then make modifications to their copy, and attempt to write their modified version back into the shared memory. If no other thread has made any modifications to the shared memory, the compare-and-swap operation allows the thread to write its changes to shared memory. If another thread has already changed the shared memory, the thread will have to obtain a new copy, make its changes and attempt to write them to shared memory again.
The reason this is called optimistic locking is that threads obtain a copy of the data they want to change and apply their changes, under the optimistic assumption that no other thread will have made changes to the shared memory in the meantime. When this optimistic assumption holds true, the thread just managed to update shared memory without locking. When this assumption is false, the work was wasted, but still no locking was applied.
Optimistic locking tends to work best with low to medium contention on the shared memory. If the content is very high on the shared memory, threads will waste a lot of CPU cycles copying and modifying the shared memory only to fail writing the changes back to the shared memory. But, if you have a lot of content on shared memory, you should anyways consider redesigning your code to lower the contention.

Optimistic Locking is Non-blocking

The optimistic locking mechanism I have shown here is non-blocking. If a thread obtains a copy of the shared memory and gets blocked (for whatever reason) while trying to modify it, no other threads are blocked from accessing the shared memory.
With a traditional lock / unlock paradigm, when a thread locks a lock - that lock remains locked for all other threads until the thread owning the lock unlocks it again. If the thread that locked the lock is blocked somewhere else, that lock remains locked for a very long time - maybe even indefinitely.

Non-swappable Data Structures

The simple compare-and-swap optimistic locking works for shared data structures where the whole data structure can be swapped (exchanged) with a new data structure in a single compare-and-swap operation. Swapping the whole data structure with a modified copy may not always be possible or feasible, though.
Imagine if the shared data structure is a queue. Each thread trying to either insert or take elements from the queue would have to copy the whole queue and make the desired modifications to the copy. This could be achieved via an AtomicReference. Copy the reference, copy and modify the queue, and try to swap the reference pointed to in the AtomicReference to the newly created queue.
However, a big data structure may require a lot of memory and CPU cycles to copy. This will make your application spend a lot more memory, and waste a lot of time on the copying. This will impact the performance of your application, especially if contention on the data structure is high. Furthermore, the longer time it takes for a thread to copy and modify the data structure, the bigger the probability is that some other thread will have modified the data structure in between. As you know, if another thread has modified the shared data structure since it was copied, all other threads have to restart their copy-modify operations. This will increase the impact on performance and memory consumption even more.
The next section will explain a method to implement non-blocking data structures which can be updated concurrently, not just copied and modified.

Sharing Intended Modifications

Instead of copying and modifying the whole shared data structure, a thread can share its *intended modification* of the shared data structure. The process for a thread wanting to make a modification to the shared data structure then becomes:

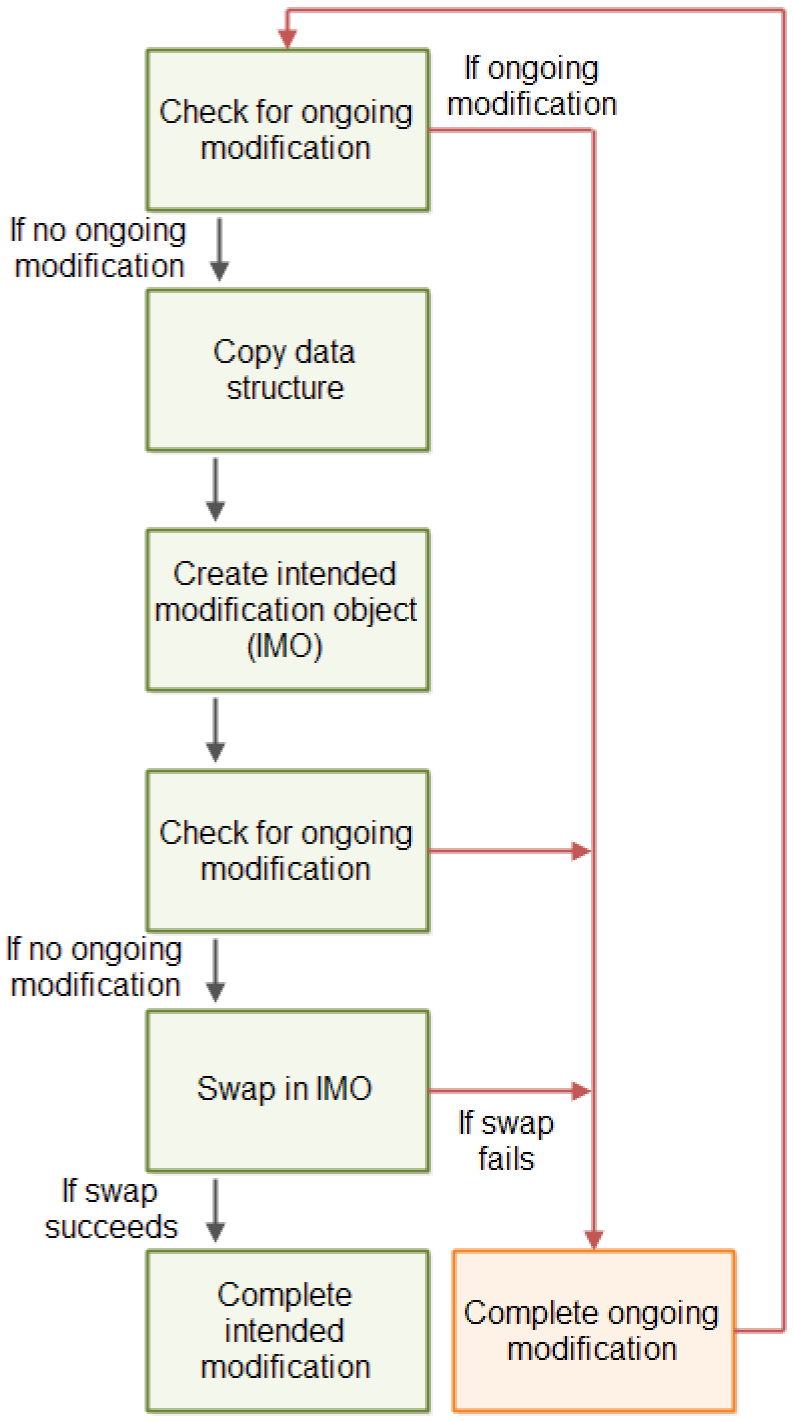
1. Check if another thread has submitted an intended modification to the data structure.
2. If no other thread has submitted an intended modification, create an intended modification (represented by an object) and submit that intended modification to the data structure (using a compare-and-swap operation).
3. Carry out the modification of the shared data structure.
4. Remove the reference to the intended modification to signal to other threads that the intended modification has been carried out.

As you can see, the second step can block other threads from submitting an intended modification. Thus, the second step effectively works as a lock of the shared data structure.

If one thread successfully submits an intended modification, no other thread can submit an intended modification until the first intended modification is carried out.
If a thread submits an intended modification and then gets blocked doing some other work, the shared data structure is effectively locked. The shared data structure does not directly block the other threads using the data structure. The other threads can detect that they cannot submit an intended modification and decide to something else. Obviously, we need to fix that.

Completable Intended Modifications

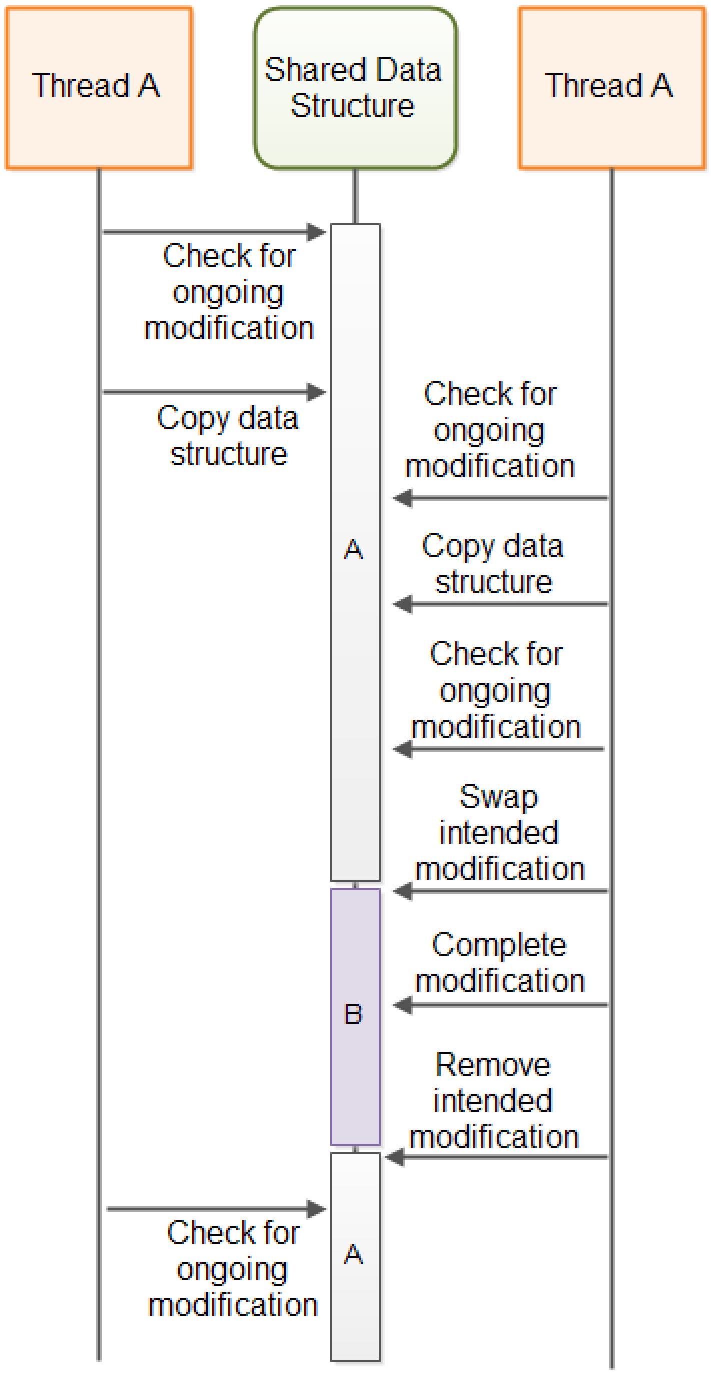
To avoid that a submitted intended modification can lock the shared data structure, a submitted intended modification object must contain enough information for another thread to complete the modification. Thus, if the thread submitting the intended modification never completes the modification, another thread can complete the modification on its behalf, and keep the shared data structure available for other threads to use.
Here is a diagram illustrating the blueprint of the above described non-blocking algorithm:


Check for ongoing modification
If ongoing modification
If no ongoing modification
Copy data structure
Create intended modification object (IMO)
Check for ongoing modification
If no ongoing modification
Swap in IMO
If swap fails
If swap succeeds
Complete intended modification
Complete ongoing modification

The modifications must be carried out as one or more compare-and-swap operations. Thus, if two threads try to complete the intended modification, only one thread will be able to carry out any of the compare-and-swap operations. As soon as a compare-and-swap operation has been completed, further attempts to complete that compare-and-swap operation will fail.

The A-B-A Problem

The above illustrated algorithm can suffer from the A-B-A problem. The A-B-A problem refers to the situation where a variable is changed from A to B and then back to A again. For another thread it is thus not possible to detect that the variable was indeed changed.
If thread A checks for ongoing updates, copies data and is suspended by the thread scheduler, a thread B may be able to access the shared data structure in the meanwhile. If thread B performs a full update of the data structure, and removes its intended modification, it will look to thread A as if no modification has taken place since it copied the data structure. However, a modification did take place. When thread A continues to perform its update based on its now out-of-date copy of the data structure, the data structure will have thread B's modification undone.
The following diagram illustrates A-B-A problem from the above situation:


Thread A
Shared Data Structure
Thread A
Check for ongoing modification
Copy data structure
Check for ongoing modification
A
Copy data structure
Check for ongoing modification
Swap intended modification
Complete modification
B
Remove intended modification
Check for ongoing modification
A

A-B-A Solutions

A common solution to the A-B-A problem is to not just swap a pointer to an intended modification object, but to combine the pointer with a counter, and swap pointer + counter using a single compare-and-swap operation. This is possible in languages that support pointers like C and C++. Thus, even if the current modification pointer is set back to point to "no ongoing modification", the counter part of the pointer + counter will have been incremented, making the update visible to other threads.

In Java you cannot merge a reference and a counter together into a single variable. Instead Java provides the AtomicStampedReference class which can swap a reference and a stamp atomically using a compare-and-swap operation.

A Non-blocking Algorithm Template

Below is a code template intended to give you an idea about how non-blocking algorithms are implemented. The template is based on the descriptions given earlier in this tutorial. NOTE: I am not an expert in non-blocking algorithms, so the template below probably has some errors. Do not base your own non-blocking algorithm implementation on my template. The template is only intended to give you an idea of how the code for a non-blocking algorithm could look. If you want to implement your own non-blocking algorithms, study some real, working non-blocking algorithm implementations first, to learn more about how they are implemented in practice.

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicStampedReference;

public class NonblockingTemplate {

  public static class IntendedModification {
    public AtomicBoolean completed =
        new AtomicBoolean(false);
  }

  private AtomicStampedReference<IntendedModification>
    ongoingMod =
      new AtomicStampedReference<IntendedModification>(null, 0);

  //declare the state of the data structure here.


  public void modify() {
    while(!attemptModifyASR());
  }

  public boolean attemptModifyASR(){

    boolean modified = false;

    IntendedModification currentlyOngoingMod =
    ongoingMod.getReference();
    int stamp = ongoingMod.getStamp();

    if(currentlyOngoingMod == null){
      //copy data structure state - for use
      //in intended modification
```

```
            //prepare intended modification
            IntendedModification newMod =
            new IntendedModification();

            boolean modSubmitted =
               ongoingMod.compareAndSet(null, newMod, stamp, stamp + 1);

            if(modSubmitted){

               //complete modification via a series of compare-and-swap operations.
               //note: other threads may assist in completing the compare-and-swap
               // operations, so some CAS may fail

               modified = true;
            }

         } else {
            //attempt to complete ongoing modification, so the data structure is freed up
            //to allow access from this thread.

            modified = false;
         }

         return modified;
   }
}
```

Non-blocking Algorithms are Difficult to Implement

Non-blocking algorithms are hard to design and implement correctly. Before attempting to implement your own non-blocking algorithms, see if there is not someone who has already developed a non-blocking algorithm for your needs.
Java already comes with a few non-blocking implementations (e.g. ConcurrentLinkedQueue) and will most likely get more non-blocking algorithm implementations in future Java versions.
In addition to Java's built-in non-blocking data structures there are also some open source non-blocking data structures you can use. For instance, the LMAX Disrupter (a queue-like data structure), and the non-blocking HashMap from Cliff Click. See my Java concurrency references page for links to more resources.

The Benefit of Non-blocking Algorithms

There are several benefits of non-blocking algorithms compared to blocking algorithms. This section will describe these benefits.

Choice

The first benefit of non-blocking algorithms is, that threads are given a choice about what to do when their requested action cannot be performed. Instead of just being blocked, the request thread has a choice about what to do. Sometimes there is nothing a thread can do. In that case it can choose to block or wait itself, thus freeing up the CPU for other tasks. But at least the requesting thread is given a choice.
On a single CPU system perhaps it makes sense to suspend a thread that cannot perform a desired action, and let other threads which can perform their work run on the CPU. But even on a single CPU system blocking algorithms may lead to problems like deadlock, starvation and other concurrency problems.

No Deadlocks

The second benefit of non-blocking algorithms is, that the suspension of one thread cannot lead to the suspension of other threads. This means that deadlock cannot occur. Two threads cannot be blocked waiting for each other to release a lock they want. Since threads are not blocked when they cannot perform their requested action, they cannot be blocked waiting for each other. Non-blocking algorithms may still result in live lock, where two threads keep attempting some action, but keep being told that it is not possible (because of the actions of the other thread).

No Thread Suspension

Suspending and reactivating a thread is costly. Yes, the costs of suspension and reactivation has gone down over time as operating systems and thread libraries become more efficient. However, there is still a high price to pay for thread suspension and reactivation.
Whenever a thread is blocked it is suspended, thus incurring the overhead of thread suspension and reactivation. Since threads are not suspended by non-blocking algorithms, this overhead does not occur. This means that the CPUs can potentially spend more time performing actual business logic instead of context switching.
On a multi CPU system blocking algorithms can have more significant impact on the overall performance. A thread running on CPU A can be blocked waiting for a thread running on CPU B. This lowers the level of parallelism the application is capable of achieving. Of course, CPU A could just schedule another thread to run, but suspending and activating threads (context switches) are expensive. The less threads need to be suspended the better.

Reduced Thread Latency

Latency in this context means the time between a requested action becomes possible and the thread actually performs it. Since threads are not suspended in non-blocking algorithms they do not have to pay the expensive, slow reactivation overhead. That means that when a requested action becomes possible threads can respond faster and thus reduce their response latency.
The non-blocking algorithms often obtain the lower latency by busy-waiting until the requested action becomes possible. Of course, in a system with high thread contention on the non-blocking data structure, CPUs may end up burning a lot of cycles during these busy

waits. This is a thing to keep in mind. Non-blocking algorithms may not be the best if your data structure has high thread contention. However, there are often ways do redesign your application to have less thread contention.

Amdahl's Law

Amdahl's law can be used to calculate how much a computation can be sped up by running part of it in parallel. Amdahl's law is named after Gene Amdahl who presented the law in 1967. Most developers working with parallel or concurrent systems have an intuitive feel for potential speedup, even without knowing Amdahl's law. Regardless, Amdahl's law may still be useful to know.
I will first explain Amdahl's law mathematically, and then proceed to illustrate Amdahl's law using diagrams.

Amdahl's Law Defined

A program (or algorithm) which can be parallelized can be split up into two parts:

- A part which cannot be parallelized
- A part which can be parallelized

Imagine a program that processes files from disk. A small part of that program may scan the directory and create a list of files internally in memory. After that, each file is passed to a separate thread for processing. The part that scans the directory and creates the file list cannot be parallelized, but processing the files can.
The total time taken to execute the program in serial (not in parallel) is called T. The time T includes the time of both the non-parallelizable and parallelizable parts. The non-parallelizable part is called B. The parallizable part is referred to as T - B. The following list sums up these definitions:

- T = Total time of serial execution
- B = Total time of non-parallizable part
- T - B = Total time of parallizable part (when executed serially, not in parallel)

From this follows that:
T = B + (T-B)

It may look a bit strange at first that the parallelizable part of the program does not have its own symbol in the equation. However, since the parallelizable part of the equation can be expressed using the total time T and B (the non-parallelizable part), the equation has actually been reduced conceptually, meaning that it contains less different variables in this form.
It is the parallelizable part, T - B, that can be sped up by executing it in parallel. How much it can be sped up depends on how many threads or CPUs you apply to execute it. The number of threads or CPUs is called N. The fastest the the parallelizable part can be executed is thus:
(T - B) / N

Another way to write this is:
(1/N) * (T - B)

Wikipedia uses this version in case you read about Amdahl's law there.
According to Amdahl's law, the total execution time of the program when the parallelizable part is executed using N threads or CPUs is thus:
T(N) = B + (T - B) / N

T(N) means total execution with with a parallelization factor of N. Thus, T could be written T(1) , meaning the total execution time with a parallelization factor of 1. Using T(1) instead of T, Amdahl's law looks like this:
T(N) = B + (T(1) - B) / N

It still means the same though.

A Calculation Example

To better understand Amdahl's law, let's go through a calculation example. The total time to execute a program is set to 1. The non-parallelizable part of the programs is 40% which out of a total time of 1 is equal to 0.4 . The parallelizable part is thus equal to 1 - 0.4 = 0.6 .
The execution time of the program with a parallelization factor of 2 (2 threads or CPUs executing the parallelizable part, so N is 2) would be:

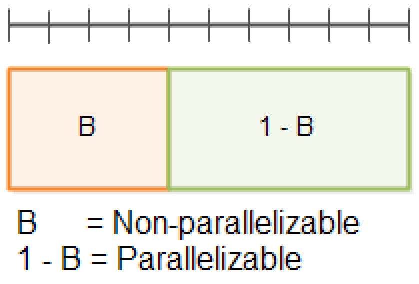
```
T(2) = 0.4 + ( 1 - 0.4 ) / 2
     = 0.4 + 0.6 / 2
     = 0.4 + 0.3
     = 0.7
```

Making the same calculation with a parallelization factor of 5 instead of 2 would look like this:

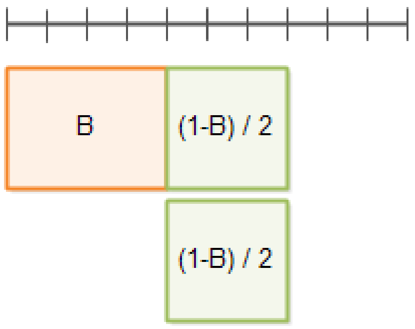
```
T(5) = 0.4 + ( 1 - 0.4 ) / 5
     = 0.4 + 0.6 / 5
     = 0.4 + 0.12
     = 0.52
```

Amdahl's Law Illustrated

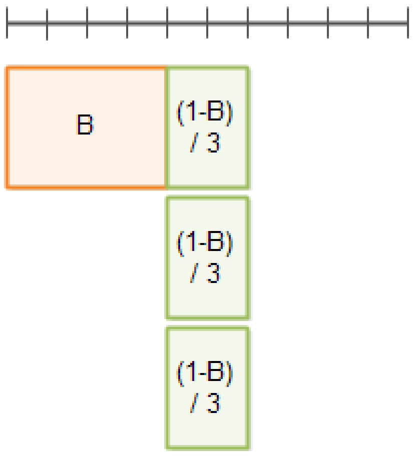
To better understand Amdahl's law I will try to illustrate how the law is derived.
First of all, a program can be broken up into a non-parallelizable part B, and a parallelizable part 1-B, as illustrated by this diagram:



The line with the delimiters on at the top is the total time T(1).
Here you see the execution time with a parallelization factor of 2:



Here you see the execution time with a parallelization factor of 3:



Optimizing Algorithms

From Amdahl's law it follows naturally, that the parallelizable part can be executed faster by throwing hardware at it. More threads / CPUs. The non-parallelizable part, however, can only be executed faster by optimizing the code. Thus, you can increase the speed and parallelizability of your program by optimizing the non-parallelizable part. You might even change the algorithm to have a smaller non-parallelizable part in general, by moving some of the work into the parallelizable part (if possible).

Optimizing the Sequential Part

If you optimize the sequential part of a program you can also use Amdahl's law to calculate the execution time of the program after the optimization. If the non-parallelizable part B is optimized by a factor of O, then Amdahl's law looks like this:
T(O,N) = B / O + (1 - B / O) / N

Remember, the non-parallelizable part of the program now takes B / O time, so the parallelizable part takes 1 - B / O time.
If B is 0.4, O is 2 and N is 5, then the calculation looks like this:
T(2,5) = 0.4 / 2 + (1 - 0.4 / 2) / 5
 = 0.2 + (1 - 0.4 / 2) / 5
 = 0.2 + (1 - 0.2) / 5
 = 0.2 + 0.8 / 5
 = 0.2 + 0.16
 = 0.36

Execution Time vs. Speedup

So far we have only used Amdahl's law to calculate the execution time of a program or algorithm after optimization or parallelization. We can also use Amdahl's law to calculate the *speedup*, meaning how much faster the new algorithm or program is than the old version.
If the time of the old version of the program or algorithm is T, then the speedup will be
Speedup = T / T(O,N)

We often set T to 1 just to calculate the execution time and speedup as a fraction of the old time. The equation then looks like this:
Speedup = 1 / T(O,N)

If we insert the Amdahl's law calculation instead of T(O,N), we get the following formula:
Speedup = 1 / (B / O + (1 - B / O) / N)

With B = 0.4, O = 2 and N = 5, the calculation becomes:
Speedup = 1 / (0.4 / 2 + (1 - 0.4 / 2) / 5)
 = 1 / (0.2 + (1 - 0.4 / 2) / 5)
 = 1 / (0.2 + (1 - 0.2) / 5)
 = 1 / (0.2 + 0.8 / 5)
 = 1 / (0.2 + 0.16)
 = 1 / 0.36
 = 2.77777 ...

That means, that if you optimize the non-parallelizable (sequential) part by a factor of 2, and paralellize the parallelizable part by a factor of 5, the new optimized version of the program or algorithm would run a maximum of 2.77777 times faster than the old version.

Measure, Don't Just Calculate

While Amdahl's law enables you to calculate the theoretic speedup of parallelization of an algorithm, don't rely too heavily on such calculations. In practice, many other factors may come into play when you optimize or parallelize an algorithm.

The speed of memory, CPU cache memory, disks, network cards etc. (if disk or network are used) may be a limiting factor too. If a new version of the algorithm is parallelized, but leads to a lot more CPU cache misses, you may not even get the desired x N speedup of using x N CPUs. The same is true if you end up saturating the memory bus, disk or network card or network connection.

My recommendation would be to use Amdahl's law to get an idea about where to optimize, but use a measurement to determine the real speedup of the optimization. Remember, sometimes a highly serialized sequential (single CPU) algorithm may outperform a parallel algorithm, simply because the sequential version has no coordination overhead (breaking down work and building the total again), and because a single CPU algorithm may conform better with how the underlying hardware works (CPU pipelines, CPU cache etc).