Sommario

Debug

Si definisce Debug l'attività che consiste nell'individuare e correggere errori rilevati in fase di programmazione ed in fase di testing o utilizzo finale dell'applicazione.
Un Debugger è un componente software che consente l'individuazione di errori.

Vediamo un esempio.

Data una lista con numeri interi con 5 elementi visualizzare la somma degli elementi di valore pari.

```
class Program
  {
      static void Main(string[] args)
      {
          List<int> numeri;
          numeri = new List<int>();
          numeri.Add(1);
          numeri.Add(2);
          numeri.Add(3);
          numeri.Add(4);
          numeri.Add(5);

          //in alternativa
          //List<int> numeri=new List<int>() { 1, 2, 3, 4, 5 };
          int somma = 0;
          for (int i = 0; i < numeri.Count; i++)
          {
              if (numeri[i] % 2 == 0)
                  somma = somma + numeri[i];
          }
          Console.WriteLine(somma);
          Console.ReadLine();
      }
  }
```

Commettiamo un errore:

```
class Program
  {
      static void Main(string[] args)
      {
          List<int> numeri;
          numeri = new List<int>();
          numeri.Add(1);
          numeri.Add(2);
          numeri.Add(3);
          numeri.Add(4);
          numeri.Add(5);

          //in alternativa
          //List<int> numeri=new List<int>() { 1, 2, 3, 4, 5 };
          int somma = 0;
          for (int i = 0; i < numeri.Count; i++)
          {
              if (numeri[i] % 2 == 0)
                  somma = somma + numeri[1];
```

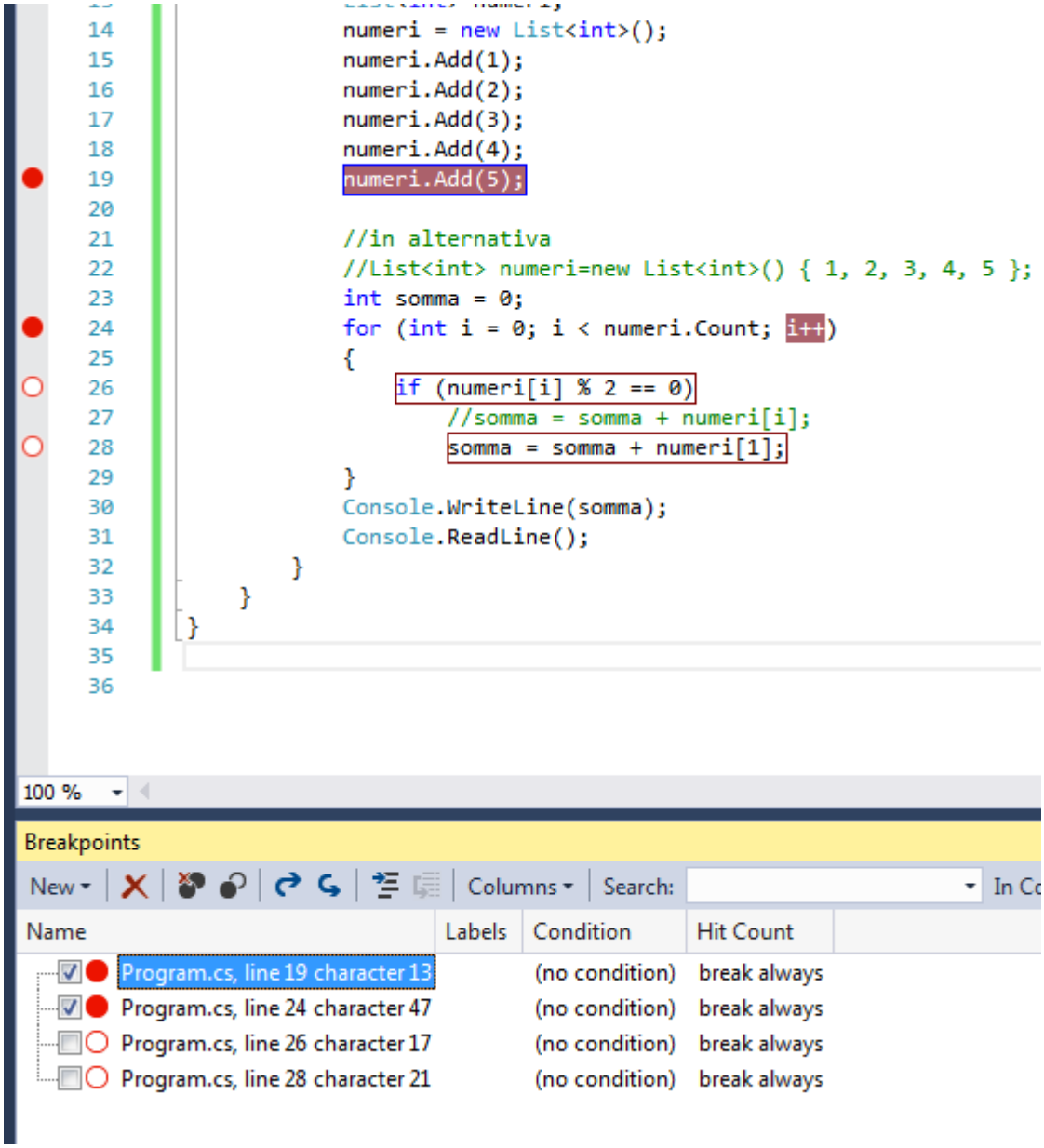
```
            }
            Console.WriteLine(somma);
            Console.ReadLine();
        }
    }
```

Per meglio controllare il codice si possono mettere dei breakpoint.

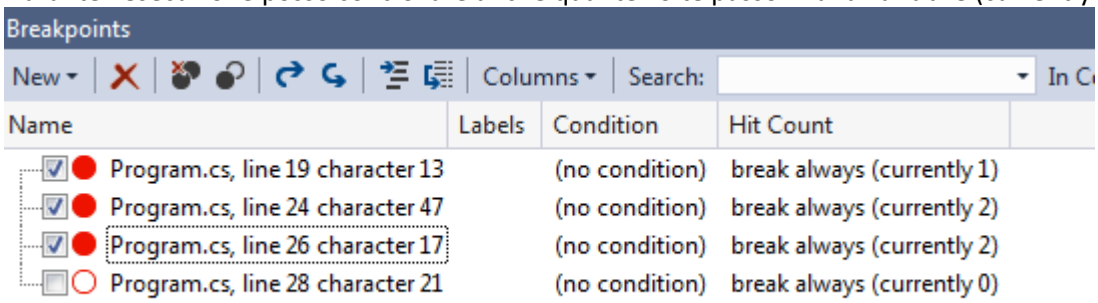
Breakpoint

Un breakpoint consente di fissare uno specifico punto del codice al cui raggiungimento il debugger si ferma in attesa che il programmatore possa effettuare certe verifiche.

Per inserire un breakpoint si stabilisce la riga del codice desiderata e si clicca sulla barra a sinistra.
Per mettere il breakpoint sulla riga del for, seleziono cosa voglio controllare, premo F9.
Avvio il programma e premendo F10 posso controllare passo passo cosa succede nel codice. Per verificare il valore della variabile che si sta controllando si preme F5 e passando sopra con il mouse posso leggerne il contenuto.
Vediamo come disabilitare il BreakPoint.
Cliccare su Debug-> Windows->Breakpoints; sia aprirà in basso una finestra contenente l'elenco dei Breakpoints.

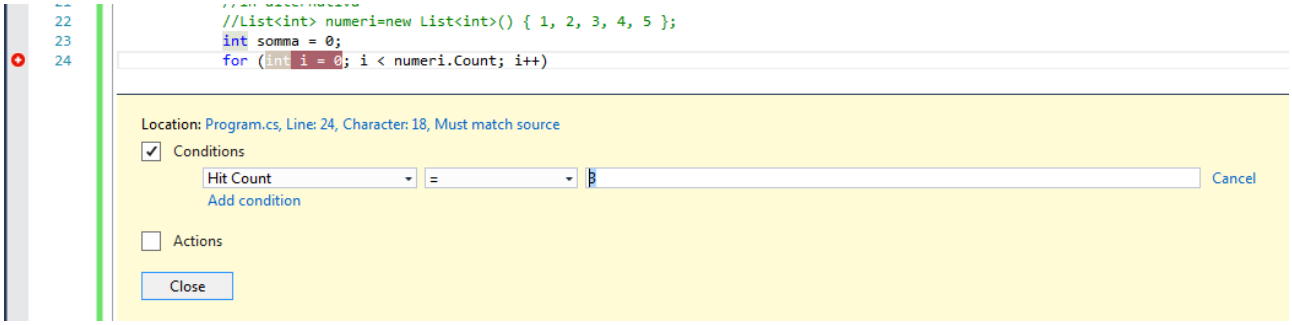


Cliccando sulla spunta è possibile disabilitarlo, ma posso anche eliminarlo.
Durante l'esecuzione posso controllare anche quante volte passo in una variabile (currently1).



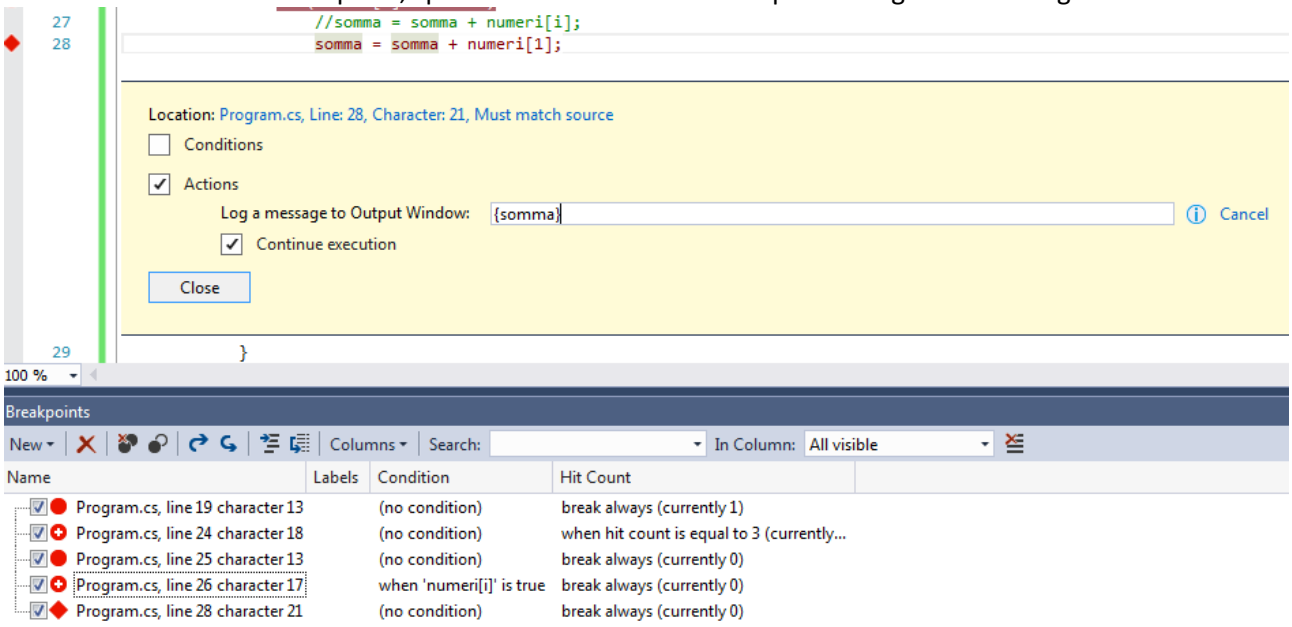
Hit Count

Posso decidere quando fermare il codice dopo n volte, o ad una certa condizione.
Menù->Breakpoint->TastoDestro



TracePoint

Stabilisce una azione nel punto stabilito, ma non ferma il codice.
Ad esempio voglio visualizzare la somma e non fermarmi.
Per far ciò cliccare sul Breakpoint, spuntare Action e scrivere tra parentesi graffe cosa voglio visualizzare.



Step Into

Trasformiamo il codice in un metodo e ricommettiamo l'errore `somma = somma + numeri[1]`.

```
static void Main(string[] args)
      {
          List<int> numeri;
          numeri = new List<int>();
          numeri.Add(1);
          numeri.Add(2);
          numeri.Add(3);
          numeri.Add(4);
          numeri.Add(5);

          //in alternativa
```

```
            //List<int> numeri=new List<int>() { 1, 2, 3, 4, 5 };
            int somma = 0;
            somma = Somma(numeri);
            Console.WriteLine(somma);
            Console.ReadLine();
        }

        private static int Somma(List<int> numeri)
        {
            int somma = 0;
            for (int i = 0; i < numeri.Count; i++)
            {
                if (numeri[i] % 2 == 0)
                    //somma = somma + numeri[i];
                    somma = somma + numeri[1];
            }

            return somma;
        }
    }
```

Se metto BreakPoint e durante l'esecuzione premo F10 non si riesce ad entrare nel metodo perché lo salta. Devo usare F11.

Step Out

Per uscire dal metodo deve usare Shift+F11.

Edit Variable Value

Durante il debugging posso modificare il valore di una variabile e continuare, oppure posso tornare indietro cliccando sulla freccia gialla e trascinare verso le righe di codice precedenti.

Esercizio

Realizzare un'applicazione console che inserita una lista di 6 numeri da parte dell'utente calcola e visualizza la somma dei numeri di posizione pari e la somma dei numeri di valore dispari.

Introduzione alla programmazione Multithreading

Fare multithreading significa che in un programma si divide l'operazione in tante parti; dal thread principale (il processo di partenza) parte almeno un thread secondario.

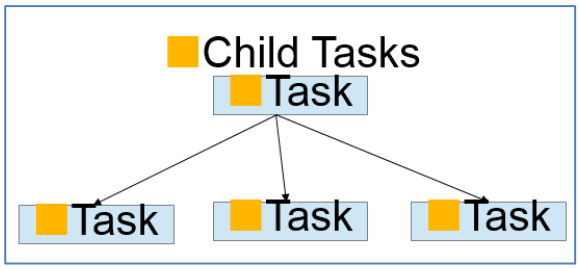
Prime definizioni.

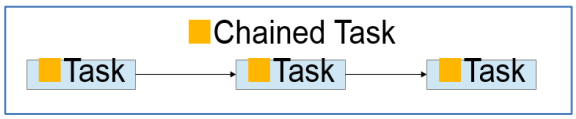
Processo: è il nome che assume il programma quando parte.

Thread: è l'unità base del processo per il quale il processore alloca tempo. Almeno un Thread è sempre presente (è il programma principale). Dal Task principale si generano i Task secondari.
Un Thread è una suddivisione di un processo in due o più filoni o sottoprocessi che vengono eseguiti concorrentemente da un sistema di elaborazione.

Application Domains sono ulteriori suddivisioni dei Thread.

Task: Astrazione di uno specifico piccolo compito (unità di lavoro, unità logica, un metodo) che funziona in modo asincrono (è il thread secondario). E' un flusso del processo.
Generalmente un task usa il pattern Completion Notification, perché notifica il suo stato.





Thread in background
Un thread principale parte e quando temina tutti i suoi thread secondari.
Generalmente i threads sono in background. La vita di un secondario dipende dal principale.

Thread in foreground
Un thread principale parte e quando finisce lascia vivo il thread secondario.
Es. "Impossibile salvare perché in uso, ma ho già chiuso il principale".
Spesso è un problema avere thread in foreground.

Timers

Vediamo un altro strumento utile per programmazione multithreading.
.Net fornisce due tipologie di Timers: Server Timer e Thread Timer.
I timers sono operazioni ricorrenti, schedulate. Possono essere attivati una volta sola oppure possono essere periodici. Fino agli anni 90 tutti i giochi funzionavano con i Timer.

Server Timer

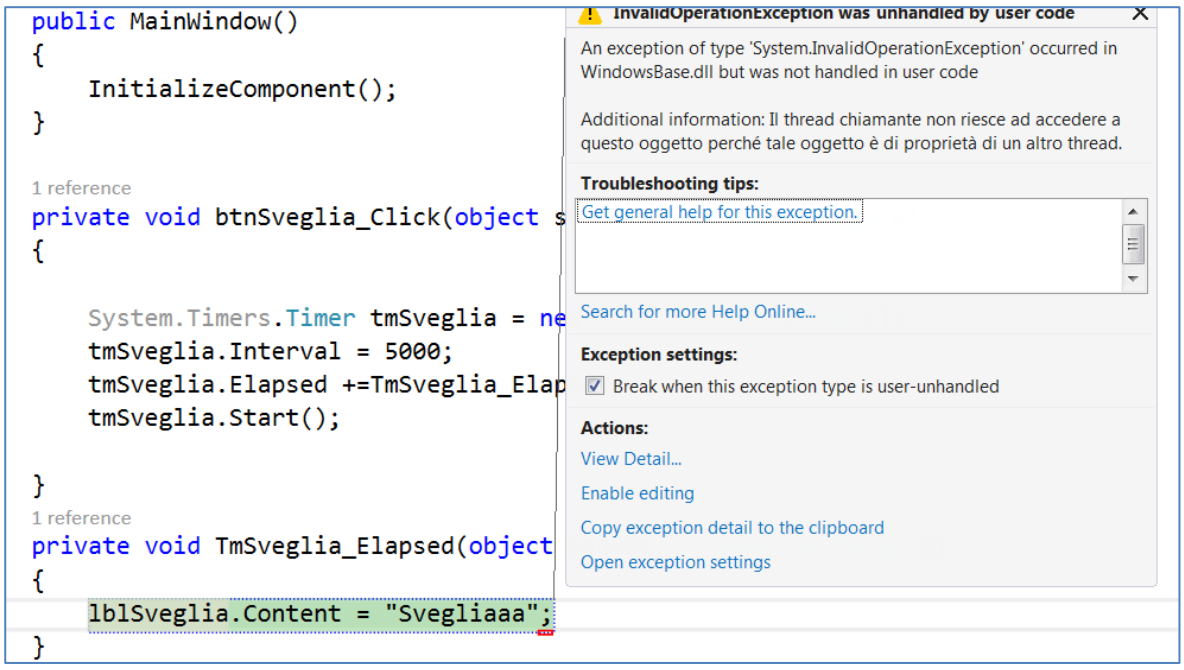
Server Timer scatena eventi su un thread; sarà l'evento a dovere svolgere un determinato compito. Al temine della sua esecuzione non ha bisogno di essere a conoscenza di cosa fare successivamente, notifica solamente lo scadere dell'intervallo di tempo.
Ha una proprietà Interval di tipo double (espressa in millisecondi; pertanto funziona bene per intervalli lunghi) ed un evento Elapsed che può essere utilizzato anche dal thread dell'interfaccia grafica.

```
private void btnSveglia_Click(object sender, RoutedEventArgs e)
        {

            System.Timers.Timer tmSveglia = new System.Timers.Timer();
            tmSveglia.Interval = 5000;
            tmSveglia.Elapsed +=TmSveglia_Elapsed;
            tmSveglia.Start();

        }
```

Il Timer scatta ogni 5 secondi e solleva un evento il quale sa cosa eseguire.

```
private void TmSveglia_Elapsed(object sender, ElapsedEventArgs e)
        {
            lblSveglia.Content = "Svegliaaa";
        }
```

L'azione solleva un'eccezione.



Questo perché la label appartiene al thread principale e non al thread, anzi al task, generato.
Pertanto è necessario effettuare una seconda modifica al codice, generando un thread secondario:

```
private void TmSveglia_Elapsed(object sender, ElapsedEventArgs e)
        {
            Dispatcher.Invoke(AggiornaTestoSveglia);
        }
```

```
        private void AggiornaTestoSveglia()
        {
            lblSveglia.Content = "Svegliaaa";
         }
```

Thread Timer

Thread Timer è un oggetto che utilizza dei metodi di callback (chiamate indietro, cioè quando scatta il periodo "dico" cosa si deve fare).
Al momento in cui scade il timer esegue un metodo che contiene i comandi da eseguire.
Ricordiamo che il programma principale è sempre il Thread principale, i Timers rappresentano i secondari.
In realtà il Thread principale per gestire tutta l'interfaccia di Windows necessita di far partire altri thread.
Ha una proprietà Interval di tipo long.

```
private void btnSvegliaThread_Click(object sender, RoutedEventArgs e)
        {
            System.Threading.Timer threadTimer;
            threadTimer = new System.Threading.Timer(SvegliaDaSuonare, null, 0, 1000);

        }
```

Analizziamo la seguente istruzione:

```
threadTimer = new System.Threading.Timer(SvegliaDaSuonare, null, 0, 1000);
```

- SvegliaDaSuonare indica il metodo invocato, quindi il Thread Timer esegue un metodo.
- Null indica lo stato del Thread
- 0 Indica il ritardo dalla partenza
- 1000 indica ogni quanto scatta il timer, è un intervallo di tempo ricorrente.

```
  private void SvegliaDaSuonare(object state)
        {
            Dispatcher.Invoke(AggiornaSvegliaThread);

        }
```

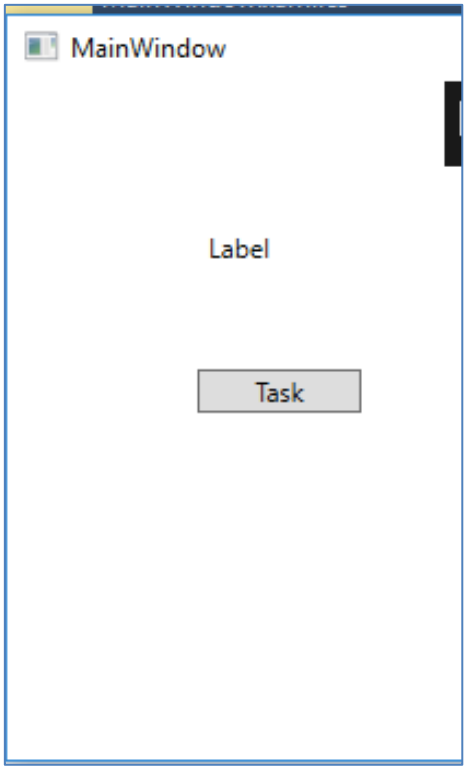
```
private void AggiornaSvegliaThread()
        {
            lblSvegliaThread.Content = "Sveglia Thread!!";
        }
```

How to Make a Game

Tutti i giochi seguono la seguente struttura: un main loop all'interno del quale vi è interazione con l'input dell'utente, si gestiscono le animazioni (sprites), si gestiscono le collisioni, si utilizzano effetti sonori.
In passato si utilizzavano solo Timers, ora il multithreading.

Prime applicazioni multithreading

Consideriamo la seguente interfaccia grafica con una Label ed un Button:



Durante l'esecuzione del seguente codice, l'interfaccia grafica non è responsiva, cioè non è reattiva, non risponde a nessuna attività.

```
        private void btnTask_Click(object sender, RoutedEventArgs e)
        {
            for (int i=0;i<1000;i++)
            { for(int j=0;j<100;j++)
                { }
            }
            lblTask.Content = "Finito!";
        }
```

Vediamo quali modifiche sono necessarie per rendere l'applicazione multithreading attraverso l'uso del task. Prima di tutto creiamo un metodo che contenga il codice.

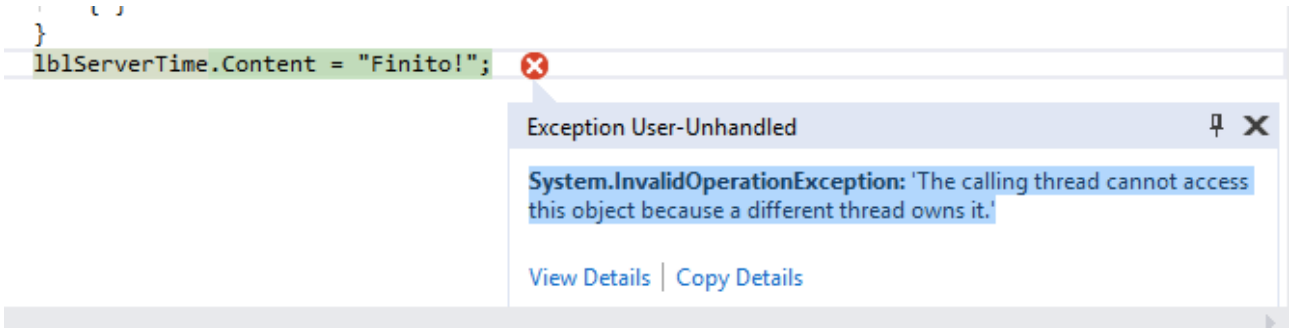
```
 private void btnTask_Click(object sender, RoutedEventArgs e)
        {
            LavoroLungoEComplicato();
        }
```

```
public void LavoroLungoEComplicato()
        {
            for (int i = 0; i < 1000; i++)
            {
                for (int j = 0; j < 100; j++)
                { }
            }
            lblTask.Content = "Finito!";
        }
```

Fin qui nulla è cambiato. L'interfaccia continua a non essere responsiva.
Una prima modifica da fare al codice è la seguente:

```
private void btnTask_Click(object sender, RoutedEventArgs e)
        {
            Task.Factory.StartNew(LavoroLungoEComplicato);

        }
```

L'applicazione è stata resa multithreading. La riga di codice `Task.Factory.StartNew(Speak);` crea un task e lo fa partire, Speak indica il metodo da far eseguire.
In alternativa si potrebbe scrivere
Durante l'esecuzione del metodo LavoroLungoEComplicato() l'interfaccia è risponde alle altre attività.
Tuttavia non basta!
La chiamata al metodo LavoroLungoEComplicato solleva un'eccezione al momento in cui viene eseguita l'istruzione `lblTask.Content = "Finito!"`.



Questo perché la label appartiene al thread principale e non al thread, anzi al task, generato.
Pertanto è necessario effettuare una seconda modifica al codice, generando un thread secondario:

```
public void LavoroLungoEComplicato()
        {
            for (int i = 0; i < 1000; i++)
            {
                for (int j = 0; j < 100; j++)
                { }
            }
            Dispatcher.Invoke(AggiornaInterfacciaUtente);
        }
```

L'aggiornamento dell'interfaccia va incapsulata in un metodo che nel nostro esempio chiameremo `AggiornaInterfacciaUtente()`.

```
public void AggiornaInterfacciaUtente()
        {

            lblTask.Content = "Finito!";
        }
```

Con l'istruzione `Dispatcher.Invoke(AggiornaInterfacciaUtente);` si chiede al thread principale di aggiornare il contenuto della label. Il Task principale ha lasciato un figlio, il figlio non può aggiornare direttamente su un componente (es. label) appartenente al principale.
L'esempio appena proposto è del topo Fire & Forget.

Task Parametrici e non Parametrici

Per lanciare il Task è necessario creare il metodo che fa un certo lavoro (nel nostro caso DoWork).

Task senza parametro

Ricordiamo come si lancia un thread senza parametri

```
        private void btnTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            Task.Factory.StartNew(DoWork);
        }
```

```
        private void DoWork()
        {
            for (int i = 0; i < 1000; i++)
            {

                Dispatcher.Invoke(UpdateUI);
            }
        }

        void UpdateUI()
        {
            lblCount.Content = "fatto";
        }
```

Osserviamo che se vogliamo passare al metodo UpdateUI un parametro (ad esempio i) si ha un errore.

```
    private void btnTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            Task.Factory.StartNew(DoWork);
        }

        private void DoWork()
        {
            for (int i = 0; i < 1000; i++)
            {
                Dispatcher.Invoke(UpdateUI(i));
            }
        }

        void UpdateUI(int i)
        {
            lblCount.Content = i.ToString();
        }
```

Prima di tutto chiediamoci che cosa vuol dire passare un metodo?

Con `Dispatcher.Invoke(UpdateUI(i));` si passa un'azione ad un altro metodo. In realtà si passa l'indirizzo della prima istruzione del metodo (il puntatore al metodo), oltre al parametro i.
Diamo le definizioni di action e functor.
Le action sono metodi che svolgono un compito, che si possono passare e non restituiscono nulla.

Functor sono metodi che svolgono compito che si possono passare e restituiscono un valore.

La trasformazione di un metodo in azione si effettua attraverso una funzione lambda.

Task con parametro

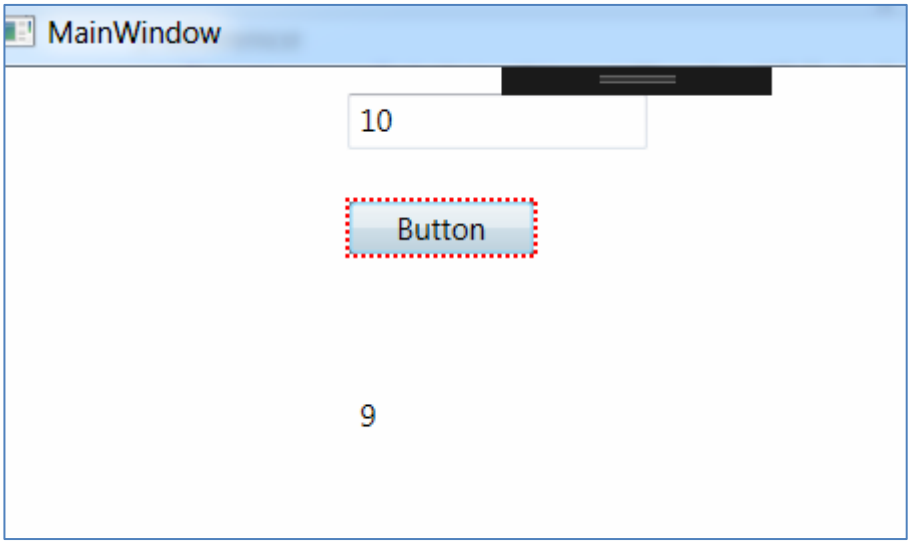
Vediamo come si risolve il problema sollevato al paragrafo precedente modificando il codice.

```
        private void btnTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            Task.Factory.StartNew(DoWork);
        }

        private void DoWork()
        {
            for (int i = 0; i < 1000; i++)
            {
                Thread.Sleep(100);
                //Dispatcher.Invoke(UpdateUI(i));
```

```
                Dispatcher.Invoke(()=>UpdateUI(i));
            }
        }

        void UpdateUI(int i)
        {
            lblCount1.Content = i.ToString();
        }
    }
```

Con l'istruzione `Thread.Sleep(100)` si rallenta l'esecuzione del ciclo. Permette di simulare un ritardo di 100 millisecondi.
La seguente sintassi `Dispatcher.Invoke(()=>UpdateUI(i));` si chiama funzione lambda. E' un modo che permette di trasformare un metodo in un'azione e di passare un parametro oltre che l'azione.

Ora, consideriamo la seguente Intefaccia, con un Button, una Label ed una TextBox. La fine del ciclo lo stabilisce l'utente scrivendo il valore nella TextBox. Il conteggio viene visualizzato nella Label ad ogni secondo.



Analizziamo il seguente codice:

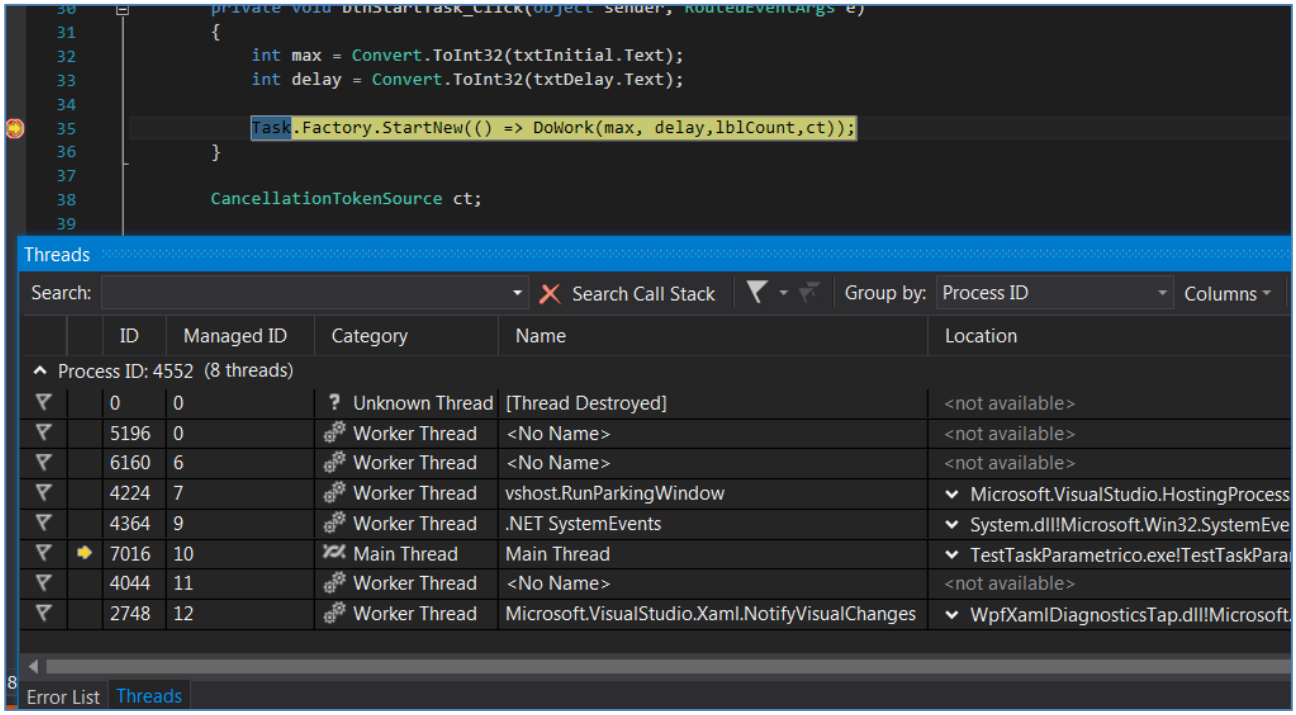
```
public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }


        private void btnStartTask_Click(object sender, RoutedEventArgs e)
        {
            int max = Convert.ToInt32(txtInitial.Text);
            Task.Factory.StartNew(()=>DoWork(max));
        }


        void DoWork(int max)
        {
            for (int i = 0; i < max; i++)
            {//simuliamo un ritardo
                Thread.Sleep(1000);
```

```
                Dispatcher.Invoke(()=>UpdateUI(i));

            }
        }
        void UpdateUI(int i)
        { lblCount.Content = i.ToString(); }
    }
}
```

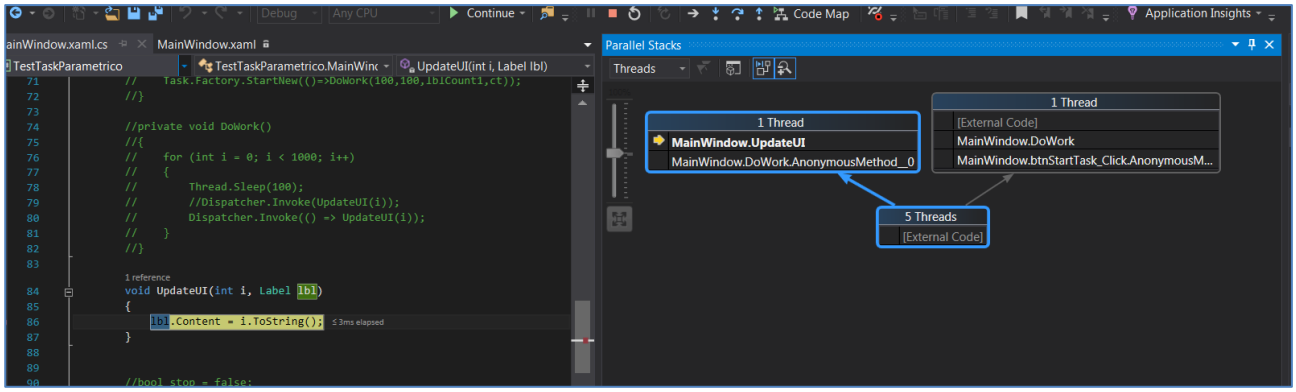
Anche per la seguente istruzione ho bisogno di usare la funzione Lambda per passare il parametro max e trasformare il metodo in azione. La sintassi che permette di effettuare ciò è `Task.Factory.StartNew(()=>DoWork(max)).`
Ad un Task posso passare più dati ed il canale di comunicazione.
Vediamone un esempio.


```
        private void btnStartTask_Click(object sender, RoutedEventArgs e)
        {
            int max = Convert.ToInt32(txtInitial.Text);
            int delay = Convert.ToInt32(txtDelay.Text);

            Task.Factory.StartNew(() => DoWork(max, delay,lblCount));
        }

        void DoWork(int max, int delay, Label lbl)
        {
            for (int i = 0; i < max; i++)
            {
                Thread.Sleep(delay);
                Dispatcher.Invoke(() => UpdateUI(i,lbl));
            }
        }
       void UpdateUI(int i, Label lbl)
        {
            lbl.Content = i.ToString();
        }
```

Osserviamo alcuni strumenti che mette a disposizione VisualStudio con il Debug.
Mettendo un breakpoint e cliccando su Debug->Windows->Threads è possibile visualizzare l'elenco di tutti i threads in esecuzione.

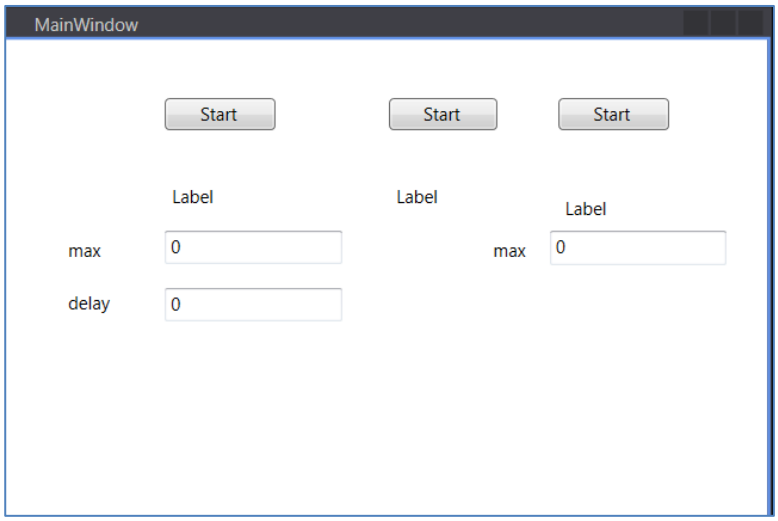


Mettendo un breakpoint e cliccando su Debug->Windows->Parallel Stacks è possibile visualizzare graficamente le relazioni tra i threads in esecuzione.



Esercizio.

Creare la seguente interfaccia grafica .



Rispettare le seguenti specifiche:

Creare 3 task in modo tale che:

1. un task deve svolgere un conteggio fino a 100 con un delay di 100 ms;
2. un task deve svolgere un conteggio fino ad un massimo stabilito dall'utente, con un delay di 100 ms;
3. un task deve svolgere un conteggio, dopo che l'utente abbia inserito in input sia il massimo sia il delay.

Soluzione

Il metodo DoWork e UpdateUI deve essere lo stesso per tutti i task.

```
private void DoWork(int max, int delay, Label lbl)
        {
            for (int i = 0; i < 100; i++)
            {
                Dispatcher.Invoke(() => UpdateUI(i,lbl));
                Thread.Sleep(100);
            }
        }
```

```
        private void UpdateUI(int i, Label lbl)
        {
            lbl.Content = i.ToString();
        }
```

Punto 1

```
        private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {

            Task.Factory.StartNew(() => DoWork(100,100, lblCountNoPar));
        }
```

Punto 2

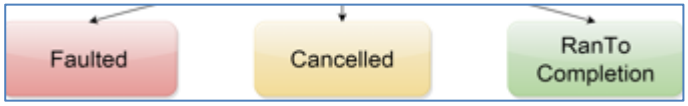
```

        private void btnStartTask_Click(object sender, RoutedEventArgs e)
        {
            int max = Convert.ToInt32(txtInitial.Text);

            Task.Factory.StartNew(() => DoWork(max, 100, lblCount)); }
```

Punto 3

```
        private void btnStartTaskLimitDelay_Click(object sender, RoutedEventArgs e)
        {
            int max2 = Convert.ToInt32(txtInitial2.Text);
            int delay = Convert.ToInt32(txtDelay2.Text);
            Task.Factory.StartNew(() => DoWork(max2, delay, lblCount2));
        }
```

Cancellare un task

Ricordiamo che un task può terminare per 3 motivi: Faulted (se si solleva un'eccezione), Cancelled (se viene cancellato), RanToCompletion (se termina la sua esecuzione).
Un task principale può richiedere al secondario di fermarsi. Il task secondario può valutare se può fermarsi in sicurezza e se è possibile salvare il suo stato.



Vediamo come è possibile fermare un Task, cioè come un task principale può richiedere al secondario (nel nostro esempio DoWork()) di fermarsi.

```
    bool stop = false;

        private void btncancTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            stop = true;
        }
```

Ricordiamo che un ciclo for può essere interrotto con un'istruzione `break;`

```
void DoWork(int max, int delay, Label lbl)
        {
            for (int i = 0; i < delay; i++)
            {
                Thread.Sleep(delay);
                Dispatcher.Invoke(() => UpdateUI(i,lbl));
                if (stop)
                    break;
            }
        }
```

```
  private void btnTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
       stop = false;
       Task.Factory.StartNew(()=>DoWork(100,100,lblCountNoPar));

        }
```

Vediamo come migliorare il codice in modo tale da rendere il meccanismo di stop valido per qualsiasi task in esecuzione.
Usiamo un oggetto di tipo `CancellationToken.` Qualsiasi task che possiede un CancellationToken può arrestarsi.
Per fermare contemporaneamente tutti i task, invece che dichiarare una variabile booleana, basta dichiarare fuori dai metodi un unico token e passarlo come parametro ai metodi.

```
CancellationTokenSource ct=new CancellationTokenSource();;
```

```
        private void btnTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            Task.Factory.StartNew(() => DoWork(100, 100, lblCount1, ct));
        }
```

In caso di richiesta di cancellazione tramite `if (ct.Token.IsCancellationRequested)` si interrompe il ciclo.

```
        void DoWork(int max, int delay, Label lbl, CancellationTokenSource ct)
        {
            for (int i = 0; i < max; i++)
            {

                Dispatcher.Invoke(() => UpdateUI(i,lbl));
                Thread.Sleep(delay);
                if (ct.Token.IsCancellationRequested)
                {
                   //Gracefully
                    break;

                }

            }
        }
```

Ed ecco come cambia il codice relativo al bottone che permette l'arresto.

```
      private void btncancTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //stop = true;
                ct.Cancel();
        }
```

Il Task principale nel momento in cui richiede ad un altro task di fermarsi non sa a che punto è arrivato. Il principio da seguire è il seguente: il task principale manda il token per richiedere di fermarsi in maniera gracefully, "con calma, in maniera aggraziata", in tal modo il secondario piò salvare in sicurezza il suo stato. Nel caso di richiesta "No Gracefully", per verificare se lo stop ha creato problemi oppure no è necessario sollevare un'eccezione. Vediamo come si modifica quindi il codice nella modalità "No Gracefully".

```
        private void DoWork(int max, int delay, Label lbl, CancellationTokenSource ct)
        {
            for (int i = 0; i < max; i++)
            {
                Dispatcher.Invoke(() => UpdateUI(i, lbl));
                Thread.Sleep(delay);
                if (ct.Token.IsCancellationRequested)
                {
                    // No Gracefully stop
                    ct.Token.ThrowIfCancellationRequested();
                    break;
                }
            }
        }
```

Il Token permette la comunicazione tra task principale e secondario e viceversa, mentre la variabile booleana non permette che il secondario possa comunicare al principale cosa sta accadendo al momento della cancellazione.

Per cancellare un task creato prima che parta si effettua la seguente modifica al codice del metodo

btnStartTaskNoPar_Click.

```
Task tsk = Task.Factory.StartNew(() => DoWork(100, 100, lblCountNoPar, ct),ct.Token);
```

Si aggiunge un parametro CancellationToken all'istruzione che crea un task.

Ecco il codice completo ed approfittiamone per effettuare altre osservazioni.
L'istruzione `ct = new CancellationTokenSource();` permette di rigenerare il token ogni volta in cui si clicca il pulsante start.

```
    private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            ct = new CancellationTokenSource();
        Task tsk = Task.Factory.StartNew(() => DoWork(100, 100, lblCountNoPar, ct),ct.Token);
            //tsk.Wait();
            MessageBox.Show("Finito");
        }
```

Se scriviamo `MessageBox.Show("Finito");` sembra che per il Task principale il secondario sia già terminato (anche se non è effettivamente così!). Occorrerebbe chiedere al Task principale di aspettare, ma l'istruzione `tsk.Wait();` blocca l'aggiornamento dell'interfaccia.

Esercizio.

Si consideri la seguente interfaccia grafica.

Si modifichi il codice secondo le seguenti specifiche:

1. Cliccando il button "Stop 1" si arresta il task attivato con "Start1".
2. Cliccando il button "Stop ultimi 2" si arrestano il task attivati con "Start2" e "Start3".
3. Cliccando il button "Stop tutti" si arrestano il task attivati con "Start1","Start2" e "Start3".

Soluzione

1. Per prima cosa creiamo un oggetto di tipo CancellationTokenSource.

```
CancellationTokenSource ct = new CancellationTokenSource();
```

Analizziamo le modifiche effettuate nel metodo relativo all'avvio del Task.

Le istruzioni in giallo sono necessarie dal secondo avvio del task, in quanto in seguito al primo stop il token è stato annullato. In alternativa, si potrebbe creare un nuovo token nel momento in cui è stato cancellato il task.

```
private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            if (ct == null)
                ct = new CancellationTokenSource();
            Task.Factory.StartNew(() => DoWork(100, 100, lblCountNoPar, ct));
        }
```

```
 private void btnStopTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
//L'if evita il sollevarsi di un'eccezione nel caso in cui si prema due volte consecutive il
button Stop1

         if (ct != null)
            {
                ct.Cancel();
                ct = null;
            }
//una volta cancellato potrei ricreare un nuovo Token

        }
```

2. E' necessario creare un secondo oggetto di tipo CancellationTokenSource.

```
CancellationTokenSource ct = new CancellationTokenSource();
CancellationTokenSource ct2 = new CancellationTokenSource();
```

Si ripete la stessa procedura vista al punto 1 su tutti i metodi interessati facendo attenzione al parametro passato `Task.Factory.StartNew(() => DoWork(max, 100, lblCount, ct2));`

```
 private void btnStartTask_Click(object sender, RoutedEventArgs e )
        {
            if (ct2 == null)
                ct2 = new CancellationTokenSource();
            int max = Convert.ToInt32(txtInitial.Text);
            Task.Factory.StartNew(() => DoWork(max, 100, lblCount, ct2));
        }
```

```
  private void btnStartTaskLimitDelay_Click(object sender, RoutedEventArgs e)
        {
            if (ct2 == null)
                ct2 = new CancellationTokenSource();
            int max = Convert.ToInt32(txtInitial2.Text);
            int delay = Convert.ToInt32(txtDelay2.Text);
            Task.Factory.StartNew(() => DoWork(max, delay, lblCount2, ct2));
        }
```

```
        private void btnStopTaskPar_Click(object sender, RoutedEventArgs e)
        {
            if (ct2 != null)
            {
                ct2.Cancel();
```

```
                ct2 = null;
            }
        }
```

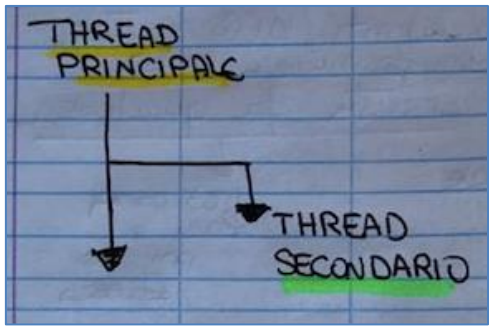
3. La modifica da fare è sul codice relativo al button “Stop tutti”.

```
private void btnStopTutti_Click(object sender, RoutedEventArgs e)
        {
            if (ct != null)
            {
                ct.Cancel();
                ct = null;
            }
            if (ct2 != null)
            {
                ct2.Cancel();
                ct2 = null;
            }
```

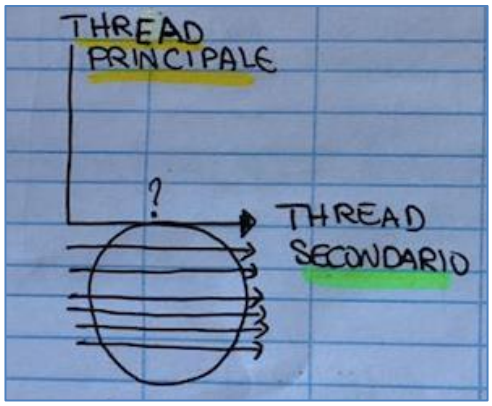
Multithreading Pattern

Fare multithreading significa che in un programma si divide l'operazione in tante parti; dal thread principale (il processo di partenza) parte almeno un thread secondario.
I due thread possono interagire in 4 modi differenti classificati in Multithreading Pattern.

Fire & Forget .
Il thread principale parte e ad un certo punto manda in esecuzione un thread secondario che lavora, ma al thread principale non interessa ciò che accade nell’esecuzione del secondario.
E’ come se si “dimenticasse” che un thread secondario sta eseguendo un lavoro).



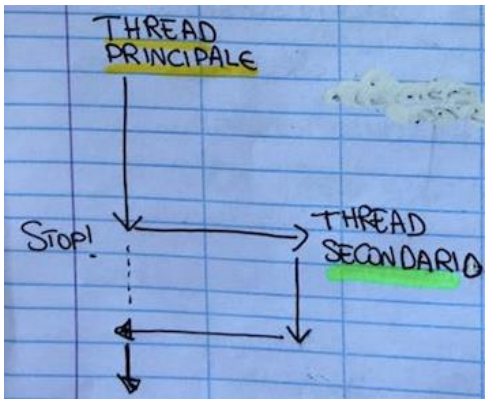
Polling:
Il thread principale fa partire un thread secondario e poi chiede a che punto è arrivato.
Il problema di questo pattern è che il thread principale perde tempo nell’interrogare il thread secondario.



Wait for Completion.

Il thread principale parte, ad un certo punto lancia un thread secondario; ma il thread principale resta fermo in attesa che il thread secondario ha terminato il suo completamento.

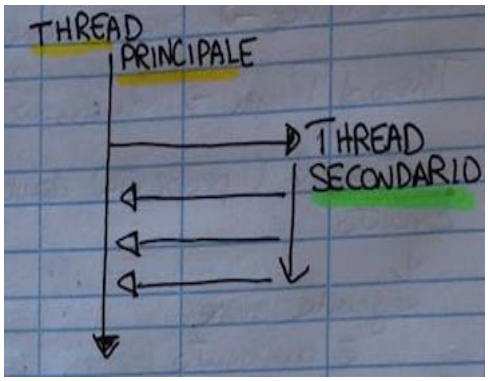
Ad esempio, finché un video non è scaricato non posso vederlo.



Completion Notification e Progress Notification.

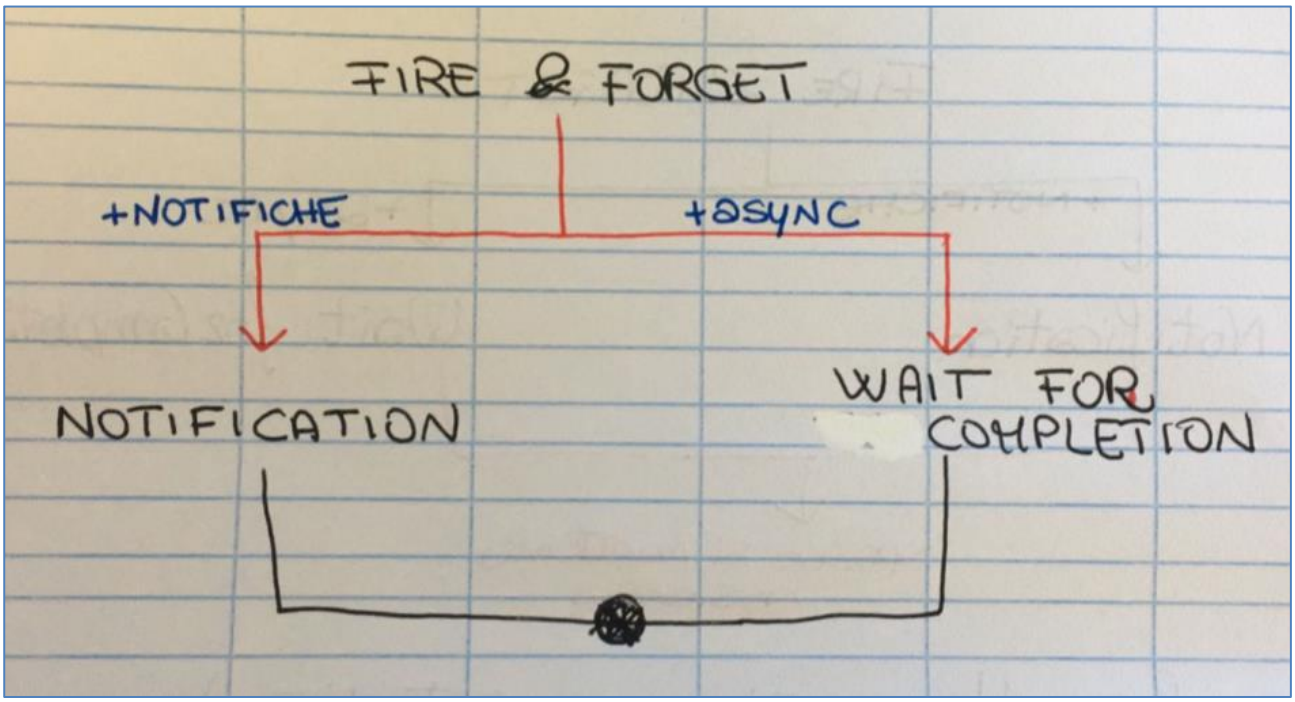
Il thread principale parte, ad un certo punto lancia un thread secondario che notificherà il suo completamento. In questo caso però il thread principale continua la sua esecuzione, non resta fermo in attesa. E' il pattern speculare del polling.

Completion Notification si specializza in Progress Notification, cioè il thread secondario, durante la sua esecuzione, aggiorna il principale con delle notifiche.



Aggiungendo al pattern multithread Fire & Forget (con un DoWork, un metodo per la creazione del Task e uno Start) altri "ingredienti si ottiene Completion Notification o Progress Notification (con notifiche finli o progressive), o Wait for Completion (con async ed await e senza notifiche.)

E’ possibile combinare anche Completion Notification e Wait for Completion, in questo modo il thread principale attende il termine dell’esecuzione del Thread secondario, ma nel frattempo ha notifiche sull’evoluzione del secondario.



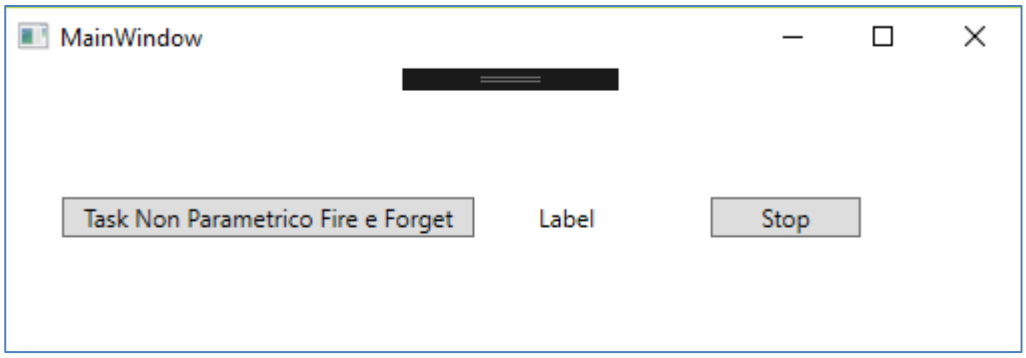
In .NET Framework sono disponibili tre modelli per l'esecuzione di operazioni asincrone:

- Modello di programmazione asincrona (APM) (detto anche modello IAsyncResult). Questo modello non è più consigliato per i nuovi sviluppi.
- Modello asincrono basato su eventi (EAP), nel caso Completion Notification.
- Modello asincrono basato su attività (TAP), che usa un unico metodo per rappresentare l'inizio e il completamento di un'operazione asincrona. TAP è stato introdotto in.NET Framework 4 ed è l'approccio consigliato per la programmazione asincrona in .NET Framework. Le parole chiave async, await e task in C# aggiungono supporto del linguaggio per TAP.

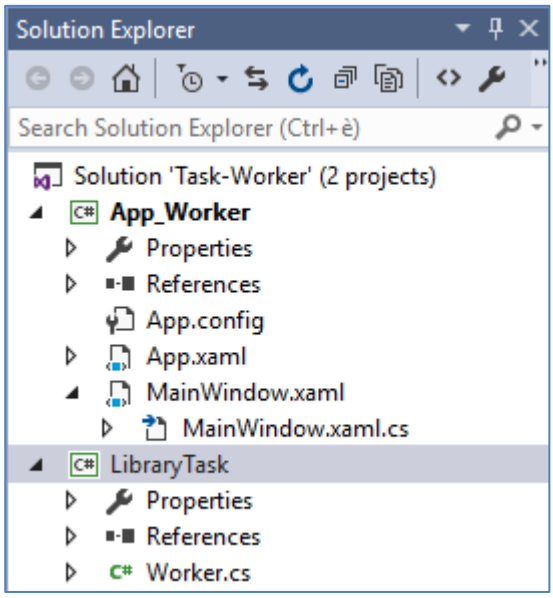
Analizziamo tutte e 4 le tipologie di pattern (Fire & Forget, Completion Notification, Wait for Completion, Completion Notification+Wait for completion).

Fire & Forget

Prima di implementare il pattern procediamo con la separazione del codice elaborativo da quello relativo all’interfaccia.
Creiamo una semplice interfaccia come quella in figura:



Per far ciò creiamo una libreria che conterrà una classe Worker; il worker sarà un oggetto che svolge il lavoro.



Spostiamo tutto il codice di elaborazione nella classe Worker.

Spostiamo nella classe il metodo che svolge il lavoro. Osserviamo che nel metodo DoWork() non sono necessari i parametri anche perché nella libreria di classi non è possibile far riferimento all'interfaccia grafica.

```
namespace LibraryTask
{

    public class Worker
    {
       //per creare l'oggetto worker sono necessarie 3 variabili: un token per fermare il
thread, un massimo per terminare il ciclo, un ritardo da considerare nel conteggio
        CancellationTokenSource _cts;
        int _max;
        int _limit;

        //costruttore
        public Worker(int max, int limit, CancellationTokenSource cts)
        {
            _limit = limit;
            _max = max;
            _cts = cts;
        }
        public void Start()
        { Task.Factory.StartNew(() => DoWork(_max, _limit)); }
```

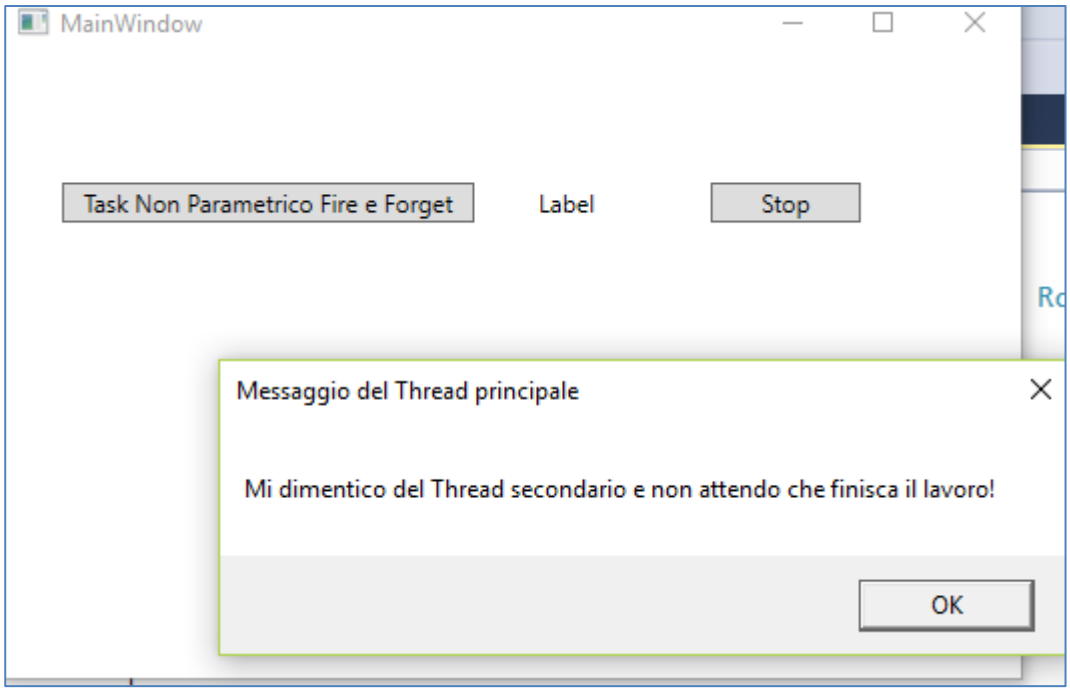
```
        // DoWork è privato perchè è utilizzato solamente all'interno della classe
        private void DoWork(int max, int delay)
        {
            for (int i = 0; i < max; i++)
            {
                Thread.Sleep(delay);
                if (_cts.IsCancellationRequested)
                    break;
            }
        }
    }
}
```

Passiamo alla parte relativa all'interfaccia.

Da ricordare di aggiungere al progetti wpf il riferimento alla libreria attraverso il percorso add-reference-LibraryTask e attraverso il comando:

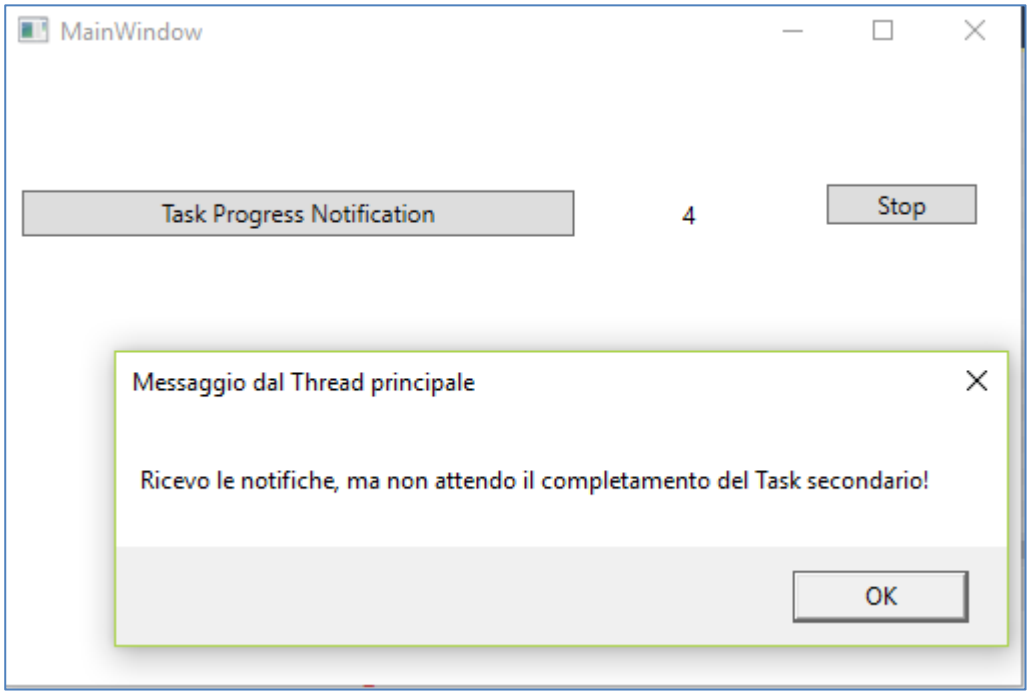
`using LibraryTask.`

```
//alter using
using LibraryTask;
using System.Threading;

namespace App_Worker
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        CancellationTokenSource cts;

        private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //creo token per fermare il task
            cts = new CancellationTokenSource();
            //creo un oggetto worker
            Worker wrk = new Worker(5, 1000, cts);
       //avvio il thread

            wrk.Start();
            //Osserviamo che in Fire e Forget appena si clicca start visualizza sua
qualsiasi istruzione successiva alal chiamata del thread, in questocaso la messageBox

            MessageBox.Show("Mi dimentico del Thread secondario e non attendo che finisca il
lavoro per visuualizzare la MessageBob!");
        }

        private void btnStop_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
                cts.Cancel();
        }
    }
}
```



Progress Notification

Vediamo come si modifica il Fire & Forget per avere notifiche durante del lavoro del thread secondario.



Vediamo le modifiche da fare alla classe worker.

```
namespace LibraryTask
{

    public class Worker
    {
        CancellationTokenSource _cts;
        int _max;
        int _limit;

        //per notifcare all'interfaccia uso un tipo di dato in grado di comunicare
all'interfaccia un valore
        IProgress<int> _progress;
```

```
        //il tipo del vaolre da notificare è tra parentesi angolari

        //anche il costruttore deve avere IProgress come parametro

        public Worker(int max, int limit, CancellationTokenSource cts, IProgress<int>
progress)
        {
            _limit = limit;
            _max = max;
            _cts = cts;
            _progress = progress;

        }
        public void Start()
        { Task.Factory.StartNew(() => DoWork(_max, _limit)); }


        private void DoWork(int max, int delay)
        {
            // per notificare il progresso è necessaria un metodo NotifyProgress da invocare

            for (int i = 0; i < max; i++)
            { NotifyProgress(_progress, i);
                Thread.Sleep(delay);
                if (_cts.IsCancellationRequested)
                    break;
            }
        }
        private void NotifyProgress(IProgress<int>progress, int i)
        {
            progress.Report(i);
        }
    }
}
```

Di seguito le modifiche al codice relativo all'interfaccia. Per aggiornare l'interfaccia ad ogni notifica è necessario il metodo UpdateUI.

```
namespace App_Worker
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;

        private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //Nel costruttore di Iprogress è necessario un'action
            //progress serve a far si che il task comunichi con l'interfaccia

            IProgress<int> progress = new Progress<int>(UpdateUI);
            //creo token per fermarlo
            cts = new CancellationTokenSource();

            //creo worker
            Worker wrk = new Worker(5, 1000, cts,progress);
            wrk.Start();

            //Osserviamo che vi è ancora una traccia del Fire e Forget
```

```
            //Infatti appena si clicca start si visualizza la messageBox mentre il thread
secondario continua a notificare!
            MessageBox.Show("Sono il Thread principale, ricevo le notifiche, ma non attendo
il completamento del Task secondario!");

        }
        private void UpdateUI(int i)
        { lblCount.Content = i.ToString(); }

        private void btnStop_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
                cts.Cancel();
        }
    }
}
```

Wait for Completion

Vediamo come si modifica il Fire & Forget per fare in modo che il thread principale aspetti il completamento del secondario per poi continuare con le proprie istruzioni.

Tali modifiche sono necessarie a partire dalla classe Worker.
Per convenzione si cambia il nome al metodo Start in `StartAsync()`

```
public void StartAsync()
        {
            Task.Factory.StartNew(() => DoWork(_max,_limit));
        }
```

Inoltre si marca il metodo con la parola chiave async:

```
public async void StartAsync()
        {
            Task.Factory.StartNew(() => DoWork(_max,_limit));
        }
```

Una volta inserito il marcatore si utilizza la seconda parola chiave await prima della chiamata al thread e si sostituisce il void con task perché il metodo deve restituire Task.

```
public async Task StartAsync()
        {
            await Task.Factory.StartNew(() => DoWork(_max,_limit));
        }
```

All'interno della classe si eliminano tutti i riferimenti relativi alle notifiche.

```

namespace LibraryTask
{

    public class Worker
    {
        CancellationTokenSource _cts;
        int _max;
```

```
        int _limit;
        //Rispetto al Completion Notification, si elemina tutte le parti di codice relative
alla notifica del progresso
        //IProgress<int> _progress;

 public Worker(int max, int limit, CancellationTokenSource cts /*,IProgress<int> progress*/)
        {
            _limit = limit;
            _max = max;
            _cts = cts;
            //_progress = progress;

        }

        //public void Start()
        //{ Task.Factory.StartNew(() => DoWork(_max, _limit)); }

        public async Task StartAsync()
        {await Task.Factory.StartNew(() => DoWork(_max, _limit)); }

        private void DoWork(int max, int delay)
        {
            //* per notificare il progresso:

            for (int i = 0; i < max; i++)
            { //se non voglio notificare
                //NotifyProgress(_progress, i);
                Thread.Sleep(1000);
                if (_cts.IsCancellationRequested)
                    break;
            }
        }
        //private void NotifyProgress(IProgress<int>progress, int i)
        //{
        //    progress.Report(i);
        //}
    }
}
```

Modifiche simili si fanno nella parte di codice relativa all'interfaccia.

```
public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;


        private async void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {

            //IProgress<int> progress = new Progress<int>(UpdateUI);

            //creo token per fermarlo
            cts = new CancellationTokenSource();

            //creo worker
            Worker wrk = new Worker(5, 1000, cts/*,progress*/);
            //invece di un Fire e Forget con notifiche
            //wrk.Start();
```

```
            //invoco il metodo startasync
            MessageBox.Show("Via al Thread secondario!!");
            await wrk.StartAsync();

            //Osserviamo che non è più Fire e Forget, ho la notifica quando il task
secondario termina!
            MessageBox.Show("Thread secondario terminato, posso riprendere il lavoro!");

        }



        private void UpdateUI(int i)
        { lblCount.Content = i.ToString(); }


        private void btnStop_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
                cts.Cancel();
        }
    }
}
```

Completion Notification + Wait for Completion

I due pattern precedenti si possono combinare insieme, semplicemente aggiungendo le notifiche al codice del Wait for Completion.
Vediamo il codice completo.

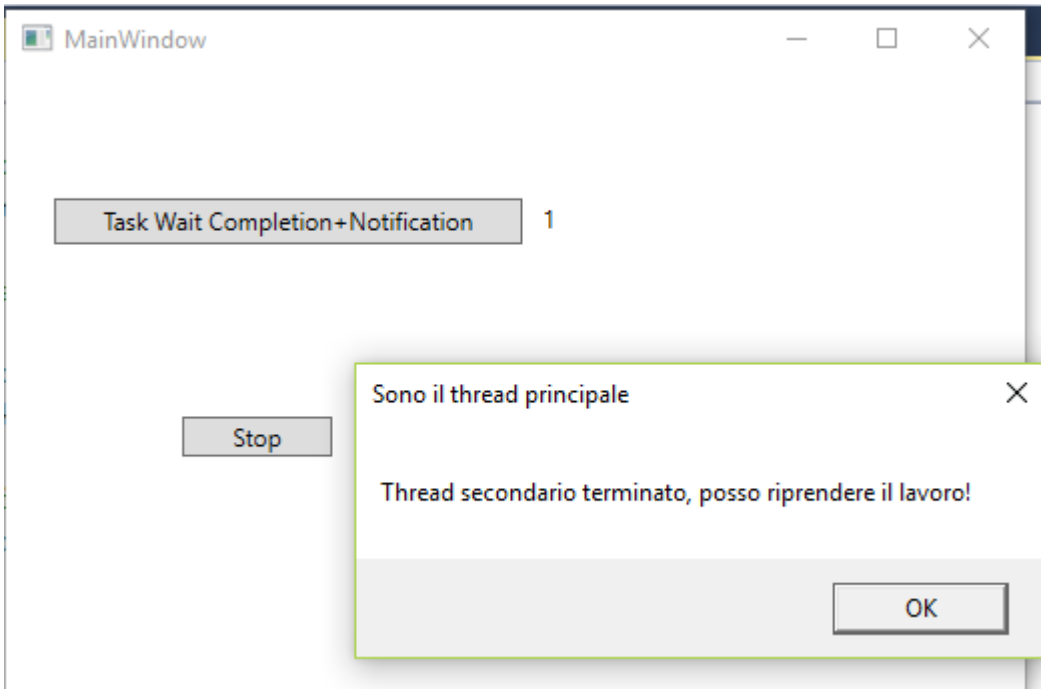
```
namespace LibraryTask
{

    public class Worker
    {

        CancellationTokenSource _cts;
        int _max;
        int _limit;

        //per notifcare all'interfaccia uso un tipo di dato in grado di comunicare
all'interfaccia un valore
        //il tipo del vaolre è tra parentesi angolari

        IProgress<int> _progress;

        //anche il costruttore deve avere IProgress come parametro

    public Worker(int max, int limit, CancellationTokenSource cts, IProgress<int> progress)
        {
            _limit = limit;
            _max = max;
            _cts = cts;
            _progress = progress;

        }
        public void Start()
        { Task.Factory.StartNew(() => DoWork(_max, _limit)); }
```

```
        public async Task StartAsync()
        {await Task.Factory.StartNew(() => DoWork(_max, _limit)); }

        // DoWork è privato perchè è utilizzato solamente all'interno della classe
        private void DoWork(int max, int delay)
        {
            //* per notificare il progresso:

            for (int i = 0; i < max; i++)
            { //se  voglio notificare
                NotifyProgress(_progress, i);
                Thread.Sleep(1000);
                if (_cts.IsCancellationRequested)
                    break;
            }

        }

        private void NotifyProgress(IProgress<int>progress, int i)
        {
            progress.Report(i);
        }
    }
}
```

```
namespace App_Worker
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;

        private async void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //Nel costruttore di Iprogress è necessario un'action
            //progress serve a far si che il task comunich con l'interfaccia

            IProgress<int> progress = new Progress<int>(UpdateUI);

            //creo token per fermarlo
            cts = new CancellationTokenSource();

            //creo worker
            Worker wrk = new Worker(5, 1000, cts,progress);
            //invece di un Fire e Forget con notifiche
            //wrk.Start();
            //invoco il metodo startasync
            MessageBox.Show("Via al Thread secondario!!");
            await wrk.StartAsync();

            //Osserviamo che non è più Fire e Forget, ho la notifica quando il task
secondario termina!
            MessageBox.Show("Thread secondario terminato, posso riprendere il lavoro!");

        }
        private void UpdateUI(int i)
```

```
        { lblCount.Content = i.ToString(); }

        private void btnStop_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
                cts.Cancel();
        }
    }
}
```

Ed ecco il risultato finale:

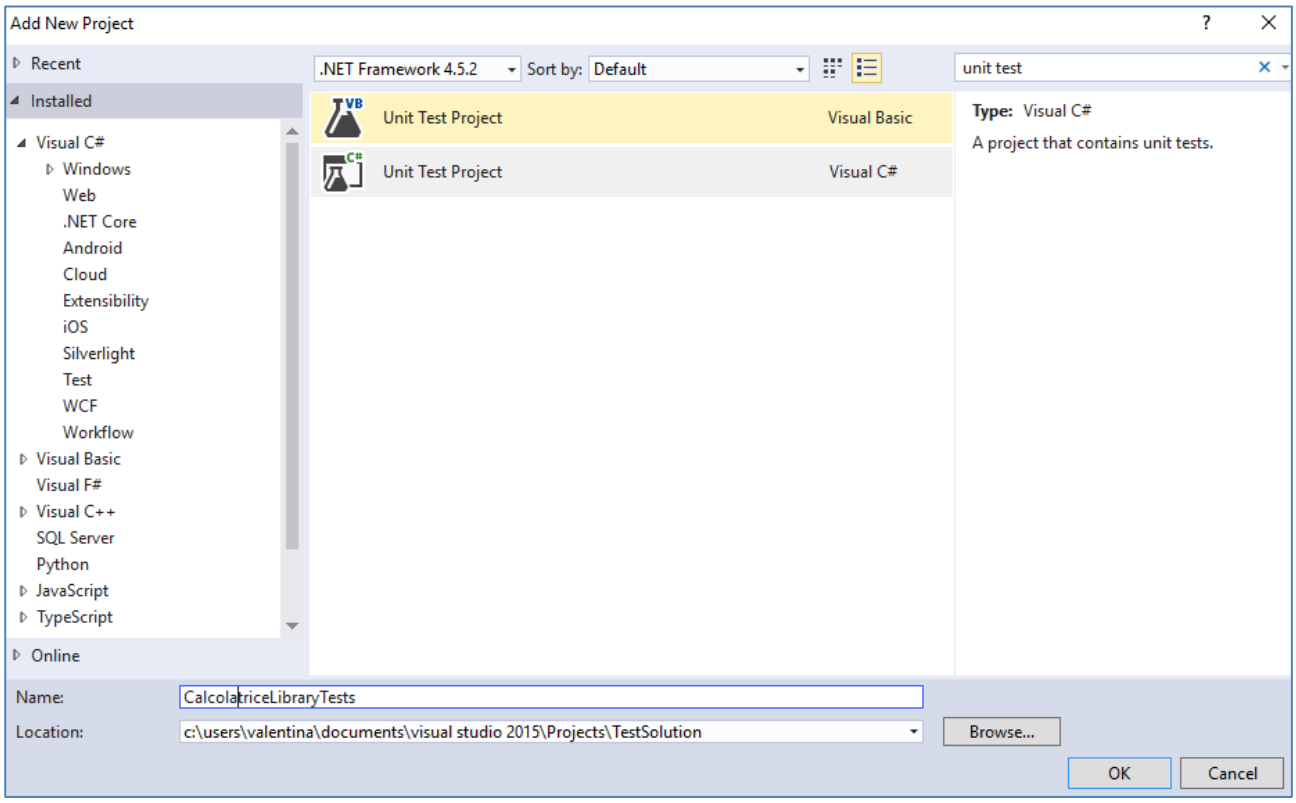


Unit Test

Lo unit testing è una porzione di codice che serve a testare una singola funzionalità, cioè una porzione di codice (metodi, funzione, classi) che ha una singola responsabilità.
Il test viene svolto anche per le User Interface per validare l'interazione con l'utente, in tal caso lo unit test prende il nome di UI testing. La metodologia utilizzata è chiamata Test Driven Development che si basa su 3 fasi ripetute: Red-Green-Refactor.
Fase Red: Si scrive il test. Regola del TDD è "mai scrivere codice di produzione senza aver scritto test rosso"
Fase Green: Si scrive il minimo codice possibile che faccia funzionare i test.
Fase Refactor: si ottimizza il codice.

Unit Test in Visual Studio

Andiamo a creare un Blank Solution con una Library ed una classe.
Per fare i test occorre creare un nuovo tipo di Progetto alla Solution (Add-NewProject-UnitTest) .
Ogni classe dell'applicazione avrà una corrispondente classe nel TestProject.
Aggiungiamo il nuovo progetto. Normalmente il progetto prende il nome della Library con l'aggiunta della parola Tests in fondo. Esiste un meccanismo che crea il automatico il test cliccando sul tasto destro sul nome del metodo da testare e selezionando CreatTest.
Ricordare di aggiungere la reference della Libreria da testare al progetto di Test.

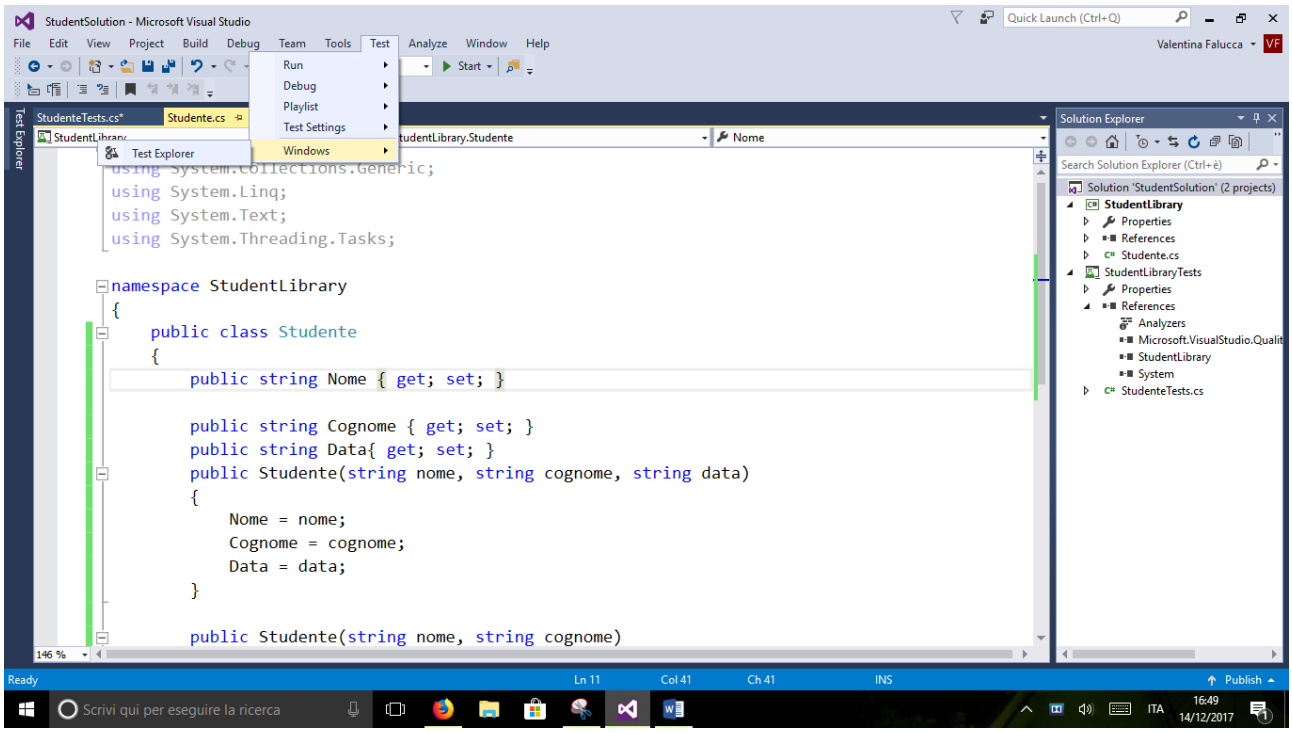


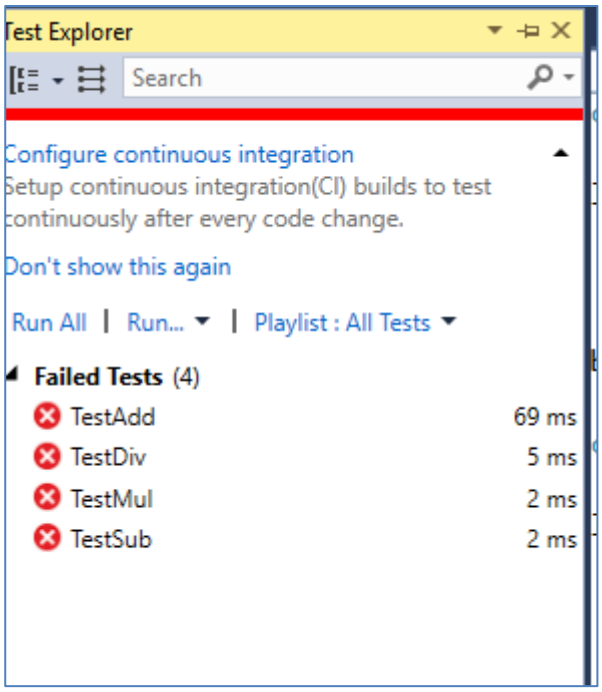
In questo esempio si è creata una Libreria denominata CalcolatriceLibrary, con una classe Calcolatrice e una libreria CalcolatriceLibraryTests che fa riferimento a Calcolatriice Library (Add-Reference-spuntare Calcolatrice Library).
Iniziamo a scrivere il codice di test per il metodo che fa la somma. Il test passerà i parametri 3 e 4 al

metodo Add della classe Calcolatrice aspettandosi come risultato 7 per far sì che il test sia passato.

```
using CalcolatriceLibrary;

namespace CalcolatriceLibraryTests
{
    [TestClass]
    public class CalcolatriceTests
    {
        //le paretesi quadre indicano che il metodo serve a fare il test
        [TestMethod]
        public void TestAdd()
        {
            int somma = Calcolatrice.Add(3, 4);
            //scriviamo il controllo
            //le righe di codice vengono chiamate "asserzione"
            //if (somma == 7)
            //    Console.WriteLine("ok");

       //pertanto voglio affermare che per passare il test 7 deve essere uguale alla somma
            Assert.AreEqual(7, somma);
        }
    }
}
```

A questo punto si apre la finestra Test Explorer (Test-Windows-Test Explorer) e cliccando su Run All vedremo che il nostro test è nella fase red. Il test non può passare perché non si è scritto il codice relativo al metodo Add della classe.



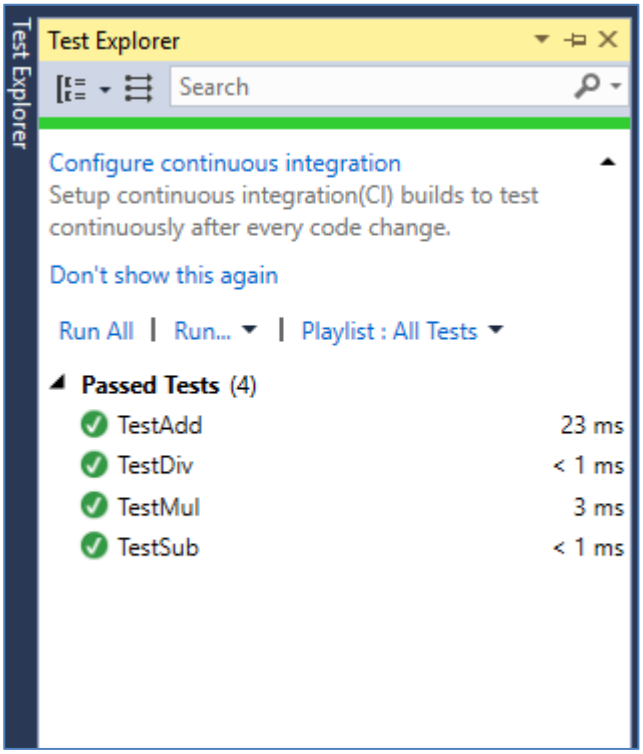


Passiamo quindi alla fase verde scrivendo, all'interno della classe Calcolastrice, il minimo codice che permetta di far passare il test rispettando la regola "Non scrivere più codice di produzione di quello necessario per far passare il test correntemente rosso."

.

```
public static int Add(int n1, int n2)
        {
            int resp = 0;
            resp = n1 + n2;

            return resp;
        }

        public static int Mul(int n1, int n2)
        {
            int resp = 0;
            resp = n1 * n2;

            return resp;
        }

        public static int Div(int n1, int n2)
        {
            int resp = 0;
            resp = n1 / n2;

            return resp;
        }

        public static int Sub(int n1, int n2)
        {
            int resp = 0;
            resp = n1 - n2;

            return resp;
        }
    }
```

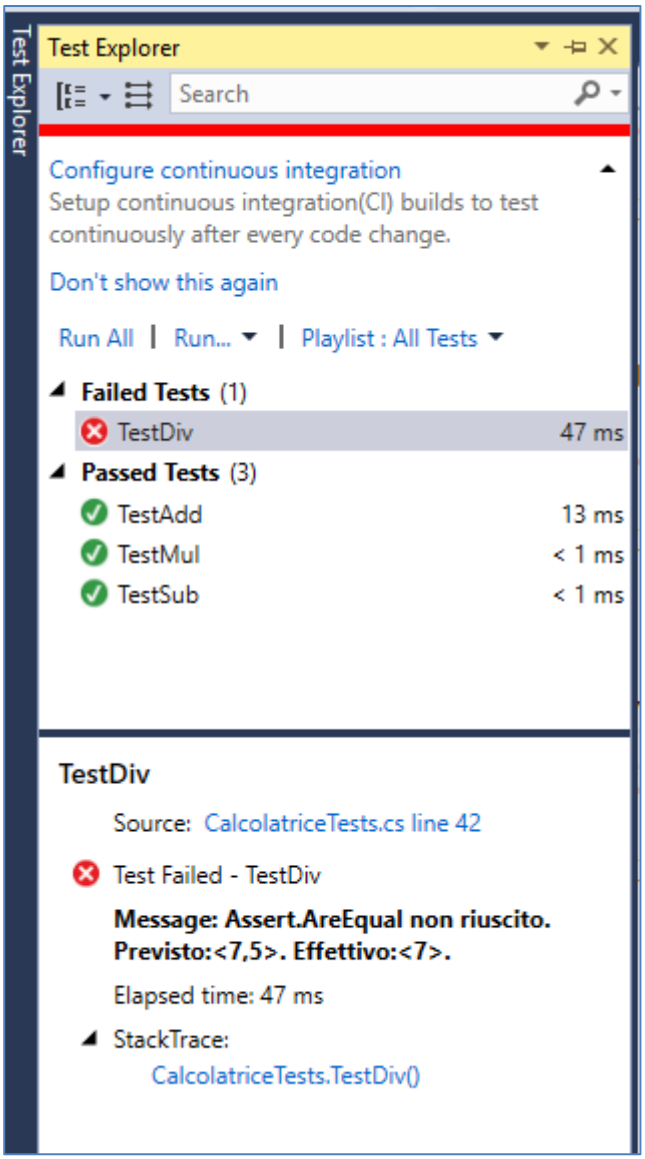
A questo punto, l'esecuzione del test restituisce esito positivo.



Si provi a modificare un metodo ad a ripetere il test.

```
[TestMethod]
        public void TestDiv()
        {
            int resp = Calcolatrice.Div(30, 4);

            Assert.AreEqual(7.5, resp);
        }
```

Ovviamnete il test dà esito negativo, fallisce, in quanto il risultato del metodo dichiarato intero) non può restituire 7.5!



E' necessario dunque modificare il codice del metodo che effettua la divisione.

```
        public static double Div(double n1, double n2)
        {
            double resp = 0;
            resp = n1 / n2;

            return resp;
        }
```

```
[TestMethod]
        public void TestDiv()
        {
            double resp = Calcolatrice.Div(30, 4);

            Assert.AreEqual(7.5, resp);
        }
```

Aggiungiamo un esempio di test che controlla la divisione per zero. Ovviamente per ogni metodo si può scrivere più di un test!

```
[TestMethod]
```

```
        public void TestDivByZero()
        {
            double resp = Calcolatrice.Div(30, 0);

            Assert.AreEqual(Double.PositiveInfinity, resp);
        }
```

Unit Test per la programmazione ad oggetti.

Esempio 1

Si prenda in considerazione la seguente classe.

```
namespace StudentLibrary
{
    public class Studente
    {
        public string Nome { get; set; }
        public string Cognome { get; set; }
        public string Data{ get; set; }
        public Studente(string nome, string cognome, string data)
        {
            Nome = nome;
            Cognome = cognome;
            Data = data;
        }

        public Studente(string nome, string cognome)
        {
            Nome = nome;
            Cognome = cognome;
        }

      public Studente()
        { }

        public override string ToString()
        {
            return $"{ Nome},{ Cognome},{ Data}";
        }

        public static Studente Parse(string str)
        {
            string[] campi = str.Split(',');
            Studente st = new Studente();
            st.Nome = campi[0];
            st.Cognome = campi[1];
            st.Data = campi[2];
            return st;
        }
    }
}
```

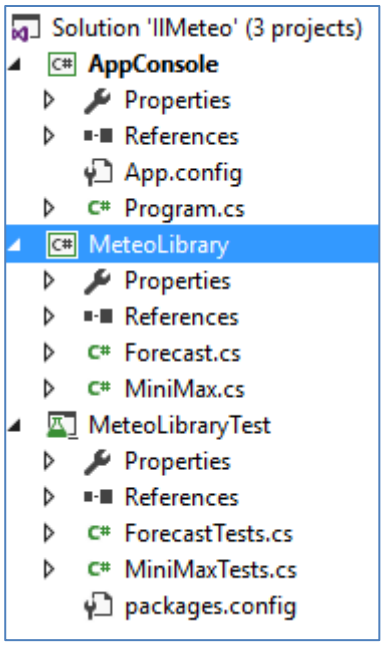
Il Corrispondente Test per ogni metodo della classe è il seguente.

```
namespace StudentLibraryTests
{
    [TestClass]
    public class StudenteTests
```

```
    {
        [TestMethod]
        public void TestCostruttore3Parametri()
        {
            Studente stud = new Studente("Valentina", "Falucca", "05/01/1982");
            //come si testa un costruttore?
            Assert.AreEqual("Valentina", stud.Nome);
            Assert.AreEqual("Falucca", stud.Cognome);
            Assert.AreEqual("05/01/1982", stud.Data);
        }

        [TestMethod]
        public void TestCostruttore2Parametri()
        {
            Studente stud = new Studente("Valentina", "Falucca");
            //come si testa un costruttore?
            Assert.AreEqual("Valentina", stud.Nome);
            Assert.AreEqual("Falucca", stud.Cognome);
        }

        [TestMethod]
        public void TestToString()
        {

            Studente stud = new Studente("Valentina", "Falucca", "05/01/1982");
            string resp = stud.ToString();
            Assert.AreEqual("Valentina,Falucca,05/01/1982", resp);
        }

        [TestMethod]
        public void TestPrse()
        {

            Studente stud = Studente.Parse("Valentina,Falucca,05/01/1982");
            Assert.AreEqual("Valentina", stud.Nome);
            Assert.AreEqual("Falucca", stud.Cognome);
            Assert.AreEqual("05/01/1982", stud.Data);
        }
    }
}
```

Esempio 2

Vediamo come per ogni metodo di una classe è possibile scrivere il corrispondente Test. Creiamo una soluzione vuota denominata IlMeteo contenente una Library (MeteoLibrary) ed un progetto Console (AppConsole). Ovviamente per ciascuna Libreria e ciascuna cclasse va creato il corrispondente UnitTest (Add-NewProject-UnitTest).
Ecco come deve essere strutturata la Solution al termine delle operazioni preliminari.



Riportiamo il codice della classe meteo ed il corrispondente UnitTest.

```
public class MiniMax
    {
        public int Min { get; set; }
        public int Max { get; set; }
  public MiniMax()
        {

        }
//codice in tabella
    }
```

```
namespace MeteoLibraryTest
{
    [TestClass]
    public class MiniMaxTests
    {
//test in tabella (colonna di destra)
    }
}
```

MiniMax.cs	MiniMaxClassTests.cs

`public MiniMax(int min, int max):this()`
`{`
`Min = min;`
`Max = max;`
`}`
	`[TestMethod]`
	`public void TestConstruttore2Int()`
	`{`
	`MiniMax m1 = new MiniMax(-5, 7);`
	`Assert.AreEqual(-5, m1.Min);`
	`Assert.AreEqual(7, m1.Max);`
	`}`

`public MiniMax(string min, string max):this()`
`{`
`Min = int.Parse(min);`
`Max = int.Parse(max);`
`}`
	`[TestMethod]`
	`public void TestConstruttore2Strin()`
	`{`
	`MiniMax m1 = new MiniMax("-5","7");`
	`Assert.AreEqual(-5, m1.Min);`
	`Assert.AreEqual(7, m1.Max)`
	`}`

MiniMax.cs

```
  public static MiniMax Parse(string str)
  {
    MiniMax resp = new MiniMax();
      try
       {
       string[] campi = str.Split('/');
       if (campi.Length == 2)
             {
                string min = campi[0];
                string max = campi[1];
                resp = new MiniMax(min, max);
             }
       }
       catch (Exception)
           {

           }
           return resp;
       }
```

Il metodo non restituisce eccezioni se la lunghezza del vettore campi è diversa da due. Pertanto si potrebbe migliorare il codice.

```
public static MiniMax Parse(string str)
     {
       MiniMax resp = new MiniMax();
       try
       {
         string[] campi = str.Split('/');
            //if (campi.Length == 2)
                {
                string min = campi[0];
                string max = campi[1];
                resp = new MiniMax(min, max);
                }
            }
            catch (Exception)
            {
                resp = null;
            }
            return resp;
       }
```

```
  public override string ToString()
       {
           return $"{Min}/{Max}";
       }
```

```
   public override int GetHashCode()
       {
           return Min ^ Max;
       }
```

```
public override bool Equals(Object obj)
{
            // Check for null values and compare run-
time types.
if (obj == null || GetType()!= obj.GetType())
          return false;


MiniMax s = (MiniMax)obj;
return (Min == s.Min && Max == s.Max);
```

MiniMaxClassTests.cs

```
 [TestMethod]
 public void TestParse()
  {
    MiniMax m = MiniMax.Parse("-7/3");
    Assert.AreEqual(-7, m.Min);
    Assert.AreEqual(3, m.Max);
   }
```

E' possibile scrivere più di un test per ogni metodo della classe. Ad esempio il seguente testa la correttezza del separatore

```
[TestMethod]
 public void TestParseNoSeparator()
   {
  MiniMax m = MiniMax.Parse("-7,3");
  Assert.AreEqual(0, m.Min);
  Assert.AreEqual(0, m.Max);
  }
```

Il test precedente, nel caso in cui il separatore sia errato, ci permette di riflettere sul reale contenuto che dovrebbe avere il minimo ed il massimo (null e non zero!) Modifichiamo quindi il test e di conseguenza il codice del metodo Parse.

```
 [TestMethod]
public void TestParseNoSeparator()
    {
       MiniMax m = MiniMax.Parse("-7,3");
       Assert.IsNull(m);

     }
```

```
[TestMethod]
   public void TestToString()
     {
        MiniMax m = new MiniMax();
        m.Min = -7;
        m.Max = 3;
        string resp = m.ToString();
        Assert.AreEqual(resp, "-7/3");
        }
```

```
[TestMethod]
  public void TestGetHasCode()
    {
      MiniMax m1 = new MiniMax(-1, 3);
      int resp = m1.GetHashCode();
      Assert.AreEqual(resp, -4);
        }
```

```
[TestMethod]
 public void TestEquals()
   {
   MiniMax m1 = MiniMax.Parse("-7/3");
   MiniMax m2 = MiniMax.Parse("-7/3");

       bool resp = m1.Equals(m2);
       Assert.AreEqual(true, resp);
   }
```

MiniMax.cs	MiniMaxClassTests.cs

}	[TestMethod]
	public void TestNotEquals()
	{
	MiniMax m1 = MiniMax.Parse("-7/3");
	MiniMax m2 = MiniMax.Parse("7/3");
	
	bool resp = m1.Equals(m2);
	Assert.AreEqual(false, resp);
	}

	[TestMethod]
	public void TestEqualsWithDifferentType()
	{
	int i = 10;
	MiniMax m = new MiniMax();
	bool resp = m.Equals(i);
	Assert.AreEqual(false, resp);
	}

	[TestMethod]
	public void TestEqualsWithNull()
	{
	MiniMax m = new MiniMax();
	bool resp = m.Equals(null);
	Assert.AreEqual(resp, false);
	}

public static bool operator ==(MiniMax s1, MiniMax s2)
{
return s1.Equals(s2);
}
	[TestMethod]
	public void TestOperatorEquals()
	{
	MiniMax m1 = new MiniMax(10, 2);
	MiniMax m2 = new MiniMax(10, 2);
	bool resp = m1 == m2;
	
	Assert.AreEqual(true, resp);
	}

public static bool operator !=(MiniMax s1, MiniMax s2)
{
return !s1.Equals(s2);
}
	[TestMethod]
	public void TestOperatorNotEquals()
	{
	MiniMax m1 = new MiniMax(10, 3);
	MiniMax m2 = new MiniMax(10, 2);
	bool resp = m1 != m2;
	
	Assert.AreEqual(true, resp);
	}

	E' possibile testare anche elementi di una lista di oggetti.
	
	[TestMethod]
	public void TestIndexOf()
	{
	MiniMax m1 = MiniMax.Parse("-7/3");
	MiniMax m2 = MiniMax.Parse("7/3");
	MiniMax m3 = MiniMax.Parse("-7/-3");
	List<MiniMax> min = new List<MiniMax>();
	min.Add(m1);
	min.Add(m2);
	min.Add(m3);
	
	int pos = min.IndexOf(m2);
	
	Assert.AreEqual(1, pos);
	
	}

Vedimo la classe Forecast.cs e la relativa classe Test ForecastTests.cs:

```
namespace MeteoLibrary
{
    public class Forecast
    {
        public string GiornoSettimana { get; set; }
        public int Giorno { get; set; }
        public MiniMax Temp { get; set; }
//Codice in tabella
}
```

```
[TestClass]
    public class ForecastTests
    {
//codice in tabella
}
```

Forecast.cs	ForecastTests.cs

`public Forecast(string giornoSettimana, string giorno, string temp)`
`{`
`GiornoSettimana = giornoSettimana;`
`Giorno = int.Parse(giorno);`
`Temp = MiniMax.Parse(temp);`
`}`
	`[TestMethod]`
	`public void TestConstruttore3Par()`
	`{`
	`Forecast f = new Forecast("Lunedì","20","-7/3");`
	`Assert.AreEqual("Lunedì", f.GiornoSettimana);`
	`Assert.AreEqual(20, f.Giorno);`
	`Assert.AreEqual(-7, f.Temp.Min);`
	`Assert.AreEqual(3, f.Temp.Max);`
	`}`

`public Forecast()`
`{ Temp =new MiniMax(); }`
	`[TestMethod]`
	`public void TestConstruttoreNoPar()`
	`{`
	`Forecast f = new Forecast();`
	`Assert.AreEqual(0, f.Temp.Min);`
	`Assert.AreEqual(0, f.Temp.Max);`
	`}`

`public static Forecast Parse(string str)`
`{`
`string[] campi = str.Split(',');`
`string giornoSettimana = campi[0];`
`string giorno = campi[1];`
`string temp = campi[2];`
`Forecast resp;`
`resp=newForecast(giornoSettimana, giorno, temp);`
`return resp;`
`}`
	`[TestMethod]`
	`public void TestParse()`
	`{`
	`Forecast f = Forecast.Parse("Mercoledì,20,-1/7");`
	`Assert.AreEqual("Mercoledì", f.GiornoSettimana);`
	`Assert.AreEqual(20, f.Giorno);`
	`Assert.AreEqual(-1, f.Temp.Min);`
	`Assert.AreEqual(7, f.Temp.Max);`
	`MiniMax temp = new MiniMax(-1, 7);`
	`Assert.AreEqual(temp, f.Temp);`
	`}`

`public override string ToString()`
`{`
`return $"{GiornoSettimana},{Giorno},{Temp}";`
`}`
	`[TestMethod]`
	`public void TestToString()`
	`{`
	`Forecast f =new Forecast();`
	`f.Giorno = 20;`
	`f.GiornoSettimana = "Mercoledì";`
	`f.Temp.Min = -1;`
	`f.Temp.Max = 7;`
	`string resp = f.ToString();`
	`Assert.AreEqual(resp,"Mercoledì,20,-1/7");`
	`}`

`public override bool Equals(Object obj)`
`{`
`// Check for null values and compare run-time types.`
`if (obj == null || GetType() != obj.GetType())`
`return false;`
`Forecast s = (Forecast)obj;`
	`[TestMethod]`
	`public void TestEquals()`
	`{`
	`Forecast f1 = Forecast.Parse("Mercoledì,20,-1/7");`
	`Forecast f2 = Forecast.Parse("Mercoledì,20,-1/7");`
	`bool resp = f1.Equals(f2);`
	`Assert.AreEqual(true, resp);`
	`}`

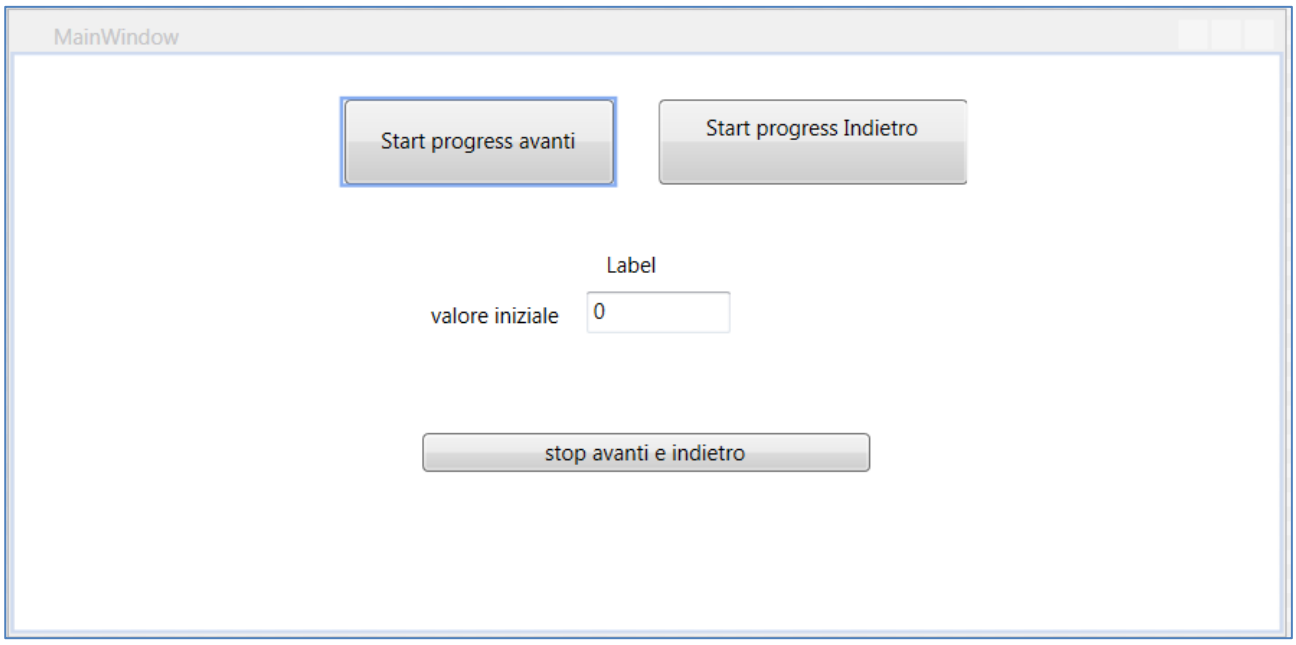
Forecast.cs	ForecastTests.cs

return (Temp == s.Temp);
}
 	
public static bool operator==(Forecast s1, Forecast s2)
{
return s1.Equals(s2);
}
 	
public static bool operator !=(Forecast s1,Forecast s2)
{
return !s1.Equals(s2);
}
	//le temperature devono essere differenti
	[TestMethod]
	public void TestNotEquals()
	{
	Forecast f1 = Forecast.Parse("Mercoledì,20,-1/7");
	Forecast f2 = Forecast.Parse("Mercoledì,20,-2/7");
	
	bool resp = f1.Equals(f2);
	Assert.AreEqual(false, resp);
	}
	
	[TestMethod]
	public void TestEqualsWithDifferentType()
	{
	int i = 10;
	Forecast f1 = Forecast.Parse("Mercoledì,20,-1/7");
	bool resp = f1.Equals(i);
	
	Assert.AreEqual(false, resp);
	}
	
	[TestMethod]
	public void TestEqualsWithNull()
	{
	Forecast f1 = new Forecast();
	bool resp = f1.Equals(null);
	
	Assert.AreEqual(resp, false);
	}
	
	[TestMethod]
	public void TestOperatorEquals()
	{
	Forecast f1 = Forecast.Parse("Mercoledì,20,-1/7");
	Forecast f2 = Forecast.Parse("Mercoledì,20,-1/7");
	bool resp = f1 == f2;
	
	Assert.AreEqual(true, resp);
	}
	
	[TestMethod]
	public void TestOperatorNotEquals()
	{
	Forecast f1 = Forecast.Parse("Mercoledì,20,-1/7");
	Forecast f2 = Forecast.Parse("Mercoledì,20,-9/7");
	bool resp = f1 != f2;
	
	Assert.AreEqual(true, resp);
	}

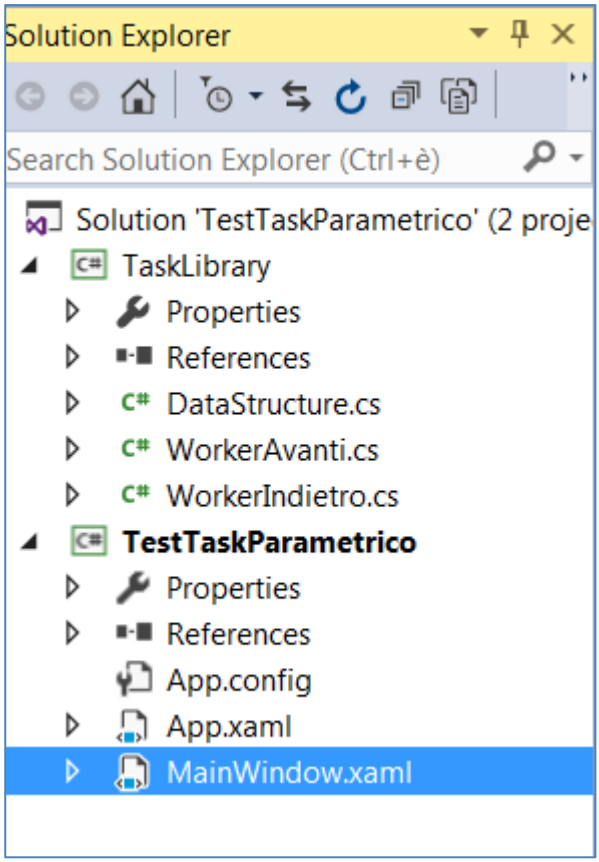
Thred che lavorano sugli stessi dati

In alcune situazioni 2 o più thread competono su dati condivisi. E' necessario sincronizzare i lavori dei thread in modo tale che non si verifichino interferenze.

Consideriamo il seguente problema: due thread lavorano su uno stessa variabile: un incrementa di 100 il valore contenuto nella variabile ed uno lo decrementa di 100.



La solution avrà una Library con 3 classi: una di queste classi è un DTO che permette di creare una struttura dati, una classe WorkerAvanti che permetterà di incrementare il valore della struttura dati ed una classe WorkerIndietro eddettuerà il decremento.



Vediamo nel dettaglio il codice relativo alle classi in questione:

```
namespace TaskLibrary
{
    public class DataStructure
    {
       public int Valore { get; set; }
    }
}
```

```
namespace TaskLibrary
{
    public class WorkerAvanti
    {
        CancellationTokenSource _cts;
        DataStructure _data;
        IProgress<int> _progress;

        public WorkerAvanti(DataStructure data, CancellationTokenSource cts, IProgress<int>
progress)
        {
            _data = data;
            _cts = cts;
            _progress = progress;

        }
        public void Start()
        {
            Task.Factory.StartNew(() => DoWork(_data));
        }
        private void DoWork(DataStructure _data)
        {
            for (int i = 0; i < 100; i++)
            {
                _data.Valore++;
                NotifyProgress(_progress, _data.Valore);
                Thread.Sleep(100);
                if (_cts.IsCancellationRequested)
                    break;
            }
        }

        private void NotifyProgress(IProgress<int> progress,int i)
        {
            progress.Report(i);
        }
    }
}
```

```
namespace TaskLibrary
{
    public class WorkerIndietro
    {
        CancellationTokenSource _cts;
        DataStructure _data;
        IProgress<int> _progress;
```

```
        public WorkerIndietro(DataStructure data, CancellationTokenSource cts,
IProgress<int> progress)
        {
            _data = data;
            _cts = cts;
            _progress = progress;
        }
        public void Start()
        {
            Task.Factory.StartNew(() => DoWork(_data));
        }
        private void DoWork(DataStructure _data)
        {
            for (int i = 0; i < 100; i++)
            {
                _data.Valore--;
                NotifyProgress(_progress, _data.Valore);
                Thread.Sleep(100);
                if (_cts.IsCancellationRequested)
                    break;
            }
        }

        private void NotifyProgress(IProgress<int> progress, int i)
        {
            progress.Report(i);
        }
    }
}
```

Vediamo le modifiche alla parte di codice relativo alla UI.

```
namespace TestTaskParametrico
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;
        DataStructure data;

        private void btnStartAvanti_Click(object sender, RoutedEventArgs e)
        {
            IProgress<int> progress = new Progress<int>(UpdateUI);
            if (cts == null)
            cts = new CancellationTokenSource();
            int valoreIniziale = int.Parse(txtInizio.Text);
            if (data == null)
                data = new DataStructure() { Valore = valoreIniziale };
            WorkerAvanti wrkav = new WorkerAvanti(data, cts, progress);
            wrkav.Start();

        }

        private void UpdateUI(int i)
```

```
        {
            lblConta.Content = i;
        }

        private void btnStartIndietro_Click(object sender, RoutedEventArgs e)
        {
            IProgress<int> progress = new Progress<int>(UpdateUI);
            if (cts == null)
            cts = new CancellationTokenSource();
            int valoreIniziale = int.Parse(txtInizio.Text);
            if (data == null)
                data = new DataStructure() { Valore = valoreIniziale };
            WorkerIndietro wrkin=new WorkerIndietro(data, cts, progress);
            wrkin.Start();
        }

        private void btnStopTutto_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)

                cts.Cancel();

        }
    }
}
```

All'avvio di entrambi i thread si verifica una situazione di interferenza tra i due thread perché entrambi si stanno contendendo la variabile.
La sezione critica si stabilisce mediante una parola chiave lock.
Vediamo come viene utilizzata modificando il metodo DoWork nella classe WorkerIndietro

```
private void DoWork(DataStructure _data)
        {
            lock(_data)
            {
            for (int i = 0; i < 100; i++)
            {
                _data.Valore--;
                NotifyProgress(_progress, _data.Valore);
                Thread.Sleep(100);
                if (_cts.IsCancellationRequested)
                    break;
            }
            }
        }
```

e nella classe WorkerAvanti

```
private void DoWork(DataStructure _data)
        {
            lock(_data)
            {
            for (int i = 0; i < 100; i++)
            {
                _data.Valore--;
                NotifyProgress(_progress, _data.Valore);
                Thread.Sleep(100);
                if (_cts.IsCancellationRequested)
```

```
                    break;
                }
                }
            }
```