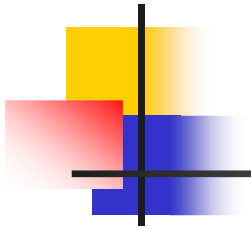
# Tecniche di Programmazione avanzata

*Corso di Laurea Specialistica in Ingegneria Telematica*
*Università Kore – Enna – A.A. 2009-2010*

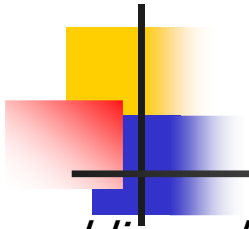
Alessandro Longheu
*http://www.diit.unict.it/users/alongheu*
*alessandro.longheu@diit.unict.it*

# Il linguaggio C#
# Thread



# Thread

- Il threading consente di eseguire elaborazioni simultanee in un programma in C#.
- Lo spazio dei nomi System.Threading fornisce classi e interfacce che supportano la programmazione multithreading
  - la creazione e l'avvio di nuovi thread,
  - la sincronizzazione di più thread,
  - la sospensione e l'interruzione di thread.
- Per incorporare il threading nel codice C#, è sufficiente creare una funzione da eseguire all'esterno del thread principale e puntarvi un nuovo oggetto Thread
- La strategia più diffusa consiste nell'utilizzare i thread di lavoro per eseguire attività lunghe o in cui il tempo riveste importanza ma che non richiedono molte delle risorse utilizzate da altri thread.



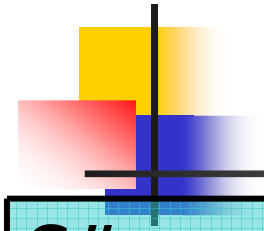
# Metodi per Thread

```
public sealed class Thread {
   public static Thread CurrentThread { get; }    // metodi statici
   public static void Sleep(int milliSeconds) {...}
   ...
   public Thread(ThreadStart startMethod) {...} // creazione dei thread
   public string Name { get; set; } // proprietà
   public ThreadPriority Priority { get; set; }
   public ThreadState ThreadState { get; }
   public bool IsAlive { get; }
   public bool IsBackground { get; set; }
   ...
   public void Start() {...} // metodi
   public void Suspend() {...}
   public void Resume() {...}
   public void Join() {...}  // il chiamante aspetta che il thread termini
   public void Abort() {...}      // emette ThreadAbortException
   public void Interrupt() {...} // chiamato solo nello stato WaitSleepState
   ...
}
public delegate void ThreadStart();      // metodo senza parametri
public enum ThreadPriority {Normal, AboveNormal, BelowNormal, Highest, Lowest}
public enum ThreadState {Unstarted, Running, Suspended, Stopped, Aborted, ...}
```

# Esempio

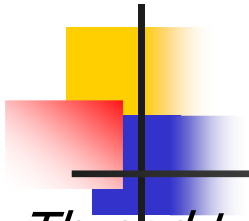
- Due thread che stampano caratteri
- Notare che il metodo Print non è statico quindi è necessario istanziare la classe che lo contiene

```
using System;
using System.Threading;

class Printer {
   char ch;
   int sleepTime;
   public Printer(char c, int t) {ch = c; sleepTime = t;}
   public void Print() {
     for (int i = 1; i < 100; i++) {
          Console.Write(ch);
          Thread.Sleep(sleepTime);        }    }    }

class Test {
   static void Main() {
     Printer a = new Printer('.', 10);
     Printer b = new Printer('*', 100);
     new Thread(new ThreadStart(a.Print)).Start();
     new Thread(new ThreadStart(b.Print)).Start();
   }}
```



# Confronto con Java

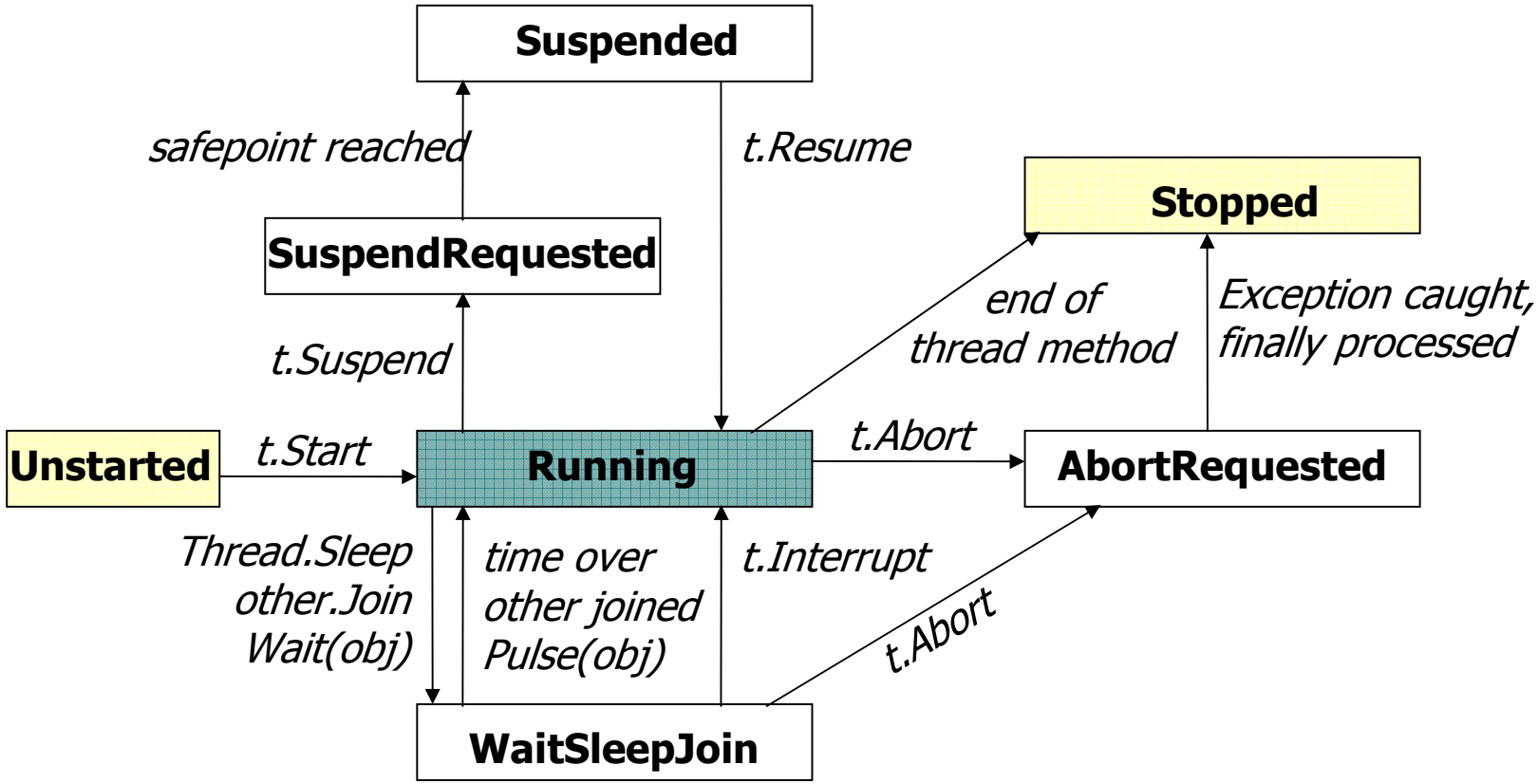
| C# | Java |
|---|---|
| void P() {<br>... *thread actions* ...<br>}<br>Thread t = new<br>Thread(newThreadStart(P)); | class MyThread extends Thread {<br>public void run() {<br>... *thread actions* ...<br>}}<br>Thread t = new MyThread(); |
| Non è necessario estendere Thread | I Thread utente sono una estensione di Thread |
| Qualsiasi metodo di tipo void senza parametri può essere un thread | Bisogna implementare il metodo run |
| *ThreadAbortException* può essere catturata, ma è rilanciata alla fine del gestore, tranne se è invocato *ResetAbort*. Tutti i blocchi finally sono eseguiti. | Il metodo stop è deprecato |

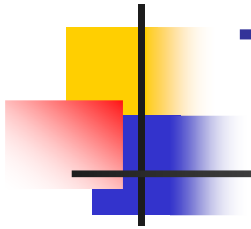


# Thread

```
Thread t = new Thread(new ThreadStart(P));
Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority,
   t.ThreadState);
t.Name = "Worker"; t.Priority = ThreadPriority.BelowNormal;
t.Start();
Thread.Sleep(1);
Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority,
   t.ThreadState);
t.Suspend();
Thread.Sleep(1);
Console.WriteLine("state={0}", t.ThreadState);
t.Resume();
Console.WriteLine("state={0}", t.ThreadState);
t.Abort();
Thread.Sleep(1);
Console.WriteLine("state={0}", t.ThreadState);
Output
   name=, priority=Normal, state=Unstarted
   name=Worker, priority=BelowNormal, state=Running
   state=Suspended          state=Running       state=Stopped
```

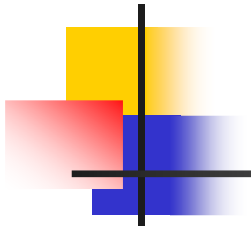
# Diagramma di stato di un Thread





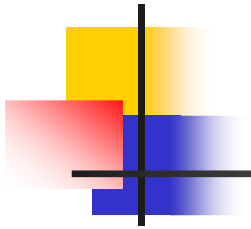
# Thread

- Creazione di una classe che esegue il lavoro (Executor)
- Implementazione di un metodo void xxx(…)
- Istanza di Executor
- Istanza del Thread inizializzato su Executor
- Esecuzione di start()



# Thread

- Modificatore **volatile**
  - La parola chiave **volatile** avvisa il compilatore che più thread accederanno al membro dati
  - Il compilatore non deve ottimizzare questo membro.
- Proprietà isAlive
- Metodo Abort()
  - Blocca il thread
- Metodo Join()
  - Sincronizza con il thread
  - Use this method to ensure a thread has terminated. The caller will block indefinitely if the thread does not terminate.



# Join

```
using System;
using System.Threading;

class Test {
   static void P() {
     for (int i = 1; i <= 20; i++) {
           Console.Write('-');
           Thread.Sleep(100);
     }    }
   static void Main() {
     Thread t = new Thread(new ThreadStart(P));
     Console.Write("start");
     t.Start();
     t.Join();       // si sincronizza con la fine del thread
     Console.WriteLine("end");
   }
}

Output
start----------end
```

- Notare che il metodo P stavolta è statico quindi NON è necessario istanziare la classe che lo contiene

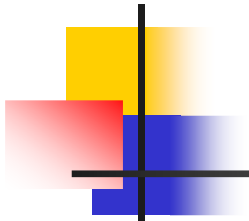
# Gestione dell'Abort

- Abort lancia una eccezione, gestibile dallo stesso thread

```
using System; using System.Threading;
class Test {
  static void P() {
        try {                    try {                    try {
                        while (true) ;
  } catch (ThreadAbortException) { Console.WriteLine("-- inner aborted");
 }} catch (ThreadAbortException) { Console.WriteLine("-- outer aborted");
 }} finally { Console.WriteLine("-- finally"); }       }
  static void Main(string[] arg) {
        Thread t = new Thread(new ThreadStart(P));
        t.Start(); Thread.Sleep(1);
        t.Abort(); t.Join(); Console.WriteLine("done");       }    }
```

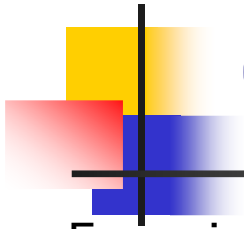
Output

```
-- inner aborted
-- outer aborted
-- finally
done
```



# Creazione di un thread

- La creazione di un nuovo oggetto **Thread** determina la generazione di un nuovo thread gestito. La classe **Thread** dispone di costruttori che accettano un delegato ThreadStart o ParameterizedThreadStart. Il delegato esegue il wrapping del metodo richiamato dal nuovo thread quando viene eseguita una chiamata al metodo Start.
- Il metodo Start restituisce immediatamente un valore, spesso prima dell'avvio effettivo del nuovo thread. È possibile utilizzare le proprietà ThreadState e IsAlive per determinare lo stato del thread, ma queste proprietà non dovrebbero mai essere utilizzate per la sincronizzazione.
- È possibile passare parametri ad un thread: il delegato **ParameterizedThreadStart** offre una tecnica semplice per passare un oggetto contenente dati a un thread quando viene chiamato l'overload del metodo System.Threading.Thread.Start(System.Object). Un'alternativa (modo 2) consiste nell'incapsulare la routine del thread e i dati in una classe di supporto e nell'utilizzare il delegato **ThreadStart**.
- Nessuno di questi delegati restituisce un valore perché non è possibile definire una posizione per la restituzione dei dati da una chiamata asincrona. Per recuperare i risultati di un metodo di thread, è possibile utilizzare un metodo di callback



# Creazione di un thread

- Esempio di **passaggio parametri ad un thread tramite modo 1:**

```
using System; using System.Threading;
public class Work      {    public static void Main(){    Thread newThread =
    new Thread(new ParameterizedThreadStart(Work.DoWork));
    // Use the overload of the Start method with an Object parameter
    newThread.Start(42);
    Work w = new Work();
    newThread = new Thread(new ParameterizedThreadStart(w.DoMoreWork));
    newThread.Start("The answer.");    }
    public static void DoWork(object data)    {
    Console.WriteLine("Static thread procedure. Data='{0}'",  data);    }
    public void DoMoreWork(object data)    {
        Console.WriteLine("Instance thread procedure. Data='{0}'", data);   }}
```

- Questa tecnica non è tuttavia indipendente dai tipi, poiché consente di passare al metodo **Thread.Start(Object)** qualsiasi oggetto. Un modo più efficiente per passare i dati a una routine del thread consiste nell'inserire sia la routine che i campi dati in un oggetto di lavoro (modo 2)

# Creazione di un thread

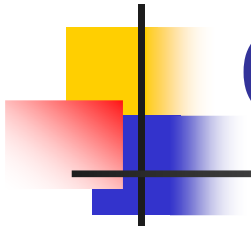
- Esempio di **passaggio parametri ad un thread tramite modo 2:**

```
using System;
using System.Threading;
public class ThreadWithState {
    private string boilerplate;    private int value;
public ThreadWithState(string text, int number)
    {       boilerplate = text;        value = number;    }
public void ThreadProc()
    {       Console.WriteLine(boilerplate, value);    }} // END support class
public class Example {
    public static void Main()
    {       // Supply the state information required by the task.
        ThreadWithState tws = new ThreadWithState(
            "This report displays the number {0}.", 42);
Thread t = new Thread(new ThreadStart(tws.ThreadProc));
    t.Start();
    Console.WriteLine("Main thread does some work, then waits.");
    t.Join();
    Console.WriteLine("Independent task completed; main thread ends.");
    }}
```

# Creazione di un thread

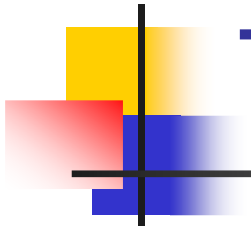
- Esempio di **recupero risultati da un thread**: nel costruttore della classe contenente i dati e il metodo di thread viene accettato anche un delegato che rappresenta il metodo di callback.
- Con Callback si intende una funzione che viene richiamata da un' altra funzione. Il procedimento comporta il passaggio della prima funzione come parametro alla funzione chiamante. In questo modo il chiamante può realizzare un compito specifico, pur non essendo noto al tempo di compilazione
- Al termine del thread, si richiama il delegato

```
using System; using System.Threading;
public class ThreadWithState {
    private string boilerplate;        private int value;
    // Delegate to execute the callback when the task is complete.
    private ExampleCallback callback;
    public ThreadWithState(string t, int n, ExampleCallback cDelegate)
    {      boilerplate = t;      value = n;      callback = cDelegate;    }
    public void ThreadProc()
    {      Console.WriteLine(boilerplate, value);
        if (callback != null)          callback(1);    }} // END support class
```



# Creazione di un thread

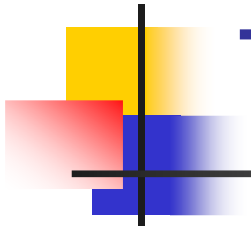
```
...
public delegate void ExampleCallback(int lineCount);

public class Example {

public static void Main()
{  ThreadWithState tws = new ThreadWithState(
   "Display number {0}.", 42,new ExampleCallback(ResultCallback));
   Thread t = new Thread(new ThreadStart(tws.ThreadProc));
   t.Start();
   Console.WriteLine("Main thread does some work, then waits.");
   t.Join();
   Console.WriteLine("Indep. task completed; main thread ends.");     }

public static void ResultCallback(int lineCount)
   {        Console.WriteLine("Independent  task  printed  {0}  lines.",
   lineCount);    }}
```



# Thread

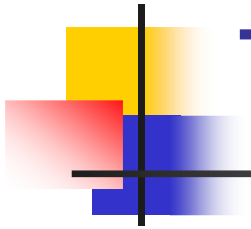
Sospensione thread usando metodi di System.Threading.Thread:

- Il metodo **Sleep**, blocca subito il thread corrente per il numero di millisecondi specificato, cedendo il resto della porzione di tempo a un altro thread. Un thread non può chiamare Sleep su un altro.
- È possibile interrompere un thread in attesa chiamando il metodo **Interrupt** sul thread bloccato per generare un'eccezione ThreadInterruptedException. Il thread dovrebbe intercettare l'eccezione ed eseguire le operazioni appropriate per continuare a funzionare. Se il thread ignora l'eccezione, l'ambiente di esecuzione la intercetta e interrompe il thread.



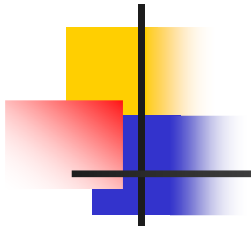
# Thread

- È' inoltre possibile sospendere un thread con **Suspend**. Quando un thread chiama Suspend su se stesso, la chiamata bloccherà il thread fino a quando non verrà ripreso da un altro thread. Quando un thread chiama Suspend su un altro, la chiamata non è bloccante e provoca la sospensione dell'altro thread. Se si chiama il metodo **Resume**, un altro thread uscirà dallo stato di sospensione e riprenderà l'esecuzione. A differenza di Sleep, Suspend non determina l'interruzione immediata: il CLR deve attendere che il thread abbia raggiunto un punto sicuro (un punto nell'esecuzione in corrispondenza del quale è possibile eseguire le operazioni di Garbage Collection).
- Suspend e Resume non si basano sulla cooperazione del thread che viene controllato, sono estremamente intrusivi e possono causare alle applicazioni seri problemi come i deadlock, che si verificano ad esempio se si sospende un thread che contiene una risorsa richiesta da un altro thread. In .NET 2.0, Suspend e Resume sono (infine) obsoleti.



# Thread

- Lo scheduling dei thread viene stabilito dalla priorità.
- I dettagli dello scheduler utilizzato per determinare l'ordine in cui i thread vengono eseguiti **varia in base al sistema operativo**.
  - In alcuni, viene sempre pianificata per prima l'esecuzione del thread con la priorità più alta, tra quelli di cui è possibile l'esecuzione. Se più thread con la stessa priorità sono tutti disponibili, l'utilità di pianificazione scorre in ciclo i thread a quella priorità, assegnando a ciascuno di essi una porzione di tempo fissa in cui effettuare l'esecuzione (**time slicing**).
  - Se un thread con una priorità più alta è disponibile, non verranno eseguiti i thread a priorità più bassa. Quando non sono più disponibili thread eseguibili a una determinata priorità, si passa alla priorità più bassa successiva e pianifica l'esecuzione dei thread che hanno quella priorità.
  - Se diventa eseguibile un thread a priorità più alta, il thread a priorità più bassa viene interrotto e viene nuovamente consentita l'esecuzione del primo.
- Nel sistema operativo possono inoltre essere regolate dinamicamente le priorità del thread man mano che l'interfaccia utente di un'applicazione viene spostata tra il primo piano e il background. In altri OS è possibile usare altri algoritmi.
- Per la gestione dei thread in .NET vs Windows, consultare http://msdn.microsoft.com/it-it/library/74169f59(VS.80).aspx



# ThreadPool

Spunti per esercitazione:

- Realizzare un programma che dato un array di elementi, calcoli in parallelo il massimo, il minimo e la media, e stampi i risultati cosi' ottenuti
- Si realizzi un programma che lancia n thread per cercare un elemento in una matrice di n per n elementi. Ognuno dei thread cerca l'elemento in una delle righe della matrice. Non appena un thread ha trovato l'elemento cercato, rende note agli altri thread le coordinate dell'elemento e tutti i thread terminano.
- Risolvere il problema dei cinque filosofi
- Realizzare un server che rimane in ascolto di richieste dai client (simulare il tutto in localhost); ad ogni richiesta di un client, sganciare un thread che ne soddisfi le richieste, mentre il thread principale torna in ascolto di nuove connessioni