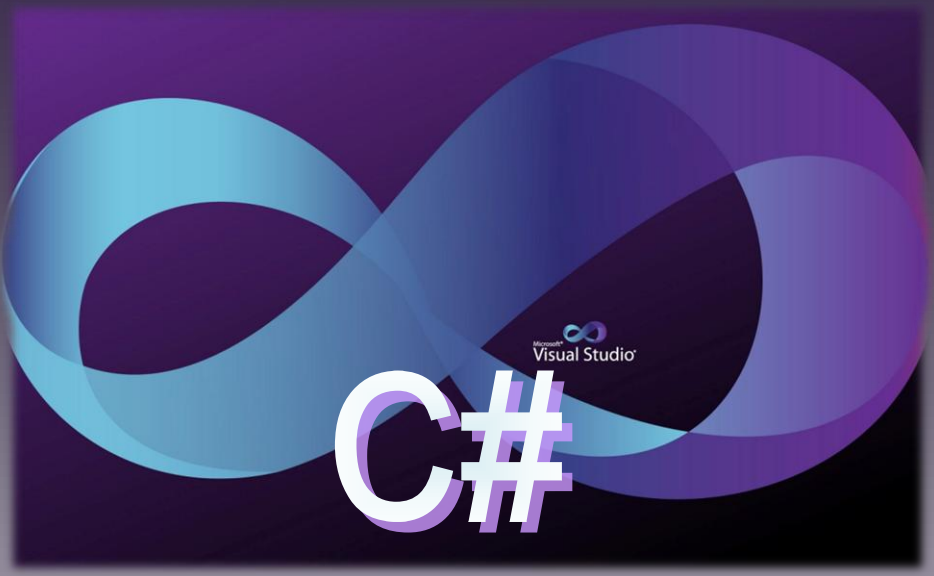
ISTITUTO TECNICO INDUSTRIALE
G. M. ANGIOY
SASSARI



CORSO DI PROGRAMMAZIONE

PROCESSI LEGGERI O THREADS

DISPENSA 17.03

17-03_Thread_[09]

Versione del: 07/11/2015
Revisione numero: 09

Prof. Andrea Zoccheddu
Dipartimento di Informatica

DIPARTIMENTO
INFORMATICA E TELECOMUNICAZIONI

PROCESSI LEGGERI

THREADS

PROCESSO, TASK E THREAD

PROGRAMMI, PROCESSI E PROCESSI LEGGERI

url

http://it.wikipedia.org/wiki/Processo_(informatica)

In Informatica per processo si intende un'istanza di un programma in esecuzione in modo sequenziale. Più precisamente è un'attività controllata da un programma che si svolge su un processore in genere sotto la gestione o supervisione del rispettivo sistema operativo.

Un programma è costituito dal codice oggetto generato dalla compilazione del codice sorgente, ed è normalmente salvato sotto forma di uno o più file. Esso è un'entità statica, che rimane immutata durante l'esecuzione.

Il processo è l'entità utilizzata dal sistema operativo per rappresentare una specifica esecuzione di un programma. Esso è quindi un'entità dinamica, che dipende dai dati che vengono elaborati, e dalle operazioni eseguite su di essi. Il processo è quindi caratterizzato, oltre che dal codice eseguibile, dall'insieme di tutte le informazioni che ne definiscono lo stato, come il contenuto della memoria indirizzata, i thread, i descrittori dei file e delle periferiche in uso.

L'uso dell'astrazione dall'hardware è necessario al sistema operativo per realizzare la multiprogrammazione (multitasking).

Definizione

Un processo è un'istanza di un programma in esecuzione su un elaboratore elettronico.

Il processo per poter avanzare deve essere affidato ad un processore (CPU).

Su un computer possono esserci più processi attivi in esecuzione concorrente.

Quando si possono avere più processi in concorrenza si parla di multitasking.

PROCESSI E THREAD

url

http://it.wikipedia.org/wiki/Processo_(informatica)

Il concetto di processo è associato, ma comunque distinto da quello di thread (abbreviazione di `thread of execution`, filo dell'esecuzione) con cui si intende invece l'unità granulare in cui un processo può essere suddiviso (sottoprocesso) e che può essere eseguito a divisione di tempo o in parallelo ad altri thread da parte del processore.

In altre parole, un thread è una parte del processo che viene eseguita in maniera concorrente ed indipendente internamente allo stato generale del processo stesso. Il termine inglese rende bene l'idea, in quanto si rifà visivamente al concetto di fune composta da vari fili attorcigliati: se la fune è il processo in esecuzione, allora i singoli fili che la compongono sono i thread.

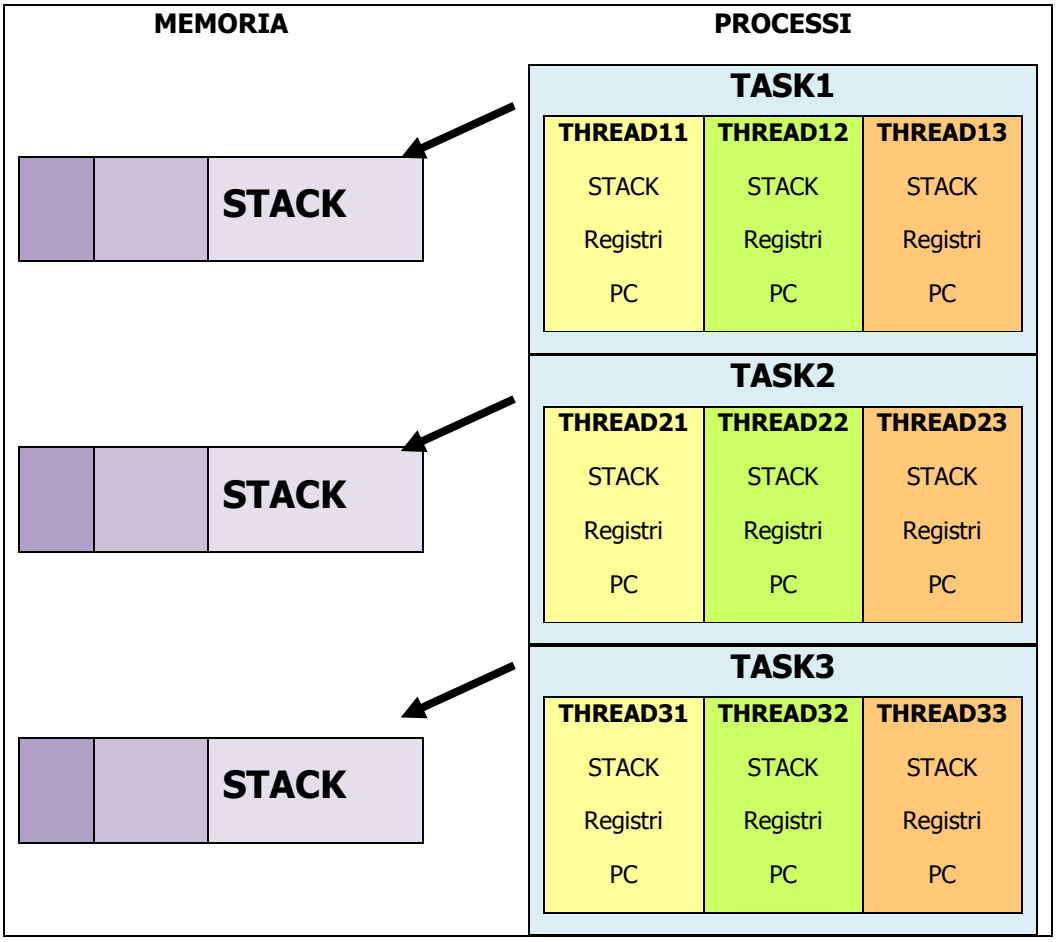
Un processo ha sempre almeno un thread (se stesso), ma in alcuni casi un processo può avere più thread che vengono eseguiti in parallelo.

Definizione

Un processo può essere formato da più frammenti di programma autonomi, detti Thread.

Un processo ha almeno un thread, se stesso. In alcuni casi un processo può affidare parti del compito ad altri pezzi di programma che, pur autonomi, non sono indipendenti dal processo.

SCHEMA DI FUNZIONAMENTO DEI TASK E DEI THREAD



Un task è sempre associato ad una porzione di memoria che deve trovare una corrispondenza in RAM nel momento in cui il task procede con la sua esecuzione. Un task è composto da almeno un thread; ciascun thread possiede i riferimenti per stack (la pila dei dati memorizzati in una parte della ram associata), i dati dei registri (con cui impostare i registri del processore al momento della sua esecuzione), il Program Counter (l'indirizzo della riga del programma da eseguire).

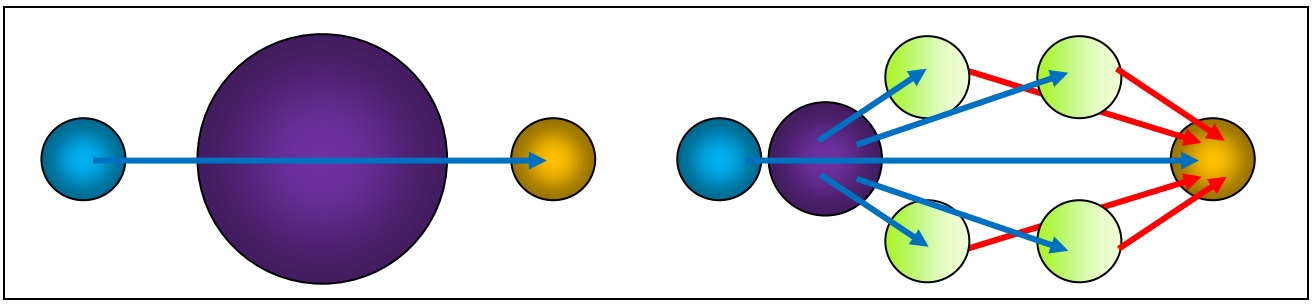
Da quanto sopra illustrato consegue che la RAM è condivisa dai thread (quindi i diversi thread di un singolo task hanno la ram in comune, come risorsa condivisa), Viceversa ciascun thread ha un suo codice (programma eseguibile) separato e gestisce singolarmente e in autonomia la rispettiva esecuzione; quindi in teoria ogni thread procede per conto proprio, distintamente dagli altri e dal thread principale (thread primario).

I THREAD IN VISUAL C#

Per impostazione predefinita, i programmi C# comprendono un unico thread, che esegue il codice del programma iniziando e terminando con il metodo Main.

Ogni comando eseguito da Main, in modo diretto o indiretto, viene eseguito dal thread predefinito, ovvero il `thread primario`, che termina quando viene restituito Main.

È tuttavia possibile creare e utilizzare `thread ausiliari` per eseguire il codice in parallelo con il thread primario. Questi thread vengono definiti `thread di lavoro`. I thread di lavoro possono essere utilizzati per eseguire attività lunghe o in cui il tempo riveste molta importanza, senza bloccare il thread primario. Ad esempio, vengono utilizzati per eseguire attività in background nelle applicazioni desktop, in modo tale che il thread principale, che gestisce gli elementi dell'interfaccia utente, garantisca tempi di risposta ottimali per le azioni dell'utente. Quando un processo è costituito da più di un thread, si parla di `multithreading`.



Il multithreading consente di risolvere i problemi legati alla velocità effettiva e alla velocità di risposta, ma può comportare problemi di condivisione delle risorse, quali i `deadlock` (Stallo - situazione in cui due o più processi si bloccano a vicenda) e le `race condition` (mancata temporizzazione dei processi concorrenti). L'utilizzo di più thread è consigliato per le attività che richiedono risorse differenti, ad esempio un browser Web, che usa un thread distinto per scaricare ogni immagine in una pagina Web che contiene più immagini.

L'assegnazione di più thread a un'unica risorsa potrebbe generare errori di sincronizzazione e se i thread vengono bloccati di frequente in attesa di altri thread lo scopo del multithreading risulta vanificato. 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. Ovviamente alcune risorse del programma devono essere accessibili a più thread. In questi casi lo spazio dei nomi `System.Threading` fornisce le classi per la sincronizzazione dei thread.

CREARE E TERMINARE THREAD

Nell'esempio che analizzeremo nel prossimo PROGETTO GUIDATO viene illustrato come creare un thread ausiliario, o di lavoro, da utilizzare per eseguire l'elaborazione in parallelo con il thread primario. Vengono inoltre descritte le procedure per mettere un thread in attesa di un altro e per terminare correttamente un thread.

Nell'esempio viene creata una classe denominata `Worker` contenente il metodo `DoWork` che verrà eseguito dal thread di lavoro. Questa è essenzialmente la funzione `Main` per il thread di lavoro. Quando verrà eseguito, il thread di lavoro chiamerà questo metodo e terminerà automaticamente quando il metodo verrà restituito.

Il metodo DoWork è analogo al seguente:

Visual C#
```
public void DoWork()
{
    while (!_shouldStop)
    {
        Console.WriteLine("unità di lavoro: sto lavorando...");
    }
    Console.WriteLine("unità di lavoro: ho terminato.");
}
```

La classe Worker contiene un altro metodo utilizzato per indicare che DoWork deve essere restituito. Questo metodo, denominato RequestStop, è analogo al seguente:

Visual C#
```
public void RequestStop()
{
    _shouldStop = true;
}
```

Il metodo RequestStop assegna semplicemente il membro dati _shouldStop a true. Poiché questo membro dati è controllato dal metodo DoWork, si ottiene l'effetto indiretto di causare la restituzione di DoWork terminando in questo modo il thread di lavoro. È tuttavia importante tenere presente che DoWork e RequestStop verranno eseguiti da thread differenti. DoWork viene eseguito dal thread di lavoro, mentre RequestStop viene eseguito dal thread primario, quindi il membro dati _shouldStop viene dichiarato volatile, come riportato di seguito:

Visual C#
```
private volatile bool _shouldStop;
```

La parola chiave volatile avvisa il compilatore che più thread accederanno al membro dati _shouldStop e che pertanto non deve formulare ipotesi di ottimizzazione sullo stato di questo membro.

L'utilizzo di volatile con il membro dati _shouldStop consente di accedere in modo sicuro a questo membro da più thread senza utilizzare le tecniche formali di sincronizzazione dei thread, ma solo perché _shouldStop è un valore bool. Questo significa che sono necessarie solo singole operazioni atomiche per modificare _shouldStop. Se tuttavia questo membro dati fosse una classe, una struttura o una matrice, l'accesso da più thread genererebbe un danneggiamento intermittente dei dati. Si consideri un thread che cambia i valori di una matrice. In Windows i thread vengono interrotti regolarmente per consentire l'esecuzione di altri thread. Pertanto, questo thread potrebbe essere interrotto dopo l'assegnazione di valori ad alcuni elementi della matrice ma prima dell'assegnazione di valori ad altri elementi. Poiché la matrice ha in questo caso uno stato non previsto dal programmatore, un altro thread che legge la stessa matrice potrebbe generare un errore. Prima di creare il thread di lavoro, la funzione Main crea un oggetto Worker e un'istanza di Thread. L'oggetto thread viene configurato per utilizzare il metodo Worker.DoWork come punto di ingresso passando al costruttore Thread un riferimento a questo metodo, come riportato di seguito:

Visual C#
```
Worker workerObject = new Worker();
Thread workerThread = new Thread(workerObject.DoWork);
```

A questo punto, anche se l'oggetto thread di lavoro esiste ed è configurato, il thread di lavoro effettivo non è ancora stato creato. Ciò si verifica solo quando un metodo (eventualmente Main) chiama il metodo Start:

Visual C#
```
workerThread.Start();
```
A questo punto il sistema avvia l'esecuzione del thread di lavoro, ma in modo asincrono rispetto al thread primario. La funzione Main continua infatti ad eseguire immediatamente il codice mentre il thread di lavoro viene sottoposto contemporaneamente a inizializzazione. Per evitare che Main tenti di terminare il thread di lavoro prima che venga eseguito, la funzione Main esegue un ciclo finché la proprietà `IsAlive` dell'oggetto thread di lavoro non viene impostata su true:

Visual C#
```
while (!workerThread.IsAlive);
```
Successivamente il thread primario viene interrotto brevemente con una chiamata a `Sleep`. In questo modo la funzione DoWork del thread di lavoro eseguirà il ciclo all'interno del metodo DoWork per alcune iterazioni prima che la funzione Main esegua altri comandi:

Visual C#
```
Thread.Sleep(1);
```
Trascorso un millisecondo, Main segnala all'oggetto thread di lavoro che deve terminare utilizzando il metodoWorker.`RequestStop` descritto in precedenza:

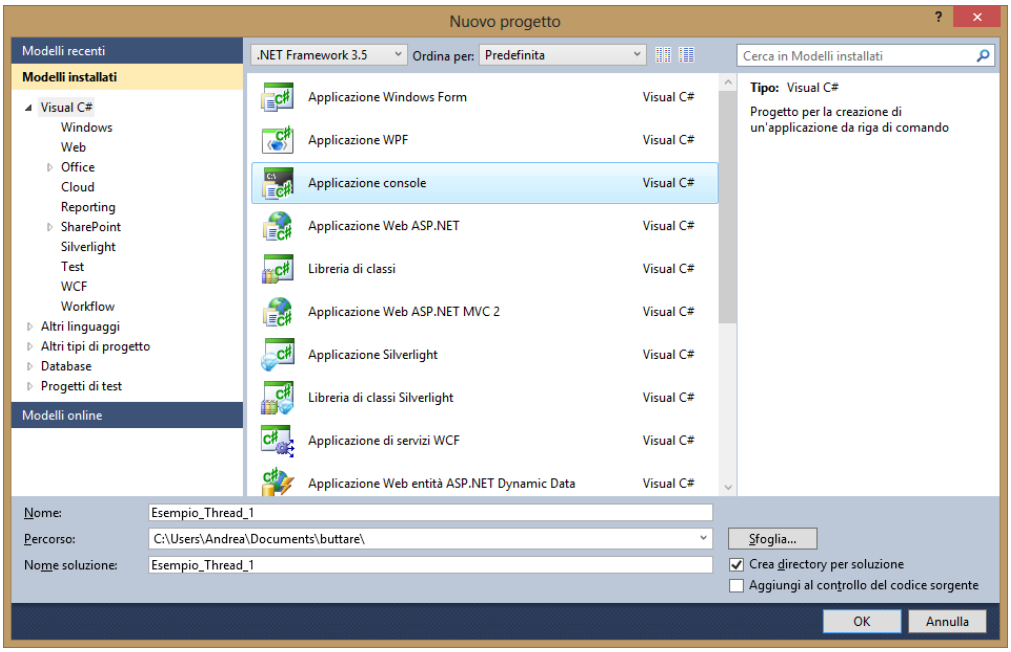
Visual C#
```
workerObject.RequestStop();
```
È inoltre possibile terminare un thread da un altro thread utilizzando una chiamata a `Abort`. In questo modo il thread interessato viene terminato in modo forzato anche se non ha completato l'attività e non consente la pulitura delle risorse. È preferibile utilizzare la tecnica illustrata in questo esempio. Infine la funzione Main chiama il metodo `Join` sull'oggetto thread di lavoro. Tramite questo metodo il thread corrente si blocca oppure attende finché non termina il thread rappresentato dall'oggetto. Pertanto Join non verrà restituito finché non viene restituito, e quindi terminato, il thread di lavoro:

Visual C#
```
workerThread.Join();
```
A questo punto esiste solo il thread primario che esegue Main. Visualizza un messaggio finale e quindi viene restituito e terminato.

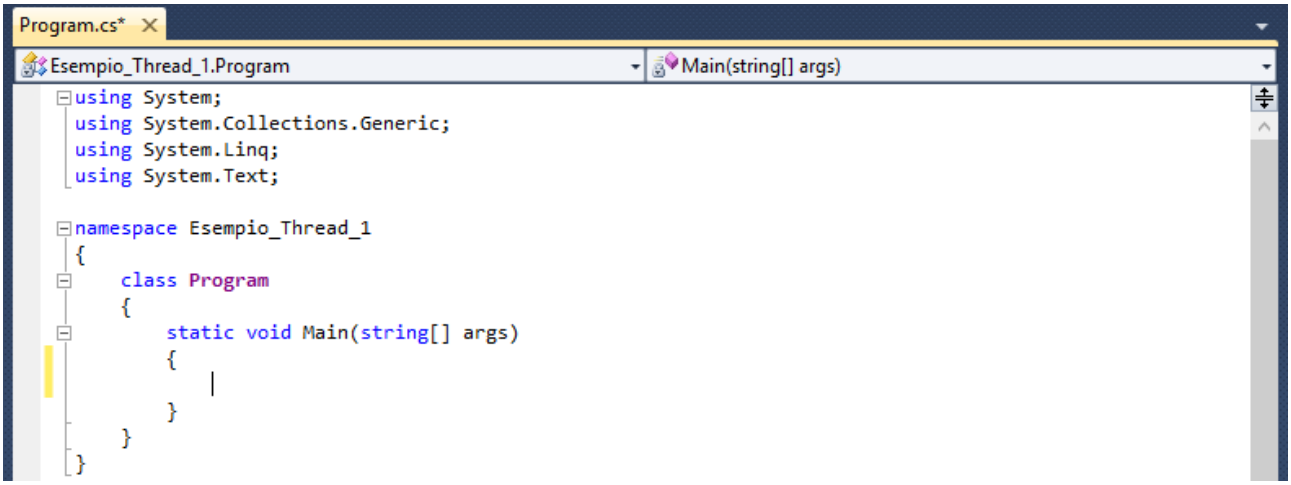
Vediamo in dettaglio il progetto:

PROGETTO GUIDATO (CONSOLE)

- Crea un nuovo progetto in modalità Console



- Appare la consueta interfaccia Console:



- Aggiungere le seguenti librerie al programma:

Visual C#

```
//librerie da aggiungere
using System.Threading;
```

- Dichiarare la classe Worker con il metodo DoWork come nel seguente esempio:

Visual C#

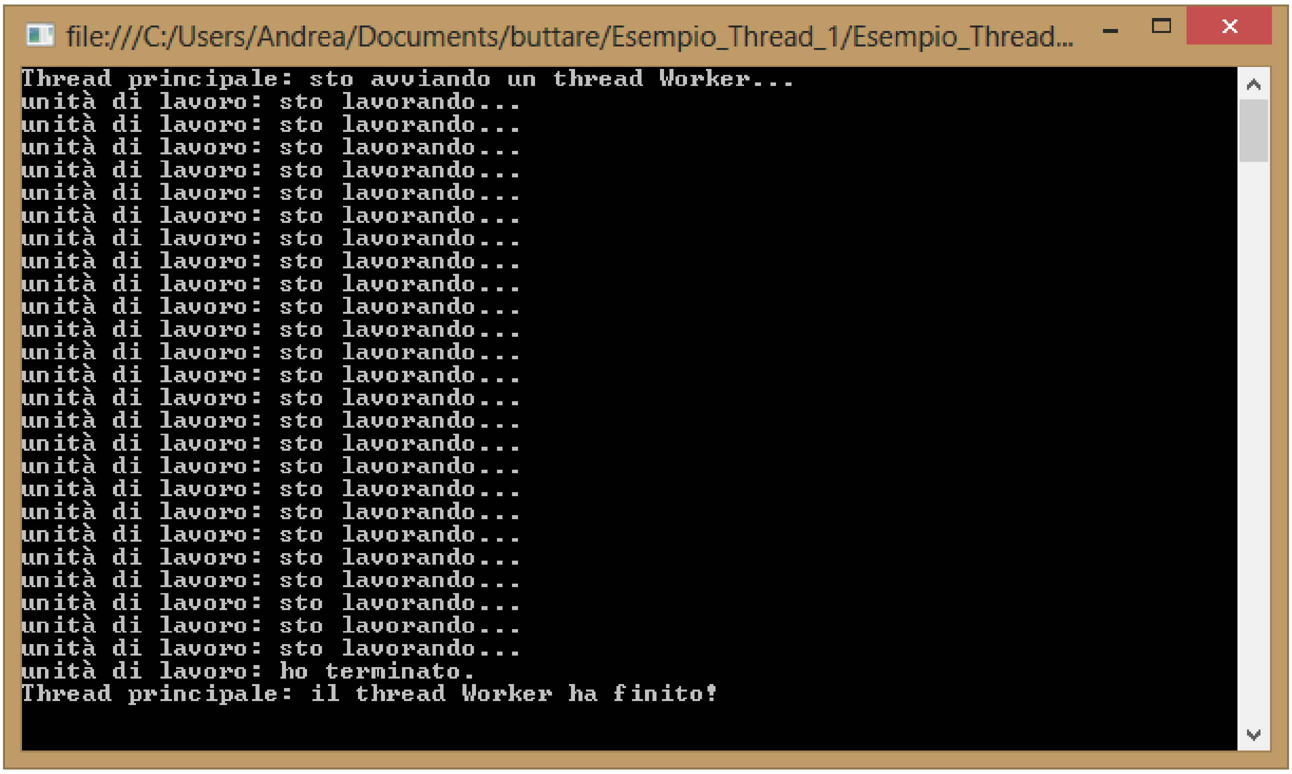
```
class Worker
{
    private volatile bool _shouldStop;

    public Worker()
    {
    }

    public void DoWork() // Questo metodo sarà chiamato dal thread ausiliario
    {
        while (!_shouldStop)
        {
            Console.WriteLine("unità di lavoro: sto lavorando...");
        }
        Console.WriteLine("unità di lavoro: ho terminato.");
    }

    public void RequestStop()
    {
        _shouldStop = true;
    }

}
```

- Modificare il metodo Main come segue;

Visual C#

```
static void Main(string[] args)
{
    // Crea l'oggetto thread, ma non lo avvia (ancora)
    Worker workerObject = new Worker(); // istanza di Worker

    // Thread su metodo DoWork
    Thread workerThread = new Thread(workerObject.DoWork);

    // Avvio del thread di tipo Worker
    workerThread.Start();
    Console.WriteLine("Thread principale: sto avviando un thread Worker...");

    // Ciclo fino a quando è attivo thread di tipo Worker
    while (!workerThread.IsAlive) ;

    // Attesa del thread principale (in sonno) per 500 millisecondi
    // per permettere al thread di tipo Worker di fare qualcosa
    Thread.Sleep(500);

    // Richiesta che il thread di tipo Worker si arresti
    workerObject.RequestStop();

    // Uso del metodo Join per fermare il thread corrente
    // fino a quando l'oggetto thread termina
    workerThread.Join();
    Console.WriteLine("Thread principale: il thread Worker ha finito!");

    Console.ReadLine();// finestra output attiva fino al clic su Invio
}
```

- Nella pagina seguente è riportato l'immagine dell'intero progetto;
- Lancia in esecuzione il progetto e osserva la finestra di output (simile alla seguente):




```
Thread principale: sto avviando un thread Worker...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: sto lavorando...
unità di lavoro: ho terminato.
Thread principale: il thread Worker ha finito!
```

- Ecco l'immagine dell'intero progetto;:

Program.cs

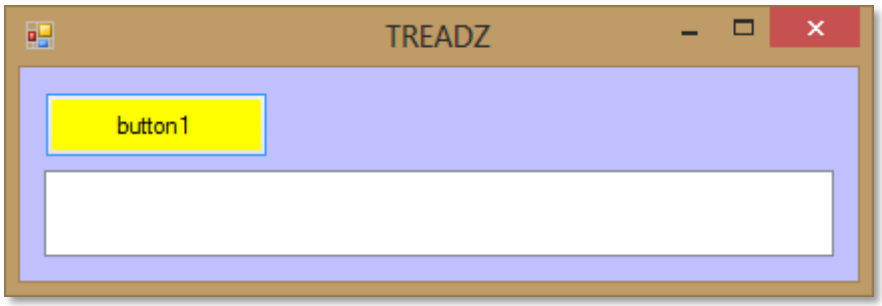
Esempio_Thread_1.Program | Main(string[] args)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

//librerie da aggiungere
using System.Threading;

namespace Esempio_Thread_1
{
    class Program
    {
        class Worker
        {
            private volatile bool _shouldStop;

            public Worker()
            {
            }

            public void DoWork() // Questo metodo sarà chiamato dal thread ausiliario
            {
                while (!_shouldStop)
                {
                    Console.WriteLine("unità di lavoro: sto lavorando...");
                }
                Console.WriteLine("unità di lavoro: ho terminato.");
            }

            public void RequestStop()
            {
                _shouldStop = true;
            }

        }
```

```
        static void Main(string[] args)
        {
            // Crea l'oggetto thread, ma non lo avvia (ancora)
            Worker workerObject = new Worker(); // istanza di Worker
            Thread workerThread = new Thread(workerObject.DoWork); // Thread su metodo DoWork

            // Avvio del thread di tipo Worker
            workerThread.Start();
            Console.WriteLine("Thread principale: sto avviando un thread Worker...");

            // Ciclo fino a quando è attivo thread di tipo Worker
            while (!workerThread.IsAlive) ;

            // Attesa del thread principale (in sonno) per 500 millisecondi
            // per permettere al thread di tipo Worker di fare qualcosa
            Thread.Sleep(500);

            // Richiesta che il thread di tipo Worker si arresti
            workerObject.RequestStop();

            // Uso del metodo Join per fermare il thread corrente
            // fino a quando l'oggetto thread termina
            workerThread.Join();
            Console.WriteLine("Thread principale: il thread Worker ha finito!");

            Console.ReadLine();
        }
    }
}
```

PROGETTO GUIDATO

- Vediamo adesso un altro progetto: crea un nuovo progetto visuale
- Disponi un pulsante e una listbox1 come nella figura



- Passa al codice e inserisci la seguente libreria:

Visual C#

```
//libreria Threading
using System.Threading;
```

- Passa al codice del progetto e dichiara la classe seguente:

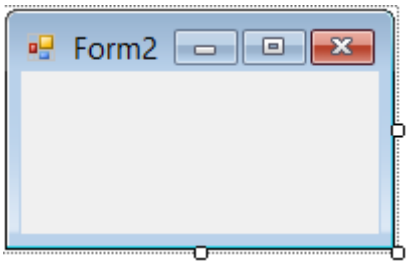
Visual C#

```
public class Worker
{
    //attributo privato condiviso (volatile)
    private volatile bool _shouldStop;

    //costruttore
    public Worker()
    {
        _shouldStop = false;
    } // ---

    //metodo 2
    public void DoWork()
    {
        while (!_shouldStop)
        {
            Form2 nuova = new Form2();
            nuova.Show();
        }
        MessageBox.Show("Thread ausiliario: ho terminato.");
    } // ---

    //metodo 2
    public void RequestStop()
    {
        _shouldStop = true;
    } // ---

}//class Worker
```

- Sotto questa dichiarazione si dichiari una variabile di tipo Worker e una variabile di tipo Thread:

Visual C#

```
// Dichiaro gli oggetti Thread, che però NON si avviano
// (non vanno in esecuzione, ancora)
Worker workerObject ;
Thread workerThread;
```



- Si aggiunga un secondo form (Form2) al progetto e lo si prepari come segue:

Visual C#

```
public partial class Form2 : Form
{
    static Random casuale = new Random();

    public Form2()
    {
        InitializeComponent();
    }

    private void Form2_Load(object sender, EventArgs e)
    {
        BackColor = Color.FromArgb(casuale.Next(256), casuale.Next(256),
casuale.Next(256));
    }
}
```

- Nel Form1 si deve associare un gestore di evento al pulsante button1:

Visual C#

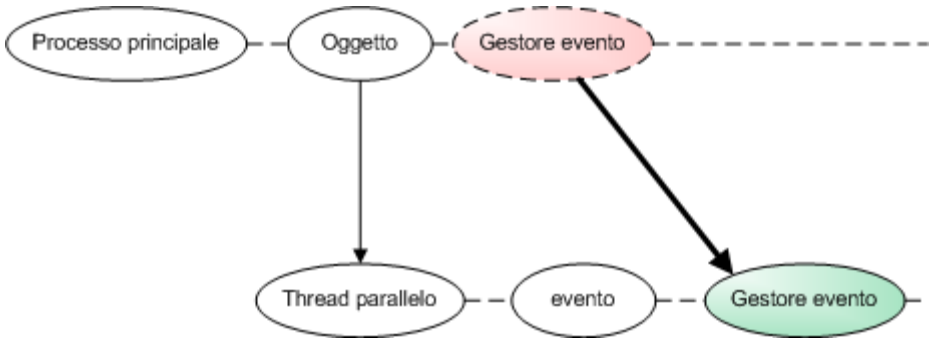
```
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    // Istanzio gli oggetti , che però NON si avviano
    // (non vanno in esecuzione, ancora)
    workerObject = new Worker();
    workerThread = new Thread(workerObject.DoWork);
    listBox1.Items.Add
                ("Processo principale: oggetti istanziati, ma non avviati... ");

    // Avvia i thread di tipo Worker .
    workerThread.Start();
    listBox1.Items.Add("Processo principale: avvio dei thread!");

    // Fa attendere il thread principale per 0.5 secondi
    // per consentire al thread Worker di avanzare col suo lavoro
    Thread.Sleep(500);

    // Richiesta al thread Worker di fermarsi
    workerObject.RequestStop();

    // Usa il metodo Join per bloccare il thread corrente
    // fino a quando l'oggetto thread è terminato
    workerThread.Join();
    listBox1.Items.Add("Processo principale: il thread Worker è terminato!");
}
```

- Si può provare il progetto …

UTILIZZO DEL THREADING

GESTIONE THREAD-SAFE DEI THREAD

Molto spesso lavorando con le GUI (interfacce grafiche) in C#, capita di dover modificare controlli da un altro thread. Questo tipo di gestione non è sbagliato come può sembrare, tanto che è un comportamento normale ad esempio nel caso descritto qui sotto:



Più semplicemente, succede che il gestore di un evento scatenato su di un thread viene gestito da quel thread anche se si trova nel codice del thread principale, quindi se in quel codice voglio modificare il valore di un controllo devo fare in modo di utilizzare un codice thread-safe. Cosa significa questo? È molto semplice. Fino al Framework 1.1 era possibile cambiare ad esempio il valore di testo di una Label da un Thread diverso. Dal framework 2.0 questo non è più possibile, per ragioni di sicurezza interna.

Ecco quindi che ci vengono in aiuto i delegati. Come si può immaginare, un delegato è qualcosa che ci viene in aiuto per risolvere un problema, qualcuno a cui chiediamo di fare qualcosa per noi. Nel vecchio C sarebbe stato qualcosa come passare ad una funzione l'indirizzo di un'altra funzione come parametro, oppure in VB sarebbe una sorta di `addressof`. Essendo il C# ad oggetti per definizione, ecco allora che tutti questi "trucchi" vengono ritradotti in oggetti, i delegati appunto.

Supponiamo di voler modificare il codice di una TextBox da un altro thread, dichiariamo quindi un nuovo semplice delegato:

Visual C#
```
private delegate void changeTextDelegate(string newText);
```

A questo punto creiamo una funzione che svolgerà il compito di cambiare il testo:

Visual C#
```
private void ChangeText(string newText)
{
    if (myTextBox.InvokeRequired)
    {
        myTextBox.Invoke(new changeTextDelegate(ChangeText), newText);
        return;
    }
    myTextBox.Text = newText;
}
```

Successivamente, nel gestore dell'evento chiameremo semplicemente la funzione `ChangeText` passando come parametro il nuovo testo da visualizzare. In questo modo, quando il gestore nell'altro thread entrerà nella funzione, la `InvokeRequired` sarà true per via del fatto che si trova in un thread diverso da quello in cui si trova la TextBox, così chiederà alla TextBox stessa di invocare il metodo stesso attraverso il delegato passandogli come parametro il nuovo testo per poi uscire con il `return`. A questo punto la nuova invocazione avverrà nel thread giusto, la `InvokeRequired` restituirà false e il codice continuerà settando il valore della TextBox a newText.

Bla bla

PARAGRAFO 1

Bla bla

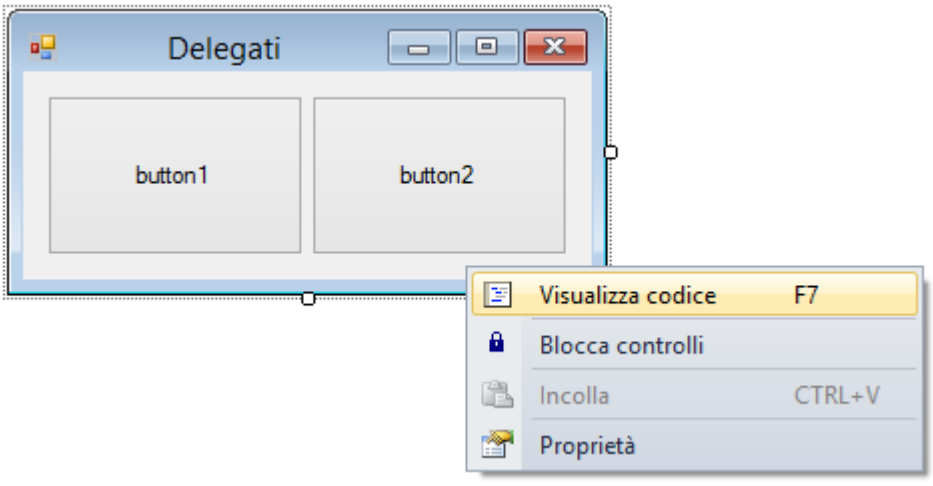
1.1

```
metodo metodo metodo metodo metodo
```

bla bla

PROGETTO GUIDATO

- Crea un nuovo progetto
- Disponi due pulsanti e prepara la finestra come nella figura



- Passa al codice e modificalo come nel seguente esempio:

```
namespace Delegati_Esempi
```

```
{
    public partial class Form1 : Form
    {
        static void InviaMessaggio(string s)
        {
            MessageBox.Show(s);
        }

        public delegate void DelegatoTest(string s);

        DelegatoTest dt = new DelegatoTest(InviaMessaggio);

        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

- Dal form1 associa il gestore di evento button1_Click e scrivi il codice seguente:

```
        private void button1_Click(object sender, EventArgs e)
        {
            dt("pulsante button1");
        }
```

- Prova il progetto.

COMMENTO AL PROGETTO

Il progetto prevede l'uso di tre parti dichiarative e una parte esecutiva.

La prima delle parti da analizzare è la dichiarazione del metodo:

```
        static void InviaMessaggio(string s)
        {
            MessageBox.Show(s);
        }
```

Il metodo è un banale metodo statico del Form1: non rende nulla (`void`), ha un parametro di tipo stringa che conterrà il testo di un messaggio da mostrare, ma soprattutto è denominato `InviaMessaggio` che sarà il nome utilizzato nella istanziazione del delegato.

La seconda parte da analizzare è

```
        public delegate void DelegatoTest(string s);
```

che costituisce il vero nucleo della dichiarazione di un delegato. La riga di codice qui riportata è la dichiarazione del delegato: il descrittore public è opinabile (lo rende visibile anche fuori dalla unità di programma); la parola chiave `delegate` è un particolare tipo di dato che lo contraddistingue come un delegato, ovvero un metodo da associare a un corpo; il resto dei comandi specifica tipo nome e parametri del delegato, ma non presuppone alcun codice di corpo.

La terza parte da analizzare è

```
        DelegatoTest dt = new DelegatoTest(InviaMessaggio);
```

che costituisce la creazione (istanziazione) del metodo delegato. In questo caso si crea un oggetto (di tipo `DelegatoTest`) istanziato dal omonimo costruttore, il quale costruttore si aspetta come parametro il nome di un metodo (con corpo) da associare al nome del delegato.

I delegati dipendono soltanto dalla firma del metodo, non dalla classe o dall'oggetto che contiene il metodo. È, inoltre, opportuno osservare che nell'esempio precedente il metodo `InviaMessaggio` è stato dichiarato come statico: questo non è un requisito, ma se si usa dentro una classe Form1 è necessario.

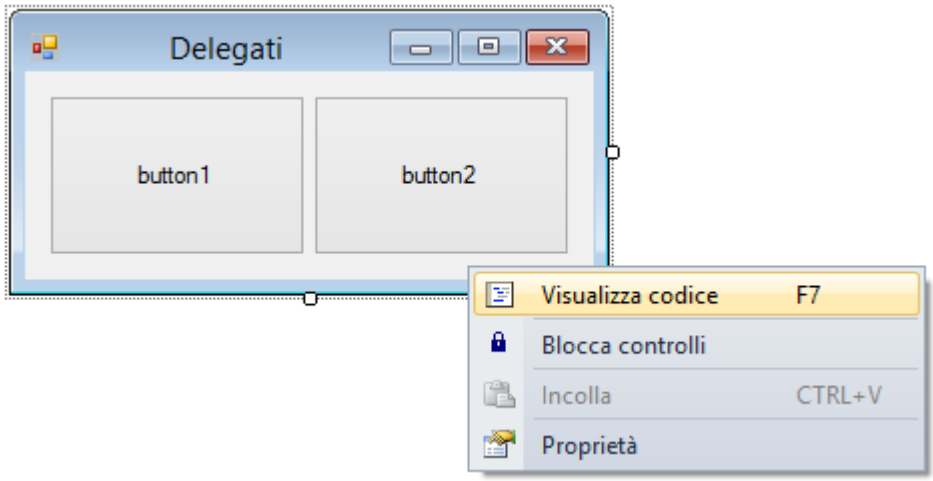
Infine al pulsante è stato associato l'invocazione del delegato, ovvero la sua "attivazione", il momento in cui il delegato è chiamato ad agire e a porre in azione l'effetto richiesto:

```
        private void button1_Click(object sender, EventArgs e)
        {
            dt("pulsante button1");
        }
```

dove la parola `dt` è l'invocazione del delegato creato in precedenza.

PROGETTO GUIDATO

- Riapri il precedente progetto



- Modificalo il codice come segue:

```
namespace Delegati_Esempi
{
    public partial class Form1 : Form
    {
        static double CalcolaMedia(int a, int b)
        {
            return (a + b) / 2.0;
        }

        public delegate double DelegatoTest(int a, int b);

        DelegatoTest dt = new DelegatoTest(CalcolaMedia);

        public Form1()
        {
            InitializeComponent();
        }
```

```
    }
}
```

- Il gestore di evento button1_Click deve essere modificato come segue:

```
        private void button1_Click(object sender, EventArgs e)
        {
            double ris = dt(13, 17);
            MessageBox.Show("" + ris);
        }
```

- Prova il progetto.

COSA È UN DELEGATO?

DEFINIZIONE DI DELEGATO

Fonte: http://msdn.microsoft.com/it-it/library/ms173172.aspx

Un delegato è un tipo in cui è incapsulato un metodo in modo sicuro.

È simile a un puntatore a una funzione in C e C++ ma, diversamente da questo, è orientato ad oggetti, indipendente dai tipi e protetto. Il tipo di un delegato è definito dal nome del delegato

Come prima definizione, quindi, un delegato è un ulteriore tipo di dato. In realtà un delegato è un nuovo tipo di dato che il programmatore può costruire e associare a un procedimento.

La dichiarazione di un delegato segue la seguente sintassi:

```
descrittori delegate TipoReso NomeDelegato(elenco_parametri);
```

I descrittori del delegato sono quelli ordinari visti per altri tipi, come public, protected, private, static, ecc… in generale in questa dispensa vedremo il descrittore public come il più frequente.

Il frammento racchiuso nella cornice rossa indica la descrizione della firma del delegato, ovvero l'insieme del tipo restituito, il suo nome e l'elenco dei parametri; in definitiva la firma del delegato è analoga alla firma di un qualsiasi metodo già studiato in precedenza.

La parola chiave delegate che separa i descrittori dalla firma, è la keyword che dichiara un nuovo delegato, ovvero il definitore del nuovo tipo (analogo per esempio a class).

Un esempio di dichiarazione di delegato è la seguente:

```
        public delegate double DelegatoTest(int a, int b);
```

dove la parola chiave delegate indica che si tratta di una dichiarazione di un delegato, l'identificatore DelegatoTest è il nome del delegato costruito (è il nuovo tipo appena dichiarato), e con le parentesi tonde si descrive la firma del metodo atteso.

Vediamo adesso come costruire il delegato appena dichiarato:

Fonte: http://msdn.microsoft.com/it-it/library/ms173172.aspx

Per la costruzione di un oggetto delegato è in genere sufficiente fornire il nome del metodo di cui il delegato eseguirà il wrapping (`confezionamento`) oppure utilizzare un metodo anonimo.

Vediamo un esempio per preparare un metodo cui il delegato eseguirà il confezionamento:

```
        static double CalcolaMedia(int a, int b)
        {
```

```
            return (a + b) / 2.0;
        }
```

E il confezionamento avverrà quando si eseguirà un'istruzione analoga alla seguente:

```
        DelegatoTest dt = new DelegatoTest(CalcolaMedia);
```

Il wrapping può essere ottenuto anche indicando il solo identificatore di un metodo associabile, come nel seguente esempio:

```
        DelegatoTest dt = CalcolaMedia;
```

L'alternativa a questo confezionamento è di utilizzare un metodo anonimo (senza nome) come nel seguente esempio:

```
        DelegatoTest myDel = delegate (int x, int y) { return ((x + y)/2.0); };
```

Nel precedente esempio il delegato confezionato è anonimo, infatti è costruito al volo con la parola chiave delegate seguito dai parametri e dal corpo.

Occorre osservare che il delegato non ammette un metodo (da confezionamento o anonimo) la cui firma non corrisponda a quella della dichiarazione; rispetto al precedente esempio darebbe errore il seguente codice:

```
DelegatoTest myDel = delegate (string s) { return (s + "errore!"); };
```

Una volta creata un'istanza di un delegato, quest'ultimo passerà al metodo le chiamate ricevute. I parametri passati al delegato dal chiamante verranno passati al metodo e l'eventuale valore restituito dal metodo verrà restituito dal delegato al chiamante.

Vediamo un altro esempio con cui confezionare un delegato, invocarlo e usare il risultato:

```
DelegatoTest prova;          //dichiarazione di un nuovo oggetto delegato
prova = CalcolaMedia;        //wrapping dell'oggetto delegato
double ris = prova(12, 21); //invocazione dell'oggetto delegato
MessageBox.Show("" + ris);   //
```

Per fare riferimento a questo processo si afferma normalmente che il delegato viene richiamato. È possibile richiamare un delegato di cui è stata creata un'istanza in modo analogo al metodo di cui è stato eseguito il `wrapping` dal delegato stesso.

DELEGATO COME TIPO

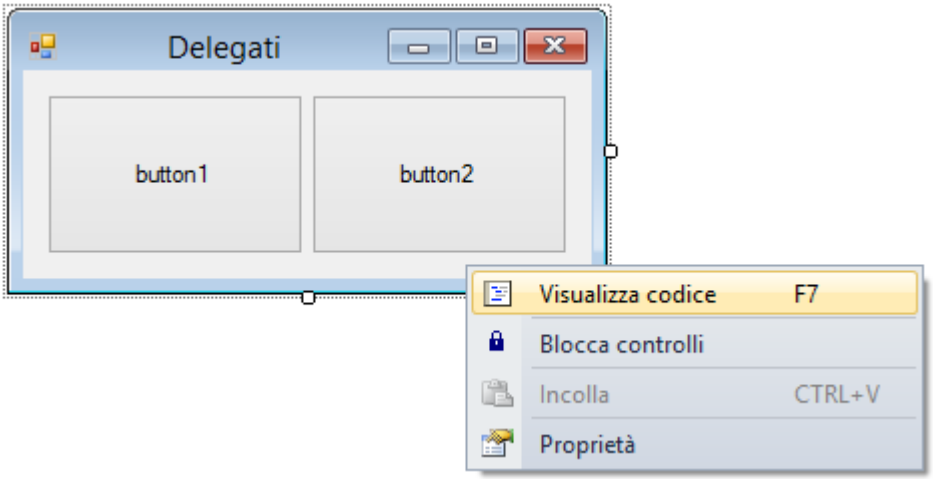
DELEGATO COME TIPO (CLASSE)

I tipi delegati vengono derivati dalla classe Delegate in .NET Framework e sono sealed (classi sigillate), ossia non possono essere utilizzati per dichiarare classi derivate da esso. Non è inoltre possibile derivare classi personalizzate da Delegate.

Poiché il delegato di cui è stata creata un'istanza è un oggetto, può essere passato come parametro o assegnato a una proprietà. **In questo modo un metodo può accettare un delegato come parametro e chiamarlo in un secondo momento.** Questa operazione è nota come `callback asincrono` e rappresenta uno dei modi più comuni per notificare a un chiamante il completamento di un processo prolungato. Quando si utilizza un delegato in questo modo, nel codice che include il delegato non deve essere necessariamente specificata l'implementazione del metodo. La funzionalità è simile all'incapsulamento fornito dalle interfacce.

PROGETTO GUIDATO

- Riapri il precedente progetto (oppure si crei uno nuovo, con la dichiarazione del delegato **DelegatoTest** :



- Aggiungi al codice la seguente dichiarazione di metodo:

```
public void MetodoConUnCallback(int param1, int param2, DelegatoTest param3)
{
      double ris;
      ris = param3(param1, param2);
      MessageBox.Show("Il risultato è: " + ris);
}
```

- Adesso associa al secondo pulsante button2 il seguente gestore di evento:

```
private void button2_Click(object sender, EventArgs e)
{
      MetodoConUnCallback(12, 13, CalcolaMedia);
}
```

- Prova il progetto ed usa il button2: funziona?
- È possibile associare al button1 un callback che invoca un metodo diverso da CalcolaMedia, per esempio un metodo che calcola il massimo, la media pesata, o la distanza dall'origine del punto le cui coordinate sono passate coi parametri?

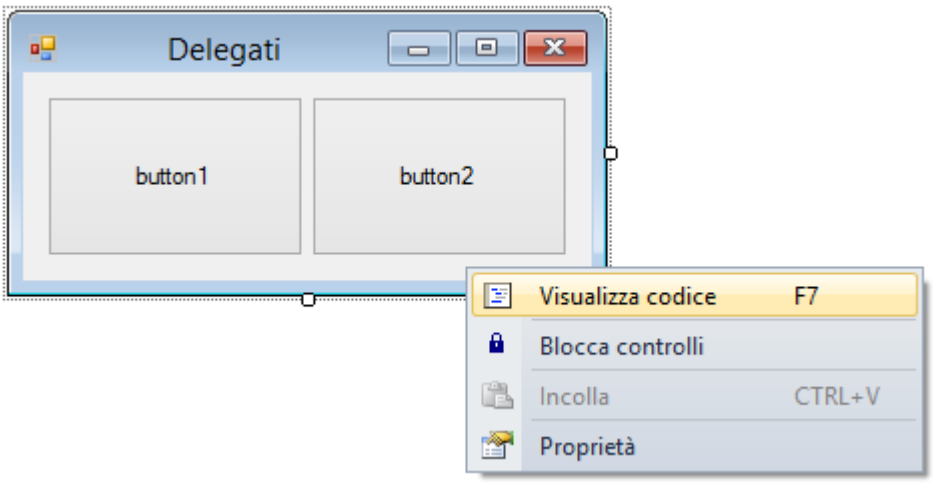
COLLEZIONE DI METODI DELEGATI

LISTA DI METODI DELEGATI

I delegati possono contenere una lista interna di delegati (chiamata `invocation list`) che contiene delegati aventi la stessa firma, si parla in questi casi di `delegati multicast`. Quando un delegato di questo tipo viene invocato esso a sua volta invoca tutti i delegati presenti nella sua invocation list. E' possibile aggiungere e rimuovere delegati da una invocation list utilizzando gli operatori di overload `+=` e `-=` rispettivamente.

PROGETTO GUIDATO

- Crea un progetto nuovo:



- Aggiungi al codice la seguente dichiarazione di metodo:

```
//-------------------------------------------------------secondo esempio
public delegate double MioDelegato (double a, double b);

public double MediaDelegata (double a, double b)
            { MessageBox.Show("1");  return ((a + b) / 2.0); }

public double MassimoDelegato (double a, double b)
            { MessageBox.Show("2"); if (a > b) return a; else return b; }

public double DistanzaDelegata (double a, double b)
            { MessageBox.Show("3"); return Math.Sqrt(a * a + b * b); }

MioDelegato deleg_1;
```

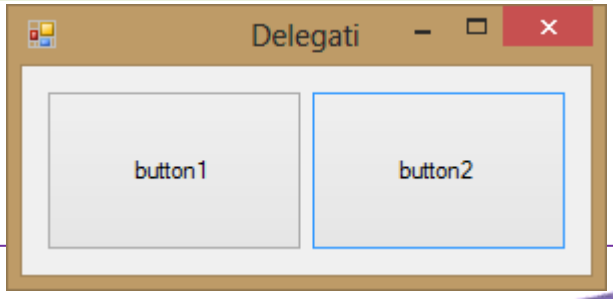
- Associa al pulsante `button1` il seguente codice:

```
//-------------------------------------------------------secondo esempio
private void button1_Click(object sender, EventArgs e)
{
      deleg_1 += MediaDelegata;

      deleg_1 += MassimoDelegato;

      deleg_1 += DistanzaDelegata;
}
```

- Associa al pulsante `button2` il seguente codice:

```
//-------------------------------------------------------secondo esempio
private void button2_Click(object sender, EventArgs e)
{
      double ris = deleg_1(12, 21);       //invocazione dell'oggetto delegato
      MessageBox.Show("Ma il risultato finale è: " + ris);

}
```

- Avvia il progetto e premi prima button1 e solo dopo button2
- Dopo premi nuovamente prima button1 e solo dopo button2
- Ripeti la precedente indicazione alcune volte



- Termina il progetto

COMMENTO AL PROGETTO

Abbiamo già affermato che i `delegati multicast` possono contenere una lista interna di delegati (chiamata invocation list) che contiene delegati tutti aventi la stessa firma.

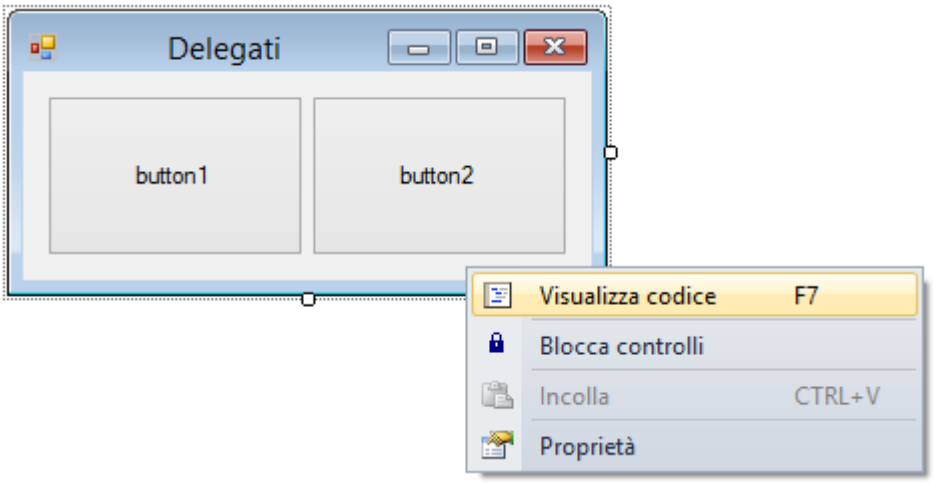
Questo significa che lo stesso delegato fa riferimento non ad un solo metodo ma ad un elenco; nell'esempio appena concluso il delegato deleg_1 fa riferimento inizialmente a nessun metodo; dopo il clic sul button1 ne vengono associati tre, nell'ordine di assegnazione; quando si preme ancora il button1 ne vengono associati altri tre, sempre nell'ordine di assegnazione. E così via.

Quando il delegato viene invocato, esso chiama l'invocazione di tutti i metodi in elenco. Se ci sono tre metodi, allora ne invoca tre. Se ci sono sei metodi, allora ne invoca sei. Se ci sono nove metodi, allora ne invoca nove. E così via.

Come è possibile aggiungere metodi al delegato è anche possibile rimuoverlo; per eliminare un metodo dalla lista occorre utilizzare l'operatore `-=`; vediamo un esempio:

PROGETTO GUIDATO

- Modifica il progetto precedente:



- Modifica il corpo del gestore di evento del button1 come segue:

```
private void button3_Click(object sender, EventArgs e)
{
    deleg_1 += MediaDelegata;       //1
    deleg_1 += MassimoDelegato;     //2
    deleg_1 += DistanzaDelegata;    //3
    deleg_1 += MediaDelegata;       //1
    deleg_1 += MassimoDelegato;     //2
    deleg_1 += DistanzaDelegata;    //3
    deleg_1 -= MassimoDelegato;     // elimino un 2
    //invocazione dell'oggetto delegato
    double ris = deleg_1(12, 21);
}
```

- Avvia il progetto e premi button1 alcune volte
- Termina il progetto

COMMENTO AL PROGETTO

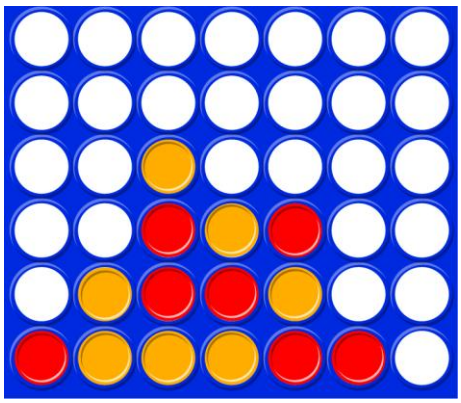
Abbiamo visto che un delegato fa riferimento non ad un solo metodo ma ad un elenco; nell'esempio appena concluso il delegato deleg_1 accoda 6 metodi ma poi ne rimuove uno; il metodo rimosso è l'ultimo dell'elenco con il nome specificato.

Quando il delegato viene invocato, esso richiama l'invocazione di tutti i metodi in elenco. La prima volta quindi invocherà 5 metodi (6 meno 1), la seconda invocherà 10 metodi (12 meno 2), alla terza invocherà 15 metodi (18 meno 3), e così via!

ESERCIZI

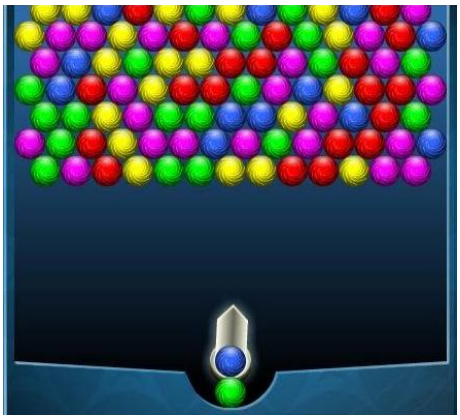
ESERCIZI

ESERCIZIO 1.



- Crea un progetto visuale
- All'avvio del programma si crea «al volo» una matrice di immagini come nella figura qui a lato (6 x 7)
- Definire un delegato che accetta due diversi metodi:
  1. Colora di rosso l'immagine cliccata
  2. Colora di giallo l'immagine cliccata
  3. Ad ogni clic il delegato rimuove il metodo associato e si aggancia all'altro

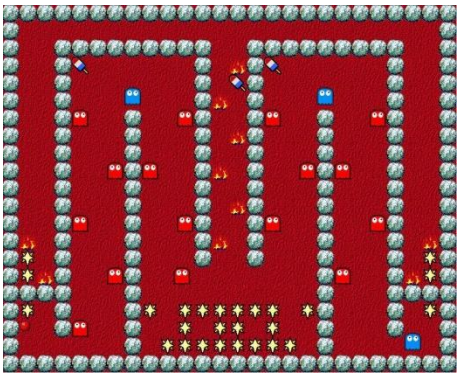
ESERCIZIO 2.



- Crea un progetto visuale
- All'avvio del programma si crea «al volo» una collezione di immagini come nella figura qui a lato

1. ?

ESERCIZIO 3.



- Crea un progetto visuale
- All'avvio del programma si crea «al volo» una collezione di immagini come nella figura qui a lato

?

SOMMARIO