MULTITHREADING

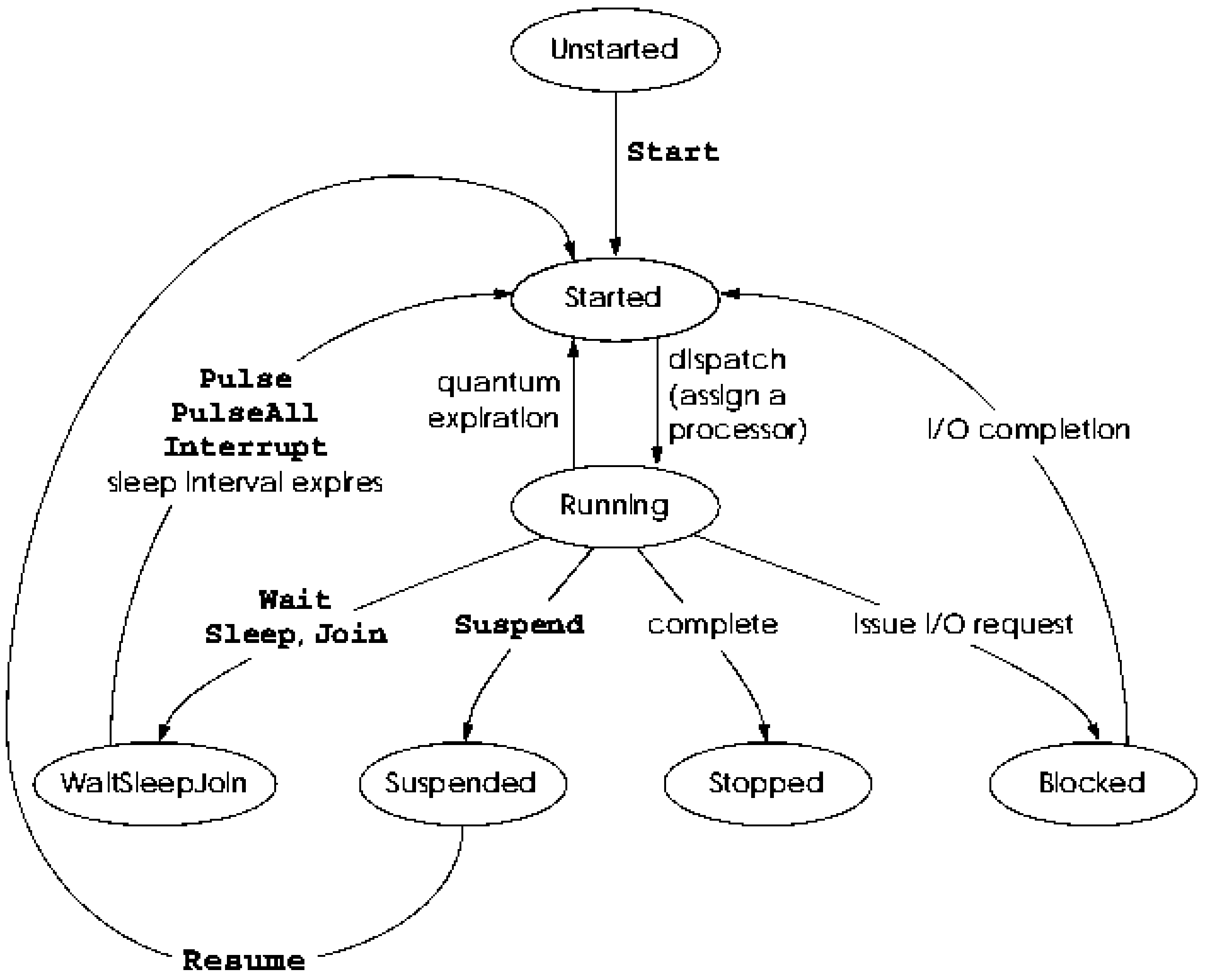
Asma Kanwal
Lecturer
Department of Computer Science
GC University, Lahore

INTRODUCTION

“threads of execution,” each thread designating a portion of a program that may execute concurrently with other threads—this capability is called multithreading.

Concurrency using synchoronization.

THREADS LIFE CYCLE



THREADS LIFE CYCLE DESCRIPTION

Start State: Ready or Runnable state

Running State: Operating system assigns a processor to the thread. When a program creates a new Thread , the program specifies the Thread 's ThreadStart delegate as the argument to the Thread constructor.

Stopped State: ThreadStart delegate terminates.

Method Abort throws a ThreadAbortException in the thread, normally causing the thread to terminate

Blocked State: Thread issues an input/output request. The operating system blocks the thread from executing until the operating system can complete the I/O for which the thread is waiting.

THREADS LIFE CYCLE DESCRIPTION

WaitSleepJoin state:

- Thread encounters code that it cannot execute yet the thread can call Monitor method Wait to enter the WaitSleepJoin state.

Monitor method ***Pulse*** or ***PulseAll.*** Method ***Pulse*** moves the next waiting thread back to the Started state. Method ***PulseAll*** moves all waiting threads back to the Started state.

- A Running thread can call Thread method Sleep to enter the WaitSleepJoin state. WaitSleepJoin state for a period of milliseconds specified as the argument to Sleep. A sleeping thread returns to the Started state when its designated sleep time expires. Sleeping threads cannot use a processor, even if one is available.

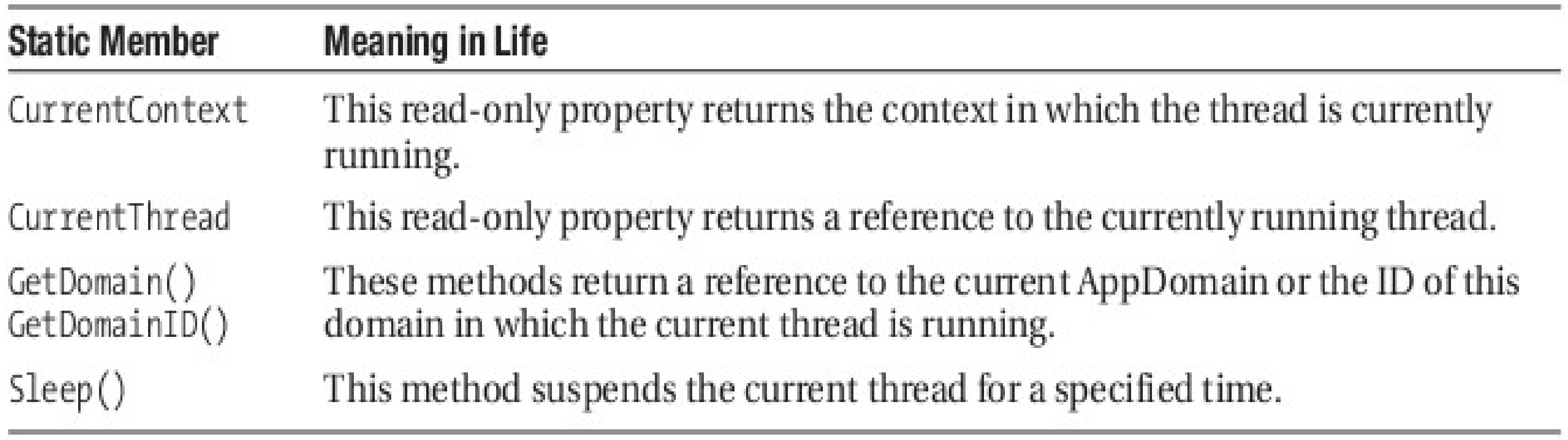
Threads Life Cycle Description

Suspended State: A Suspended thread returns to the Started state when another thread in the program invokes the Suspended thread’s Resume method.

System.Threading namespace

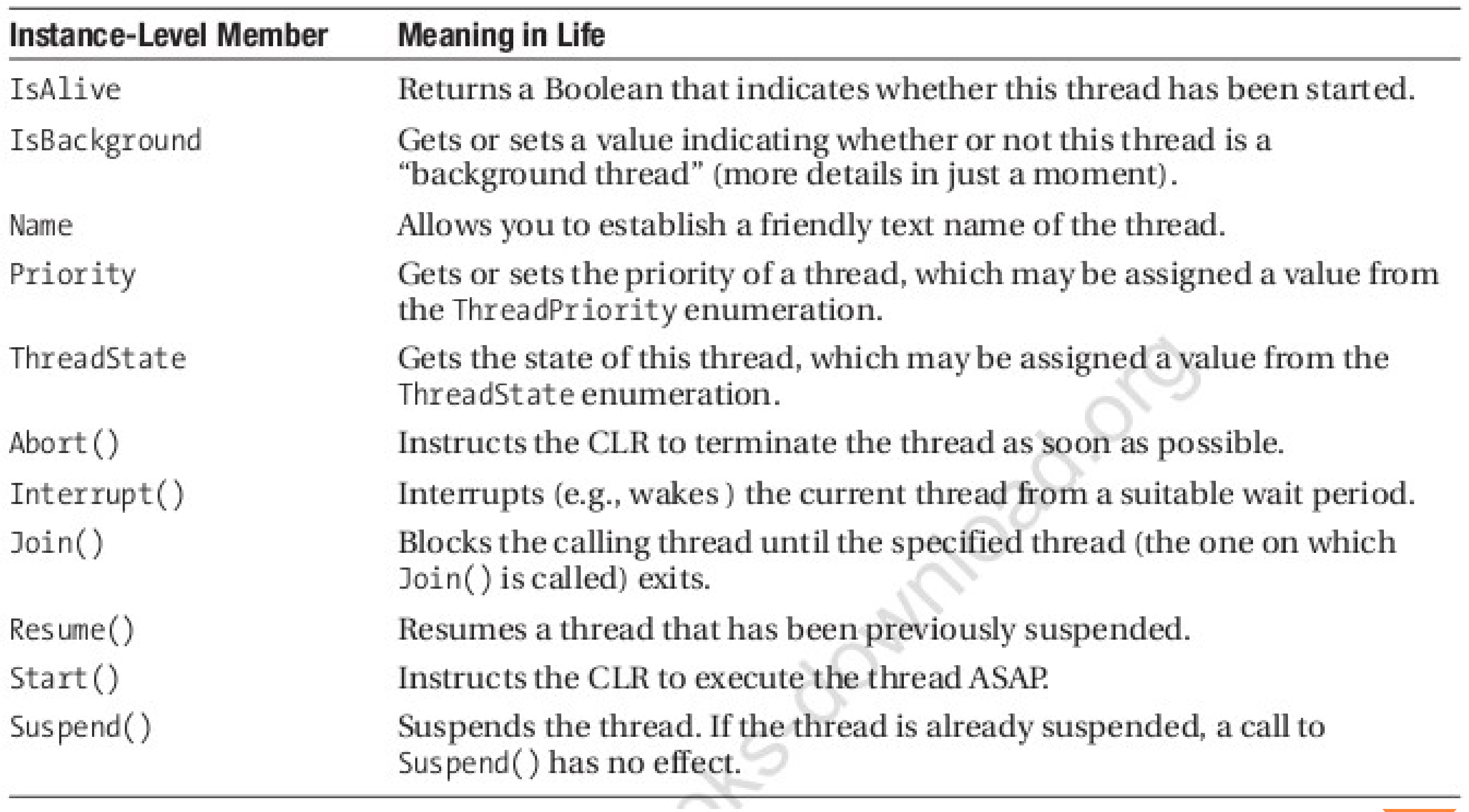
Type	Meaning in Life
`Interlocked`	This type provides atomic operations for types that are shared by multiple threads.
`Monitor`	This type provides the synchronization of threading objects using locks and wait/signals. The C# `lock` keyword makes use of a `Monitor` type under the hood.
`Mutex`	This synchronization primitive can be used for synchronization between application domain boundaries.
`ParameterizedThreadStart`	This delegate allows a thread to call methods that take any number of arguments.
`Semaphore`	This type allows you to limit the number of threads that can access a resource, or a particular type of resource, concurrently.
`Thread`	This type represents a thread that executes within the CLR. Using this type, you are able to spawn additional threads in the originating AppDomain.
`ThreadPool`	This type allows you to interact with the CLR-maintained thread pool within a given process.
`ThreadPriority`	This enum represents a thread's priority level (`Highest`, `Normal`, etc.).
`ThreadStart`	This delegate is used to specify the method to call for a given thread. Unlike the `ParameterizedThreadStart` delegate, targets of `ThreadStart` must match a fixed prototype.
`ThreadState`	This enum specifies the valid states a thread may take (`Running`, `Aborted`, etc.).
`Timer`	This type provides a mechanism for executing a method at specified intervals.
`TimerCallback`	This delegate type is used in conjunction with `Timer` types.

THREAD STATIC MEMBER



Static Member	Meaning in Life
`CurrentContext`	This read-only property returns the context in which the thread is currently running.
`CurrentThread`	This read-only property returns a reference to the currently running thread.
`GetDomain()` `GetDomainID()`	These methods return a reference to the current AppDomain or the ID of this domain in which the current thread is running.
`Sleep()`	This method suspends the current thread for a specified time.

Instance Level Member



Instance-Level Member	Meaning in Life
`IsAlive`	Returns a Boolean that indicates whether this thread has been started.
`IsBackground`	Gets or sets a value indicating whether or not this thread is a “background thread” (more details in just a moment).
`Name`	Allows you to establish a friendly text name of the thread.
`Priority`	Gets or sets the priority of a thread, which may be assigned a value from the `ThreadPriority` enumeration.
`ThreadState`	Gets the state of this thread, which may be assigned a value from the `ThreadState` enumeration.
`Abort()`	Instructs the CLR to terminate the thread as soon as possible.
`Interrupt()`	Interrupts (e.g., wakes) the current thread from a suitable wait period.
`Join()`	Blocks the calling thread until the specified thread (the one on which `Join()` is called) exits.
`Resume()`	Resumes a thread that has been previously suspended.
`Start()`	Instructs the CLR to execute the thread ASAP.
`Suspend()`	Suspends the thread. If the thread is already suspended, a call to `Suspend()` has no effect.

EXAMPLE

```
public class Mythread
   {
         public static void Thread1()
         {
            for (int i = 0; i < 10; i++)
            {
               Console.WriteLine("Thread1 {0}", i);
            }
         }

         public static void Thread2()
         {
            for (int i = 0; i < 10; i++)
            {
               Console.WriteLine("Thread2 {0}", i);
            }
         }

   }
```

EXAMPLE

```
static void Main(string[] args)
        {
            Console.WriteLine("Before start thread");
       //     Thread tid1 = new Thread(new ThreadStart(Mythread.Thread1));
       //     Thread tid2 = new Thread(new ThreadStart(Mythread.Thread2));

            Thread tid1 = new Thread(Mythread.Thread1);
            Thread tid2 = new Thread(Mythread.Thread2);

            tid1.Start();
            tid2.Start();
           // tid1
            Console.Read();
        }
```

Monitor example

```
using System;
using System.Threading;
namespace waitndpulesmethod
{
class TickTock
    {
        public void tick(bool running)
        {
            lock (this)
            {
                if (!running)
                { //  stop  the  clock
                    Monitor.Pulse(this);  //  notify  any  waiting  threads
                    return;
                }
                Console.Write("Tick  ");
                Monitor.Pulse(this);  //  let  tock()  run
                Monitor.Wait(this);  //  wait  for  tock()  to  complete
            }
        }
```

Monitor example

```
public void tock(bool running)
      {
         lock (this)
         {
            if (!running)
            { //  stop  the  clock
               Monitor.Pulse(this);  //  notify  any  waiting  thr
eads    |
               return;
            }
            Console.WriteLine("Tock");
            Monitor.Pulse(this);  //  let  tick()  run
            Monitor.Wait(this);  //  wait  for  tick()  to  complet
e
         }
      }
   }
```

MONITOR EXAMPLE

```
class MyThread
   {
      public Thread thrd;
      TickTock ttOb;
      //  Construct  a  new  thread.
      public MyThread(string name, TickTock tt)
      {
         thrd = new Thread(this.run);
         ttOb = tt;
         thrd.Name = name;
         thrd.Start();
      }
      //  Begin  execution  of  new  thread.
      void run()
      {
         if (thrd.Name == "Tick")
         {
            for (int i = 0; i < 5; i++) ttOb.tick(true);
            ttOb.tick(false);
         }
         else
         {
            for (int i = 0; i < 5; i++) ttOb.tock(true);
            ttOb.tock(false);
         }
      }
   }
```

Monitor example

```
class TickingClock
    {
        public static void Main()
        {
            TickTock tt = new TickTock();
            MyThread mt1 = new MyThread("Tick", tt);
            MyThread mt2 = new MyThread("Tock", tt);

            mt1.thrd.Join();
            mt2.thrd.Join();
            Console.WriteLine("Clock  Stopped");
            Console.Read();
        }
    }
}
```