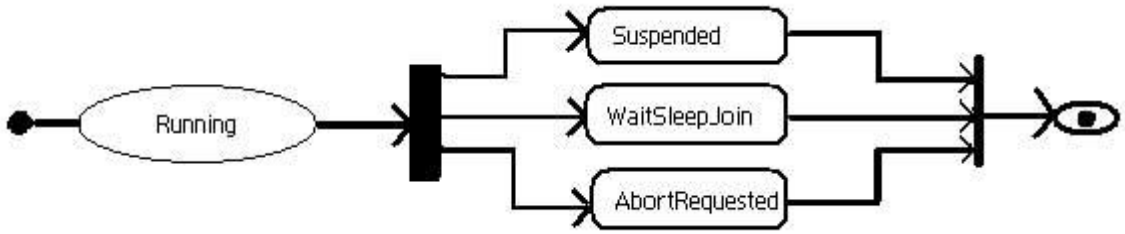
Threads in C#

During its lifecycle, each thread goes through a state. The following diagram illustrates various threads states. Thread always states in Unstrated state and the only transition is to start the thread and state changes to Running. A running thread has three transitions, Suspended, Sleep, and AbortRequested. More states are explained in the table below.

The **ThreadState** property returns the current state of a thread. You cannot set a thread's state using this property. A thread can be in more than one state at a time.



Description	State
A thread is created	Unstarted
Another thread calls the Thread.Start method on the new thread, and the call returns. The Start method does not return until the new thread has started running. There is no way to know at what point the new thread will start running, during the call to Start.	Running
The thread calls Sleep	WaitSleepJoin
The thread calls Wait on another object.	WaitSleepJoin
The thread calls Join on another thread.	WaitSleepJoin
Another thread calls Interrupt	Running
Another thread calls Suspend	SuspendRequested
The thread responds to a Suspend request.	Suspended
Another thread calls Resume	Running
Another thread calls Abort	AbortRequested
The thread responds to a Abort request.	Stopped
A thread is terminated.	Stopped
The thread state includes AbortRequested and the thread is now dead, but its state has not yet changed to Stopped.	Aborted
If thread is a background thread	Background

Join, Sleep and IsAlive

The **Join** method of Thread class in C# blocks the current thread and makes it wait until the child thread on which the Join method invoked completes its execution. There are three overloaded versions available for the Join Method in Thread class as shown below.

```
public void Join();
public void Join(int milSecTimeOut);
```

```
public void Join(TimeSpan timeOut);
```

The first version of the Join method which does not take any parameter will block the calling thread (i.e. the Parent thread) until the thread (child thread) completes its execution. In this case, the calling thread is going to wait for indefinitely time until the thread on which the Join Method invoked is completed.

The second version of the Join Method allows us to specify the time out. It means it will block the calling thread until the child thread terminates or the specified time elapses. This overloaded takes the time in milliseconds. This method returns true if the thread has terminated and returns false if the thread has not terminated after the amount of time specified by the milSecTimeOut parameter has elapsed.

The third overloaded version of this method is the same as the second overloaded version. The only difference is that here we need to use the TimeSpan to set the amount of time to wait for the thread to terminate.

The **Sleep** method of Thread class is used to suspend the current thread for the specified amount of time. There are two overloaded versions for the Sleep method in Thread class.

```
public static void Sleep (int millisecondsTimeout);
public static void Sleep (TimeSpan timeout);
```

IsAlive is the readonly property that gets a value indicating the execution status of the current thread. It will return true if this thread has been started and has not terminated normally or aborted; otherwise, false.

Example:

```
using System;
using System.Threading;
public class Example
{
   static Thread thread1, thread2;
   public static void Main()
   {
      thread1 = new Thread(ThreadProc);
      // Set the name of thread1
      thread1.Name = "Thread1";
      thread1.Start();
      // join thread1 with Main thread
      thread1.Join();
      thread2 = new Thread(ThreadProc);
      thread2.Name = "Thread2";
      thread2.Start();
      // Checking thread1 is terminated or not.
      if (thread1.IsAlive)
      {
            Console.WriteLine("Thread1 is still doing its work");
      }
      else
      {
            Console.WriteLine("Thread1 Completed its work");
      }
      Console.WriteLine(" Main thread has termminated.");
```

```
   }

   private static void ThreadProc()
   {
      Console.WriteLine("\nCurrent thread: {0}", Thread.CurrentThread.Name);
      if (Thread.CurrentThread.Name == "Thread1" &&
          thread2.ThreadState != ThreadState.Unstarted)
         if (thread2.Join(2000))
            Console.WriteLine("Thread2 has termminated.");
         else
            Console.WriteLine("The timeout has elapsed and Thread1 will
resume.");
      // suspend current thread executing this method for 4 sec.
      Thread.Sleep(4000);
      Console.WriteLine("\nCurrent thread: {0}", Thread.CurrentThread.Name);
      Console.WriteLine("Thread1: {0}", thread1.ThreadState);
      Console.WriteLine("Thread2: {0}\n", thread2.ThreadState);
   }
}
// The example displays the following output:
//       Current thread: Thread1
//
//       Current thread: Thread2
//       The timeout has elapsed and Thread1 will resume.
//
//       Current thread: Thread2
//       Thread1: WaitSleepJoin
//       Thread2: Running
//
//
//       Current thread: Thread1
//       Thread1: Running
//       Thread2: Stopped
//       Thread1 Completed its work
//       Main thread has terminated.
```

Retrieving data from a thread function

In order to retrieve the data from a thread function, first, you need to encapsulate the thread function and the data it requires in a helper class. To the constructor of the **Helper class**, you need to pass the required data as well as a delegate representing the callback method. From the thread function body, you need to invoke the callback delegate just before the thread method ends. And one more thing you need to take care that the callback delegate and the actual callback method signature should be the same.

Now, we need to create the callback method whose signature should be the same as the signature of the CallBack Delegate. In our example, **ResultCallBackMethod** is the callback method and its signature is the same as the signature of the **ResultCallbackDelegate** delegate. Within the Main method, we need to create an instance of the **ResultCallbackDelegate** delegate and while creating the instance we need to pass the **ResultCallBackMethod** as the parameter to its constructor. So when we invoke the delegate it will call the **ResultCallBackMethod** method.

Example:

```
using System;
namespace ThreadingDemo
{
    // First Create the callback delegate with the same signature of the callback
    // method.
    public delegate void ResultCallbackDelegate(int Results);

    //Creating the Helper class
    public class NumberHelper
    {
        //Creating two private variables to hold the Number and ResultCallback
        // instance
        private int _Number;
        private ResultCallbackDelegate _resultCallbackDelegate;
        //Initializing the private variables through constructor
        //So while creating the instance you need to pass the value for Number
        //and callback delegate
        public NumberHelper(int Number, ResultCallbackDelegate
resultCallbackDelagate)
        {
            _Number = Number;
            _resultCallbackDelegate = resultCallbackDelagate;
        }
        //This is the Thread function which will calculate the sum of the numbers
        public void CalculateSum()
        {
            int Result = 0;
            for (int i = 1; i <= _Number; i++)
            {
                Result = Result + i;
            }
            //Before the end of the thread function call the callback method
            if (_resultCallbackDelegate != null)
            {
                _resultCallbackDelegate(Result);
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Create the ResultCallbackDelegate instance and to its constructor
            //pass the callback method name
            ResultCallbackDelegate resultCallbackDelegate = new
ResultCallbackDelegate(ResultCallBackMethod);
            int Number = 10;
            //Creating the instance of NumberHelper class by passing the Number
            //the callback delegate instance
            NumberHelper obj = new NumberHelper(Number, resultCallbackDelegate);
```

```
            //Creating the Thread using ThreadStart delegate
            Thread T1 = new Thread(new ThreadStart(obj.CalculateSum));
            T1.Start();
            Console.Read();
        }
        //Callback method and the signature should be the same as the callback
delegate signature
        public static void ResultCallBackMethod(int Result)
        {
            Console.WriteLine("The Result is " + Result);
        }
    }
}
```

Others methods of Thread class

1. **Abort**()
   Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
2. **Resume**()
   Resumes a thread that has been suspended.
3. **Suspend**()
   Either suspends the thread, or if the thread is already suspended, has no effect.
4. **Interrupt**()
   Interrupts a thread that is in the WaitSleepJoin thread state.