# Advanced Programming Methods
# Lecture 12 - C# Concurrency

# Content

- Basic Concurrency
- Thread pooling
    - Task Parallel Library
- Synchronization mechanisms
- Asynchronous programming

# References

**NOTE: The slides are based on the following free tutorials. You may want to consult them too.**


**1. https://msdn.microsoft.com/en-us/library/hh156548(v=vs.110).aspx**

**2. https://msdn.microsoft.com/en-us/library/hh191443.aspx**

**3. http://www.albahari.com/threading/**

# Threads

- a C# program starts in a single thread (the "main" thread) created automatically by the CLR (Common Language Runtime) and operating system , and is made multithreaded by creating additional threads
- CLR assigns each thread its own memory stack so that local variables are kept separate.
- Threads share data if they have a common reference to the same object instance.

```
using System;
using System.Threading;

class ThreadTest{
  static void Main(){
    Thread t = new Thread (WriteY);         // Kick off a new thread
    t.Start();                              // running WriteY()

    // Simultaneously, do something on the main thread.
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }

  static void WriteY() {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}
```

# Threads

- once started, a thread's IsAlive property returns true, until the point where the thread ends.

- a thread ends when the delegate passed to the Thread's constructor finishes executing.

- once ended, a thread cannot restart.

# Thread life cycle

- The Unstarted State: it is the situation when the instance of the thread is created but the Start method has not been called.

-The Ready State: it is the situation when the thread is ready to run and waiting CPU cycle.

- The Not Runnable State: a thread is not runnable, when:

  - Sleep method has been called
  - Wait method has been called
  - Blocked (e.g. by I/O operations)

- The Dead State: it is the situation when the thread has completed execution or has been aborted.

# Threads scheduler

- it manages multithreading

- it is a function that the CLR typically delegates to the operating system

- it ensures all active threads are allocated appropriate execution time, and that threads that are waiting or blocked (for instance, on an exclusive lock or on user input) do not consume CPU time

# Thread's unsafety

```
class ThreadTest {
  static bool done;    // Static fields are shared between all threads
  static void Main(){
    new Thread (Go).Start();
    Go();
  }

  static void Go(){
    if (!done) { Console.WriteLine ("Done");done = true;  }
  }
}
//How many times and which “Done” is printed first?
```

# Thread's Safety

```
class ThreadSafe {
  static bool done;
  static readonly object locker = new object();
 static void Main() {
    new Thread (Go).Start();
    Go();
  }
  static void Go(){
    lock (locker){  // only one thread can execute,
                    //other threads are blocked without consuming CPU
      if (!done) { Console.WriteLine ("Done"); done = true; }
    }
}}
```

# Join and Sleep

- a thread can wait for a second thread to end by calling the second thread Join method.

- Thread.Sleep pauses the current thread for a specified period

- While waiting on a Sleep or Join, a thread is blocked and so does not consume CPU resources.

# Join example

```
static void Main()
{
  Thread t = new Thread (Go);
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended!");
}

static void Go()
{
  for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```

# Creating and Starting Threads

- threads are created using the Thread class's constructor, passing in a ThreadStart delegate which indicates where execution should begin

- ThreadStart delegate is defined:

```
public delegate void ThreadStart();
```

```
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (new ThreadStart (Go));

    t.Start();   // Run Go() on the new thread.
    Go();        // Simultaneously run Go() in the main thread.
  }

  static void Go(){
    Console.WriteLine ("hello!");
  }
}
```

# Passing data to a thread

```
static void Main()
{
// use a lambda expression to pass data to the thread's target method
  Thread t = new Thread ( () => Print ("Hello from t!") );
  t.Start();
}

static void Print (string message)
{
  Console.WriteLine (message);
}
```

# Passing data to a thread

```
static void Main(){
  Thread t = new Thread (Print);
//pass an argument into Thread's Start method
  t.Start ("Hello from t!");
}

static void Print (object messageObj){
  string message = (string) messageObj;   // We need to cast here
  Console.WriteLine (message);
}
```

# Naming Threads

```
class ThreadNaming {
  static void Main() {
    Thread.CurrentThread.Name = "main";
    Thread worker = new Thread (Go);
    worker.Name = "worker";
    worker.Start();
    Go();
  }
  static void Go() {
    Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
  }}
```

//thread's name can be set using Thread.CurrentThread property

# Foreground/Background threads

Foreground Threads:

- have the ability to prevent the current application from terminating.

- CLR will not shut down an application until all foreground threads have ended.

- by default, every thread we create via the Thread.Start() method is automatically a foreground thread

# Foreground/Background threads

Background Threads (also called daemon threads)

- are viewed by the CLR as expendable paths of execution that can be ignored at any point in time even if they are currently active doing work.

- if all foreground threads have terminated, all background threads are automatically terminated

We can query or change a thread's background status using its IsBackground property

```
using System;
using System.Threading;
namespace MyThread{
    public class BackgroundThread{
        public static void Main(string[] args){
            Thread worker = new Thread(delegate() {
                    Console.ReadLine(); });
            if (args.Length > 0) {
                    //the worker is assigned background status, and the
                    //program exits almost immediately as the main thread
                    //ends (terminating the ReadLine)
                   worker.IsBackground = true;
        } else{
          //the main thread exits, but the application keeps running
    // because a foreground thread is still alive
          } worker.Start();}}}
```

# Thread priority

- a thread's Priority property determines how much execution time it gets relative to other active threads in the operating system

enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }

- it is relevant only when multiple threads are simultaneously active.

- elevating a thread's priority doesn't make it capable of performing real-time work, because it's still throttled by the application's process priority

# Exception handling

- Any try/catch/finally blocks in scope when a thread is created are of no relevance to the thread when it starts executing

```
public static void Main(){
  try{
    new Thread (Go).Start();
  }catch (Exception ex){
    // We'll never get here!
    Console.WriteLine ("Exception!");
  }
}

static void Go() { throw null; }   // Throws a NullReferenceException
```

# Exception handling

```
public static void Main(){
   new Thread (Go).Start();
}
static void Go(){
  try{
    // ...
    throw null;    // The NullReferenceException will get caught below
    // ...
  }catch (Exception ex){
    // Typically log the exception, and/or signal another thread that we've
     come unstuck
    // ...
  }}
```

# Thread pooling

# Thread pool

- when a thread starts, a few hundred microseconds are spent organizing such things as a fresh private local variable stack.
- each thread also consumes (by default) around 1 MB of memory.
- the thread pool cuts these overheads by sharing and recycling threads, allowing multithreading to be applied at a very granular level without a performance penalty.

# Thread pool

Ways to enter the thread pool:

- By calling ThreadPool.QueueUserWorkItem
- Via asynchronous delegates
- Via BackgroundWorker
- Via the Task Parallel Library (from Framework 4.0 is the easiest way)

# ThreadPool.QueueUserWorkItem

**- it is called with a delegate that you want to run on a pooled thread**

```
static void Main(){
  ThreadPool.QueueUserWorkItem (Go);
  ThreadPool.QueueUserWorkItem (Go, 123);
  Console.ReadLine();}

//satisfies WaitCallback delegate
static void Go (object data){   // data will be null with the first call
  Console.WriteLine ("Hello from the thread pool! " + data);}
```

**- doesn't return an object to help you subsequently manage execution**

# Asynchronous delegates

1. Instantiate a delegate targeting the method you want to run in parallel (typically one of the predefined Func delegates).
2. Call BeginInvoke on the delegate, saving its IAsyncResult return value.BeginInvoke returns immediately to the caller. You can then perform other activities while the pooled thread is working.
3. When you need the results, call EndInvoke on the delegate, passing in the saved IAsyncResult object.

# Asynchronous delegates

```
static void Main() {
  Func<string, int> method = Work;
  IAsyncResult cookie = method.BeginInvoke ("test", null, null);
  //
  // ... here's where we can do other work in parallel...
  //
  int result = method.EndInvoke (cookie);
//EndInvoke waits for the asynchronous delegate to finish executing
//and it receives the return value
  Console.WriteLine ("String length is: " + result);
}

static int Work (string s) { return s.Length; }
```

# BackgroundWorker

1. Instantiate BackgroundWorker and handle the DoWork event.

2. Call RunWorkerAsync, optionally with an object argument. Any argument passed to RunWorkerAsync will be forwarded to DoWork's event handler, via the event argument's Argument property.

# BackgroundWorker

```
class Program {
  static BackgroundWorker _bw = new BackgroundWorker();

  static void Main() {
    _bw.DoWork += bw_DoWork;
    _bw.RunWorkerAsync ("Message to worker");
    Console.ReadLine();
  }

  static void bw_DoWork (object sender, DoWorkEventArgs e){
    // This is called on the worker thread
    Console.WriteLine (e.Argument);        // writes "Message to worker"
    // Perform time-consuming task...
  }}
```

# Task Parallel Library

# Via Task Parallel Library

**- enter the thread pool using the Task class from System.Threading.Tasks**

```
static void Main()   {
  Task.Factory.StartNew (Go);
/*Task.Factory.StartNew returns a Task object, which can
   then be used to monitor the task — for instance, you
   can wait for it to complete by calling its Wait method */
}

static void Go(){
  Console.WriteLine ("Hello from the thread pool!");
}
```

```
static void Main(){
  //Task<TResult> class lets you get a return value back from
  //the task after it finishes executing
  Task<string> task = Task.Factory.StartNew<string>
    ( () => DownloadString ("http://www.aaa.com") );
// do other work here and it will execute in parallel
  RunSomeOtherMethod();

   // When we need the task's return value, we query its Result property:
  // If it's still executing, the current thread will now block (wait)
  // until the task finishes:
  string result = task.Result;}
 static string DownloadString (string uri){
  using (var wc = new System.Net.WebClient())
    return wc.DownloadString (uri);}
```

# Waiting on Tasks

You can explicitly wait for a task to complete in two ways:

- Calling its Wait method (optionally with a timeout)
- Accessing its Result property (in the case of Task<TResult>)

You can also wait on multiple tasks at once via the static methods:

- Task.WaitAll (waits for all the specified tasks to finish)
- Task.WaitAny (waits for just one task to finish).

# Exception handling

```
int x = 0;
Task<int> calc = Task.Factory.StartNew (() => 7 / x);
try
{
  Console.WriteLine (calc.Result);
}
catch (AggregateException aex)
{
  Console.Write (aex.InnerException.Message);
 // Attempted to divide by 0
}
```

# Canceling tasks

- You can optionally pass in a cancellation token when starting a task
- The cancel can be done by calling cancel on the CancellationTokenSource

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;
Task task = Task.Factory.StartNew (() => {
  // Do some stuff...
  token.ThrowIfCancellationRequested();  // Check for cancellation request
  // Do some stuff...
}, token);
...
cancelSource.Cancel();
```

# Canceling tasks

- To detect a canceled task:

```
try
{
  task.Wait();
}
catch (AggregateException ex)
{
  if (ex.InnerException is OperationCanceledException)
    Console.Write ("Task canceled!");
}
```

# Continuations

- it's useful to start a task right after another one completes (or fails).
- the ContinueWith method on the Task class does exactly this:

```
Task.Factory.StartNew<int> (() => 8)
  .ContinueWith (ant => ant.Result * 2)
  .ContinueWith (ant => Math.Sqrt (ant.Result))
  .ContinueWith (ant => Console.WriteLine (ant.Result));   // 4
```

# Parallel class

- a basic form of structured parallelism via three static methods in the Parallel class:
  - Parallel.Invoke: executes an array of delegates in parallel
  - Parallel.For: performs the parallel equivalent of a C# for loop
  - Parallel.ForEach: performs the parallel equivalent of a C# foreach loop

# Synchronization

# Synchronization

- coordinating the actions of threads for a predictable outcome
- its constructs can be divided into four categories:

1. Simple blocking methods (e.g. Sleep, Join): wait for another thread to finish or for a period of time to elapse

# Synchronization

2. Locking constructs (e.g. Lock): limit the number of threads that can perform some activity or execute a section of code at a time.

3. Signaling constructs: allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling.

4. Nonblocking synchronization constructs (e.g. Volatile, Interlocked): protect access to a common field by calling upon processor primitives

# Blocking

- thread execution is paused for some reason
- thread consumes no processor time until blocking condition is satisfied

Unblocking happens in one of four ways:

- by the blocking condition being satisfied
- by the operation timing out (if a timeout is specified)
- by being interrupted via Thread.Interrupt
- by being aborted via Thread.Abort

# Blocking Versus Spinning

A thread must pause until a certain condition is met:

- efficiently: signaling and locking constructs achieve this by blocking until condition is satisfied

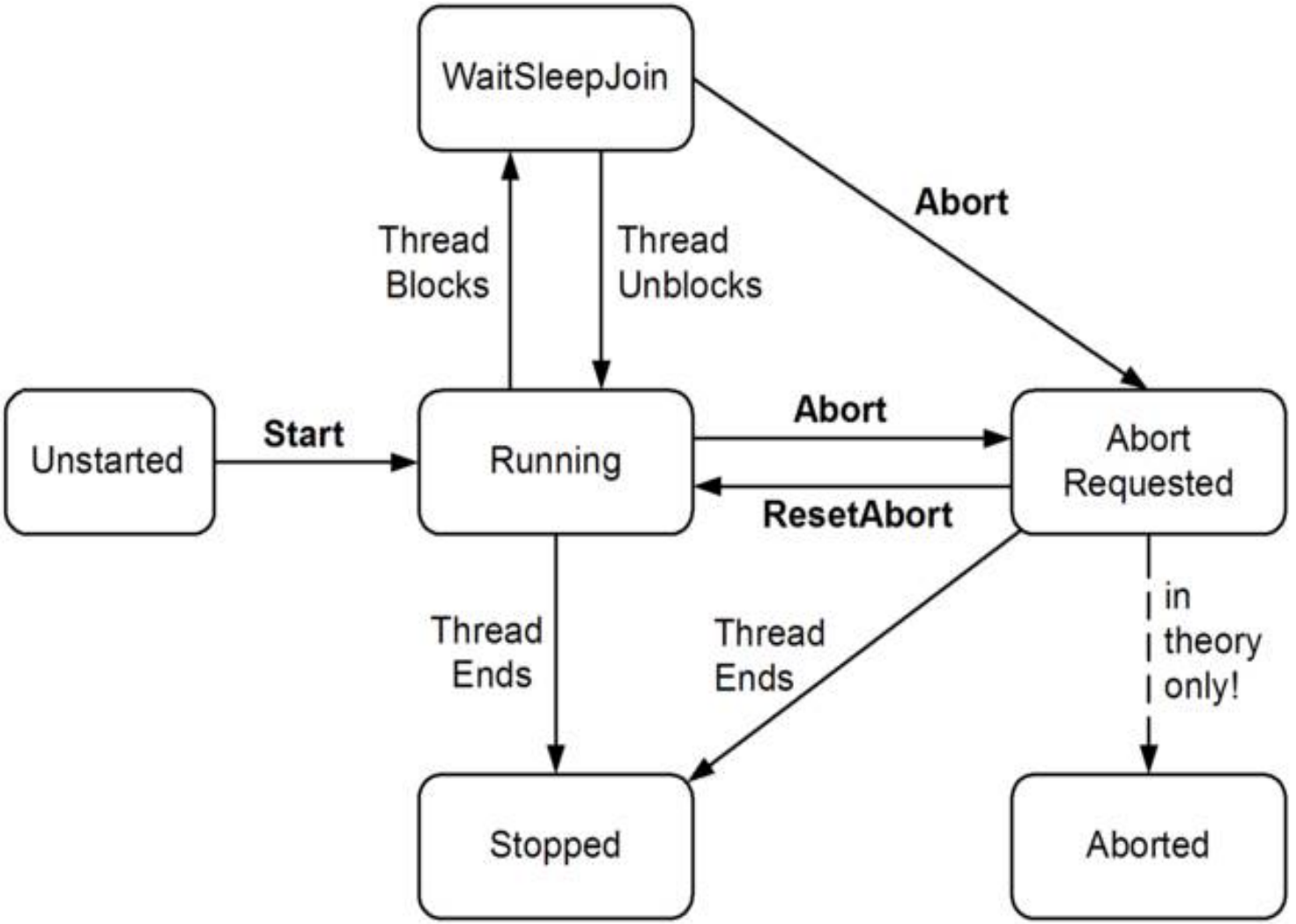
- simply but ineffiently: by spinning in a polling loop, e.g.:

```
while (!proceed);
```

or in a mixed way:

```
while (!proceed) Thread.Sleep (10);
```

# ThreadState property



# Locking

- **Exclusive locking is used to ensure that only one thread can enter particular sections of code at a time**

```
class ThreadSafe{
  static readonly object  _locker = new object();
  static int _val1, _val2;

  static void Go(){
    lock (_locker){
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
}}
```

# Locking

- lock statement is in fact a syntactic shortcut for a call to the methods Monitor.Enter and Monitor.Exit

```
Monitor.Enter (_locker);
try
{
  if (_val2 != 0) Console.WriteLine (_val1 / _val2);
  _val2 = 0;
}
finally { Monitor.Exit (_locker); }
```

# Choosing the Synchronization Object

```
1. class ThreadSafe{
  List <string> _list = new List <string>();
  void Test(){
    lock (_list) {
      _list.Add ("Item 1");
      ...
```

2. lock the entire object: `lock (this) { ... }`

3. in case of static fields/methods: `lock (typeof (Widget)) { ... }`

# When to Lock

- need to lock around accessing any writable shared field

```
class ThreadSafe{
  static readonly object  _locker = new object();
  static int _x;

  static void Increment() { lock (_locker)  _x++; }
  static void Assign()    { lock (_locker)  _x = 123; }
}
```

# Locking and Atomicity

- if a group of variables are always read and written within the same lock, we can say the variables are read and written atomically

- for example x and y are accessed atomically:

**lock (locker) { if (x != 0) y /= x; }**

# Nested Locking

```
static readonly object _locker = new object();


static void Main(){
  lock (_locker){
     AnotherMethod();
     // We still have the lock - because locks are reentrant.
  }
}
static void AnotherMethod(){
  lock (_locker) { Console.WriteLine ("Another method"); }
}
```

# Deadlocks

- when two threads each waits for a resource held by the other, so neither can proceed

```
object locker1 = new object();

object locker2 = new object();

 new Thread (() => {
                lock (locker1) {
                  Thread.Sleep (1000);
                  lock (locker2);      // Deadlock
                }
               }).Start();
lock (locker2){
  Thread.Sleep (1000);
  lock (locker1);                    // Deadlock
}
```

# Mutex

- is like a lock, but it can work across multiple processes
- can be computer-wide as well as application-wide
- Mutex class:
    - WaitOne method to lock
    - ReleaseMutex to unlock
- a Mutex can be released only from the same thread that obtained it.

Example: A common use for a cross-process Mutex is to ensure that only one instance of a program can run at a time

```
class OneAtATimePlease{
  static void Main() {
    // Naming a Mutex makes it available computer-wide.
   using (var mutex = new Mutex (false, "UniqueName")){
     // Wait a few seconds, in case another instance
     // of the program is still in the process of shutting down.
     if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false)) {
       Console.WriteLine ("Another app instance is running. Bye!");
       return; }
     RunProgram();
   } }

 static void RunProgram(){
    Console.WriteLine ("Running. Press Enter to exit");
    Console.ReadLine();
 }}
```

# Semaphore

- preventing too many threads from executing a particular piece of code at once.

- is like a room, it has a certain capacity. Once it's full, no more people can enter, and a queue builds up outside. Then, for each person that leaves, one person enters from the head of the queue.

- it has no owner, any thread can call release on a semaphore

```
class TheRoom {
  static SemaphoreSlim _sem = new SemaphoreSlim (3);    // Capacity of 3

  static void Main(){
    for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
  }

  static void Enter (object id){
    Console.WriteLine (id + " wants to enter");
    _sem.Wait();
    Console.WriteLine (id + " is in!");          // Only three threads
    Thread.Sleep (1000 * (int) id);              // can be here at
    Console.WriteLine (id + " is leaving");      // a time.
    _sem.Release();
  }
}
```

# Signaling with Event Wait Handles

Signaling:

- when one thread waits until it receives notification from another

Event wait handles:

- are the simplest of the signaling constructs
- are unrelated to C# events
- come in three flavors:
    - AutoResetEvent,
    - ManualResetEvent,
    - CountdownEvent

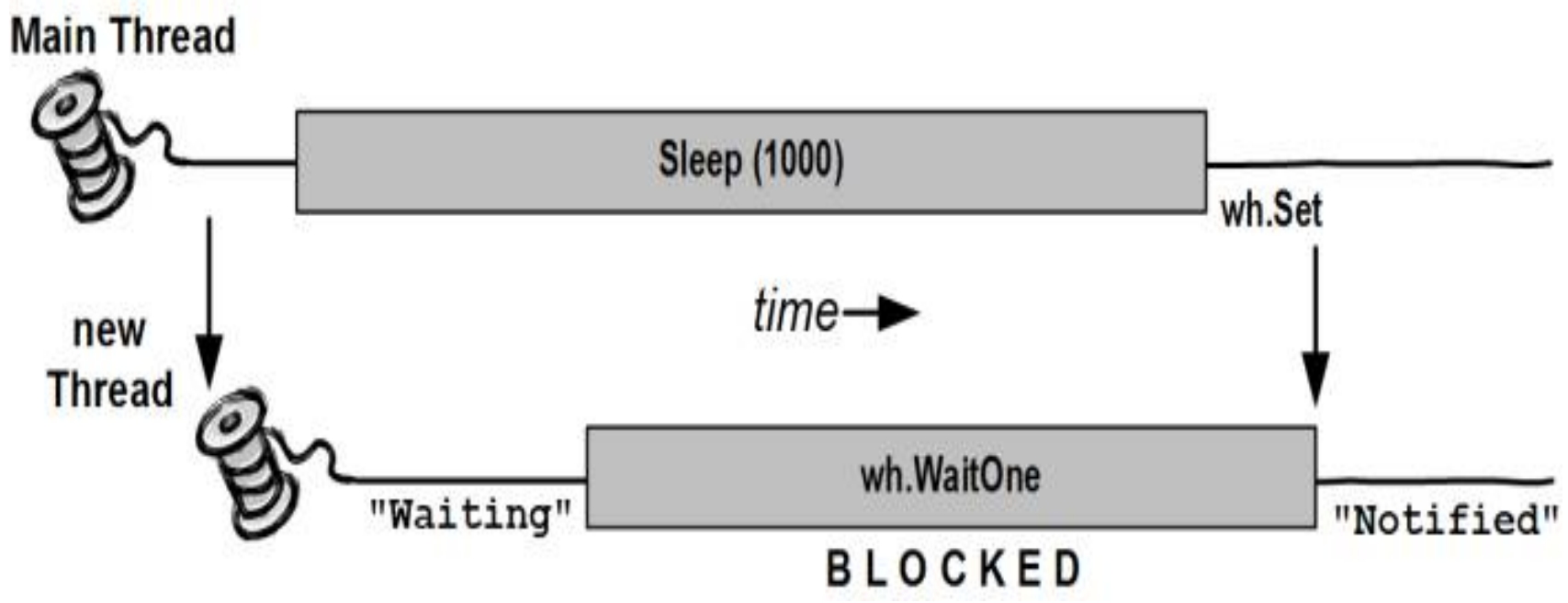
# AutoResetEvent

- is like a ticket turnstile: inserting a ticket lets exactly one person through
- “auto” in the class’s name refers to the fact that an open turnstile automatically closes or “resets” after someone steps through
- a thread waits, or blocks, at the turnstile by calling WaitOne
- a ticket is inserted by calling the Set method

# AutoResetEvent

- if a number of threads call WaitOne, a queue builds up behind the turnstile
- any thread with access to the AutoResetEvent object can call Set on it to release one blocked thread
- If Set is called when no thread is waiting, the handle stays open for as long as it takes until some thread calls WaitOne
- calling Set repeatedly on a turnstile at which no one is waiting: only the next single person is let through and the extra tickets are “wasted.”

Example: a thread is started whose job is simply to wait until signaled by another thread:



```
class BasicWaitHandle {
  static EventWaitHandle _waitHandle = new AutoResetEvent (false);

  static void Main(){
    new Thread (Waiter).Start();
    Thread.Sleep (1000);                // Pause for a second...
    _waitHandle.Set();                  // Wake up the Waiter.
  }

  static void Waiter(){
    Console.WriteLine ("Waiting...");
    _waitHandle.WaitOne();              // Wait for notification
    Console.WriteLine ("Notified");
  }
}
```

# Two-way Signaling

Example:

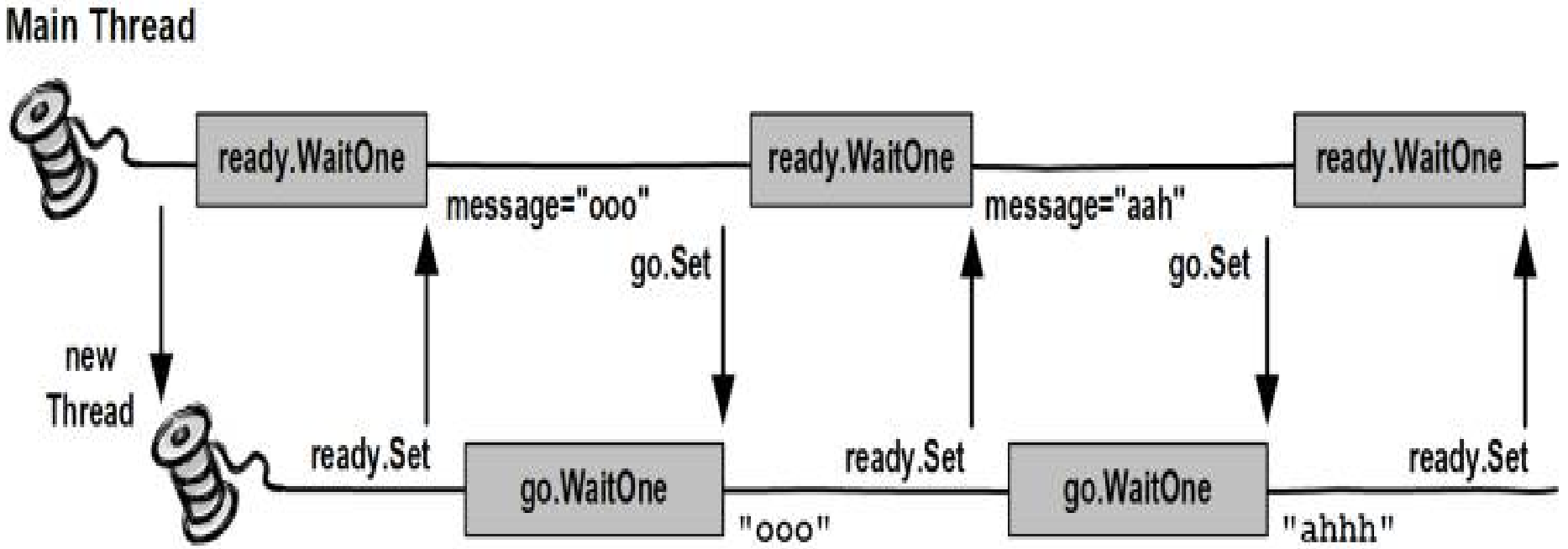
the main thread must signal a worker thread three times in a row

Bad solution:

- If the main thread simply calls Set on a wait handle several times in rapid succession, the second or third signal may get lost, since the worker may take time to process each signal.

Correct solution:

- the main thread must wait until the worker is ready before signaling it (using 2 AutoResetEvent)



```
class TwoWaySignaling {
  static EventWaitHandle _ready = new AutoResetEvent (false);
  static EventWaitHandle _go = new AutoResetEvent (false);
  static readonly object _locker = new object();
  static string _message;

  static void Main() {
    new Thread (Work).Start();
    _ready.WaitOne();                // First wait until worker is ready
    lock (_locker) _message = "ooo";
    _go.Set();                       // Tell worker to go

    _ready.WaitOne();
    lock (_locker) _message = "ahhh";  // Give the worker another message
    _go.Set();
```

```
  _ready.WaitOne();
    lock (_locker) _message = null;    // Signal the worker to exit
    _go.Set();
  }

  static void Work(){
    while (true){
      _ready.Set();                    // Indicate that we're ready
      _go.WaitOne();                     // Wait to be kicked off...
      lock (_locker){
        if (_message == null) return;       // Gracefully exit
        Console.WriteLine (_message);
      }
    }
}}
```

# Producer/consumer queue

A producer/consumer queue works as follows:

- a queue is set up to describe work items — or data upon which work is performed.
- when a task needs executing, it's enqueued, allowing the caller to get on with other things.
- one or more worker threads plug away in the background, picking off and executing queued items.

```
using System;
using System.Threading;
using System.Collections.Generic;
 class ProducerConsumerQueue : IDisposable {
  EventWaitHandle _wh = new AutoResetEvent (false);
  Thread _worker;
  readonly object _locker = new object();
  Queue<string> _tasks = new Queue<string>();

  public ProducerConsumerQueue() {
    _worker = new Thread (Work);
    _worker.Start();}

  public void EnqueueTask (string task){
    lock (_locker) _tasks.Enqueue (task);
    _wh.Set(); }
```

```
void Work() {
  while (true){
    string task = null;
    lock (_locker)
      if (_tasks.Count > 0) {
        task = _tasks.Dequeue();
        if (task == null) return;
      }
    if (task != null){
      Console.WriteLine ("Performing task: " + task);
      Thread.Sleep (1000);  // simulate work...
    } else
      _wh.WaitOne();        // No more tasks - wait for a signal
  }
}
```

```
public void Dispose(){
   EnqueueTask (null);     // Signal the consumer to exit.
   _worker.Join();         // Wait for the consumer's thread to finish.
   _wh.Close();            // Release any OS resources.
 }
static void Main(){
 using (ProducerConsumerQueue q = new ProducerConsumerQueue()){
   q.EnqueueTask ("Hello");
   for (int i = 0; i < 10; i++) q.EnqueueTask ("Say " + i);
   q.EnqueueTask ("Goodbye!");
 }
 // Exiting the using statement calls q's Dispose method, which
 // enqueues a null task and waits until the consumer finishes.
}
}
```

# ManualResetEvent

- is useful in allowing one thread to unblock many other threads

- functions like an ordinary gate

- calling Set opens the gate, allowing any number of threads calling WaitOne to be let through

- calling Reset closes the gate

-threads that call WaitOne on a closed gate will block; when the gate is next opened, they will be released all at once.

# CountdownEvent

- allows waiting on more than one thread

- the class is instantiated with the number of threads or “counts” that have to be waited on
- calling Signal decrements the “count”
- calling Wait blocks until the count goes down to zero

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main(){
  new Thread (SaySomething).Start ("I am thread 1");
  new Thread (SaySomething).Start ("I am thread 2");
  new Thread (SaySomething).Start ("I am thread 3");
  _countdown.Wait();   // Blocks until Signal has been called 3 times
  Console.WriteLine ("All threads have finished speaking!");
}

static void SaySomething (object thing){
  Thread.Sleep (1000);
  Console.WriteLine (thing);
  _countdown.Signal();
}
```

# Barrier class

- A barrier is a user-defined synchronization primitive that enables multiple threads (known as participants) to work concurrently on an algorithm in phases.
- Each participant executes until it reaches the barrier point in the code. The barrier represents the end of one phase of work.
- When a participant reaches the barrier, it blocks until all participants have reached the same barrier.
- After all participants have reached the barrier, you can optionally invoke a post-phase action. This post-phase action can be used to perform actions by a single thread while all other threads are still blocked. After the action has been executed, the participants are all unblocked.

# Barrier class

```
static Barrier _barrier = new Barrier (3);
static void Main(){
  new Thread (Speak).Start();
  new Thread (Speak).Start();
  new Thread (Speak).Start();
}
static void Speak(){
  for (int i = 0; i < 5; i++){
    Console.Write (i + " ");
    _barrier.SignalAndWait();
  }}
//0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

# Asynchronous programming patterns

Asynchronous Programming Model (APM) pattern (also called the IAsyncResult pattern)

- asynchronous operations require Begin and End methods (for example, BeginWrite and EndWrite for asynchronous write operations).
- This pattern is no longer recommended for new development

# Event-based Asynchronous Pattern (EAP)

- requires a method that has the Async suffix, and also requires one or more events, event handler delegate types, and EventArg-derived types.
- was introduced in the .NET Framework 2.0.
- It is no longer recommended for new development.

# Task-based Asynchronous Pattern (TAP)

- uses a single method to represent the initiation and completion of an asynchronous operation.
- was introduced in the .NET Framework 4 and is the recommended approach to asynchronous programming in the .NET Framework.
- The async and await keywords add language support for TAP.

# Async methods

- The method signature includes an async modifier.
- The name of an async method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:
  - Task<TResult> if your method has a return statement in which the operand has type Tresult.
  - Task if your method has no return statement or has a return statement with no operand.
  - Void if you're writing an async event handler.

# Async methods

- The method usually includes at least one await expression,
  - which marks a point where the method can't continue until the awaited asynchronous operation is complete.
  - In the meantime, the method is suspended, and control returns to the method's caller.

```
// Three things to note in the signature:
//  - The method has an async modifier.
//  - The return type is Task<int> because the return statement
//returns an integer.
//  - The method name ends in "Async."
async Task<int> AccessTheWebAsync() {
    // You need to add a reference to System.Net.Http to declare client.
    HttpClient client = new HttpClient();


    // GetStringAsync returns a Task<string>.
    //That means that when you await the
    // task you'll get a string (urlContents).
    Task<string> getStringTask =
     client.GetStringAsync("http://msdn.microsoft.com");
```

```
 // You can do work here that doesn't rely on the string
// from GetStringAsync.
    DoIndependentWork();

    // The await operator suspends AccessTheWebAsync.
    //  - AccessTheWebAsync can't continue until
    //getStringTask is  complete.
    //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    //  - Control resumes here when getStringTask is complete.
    //  - The await operator then retrieves the string
   //  result from getStringTask.
    string urlContents = await getStringTask;
```

```
    // The return statement specifies an integer result.
    // Any methods that are awaiting AccessTheWebAsync
    //retrieve the length value.
    return urlContents.Length;
}
```