Advanced Programming Methods

Lecture 12

This Lecture Overview

- Concurrency in C#

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
    // ...
  }}
```

Further Readings

- C# concurrency tutorial from MSDN