G. Cabri, G. Carenni, M. Badella, G. Malnati, L. Tessitore, C.Barberis 2003-18

Multithreading in .NET e C#

A.A. 2017-18

AppeAL
Applied Pervasive Architectures Lab

Threading in .NET

- Il namespace System.Threading contiene tutti i tipi:
  - Threads
  - Monitors
  - Eventi
  - Thread pools
  - Eccezioni relative
  - ....

Threads

- Si crea un thread (“gestito”) attraverso la classe System.Threading.Thread
- Ogni oggetto di tale classe ingloba un thread di sistema (thread “nativo”) – NON vale il viceversa
- Due tipi di thread
  - Foreground: l’applicazione non termina finchè esiste un thread di tale tipo in esecuzione
  - Background: quando l’applicazione termina, dopo che TUTTI i thread foreground sono finiti, il runtime forza la chiusura di tali thread (con il metodo Abort()).

Membri della classe Thread

```
public sealed class Thread
{
   public Thread(ThreadStart start);
   public void Start();
   public void Join();
   public void Abort();
   ...
}
```

Costruttore dell'oggetto Thread.
Il parametro è il "metodo" che deve essere eseguito dal thread.

Crea il thread nativo vero e proprio e lo manda in esecuzione.

Si blocca, in attesa che il thread termini (indefinitamente o per un certo tempo) la propria esecuzione.

Cerca di terminare il thread lanciando una eccezione di tipo ThreadAbortException sul thread stesso.

Membri della classe Thread

```
public sealed class Thread
{
        //proprietà
        public IsAlive{get;}
        public IsBackground{get;set;}

        //membri statici
        public static Thread CurrentThread{get;}
        public static void Sleep(int ms);
}
```

Restituisce true se il thread è in esecuzione (eventualmente in pausa)

Restituisce true se si tratta di un Background thread Per default è false

Mette in pausa il thread corrente per un numero prefissato di microsecondi. NOTA: Sleep() effettua sempre una "yield".

Restituisce l'oggetto che rappresenta il thread corrente.

Costruttore della classe Thread

```
public Thread(ThreadStart start);

public delegate void ThreadStart();
```

- Senza valore di ritorno

- Tale delegato può essere costruito a partire da un metodo statico o da un metodo legato all'istanza di un oggetto

Delegato ThreadStart

```
class foo
{
   private int a;
   private int b;
   private void ThreadFcn()
   {
      int i;
      for ( i = 0 ; i < this.a ; i++ )
         Console.WriteLine("Number {0}", i);
   }

   public void ExecuteMyThread(int n)
   {
      a = n;
      ThreadStart MyDelegate = new ThreadStart(this.ThreadFcn);
      Thread MyThread = new Thread(MyDelegate);
      MyThread.Start();
      MyThread.Join();
   }
}
```

Output:

```
Number 0
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
....
```

Delegato ThreadStart

```
class foo
{
   private int a;
   private int b;

   private void ThreadFcnA()
   {
      int i;
      for ( i = 0 ; i < this.a ; i++ )
         Console.WriteLine("Number A {0}", i);
   }

   private void ThreadFcnB()
   {
      int i;
      for ( i = 0 ; i < this.b ; i++ )
         Console.WriteLine("Number B {0}", i);
   }

   public void ExecuteMyThread(int n)
   {
      a = n;
      b = n;
      ThreadStart MyDelegate = new ThreadStart(this.ThreadFcnA);
      MyDelegate += new ThreadStart(this.ThreadFcnB);
      Thread MyThread = new Thread(MyDelegate);
      MyThread.Start();
      MyThread.Join();
   }
}
```

Output:
Number A 0
Number A 1
Number A 2
....
Number B 0
Number B 1
Number B 2
....

Viene creato UN SOLO thread

Le due funzioni registrate sul delegato vengono eseguite SEQUENZIALMENTE

Terminazione di un thread

- Esiste un metodo ad-hoc, Abort()
- Lancia una eccezione (ThreadAbortException) sul thread chiamato
- Il thread deve gestire questa eccezione come qualunque altra (blocchi catch e finally)
- Tale eccezione NON è ignorabile (tramite goto o codice fuori da finally) perchè viene rilanciata automaticamente dal runtime all'uscita del blocco finally (se esiste)
- L'unico modo per "fermare" l'eccezione è utilizzare il metodo ResetAbort()

Terminazione di un thread

- Abort() è sconsigliabile perchè
  - genera una eccezione “diversa” dalle altre
- Un’eccezione è un evento “eccezionale”, la terminazione di un thread NON lo è
- Termina l’esecuzione del codice in maniera brutale
  - Cosa succede se l’eccezione è lanciata mentre il codice stava aggiornando due contatori in una sezione critica?
  - Cosa succede se si stava gestendo un’altra eccezione?
- Gestire il “recovery” di una eccezione spesso non è banale
  - Chiudere tutte le risorse, riportarsi in uno stato consistente...
- Come terminare correttamente un thread?

Terminazione di un thread

```
class foo
{
   public volatile bool TerminateThread = false;

   public void ThreadFcn()
   {
      ...
      while ( !this.TerminateThread /* && LoopCount < 1234 */ )
      {
         //do something
         //update LoopCount
         //...
      }
      ...
   }
   ...
}
...
//other thread
fooObject.TerminateThread = true;
MyThread.Join();
...
```

Terminazione di un thread

```
class foo
{
   public AutoResetEvent TerminateEvent = new AutoResetEvent();

   public void ThreadFcn()
   {
      WaitHandle []Handles = new WaitHandle[2];
      Handles[0] = TerminateEvent;
      //Handles[1] = some other event related to IO

      while ( WaitHandle.WaitAny(Handles) == 1 /* && other */)
      {  //the thread must not terminate
         //do something
         ...
      }
      ...
   }
   ...
}
...
//other thread
fooObject.TerminateEvent.Set();
MyThread.Join();
...
```

Considerazioni sui thread

- In quale classe definire il metodo ThreadFcn?
  - Classe dedicata al thread
  - Classe che possiede l'oggetto Thread
- Non registrare più di un metodo sul delegato ThreadStart
- Attenzione alla gestione delle eccezioni nei thread
  - se non sono "catturate" il runtime le segnala, ma il thread principale prosegue l'esecuzione!

Thread e applicazioni grafiche

- Si usano i thread perché l'interfaccia grafica non risulti bloccata durante una lunga elaborazione (>10 ms)
  - Interrogazioni su un Database
  - Scaricamento della posta
- Tutti i metodi degli oggetti grafici DEVONO essere richiamati dal thread principale (quello che gestisce il message loop)
  - Tutte le classi derivanti da System.Windows.Forms.Control

Thread e applicazioni grafiche

- La classe System.Windows.Forms.Control dispone di una serie di metodi per invocare un metod

```
public class Control
{
   public object Invoke(
                     Delegate method,
                     object []args);

   public IAsyncResult BeginInvoke(
                     Delegate method,
                     object []args);

   public object EndInvoke(
                     IAsyncResult async);
...
}
```

Questo metodo riceve un delegato contenente il metodo da eseguire, e i relativi parametri.
La chiamata è bloccante.

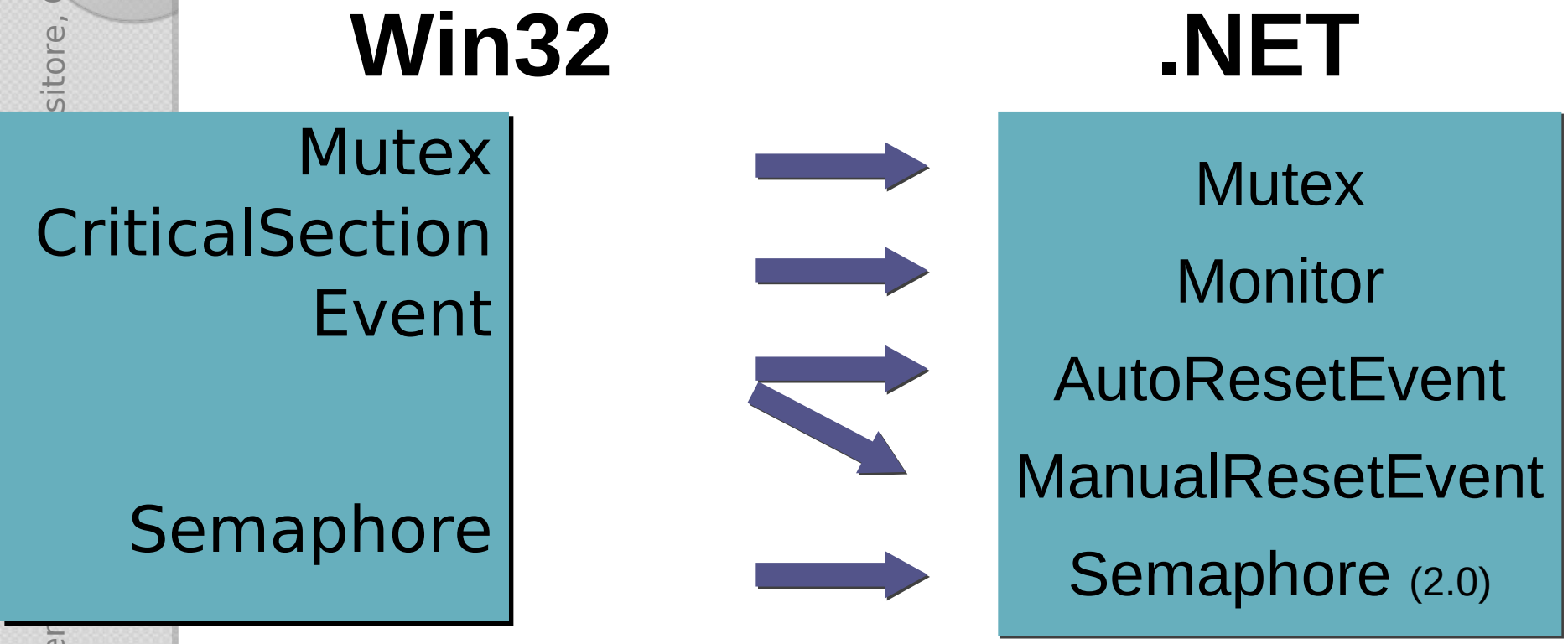
Come Invoke, ma la chiamata NON è bloccante.

Attende la terminazione di una chiamata BeginInvoke.

Sincronizzazione

- Motivazioni
- Proteggere risorse condivise da thread differenti
  - Esempio: accesso ad un contatore
- Gestire l'interazione
  - Esempio: Produttore – consumatore

Oggetti Win32 e .NET



Monitor

- Permette di definire una regione critica di codice (“solo un thread alla volta all’interno dello stesso processo”)

```
public sealed class Monitor
{
   private Monitor();

   public static void Enter(object obj);
   public static void Exit(object obj);

   public static void Wait(object obj);
   public static void Pulse(object obj);
   public static void PulseAll(object obj);
}
```

Non si può istanziare

“Entrata” nella regione critica

“Uscita” dalla regione critica

Condition variable associata al monitor

Monitor

- Ricevono un generico oggetto obj
  - obj è l'entità su cui ci si sincronizza
  - Un solo thread alla volta può entrare in una sezione critica identificata da un certo oggetto obj
  - obj deve essere un reference type!
    - Se viene utilizzato un value type, il codice viene compilato, ma NON è corretto (boxing)

```
void Enter(object obj);
void Exit(object obj);
```

Monitor - esempio

```
class foo
{
  private int a = 0;
  private int b = 0;

  public void MyMethod()
  {
    Monitor.Enter(this);
    a++;
    if (a == 12)
      b++;
    Monitor.Exit(this);
  }
}
```

```
class foo
{
  private int a = 0;
  private int b = 0;

  public void MyMethod()
  {
    try
    {
      Monitor.Enter(this);
      a++;
      if (a == 12)
        b++;
    }
    finally
    {
      Monitor.Exit(this);
    }
  }
}
```

Costrutto C# lock()

- Si utilizza per creare una sezione critica all'interno del codice C#
- Sintassi

```
lock(object obj)
{
  //code
}
```

```
class foo
{
  private int a = 0;
  private int b = 0;

  public void MyMethod()
  {
    lock(this)
    {
      a++;
      if (a == 12)
        b++;
    }
  }
}
```

Costrutto C# lock() - 2

- Si tratta di “syntactic sugar”
- Viene sostituito dal compilatore C# con chiamate a Monitor.Enter(obj)/Exit(obj), inglobate in un blocco try/finally
- Valgono le consid
  Monitor

```
lock(obj)
{
  //code
}
```

```
try
{
  Monitor.Enter(obj);
  //code
}
finally
{
  Monitor.Exit(obj);
}
```

Gerarchia WaitHandle

- Oggetti di sincronizzazione
  - Molto simili agli HANDLE Win32
- Due possibili stati: segnalato e non segnalato

```
public abstract class WaitHandle
{
   public virtual IntPtr Handle{get;set;};

   public virtual bool WaitOne();

   public static bool WaitAll(WaitHandle []handles);
   public static int WaitAny(WaitHandle []handles);
}
```

Proprietà che rappresenta l'handle nativo Win32 (HANDLE) dell'oggetto

Chiamata bloccante, aspetta che l'oggetto sia segnalato

Chiamate bloccanti, aspettano che almeno uno (o tutti) gli oggetti siano segnalati.

I metodi Wait sono analoghi alle chiamate WaitForSingleObject() WaitForMultipleObjects()

Gerarchia WaitHandle

- Mutex
  - Wrapper del Mutex Win32
  - HANDLE hMutex = CreateMutex(...)
- ManualResetEvent
  - Wrapper dell'evento Win32 con reset manuale
  - HANDLE hEvent = CreateEvent(..., TRUE, ...)
- AutoResetEvent
  - Wrapper dell'evento Win32 con reset automatico
  - HANDLE hEvent = CreateEvent(..., FALSE, ...)

Mutex

- Si utilizza per creare sezioni critiche di codice (“un thread alla volta”)

```
public sealed class Mutex : WaitHandle
{
   ...
   public override bool WaitOne();
   public void ReleaseMutex();
   ...
}
```

Utilizzato per acquisire il mutex

Utilizzato per rilasciare il mutex

- È consigliabile usare un blocco try/finally
- Un Mutex acquisito NON è segnalato, un Mutex rilasciato è segnalato

Mutex – Esempio

```
class foo
{
   private int a = 0;
   private int b = 0;
   private Mutex MyMutex = new Mutex();

   public void MyMethod()
   {
      try
      {
         MyMutex.WaitOne();
         a++;
         if (a == 12)
            b++;
      }
      finally
      {
         MyMutex.ReleaseMutex();
      }
   }
}
```

Creo un oggetto Mutex.

NOTA: questo è “syntactic sugar”, questo codice è inserito nel costruttore dal compilatore C#

Acquisisco il Mutex all’interno del blocco try

Rilascio il Mutex all’interno del blocco finally

Mutex – Note

- All'interno dello stesso thread si può acquisire uno stesso Mutex più volte, a patto di rilasciarlo lo stesso numero di volte
- Non si può rilasciare un Mutex acquisito da un altro thread (ApplicationException)
- Un Mutex NON rilasciato al termine di un thread viene rilasciato automaticamente dal runtime

Mutex e Monitor

- Analogie
  - Si utilizzano per creare sezioni critiche
- Differenze
  - Un Monitor si sincronizza su un oggetto
    - Consigliabile per sincronizzazione intra-oggetto
  - Un Mutex è un oggetto a sé stante
    - Consigliabile per sincronizzazione tra oggetti
  - Un Mutex può essere utilizzato per la sincronizzazione tra processi

Eventi

- Sono oggetti che possono essere settati (segnalati) e resettati (non segnalati)

```
public sealed class ManualResetEvent : WaitHandle
public sealed class AutoResetEvent : WaitHandle
{
   ...
   public override bool WaitOne();
   public void Set();
   public void Reset();
   ...
}
```

ManualResetEvent

- Caratteristiche
  - Devono essere resettati (non segnalati) manualmente
  - Tutti i thread in attesa (Wait) vengono svegliati non appena l'evento viene segnalato
- Utilizzi
  - Si usano per notificare più thread di un certo evento
  - NotificationEvent nel kernel di Windows
  - Esempio: far partire più thread simili "nello stesso istante"

AutoResetEvent

- Caratteristiche
  - Il primo thread in attesa sulla Waiting Queue viene svegliato, e l'evento viene resettato automaticamente
  - Se nessun thread è in attesa sull'evento, questo rimane segnalato
  - Molto simile a un semaforo a 1 posizione
- Utilizzi
  - Sincronizzazione vera e propria tra thread
  - SynchronizationEvent nel kernel di Windows
  - Esempio: problema del produttore-consumatore

Altri classi relative al threading

- System.Threading.ThreadPool
  - Permette l'accesso al thread pool nativo attraverso i propri metodi statici
    - QueueUserWorkItem
  - Si ottengono fuzionalità simili utilizzando il costrutto BeginInvoke() su un delegato
- System.Threading.ReaderWriterLock
  - Si usa per implementare il pattern dei Readers e Writers
  - Si tratta di un insieme di mutex e/o eventi inglobati in una classe

Altre classi relative al threading

- System.Threading.Interlocked
  - Contiene una serie di metodi statici per effettuare operazioni matematiche atomiche sugli interi
  - Incremento, decremento, test-and-set, scambio
  - Analoga ai metodi InterlockedXXX dell'API Win32
- System.Threading.Timer
  - Richiama un metodo (tramite delegato) ad un istante predefinito
  - Utilizza un thread del ThreadPool (e il delegato è quindi chiamato in un thread diverso dal thread principale)