

# Threading in Java and C#: A Focused Language Comparison

Shannon Hardt

## Overview

Both Java and .NET have built-in support for threads. In Java, this support is in three main forms: a mutex variable is associated with every object; there is a set of methods defined on Object that support coordination of threads; and a set of classes in java.lang that allows programmers to create and manage threads. Similarly, in .NET there is a Monitor (or lock) associated with every object instance. The System.Threading namespace contains classes that allow thread creation and manipulation.

While the APIs available to programmers are constant, the underlying implementation of threads at the operating system level is not specified in the run time specifications and may be different depending on the run time implementation of Java or .NET that is being used. In the standard implementations on Windows, user threads created in Java or .NET correspond 1 to 1 to user threads at the operating system level. The following diagram illustrates the relationship between user threads, operating system threads, and processes in Windows.

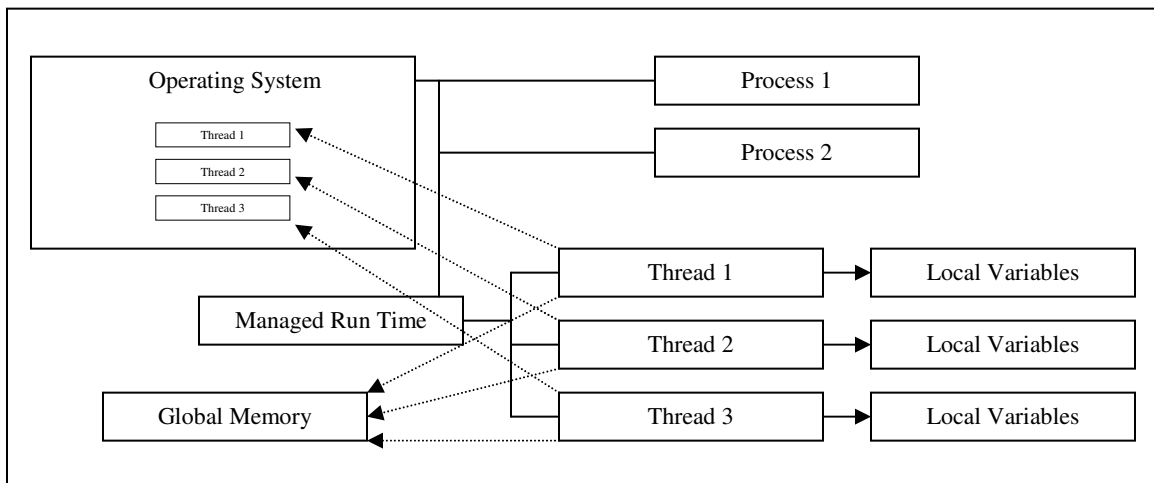


Figure 1: Threading Paradigm

## Creating & Running Threads

There are two main techniques for creating a thread in Java: subclass `java.lang.Thread` or implement the `java.lang.Runnable` interface. In both cases, you must implement a method with the following signature: `public void run()`. When you create the thread and tell it to start, the `run` method will be executed. A simple example is provided below.

```
public class Counter implements Runnable {
    private int count;
    public Counter(int val) { this.count = val; }
    public void run() {
        for(int i=0; i < count; i++)
            System.out.println("Hello World");
    }
    public static void main(String args[]) {
        Counter c = new Counter(10);
        Thread t = new Thread(c);
        t.start();
    }
}
```

In this example, when `main` is executed, it will create a new thread of control using `Counter`, which when started, will print "Hello World" ten times and then exit.

In C#, the pattern is similar. However, instead of implementing an interface or subclassing `Thread`, you use a `ThreadStart` delegate. The delegate is then used to instantiate the `Thread` object. The `ThreadStart` delegate requires a void method which takes no parameters. The previous example implemented in C# is provided below.

```
using System;
using System.Threading;
namespace SimpleThreadExample {
    class Counter {
        private int count;
        public Counter(int val) { this.count = val; }
        public void DoCount() {
            for(int i=0; i < count; i++)
                System.Console.WriteLine("Hello World");
        }
        [STAThread]
        static void Main(String[] args) {
            Counter c = new Counter(10);
            Thread t = new Thread(new ThreadStart(c.DoCount));
            t.start();
        }
    }
}
```

An argument could be made in favor of the C# model, as any method that returns void and accepts no arguments could be run as a new thread without changing the class itself. However, this could potentially lead to problems if a method that was not intended to be thread-safe was used to generate a new thread. Of course, this would not be due to a problem with the language, just its improper use. With Java, implementing `Runnable` should serve as a reminder to implement

---

thread-safe code and should be an indication that the code was intended to be run as a separate thread.

When threads are created in the default manner, they are created as foreground threads in both Java and .NET. Foreground threads will keep the execution environment alive even if the main thread of control is finished. It is also possible to create background (or daemon) threads in Java and .NET. Background threads will not keep the managed execution environment alive if all other foreground threads have terminated. In Java, a thread is marked as background using the `Thread.setDaemon(boolean)` method, while in .NET, there is a boolean property that can be set: `Thread.IsBackground`.

## Thread Life Cycles

A thread progresses through several states during its life. In Java, the states are New, Active, Inactive, Suspended, and Dead. A thread is New if it exists but hasn't done anything yet. A thread is Active if it is running, it is in the middle of performing operations, and it actually occupies a processor at the current time. A thread is Inactive if it is running, it is in the middle of performing operations, but it does not actually occupy a processor at the current time. A thread is Suspended if it cannot simply start processing. That is, it isn't running and wouldn't be able to run even if it were given some CPU time. A thread is Dead if it has no further operations to perform (the run method has terminated). The life cycle is depicted below.

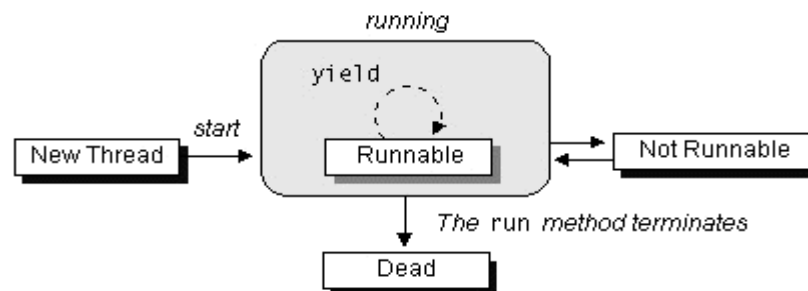


Figure 2: Thread Life Cycle (from Java Tutorial)

In .NET, the thread states are Unstarted, Running, WaitSleepJoin, SuspendRequested, Suspended, AbortRequested, and Stopped. A thread is Unstarted if the `Thread.Start` method has not been invoked on the thread. A thread is Running if the thread has been started, it is not blocked, and there is no pending `ThreadAbortException`. A thread is WaitSleepJoin if the thread is blocked as a result of a call to `Wait`, `Sleep`, or `Join`. A thread is SuspendRequested if the thread is being requested to suspend. A thread is Suspended if the thread has been suspended. A thread is AbortRequested if the `Thread.Abort` method has been invoked on the thread, but the thread has not yet received the pending `ThreadAbortException`. A thread is Stopped if the thread has finished execution.

There are more thread states in .NET as opposed to Java. This is to allow the proper execution of suspend, resume and stop requests, which were deprecated in Java (more about that later).

## Managing Threads

Once a thread has been created and started, a common requirement is for one thread to wait until another has finished processing before continuing its execution. This is handled by the `join` method. In both Java and C#, the `Thread` class has 3 `join` methods: an unconditional `join` which will cause the joined thread to wait indefinitely; a conditional `join` that will wait a specified number of milliseconds before continuing execution; and a conditional `join` that will wait a specified number of milliseconds and nanoseconds before continuing execution. I have modified the previous example to illustrate `join`.

```
public static void main(String args[]) {
    Counter c = new Counter(100);
    Thread t = new Thread(c);
    t.start();

    // can start other threads, do some processing, etc.
    System.out.println("Before joining threads");

    t.join(); // execution will stop here until Thread t has
              // completed, at which point this thread will continue
    System.out.println("After joining threads");
}
```

The string “Before joining threads” will be printed somewhere in the middle of Thread `t`’s output, assuming Thread `t` hasn’t completed its processing before reaching the print statement. However, the string “After joining threads” will only be printed after Thread `t` has completed its processing. The example above is in Java, but the semantics are the same in C# (except `Join` is capitalized).

The `Thread` class also has `suspend` and `resume` methods. In Java, these methods have been deprecated (as of Java 2) because they can leave objects in inconsistent states. In .NET, this is not the case. Invoking `Suspend` on a thread in .NET does not necessarily suspend the thread immediately – the runtime allows the thread to run until it reaches a point where it may be safely suspended. `Resume` allows the thread to continue execution from the point at which it was suspended. However, a suspended thread still maintains any locks it has acquired, so suspending a thread could lead to deadlock. It is considered better design in both Java and C# to have a thread wait or sleep at specific points in its execution (when it is using no critical resources) rather than having a different thread call `suspend/resume`.

Similarly, there is a `stop` method in Java which will immediately stop an executing thread, throwing an instance of `ThreadDeath`. This method has also been deprecated, since it could leave objects and resources in an inconsistent state. In .NET, there is `Thread.Abort`. This method is used to stop a thread permanently, although it is not guaranteed to be immediate. `Thread.Abort` causes a `ThreadAbortException` to be raised in the thread, which can be caught and handled within the thread (for cleaning up resources, etc). However, although you can catch the exception, you cannot suppress it – it will automatically be rethrown once your catch and finally blocks have executed. It is considered better design to have your thread determine when it should terminate itself. One common way to do this would be to monitor a “flag” variable that would indicate to the thread that it should terminate. For example:

---

## Threading in Java and C#: A Focused Language Comparison

Shannon Hardt

```
public class HelloWorld implements Runnable {
    private volatile boolean continueRunning = true;
    public void stopRunning() { this.continueRunning = false; }
    public void run() {
        while ( continueRunning ) {
            System.out.println("Hello World!");
            Thread.sleep(1000);
        }
    }
    public static void main(String args[]) {
        Counter c = new Counter(10);
        Thread t = new Thread(c);
        t.start();
        // when ready for Thread t to stop executing
        t.stopRunning();
    }
}
```

When `continueRunning` is set to false, the thread will exit the while loop and terminate gracefully. One potential pitfall here is avoided by using the Java keyword `volatile`. Whenever a volatile attribute is accessed or changed, its value will be refreshed from/written to main memory (as opposed to being read from the local thread's cache). This prevents the possibility that the thread will not be able to "see" that the value of the variable has been changed by a different thread and, hence, never stop running. Another alternative would be to create synchronized methods to access/modify the `continueRunning` attribute (more on synchronization later).

In Java, `Thread.yield()` will stop executing the current thread, giving up its remaining time slice, and allow another thread to execute. To accomplish this in .NET, call `Thread.Sleep(0)`. If you call `Thread.Sleep(value)` with a positive value, it will cause the currently executing thread to temporarily halt execution and wait for value milliseconds before continuing execution. This behavior is the same in Java's `Thread.sleep()` method.

Java defines a few methods on the `Object` class to manage threads: `wait()`, `notify()`, and `notifyAll()`. The `wait()` method is overloaded to accept no arguments, a timeout in milliseconds, and a timeout in milliseconds and nanoseconds. A call to this method causes the current thread to wait until another thread invokes the `notify()` or the `notifyAll()` method for the object. The thread must own the object's monitor in order to call `wait()`. The current thread will place itself in the wait set for this object and then release all synchronization claims on this object. The thread becomes disabled for thread scheduling purposes and lies dormant until one of four things happens: another thread invokes the `notify` method for this object and the thread happens to be arbitrarily chosen as the thread to be awakened; another thread invokes `notifyAll` for this object; another thread interrupts the thread; or the specified amount of time has elapsed (will wait indefinitely if no arguments provided). The corresponding .NET versions are `Monitor.Wait(Object)`, `Monitor.Pulse(Object)`, and `Monitor.PulseAll(Object)`. The code below illustrates their use.

## Threading in Java and C#: A Focused Language Comparison

Shannon Hardt

```
// Java code using wait and notify
synchronized ( this ) {
    while ( !readyToRun ) {
        this.notifyAll(); // wakes any waiting threads
        this.wait(); // relinquishes lock on this and
                    // waits for notify or notifyAll
    }
    // do some work
}

// Corresponding C# code
lock( this ) {
    while ( !readyToRun ) {
        Monitor.PulseAll(this);
        Monitor.Wait(this);
    }
}
```

In the code above, the `notifyAll` and `PulseAll` methods will wake any threads waiting on this and allow them to contend for the monitor associated with this. All of the threads wakened will be blocked until the `wait` method call, which relinquishes the current thread's lock and sets the current thread into the wait set for the object. The newly awoken threads will then contend for the object's lock, one will be granted the lock, and all others will re-enter the wait set. The semantics are the same for Java and .NET.

A facility specific to .NET is the `ThreadPool` class. There is no corresponding functionality included in Java – you would have to roll your own pool. The .NET `ThreadPool` can be used to make much more efficient use of multiple threads. Using thread pooling provides a pool of worker threads that are managed by the system – it enables the system to optimize for better throughput not only for this process but also with respect to other processes on the computer. `ThreadPool` is used by calling `ThreadPool.QueueUserWorkItem` and passing a `WaitCallback` delegate wrapping the method that you want to add to the queue. There is only one `ThreadPool` object per process. One thread monitors all tasks that have been queued and, when a task is complete, a thread from the pool executes the corresponding callback method. There is no way to cancel a work item after it has been queued.

## Synchronization

When multiple threads can access and modify shared data, it is important that only one thread have access to the data at a time. Otherwise, one thread might interrupt what another thread is doing, and the data could be left in an invalid or inconsistent state. This is called synchronizing access to a resource. In Java, a mutex is associated with every object instance, including class objects that are instances of `java.lang.Class`. A mutex, short for “mutual exclusion”, is a locking mechanism guaranteed to be atomic. In other words, only one thread can access a mutex at a time. To request a lock, use the `synchronized` keyword. You create a synchronized block by using the keyword `synchronized` in conjunction with a particular object instance.

---

## Threading in Java and C#: A Focused Language Comparison

Shannon Hardt

```
// ...
Object foo;
// ...
synchronized(foo) {
    // processing that occurs once the lock is granted
}
```

This will guarantee that only one thread will be executing the synchronized code at a time. Methods can also have a synchronized modifier, which will obtain the mutex of the this object before the method is executed. C# does not have a synchronized keyword. In C#, you can explicitly request a lock on an object by using `Monitor.Enter(Object)` and `Monitor.Exit(Object)`. When calls are nested, you must be careful to invoke `Exit` as many times as you invoke `Enter` because a count is maintained. There is a convenience method `lock(Object)` which basically wraps the associated code block with a try-catch-finally block using `Monitor.Enter` and `Monitor.Exit`. The following two code examples are functionally the same:

```
public void SynchronizedMethod1() {
    try {
        Monitor.Enter(this);
        // do some work
    }
    finally {
        Monitor.Exit(this);
    }
}

public void SynchronizedMethod2() {
    lock( this ) {
        // do some work
    }
}
```

The `Monitor.TryEnter(Object, milliseconds)` is another method in C# that will attempt to acquire a lock on the object, but if it still has not successfully obtained the lock after milliseconds time, it will return false.

.NET distinguishes between a monitor and a mutex: they are similar in that only one thread may own it at any given point in time. However, the set of threads that may own a mutex is not restricted to a single process – any thread in the system can own it. Another difference is that you can wait on multiple mutexes. If you need to acquire one of a set of mutexes, you can use `WaitHandle.WaitAny`. If you need to own all of a set of mutexes, you can use `WaitHandle.WaitAll`. Once finished with the mutex, `ReleaseMutex` releases it. The `WaitHandle` class encapsulates Win32 synchronization handles, from which `Mutex` is derived.

Another facility available in .NET with no corresponding Java functionality is Interlocked access. The Interlocked methods `CompareExchange`, `Decrement`, `Exchange`, and `Increment` provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The `Increment` and `Decrement` functions combine the operations of incrementing or decrementing the variable and checking the resulting value into one atomic operation. The `Exchange` function atomically exchanges the values of the specified variables. The `CompareExchange` function

---

## Threading in Java and C#: A Focused Language Comparison

Shannon Hardt

combines two operations: comparing two values and storing a third value in one of the variables, based on the outcome of the comparison.

In .NET, there is another special locking facility that allows multiple threads to read a resource concurrently, but requires a thread to wait for an exclusive lock in order to write to the resource: `ReaderWriterLock`. In cases where most accesses are reads, and writes are infrequent and of short duration, `ReaderWriterLock` provides better throughput than a simple one-at-a-time lock such as `Monitor`. There is no corresponding functionality in Java.

Both Java and .NET provide a higher level of abstraction for simple threaded applications where the programmer does not have to be concerned with the lower-level details of thread management. In Java, the `java.util.Timer` and `java.util.TimerTask` classes can be used. The `Timer` class allows scheduling of tasks for future execution. Tasks may be scheduled for one-time execution or for repeated execution at regular intervals. Corresponding to each `Timer` object is a single background thread that is used to execute all of the timer's tasks, sequentially. After the last live reference to a `Timer` object goes away and all outstanding tasks have completed execution, the timer's task execution thread terminates gracefully. By default, the task execution thread does not run as a daemon thread, so it is capable of keeping an application from terminating. If a caller wants to terminate a timer's task execution thread rapidly, the caller should invoke the timer's `cancel` method. `TimerTask` subclasses are the tasks executed by the `Timer`.

In .NET, the `System.Threading.Timer` objects are lightweight objects that enable you to specify a delegate to be called at a specified time. A thread in the thread pool performs the wait operation. To use it, create a `Timer` passing a `TimerCallback` delegate to the callback method, an object representing state that will be passed to the callback, an initial raise time, and a time representing the period between callback invocations. To cancel a pending timer, call the `Timer.Dispose` function.

The main difference between the Java and C# `Timer` classes is that in Java, `TimerTask` subclasses are used to code the unit of work, whereas in C#, method delegates are used. Example code using the `Timer` classes is provided in the Appendix.

## Concluding Remarks

Both Java and .NET have threading facilities built into the language. The concepts between the two are very similar – both provide for thread creation and manipulation, and both provide higher-level abstractions of the low-level thread manipulation APIs. However, .NET provides more fine-grained control over threads and allows for better performance in certain situations by using facilities such as `Interlocked`, `ReaderWriterLock`, and `ThreadPool`. If you are familiar with threading in one language, it should be easy to learn the threading capabilities of the other.

---

## References

*.NET Development for Java Programmers* by Gibbons. Apress, 2002.

*Effective Java Programming Language Guide* by Bloch. Addison Wesley, 2003.

*Java Programming Language 3<sup>rd</sup> Edition* by Arnold, Gosling, Holmes. Addison Wesley, 2000.

*Java RMI* by Grosso. O'Reilly, 2002.

*Java Tutorial: Threads*

<http://java.sun.com/docs/books/tutorial/essential/threads/>

*Java Threads 2<sup>nd</sup> Edition* by Oaks, Wong. O'Reilly, 1999.

*MSDN Online: .NET Framework Developer's Guide: Threading*

<http://msdn.microsoft.com/library/en-us/cpguide/html/cpconthreading.asp>

*Programming C# 3<sup>rd</sup> Edition* by Liberty. O'Reilly, 2003.

---

## Appendix: Timer Classes

```
/* Timer classes in Java */
import java.util.Timer;
import java.util.TimerTask;

public class Reminder {
    Timer timer;

    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds * 1000);
    }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Time's up!");
            timer.cancel(); // terminates the timer thread
        }
    }

    public static void main(String args[]) {
        System.out.println("About to schedule task");
        new Reminder(5);
        System.out.println("Task scheduled");
    }
}

// OUTPUT:
// About to schedule task
// Task scheduled
// ... 5 seconds later ...
// Time's up!
```

---

## Threading in Java and C#: A Focused Language Comparison

Shannon Hardt

```
/* Timer class in C# */
using System;
using System.Threading;

public class Reminder {
    public ManualResetEvent timerevent;
    public Reminder(int seconds) {
        timerevent = new ManualResetEvent(false);
        Timer timer = new Timer(
            new TimerCallback(this.TimerMethod),
            null,
            TimeSpan.FromSeconds(seconds),
            TimeSpan.FromSeconds(seconds)
        );
    }

    public void TimerMethod(object state) {
        Console.WriteLine("Time's up!");
        timerevent.Set();
    }

    public static void Main() {
        Console.WriteLine("About to schedule task");
        new Reminder(5);
        Console.WriteLine("Task scheduled");
    }
}

// OUTPUT:
// About to schedule task
// Task scheduled
// ... 5 seconds later ...
// Time's up!
```

---