VTC Academy
Thinking Ahead, Beyond Boundaries

Object Oriented Programming

Multithreading Programming

Session 10

.NET Core C#

Objectives

- Introduction to Thread
- Working with Thread
- Mulithreding
- Thread synchronization

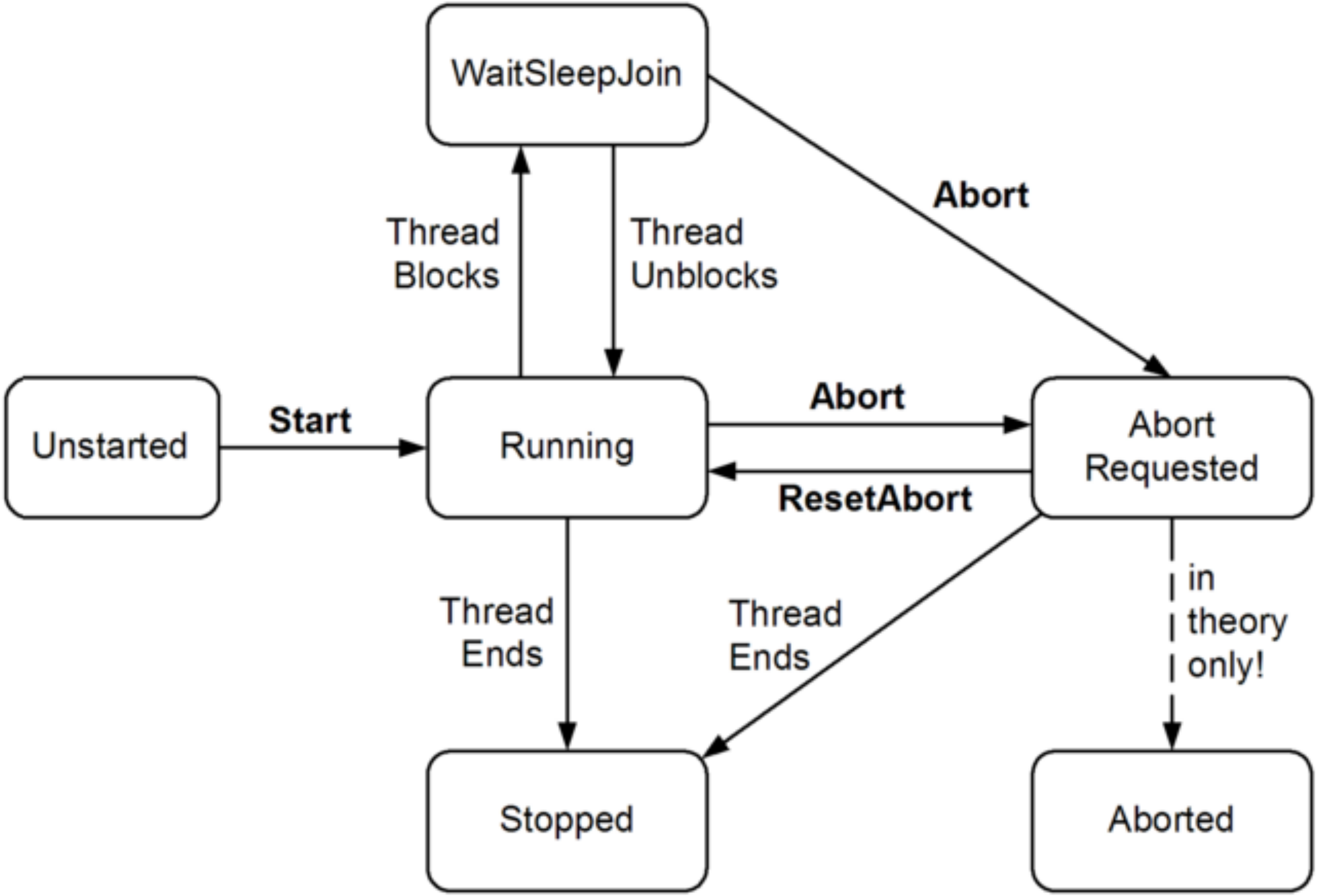
Introduction to Thread

- A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control.
- If your application involves complicated and time consuming operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
- Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

Thread Life Cycle

- **Unstarted** state: It is the situation when the instance of the thread is created but the Start method is not called
- **Ready** state: It is the situation when the thread is ready to run and waiting CPU cycle
- **Not Runnable** state: A thread is not executable, when:
  - Sleep method has been called
  - Wait method has been called
  - Blocked by I/O operations
- **Dead** state: It is the situation when the thread completes execution or is aborted

Thread Life Cycle



Main Thread

- In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.
- When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread.
- You can access a thread using the **CurrentThread** property of the Thread class.

Main Thread Example

```
using System;
using System.Threading;

namespace MainThreadDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         Thread mthread = Thread.CurrentThread;
         mthread.Name = "Main Thread";
         Console.WriteLine("This is {0}", mthread.Name);
         Console.ReadKey();
      }
   }
}
```

Creating Threads

- Threads are created by extending the Thread class. The extended Thread class then calls the **Start()** method to begin the child thread execution.

- Creating a Thread:

```
System.Threading.Thread newThread = new System.Threading.Thread(anObject.AMethod);
```

- Starting a Thread:

```
newThread.Start();
```

Creating Threads Example

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         Console.WriteLine("Child thread starts");
      }

      static void Main(string[] args)
      {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

Properties of Thread Class

Property	Description
CurrentThread	Gets the currently running thread
IsAlive	Gets a value indicating the execution status of the current thread
IsBackground	Gets or sets a value indicating whether or not a thread is a background thread
ManagedThreadId	Gets a unique identifier for the current managed thread
Name	Gets or sets the name of the thread
Priority	Gets or sets a value indicating the scheduling priority of a thread
ThreadState	Gets a value containing the states of the current thread

Methods of Thread Class

No.	Methods
1	**public void Abort()** Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread
2	**public void Interrupt()** Interrupts a thread that is in the WaitSleepJoin thread state
3	**public void Join()** Blocks the calling thread until a thread terminates, while continuing to perform standard COM and SendMessage pumping
4	**public void Start()** Starts a thread
5	**public static void Sleep(int millisecondsTimeout)** Makes the thread pause for a period of time

Managing Threads

Thread class provides various methods for managing threads:

```
using System;
using System.Threading;
namespace ManagingThreadsDemo
{
   class Program
   {
      public static void CallToChildThread()
      {
         Console.WriteLine("Child thread starts");
         // the thread is paused for 5000 milliseconds
         int sleepfor = 5000;
         Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor / 1000);
         Thread.Sleep(sleepfor);
         Console.WriteLine("Child thread resumes");
      }
      static void Main(string[] args)
      {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

Destroying Threads

- The Abort() method is used for destroying threads
- Method 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.
- After destroying, Thread can not be restart

Destroying Threads Example

```
using System;
using System.Threading;
namespace DetroyingThreadsDemo {
   class Program {
      public static void CallToChildThread() {
         try {
            Console.WriteLine("Child thread starts");
            // do some work, like counting to 10
            for (int counter = 0; counter <= 10; counter++) {
               Thread.Sleep(500);
               Console.WriteLine(counter);
            }
            Console.WriteLine("Child Thread Completed");
         } catch (ThreadAbortException e) {
            Console.WriteLine("Thread Abort Exception");
         } finally {
            Console.WriteLine("Couldn't catch the Thread Exception");
         }
      }
      static void Main(string[] args) {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         // stop the main thread for some time
         Thread.Sleep(2000);
         // now abort the child
         Console.WriteLine("In Main: Aborting the Child thread");
         childThread.Abort();
         Console.ReadKey();
      }
   }
}
```

Multitasking vs Multithreading

- Multitasking is the ability to run one or more programs concurrently

- Operating system controls the way in which these programs run by scheduling them
- Time elapsed between switching of programs is minuscule.
- Multithreading is the ability to execute different parts of a program called threads, simultaneously

Benefits of Multithreading

- Multithreading requires less overhead than multitasking:
  - In multitasking, processes run in their own different address space
  - Tasks involved in multithreading can share the same address space
  - Inter-process calling involves more overhead than inter-thread communication
- Multithreading allows us to write efficient programs that make maximum use of CPU
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause

Threads Synchronization

- In multithreaded programs, several threads may simultaneously try to update the same resource, such as a file. This leaves the resource in an undefined or inconsistent state. This is called race condition.
- In general, race conditions in a program occur when
  - Two or more threads share the same data between them
  - Two or more threads try to read and write the shared data simultaneously

Threads Synchronization

- Features and classes that can be used to synchronize access to resources in multithreaded applications:
  - The lock Keyword
  - Monitors
  - Synchronization Events and Wait Handles
  - Mutex Object

The lock Keyword

- The C# lock statement can be used to ensure that a block of code runs to completion without interruption by other threads. This is accomplished by obtaining a mutual-exclusion lock for a given object for the duration of the code block.
- Example:

```
public class TestThreading
{
    private System.Object lockThis = new System.Object();
    public void Process()
    {
        lock (lockThis)
        {
            // Access thread–sensitive resources.
        }
    }

}
```

Monitors

- Like the lock keyword, monitors prevent blocks of code from simultaneous execution by multiple threads. The Enter method allows one and only one thread to proceed into the following statements; all other threads are blocked until the executing thread calls Exit.

- Example:

```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
    DoSomething();
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
```

Synchronization Events & Wait Handles

- Using a lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but these constructs do not allow one thread to communicate an event to another.
- This requires synchronization events, which are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads.
- Threads can be suspended by being made to wait on a synchronization event that is unsignaled, and can be activated by changing the event state to signaled.
- If a thread attempts to wait on an event that is already signaled, then the thread continues to execute without delay.

Synchronization Events & Wait Handles

```
class Program
{
    static AutoResetEvent autoEvent;
    static void DoWork()
    {
        Console.WriteLine("Worker thread started, now waiting on event...");
        autoEvent.WaitOne();
        Console.WriteLine("Worker thread reactivated, now exiting...");
    }
    static void Main()
    {
        autoEvent = new AutoResetEvent(false);
        Console.WriteLine("Main thread starting worker thread...");
        Thread t = new Thread(DoWork);
        t.Start();
        Console.WriteLine("Main thread sleeping for 1 second...");
        Thread.Sleep(1000);
        Console.WriteLine("Main thread signaling worker thread...");
        autoEvent.Set();
    }
}
```

Summary

- A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control.
- The first thread to be executed in a process is called the **main** thread.
- In multithreaded programs, several threads may simultaneously try to update the same resource, such as a file. This leaves the resource in an undefined or inconsistent state. This is called race condition.
- Using a lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but these constructs do not allow one thread to communicate an event to another.