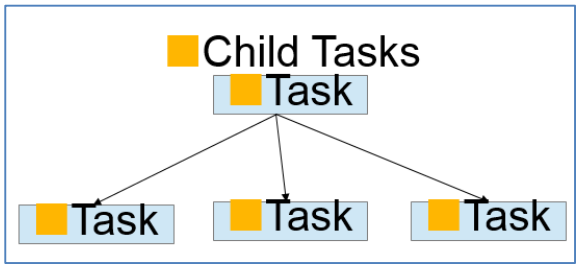
Sommario

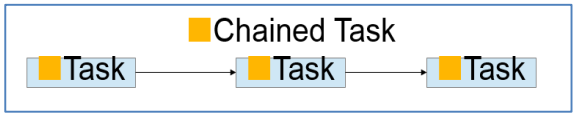
Introduzione alla programmazione Multithreading

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.
I due thread possono interagire in 4 modi differenti classificati in Multithreading Pattern.

Prime definizioni.

Thread: unità base del processo. Almeno un Thread è sempre presente (è il programma principale). Dal Task principale si generano i Task secondari.
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.

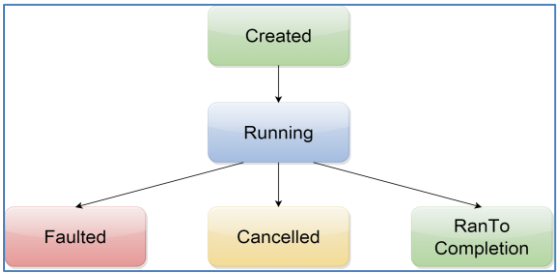




Thread in background
Un thread principale parte e quando temina tutti i suoi thread secondari.
Generalmente i threads sono in background. La vita di un secondario dipende dal principale.

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

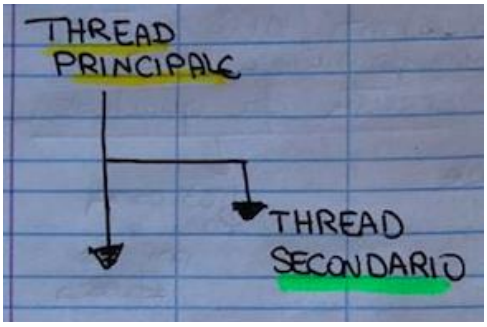
Stati del thread.
A differenza dalla teoria, nella pratica si usano 3 stati (Creato, In esecuzione, Fermo perché o è fallito, cancellato o terminato).



Multithreading Pattern

Fire & Forget .

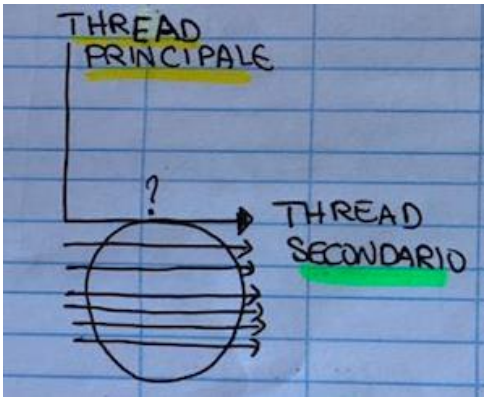
Il thread principale parte e ad un certo punto manda in esecuzione un thread secondario che lavora, ma al thread principale non interessa ciò che accade (se ne dimentica).



Polling:

Il thread principale fa partire un thread secondario e poi chiede a che punto è arrivato.

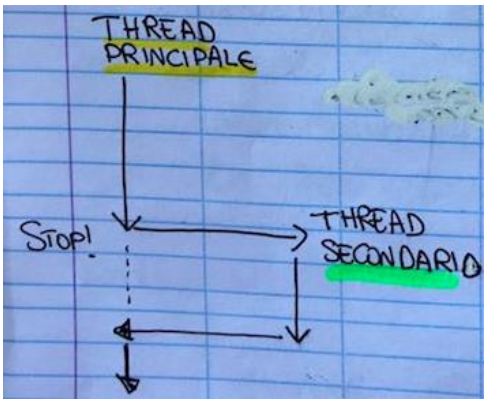
Il problema di questo pattern è che il thread principale perde tempo nell'interrogare il thread secondario.



Wait for Completion.

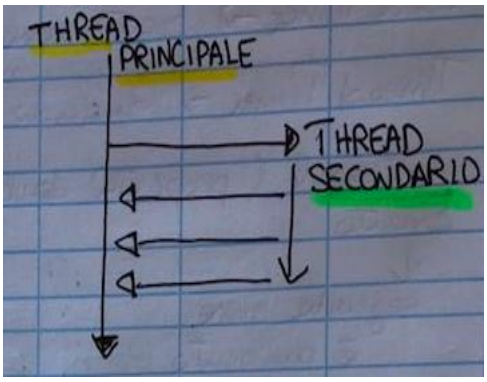
Il thread principale parte, ad un certo punto lancia un thread secondario; ma il thread principale resta fermo in attesa che il thread secondario ha terminato il sui completamento.

Ad esempio finché il video non è scaricato non posso vederlo.



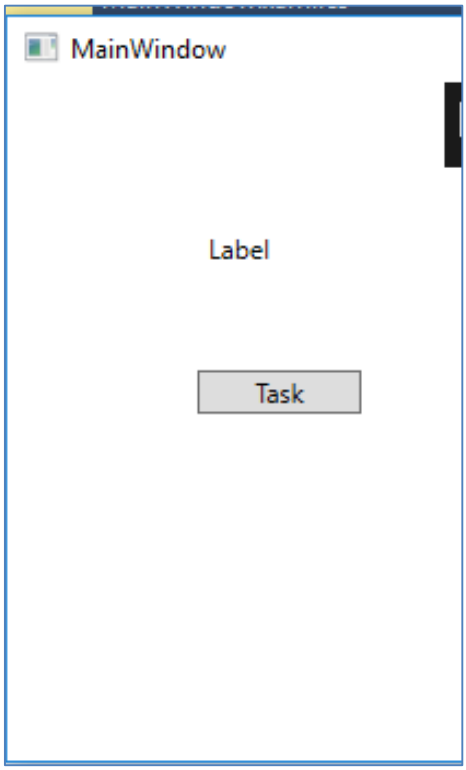
Completion Notification.

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.



Prime applicazioni multithreading

Consideriamo la seguente interfaccia grafica con una label ed un botton:



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<100;i++)
            { for(int j=0;j<1000;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.

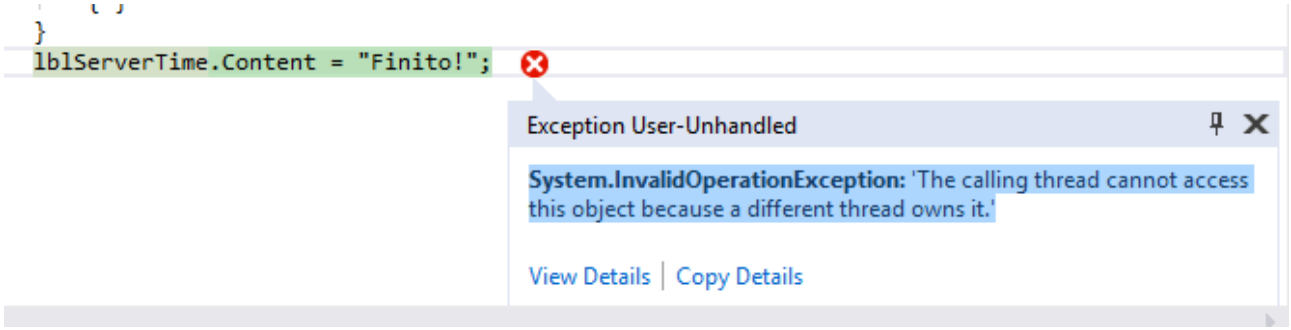
```
 private void btnTask_Click(object sender, RoutedEventArgs e)
        {
            LavoroLungoEComplicato();
        }
```

```
public void LavoroLungoEComplicato()
        {
            for (int i = 0; i < 100; i++)
            {
                for (int j = 0; j < 1000; 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. 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 < 1000000; i++)
            {
                for (int j = 0; j < 1000; j++)
                { }
            }
            Dispatcher.Invoke(AggiornaInterfacciaUtente);
        }
```

L'aggiornamento dell'interfaccia va separata 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.

Timers

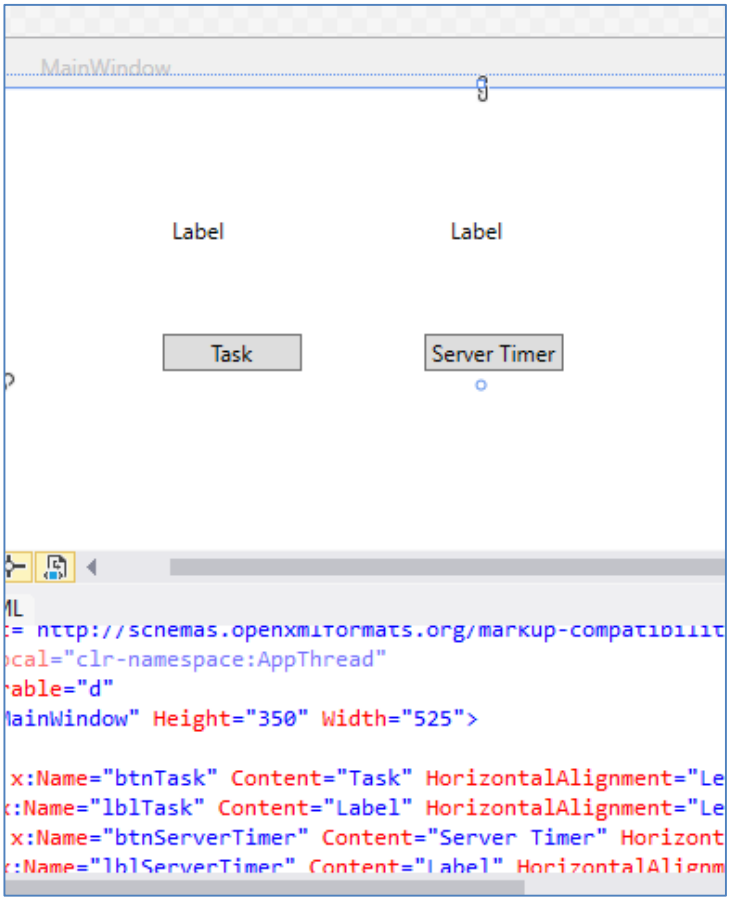
Vediamo un altro strumento utile per programmazione multithreading.
.Net fornisce due tipologie di Timers: Server Timer e Thread Timer.
I timers sono operazioni ricorrenti, schedulate. Possono essere attivati una volta sola oppure possono essere periodici. Fino agli anni 90 tutti i giochi funzionavano 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 btnServerTimer_Click(object sender, RoutedEventArgs e)
        {
            System.Timers.Timer serverTimer = new System.Timers.Timer(5000);
            serverTimer.Elapsed += serverTimer_Elapsed;
            serverTimer.Start();
        }
```

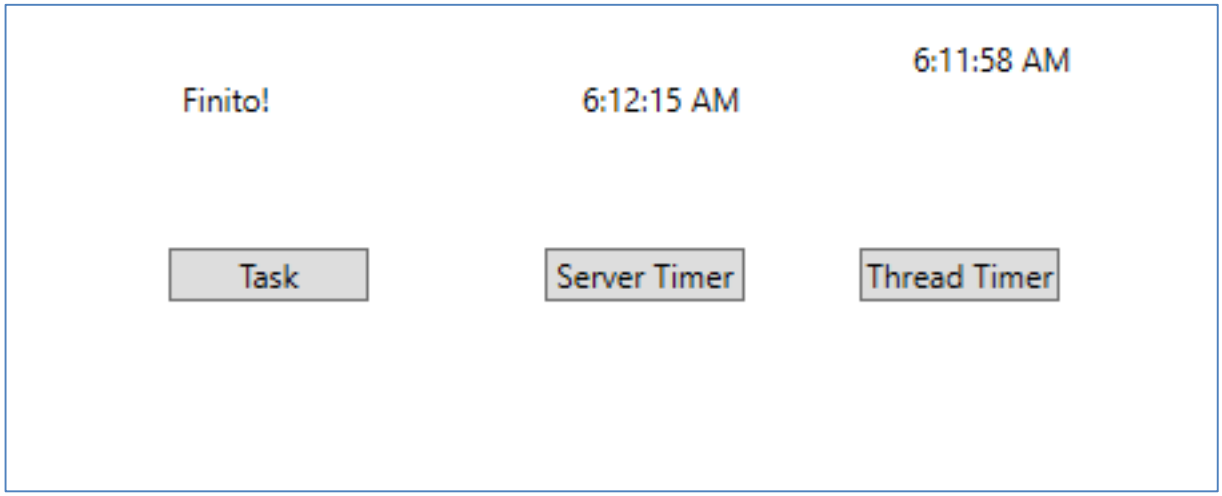
Il Timer scatta ogni 5 secondi e solleva un evento il quale sa cosa eseguire.

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

```
private void AggiornaInterfacciaServerTimer()
        {
            lblServerTimer.Content = DateTime.Now.ToLongTimeString();
        }
```

Thread Timer

Thread Timer è un oggetto che utilizza dei metodi di callback. 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 btnThreadTimer_Click(object sender, RoutedEventArgs e)
        {
            System.Threading.Timer threadTimer;
     threadTimer=new System.Threading.Timer(LavorodaFareNelThread, null, 0, 1000);
        }
```

Analizziamo la seguente istruzione:

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

- LavorodaFareNelThread 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 LavorodaFareNelThread(object state)
        {
            Dispatcher.Invoke(AggiornaInterfacciaThreadTimer);
        }
```

```
private void AggiornaInterfacciaThreadTimer()
        {
            lblThreadTimer.Content= DateTime.Now.ToLongTimeString();
        }
```

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.

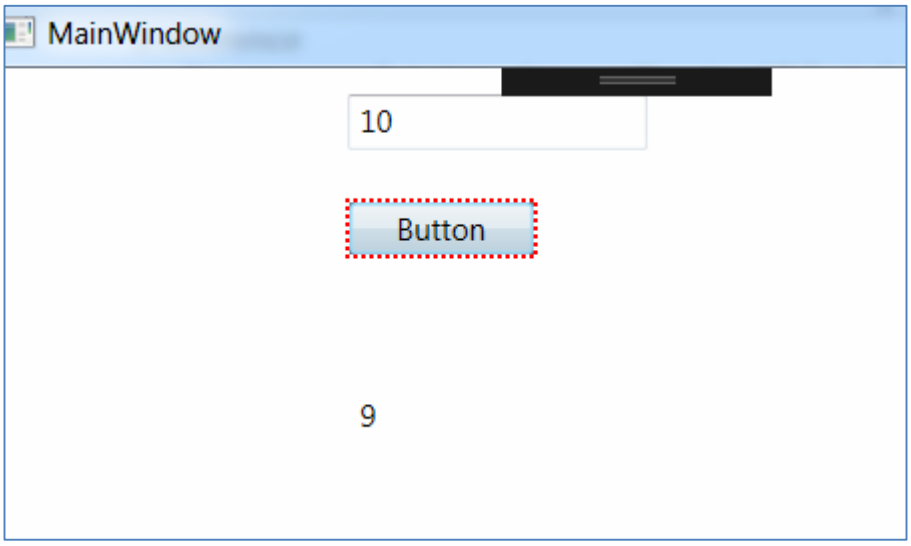
Task con parametro

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.



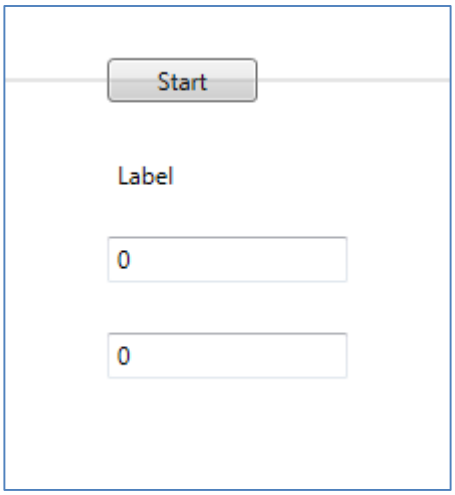
Analizziamo il seguente codice:

```
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 si ha 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)).`

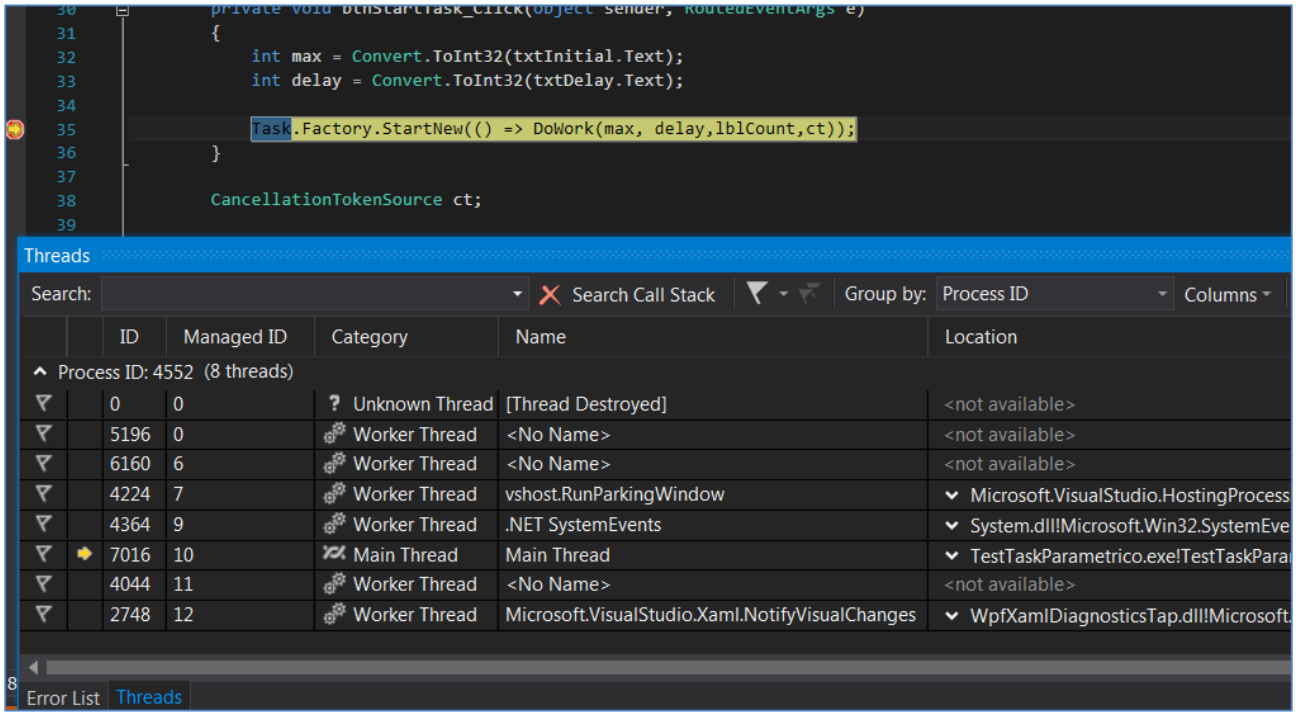
Ad un Task posso passare più dati ed il canale di comunicazione.

Vediamone un esempio.

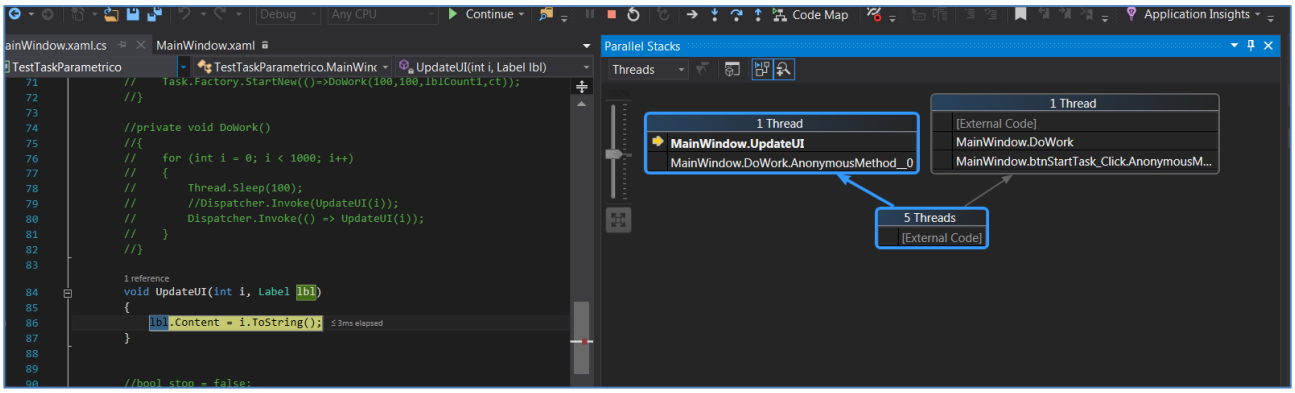


```
        private void btnStartTask_Click(object sender, RoutedEventArgs e)
        {
            int max = Convert.ToInt32(txtInitial.Text);
            int delay = Convert.ToInt32(txtDelay.Text);

            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.

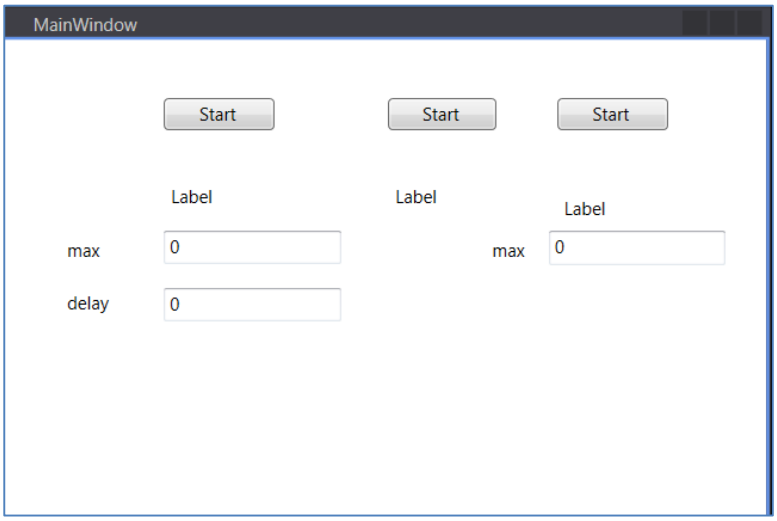


Mettendo un breakpoint e cliccando su Debug->Windows->Parallel Stacks è possibile visualizzare graficamente le relazioni tra i threads in esecuzione.



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

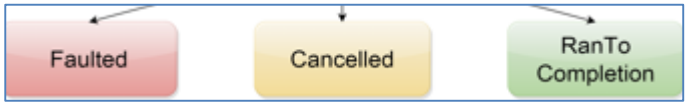
```
        private void btnStartTask_Click(object sender, RoutedEventArgs e)
        {
            int max = Convert.ToInt32(txtInitial.Text);

            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.

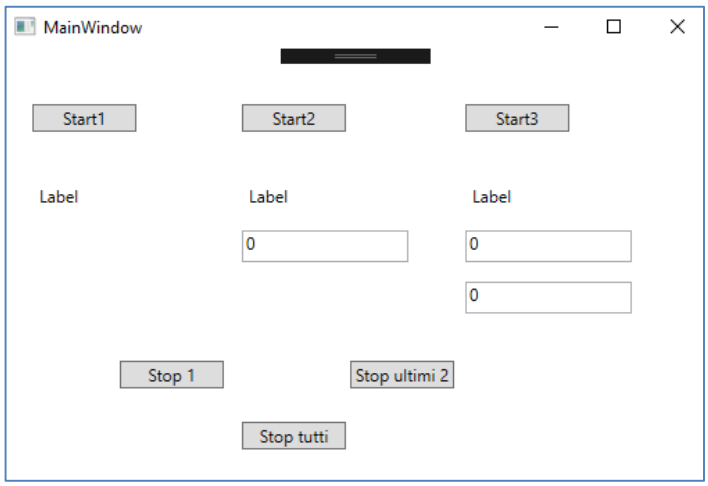
Ecco il codice completo ed approfittiamone per effettuare altre osservazioni.
L'istruzione `ct = new CancellationTokenSource();` permette di rigenerare il token ogni volta in cui si clicca il pulsante start.

```
    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;
            }
        }
```

Creazione di WorkerTask con Event-Based-Pattern

Creiamo una libreria che conterrà tutto ciò che serve per separare il codice relativo all’interfaccia grafica da quello relativo all’elaborazione.
Creiamo una classe Worker, il worker sarà un oggetto che svolge il lavoro.
Spostiamo tutto il codice di elaborazione nella classe.

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

```
        private void DoWork()
        {
            for (int i = 0; i < 100; i++)
            {

                Thread.Sleep(100);
            }

        }
```

Creiamo un metodo che crea il task.

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

Non passando parametri si potrebbe anche non utilizzare la funzione lambda.

Torniamo al codice relativo all'interfaccia.
Nel codice dell'interfaccia aggiungiamo il riferimento alla libreria.

```
using TaskLibrary;
```

Per far partire il task è necessrio creare il worker attraverso l'istruzione `Worker work = new Worker()` ed avviarlo attraverso il comando `work.Start()`.

```
        private void btnStart_Click(object sender, RoutedEventArgs e)
        {
            lblCount.Content = "Via!";
            Worker work = new Worker();
            work.Start();
        }
```

Se si avvia il progetto non riusciamo a vedere cosa sta avvenendo nella classe (Fire & Forget); la classe non notifica nulla all'interfaccia.
Pe risolvere il problema è possibile implementare un pattern Completion Notification: il Worker parte e quando ha terminato comunica alla UI che ha terminato. Per notificare da un task si utilizzano gli eventi.
Il pattern Completion Notification viene chiamato anche Event Based pattern (EBD).

Vediamo come è necessario modificare la classe Worker.

Si dichiara un "gestore di eventi" e si crea un evento chiamato OnCompleted.

```
public EventHandler OnCompleted;
```

Creiamo un metodo che ci permette di notificare il completamento.

```
      private void NotifyCompleted()
        {
                //se qualcuno è in ascolto
                if (OnCompleted != null)
                    //sollevo l'evento'
                    OnCompleted(this, new EventArgs());
            }
```

OnCompleted ha due parametri, nel primo si indica chi notifica la terminazione cioè il task, indicato con this ("me stesso"), nel secondo si indica il motivo della terminazione.
Nell'eventualità in cui volessi sollevare un evento il secondo parametro `OnCompleted(this, new EventArgs());` se invece si vuole solo notificare `OnCompleted(this, null)`.
Il metodo va chiamato dal DoWork().

```
        private void DoWork()
        {
            for (int i = 0; i < 100; i++)
            {

                Thread.Sleep(100);
            }
            NotifyCompleted();
        }
```

Torniamo all'interfaccia. Per ricevere la notifica si chiama il metodo `OnCompleted_Handler`

```
        private void btnStart_Click(object sender, RoutedEventArgs e)
        {
            lblCount.Content = "Via!";
            Worker work = new Worker();

            work.OnCompleted += OnCompleted_Handler;
        //se non mi sottoscrivo a nessun evento si tratta di un fire&forget

            work.Start();
        }
```

Per aggiornare l'interfaccia è necessario utilizzare `Dispatcher.Invoke();`

```
       private void OnCompleted_Handler(object sender, EventArgs e)
        {
             Dispatcher.Invoke(UpdateUIFatto);
        }
```

```
        void UpdateUIFatto()
        {
            lbl.Content = i.ToString();
        }
```

Il pattern Completion Notification si può articolare un una seconda tipologia di pattern, il Progress Notification. In questo caso, mentre il worker lavora , ho alcuni aggiornamenti.
La differenza tra Completion e Progress consiste che nel fatto che quest'ultimo, quando comunica se è successo qualcosa, deve anche comunicare il valore relativo al dato su quale si è sollevato l'evento.
Nella classe Worker è necessario utilizzare un nuovo evento di notifica che trasporta dati.

```
public EventHandler<ProgressEventArgs> OnProgress;
```

Le parentesi angolari indicano il tipo di dato che si vuole comunicare.
Creiamo un DataTransferObject, un tipo di oggetto che serve a trasportare dati legati ad un evento. Un DTO per convenzione è chiamato ProgressEventArgs perché rappresenta i dati dell'evento.
Osserviamo che ProgressEventArgs è di tipo EventArgs (relazione di ereditarietà).

```
namespace TaskLibrary
{
  public class ProgressEventArgs:EventArgs
    {
        public int Value { get; set; }
    }
}
```

Nella classe Worker è necessario un metodo che notifica il progresso.

```
        private void DoWork()
        {
            for (int i = 0; i < 100; i++)
            {
            NotifyProgress();
```

```
                Thread.Sleep(100);
            }
            NotifyCompleted();
        }
```

Nel metodo NotifyProgress() è necessario creare il DTO
`ProgressEventArgs args = new ProgressEventArgs()`, chiamato “carico pagante-payload” e poi notificarlo `OnProgress(this, args);`

```
private void NotifyProgress(int i)
        {

            if (OnProgress != null)
            {
                ProgressEventArgs args = new ProgressEventArgs();
                args.Value = i;

                OnProgress(this, args);
            }
        }
```

Nell’interfaccia modifico il codice.

```
        private void btnStart_Click(object sender, RoutedEventArgs e)
        {
            lblCount.Content = "Via!";
            Worker work = new Worker();

            work.OnCompleted += OnCompleted_Handler;
            work.OnProgress += OnProgress_Handler;
            work.Start();
        }
```

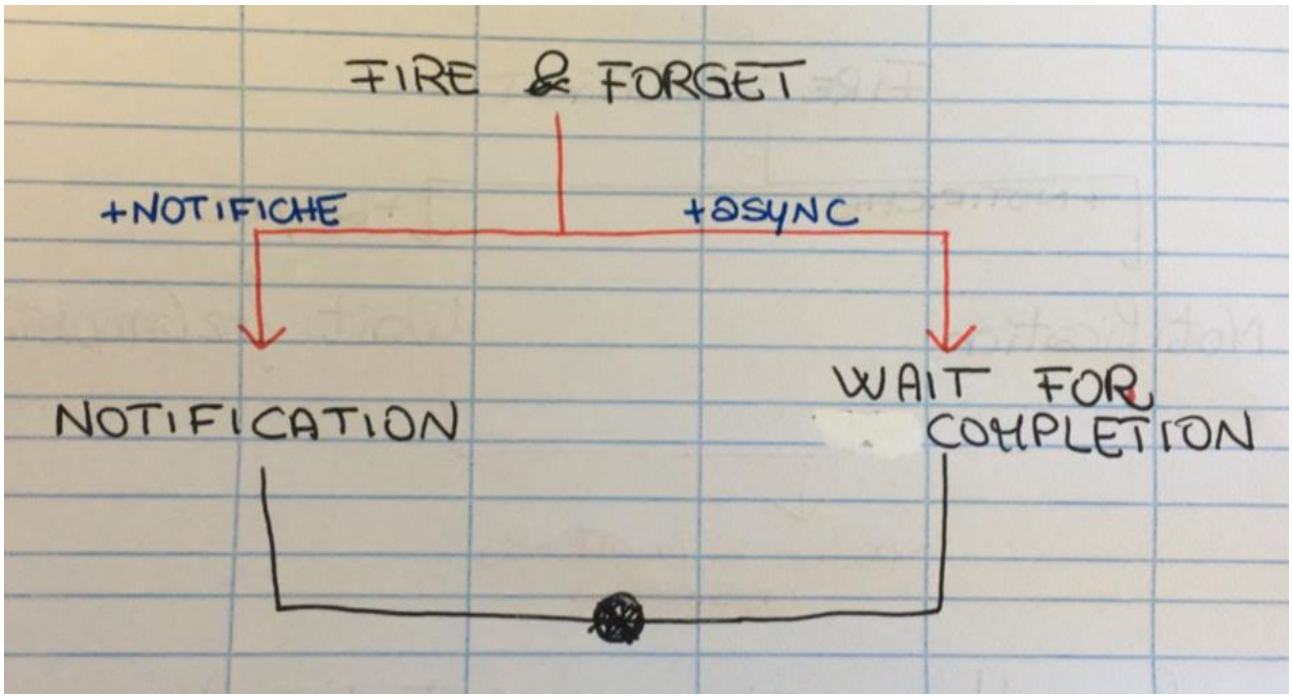
```
private void OnProgress_Handler(object sender, TaskDataEventArgs e)
        {
            Dispatcher.Invoke(() => UpdateUI(e.Value, lblCount));
        }
```

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

Esercizio: implementare il sistema di notifiche a tutti i pulsanti presenti nella seguente interfaccia grafica.

Modelli di programmazione asincrona.

Riprendiamo i pattern multithreading. Aggiungendo al pattern multithread Fire e& Forget (con un DoWork, una crreazione del Task e uno Start) altri "ingredienti si ottiene Completion Notification (con notifiche), o Wait for Completion (con async ed await e senza notifiche.) E' possibile combinare anche Completion Notification e Wait for completion.



Ci sono dei casi in cui è necessario attendere il completamento del task secondario (Pattern Wait for Completion).

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), visto nl 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.

Riprendiamo l'esempio visto precedentemente.

Osserviamo che per avere solo un Fire & Forget senza avere notifiche è sufficiente eliminare le due righe di btnStart() e tutti i metodi relativi alle notifiche.

```
            work.OnCompleted += OnCompleted_Handler;
            work.OnProgress += OnProgress_Handler;
```

Riprendiamo quindi il codice nella classe Worker.

```
    public class Worker
    {
        //public EventHandler<TaskDataEventArgs> OnValueChanged;
        //public EventHandler OnTaskCompleted;

        public int Max { get; private set; }
        public Worker(int max)
        {
            Max = max;
        }

        public void Start()
        {
            Task.Factory.StartNew(DoWork);
        }

        private void DoWork()
        {
            for (int i = 0; i < Max; i++)
            {
                //NotifyProgress(i);
                Thread.Sleep(100);
            }
            //NotifyCompletion();
        }
        //private void NotifyCompletion()
        //{
        //    if (OnTaskCompleted != null)
        //    {
        //        if (OnTaskCompleted != null)
        //            OnTaskCompleted(this, new EventArgs());
        //    }
        //}
        //private void NotifyProgress(int i)
        //{
        //    if (OnValueChanged != null)
        //    {
```

```
        //        TaskDataEventArgs args = new TaskDataEventArgs();
        //        args.Value = i;
        //        if (OnValueChanged != null)
        //            OnValueChanged(this, args);
        //    }
        //}
```

Per implementare il Wait for Completion e pertanto aspettare e svolgere in lavoro solo al termine del completamento del thread secondario, in questo esempio visualizzare la Message Box, è necessario effettuare alcune modifiche al codice.
Il seguente codice, che implementa un Fire &Forget, attualmente, visualizza solo la MessageBox appena si clicca il pulsante Start senza aspettare il termine del lavoro del Thread secondario.

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

        void UpdateUI(int i, Label lbl)
        {
            lbl.Content = i.ToString();
        }
        private void btnStart_Click(object sender, RoutedEventArgs e)
        {
            Worker work = new Worker(100);
            //work.OnTaskCompleted += OnTaskCompleted_Handler;
            //work.OnValueChanged += OnValueChanged_Handler;
            work.Start();
            MessageBox.Show("fatto");
        }

        //private void OnTaskCompleted_Handler(object sender, EventArgs e)
        //{
        //    Dispatcher.Invoke(() =>lblCount.Content = "Fatto");
        //}

        //private void OnValueChanged_Handler(object sender, TaskDataEventArgs e)
        //{
        //    Dispatcher.Invoke(() => UpdateUI(e.Value, lblCount));
        //}
    }
```

Per ottenere in Wait for Completion è necessario effettuare delle modifiche al Fire e Forget. Attraverso tale modifiche il task principale aspetta il termine del lavoro del task secondario per far visualizzare la Message Box.
Tali modifiche sono necessarie a partire dalla classe Worker.
Per convenzione si cambia il nome al metodo `StartAsync()`

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

Inoltre si marca il metodo con la parola chiave async:

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

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(progress));
        }
```

Le stesse parole chiave si utilizzano anche nel Main:

```
    private async void btnStart_Click(object sender, RoutedEventArgs e)
        {
            Worker work = new Worker(100);
            await work.StartAsync();
            MessageBox.Show("fatto");
        }
```

In questo modo si attende che finisca il task secondario per poi visualizzare la Message Box e l'interfaccia continua ad essere reattiva.

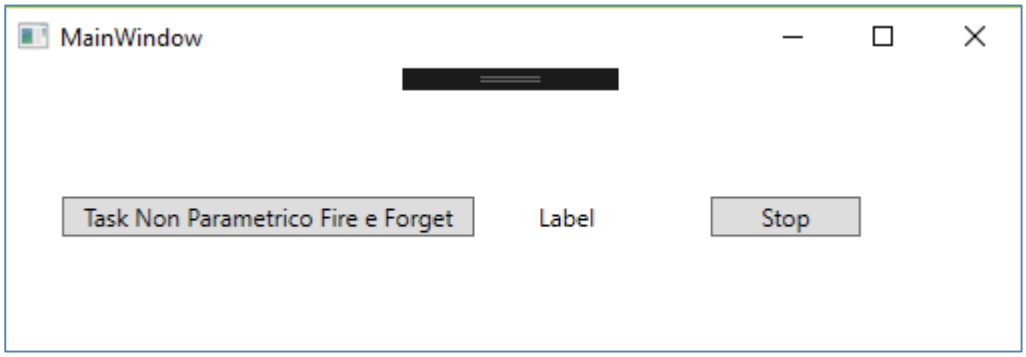
Esercitazione riassuntiva

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

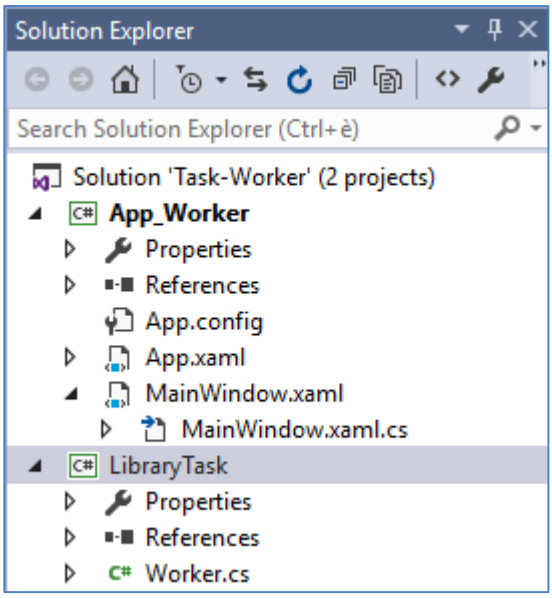
Fire & Forget

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.



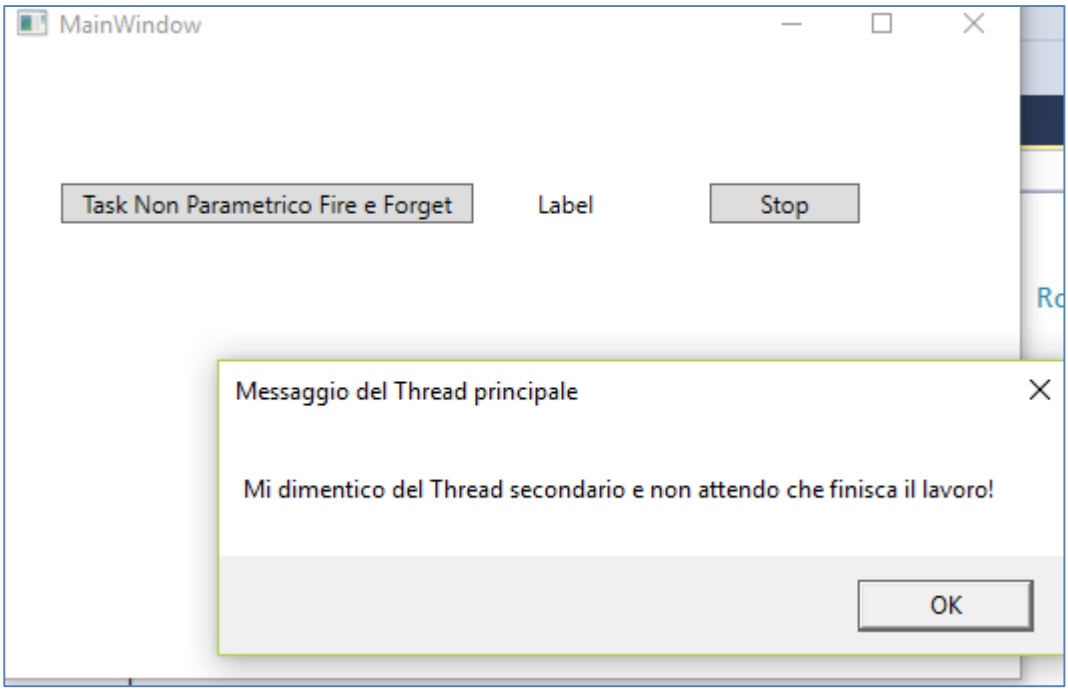
Spostiamo tutto il codice di elaborazione nella classe Worker.

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
       public CancellationTokenSource Cts { get; set; }
        public int Max { get; private set; }
        public int Limit { get; private set; }

        //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
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;
        private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //creo token per fermarlo
            cts = new CancellationTokenSource();
            //creo worker
            Worker wrk = new Worker(5, 1000, cts);
            wrk.Start();
            //Osserviamo che in Fire e Forget appena clicco start crede di aver fatto anche
se non è vero!
            MessageBox.Show("Mi dimentico del Thread secondario e non attendo che finisca il
lavoro!", "Messaggio dal Thread principale");
        }

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



Completion Notification

Rivediamo l'esempio aggiungendo la notifica della conclusione del lavoro del thread secondario.
Nel codice sono sottolineate le modifiche da fare a partire dal Fire & Forget.

```
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 considerrae nel conteggio
       public CancellationTokenSource Cts { get; set; }
        public int Max { get; private set; }
        public int Limit { get; private set; }

        public EventHandler OnCompleted;

        //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;
            }
            NotifyCompleted();
        }
```

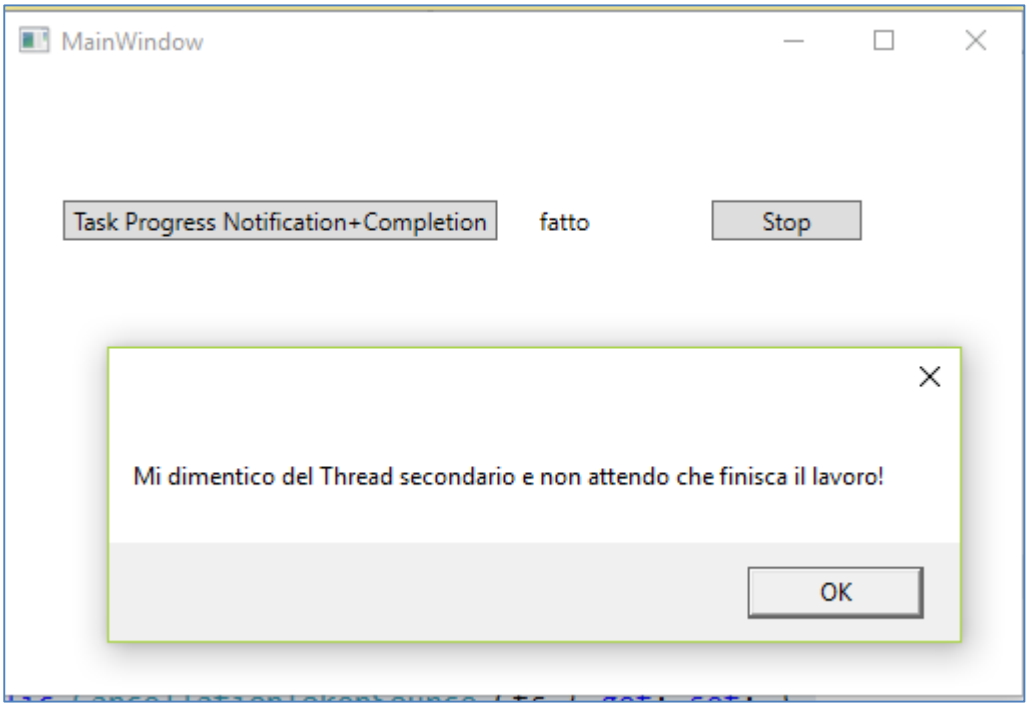
```
        private void NotifyCompleted()
        {
            //se qualcuno è in ascolto
            if (OnCompleted != null)
                //sollevo l'evento'
                OnCompleted(this, new EventArgs());
        }

    }
```

Vediamo anche le modifiche al Main:

```
namespace App_Worker
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;
        private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //creo token per fermarlo
            cts = new CancellationTokenSource();
            //creo worker
            Worker wrk = new Worker(5, 1000, cts);
            wrk.Start();

            wrk.OnCompleted += OnCompleted_Handler;
            //Osserviamo che siamo ancora in  Fire e Forget !
            MessageBox.Show("Mi dimentico ancora del Thread secondario, ma al termine del
suo lavoro ho una notifica!", "Messaggio del Thread principale");
        }

        private void OnCompleted_Handler(object sender, EventArgs e)
        {
            Dispatcher.Invoke(UpdateUI);
        }

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

        }

        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.



Creiamo una classe DTP per notificare il dato durante l'esecuzione del thread secondario.

```
    public class ProgressEventArgs : EventArgs
    {
        public int Value { get; set; }
    }
```

Vediamo le modifiche da fare alla classe worker.

```
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 considerrae nel conteggio
       public CancellationTokenSource Cts { get; set; }
        public int Max { get; private set; }
        public int Limit { get; private set; }

        public EventHandler OnCompleted;
        public EventHandler<ProgressEventArgs> OnProgress;

        //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++)
            {
                NotifyProgress(i);
                Thread.Sleep(delay);
                if (Cts.IsCancellationRequested)
                    break;
            }
            NotifyCompleted();
        }

        private void NotifyProgress(int i)
        {
            if (OnProgress != null)
            {
                ProgressEventArgs args = new ProgressEventArgs();
                args.Value = i;

                OnProgress(this, args);
            }

        }

        private void NotifyCompleted()
        {
            //se qualcuno è in ascolto
            if (OnCompleted != null)
                //sollevo l'evento'
                OnCompleted(this, new EventArgs());
        }

    }
}
```

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

```
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;
        private void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //creo token per fermarlo
            cts = new CancellationTokenSource();
            //creo worker
            Worker wrk = new Worker(5, 1000, cts);
            wrk.Start();

            wrk.OnCompleted += OnCompleted_Handler;

            wrk.OnProgress += OnProgress_Handler;
            //Osserviamo che siamo ancora in  Fire e Forget !
```

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

        private void OnProgress_Handler(object sender, ProgressEventArgs e)
        {
            Dispatcher.Invoke(() => UpdateUI(e.Value, lblCount));
        }

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

        private void OnCompleted_Handler(object sender, EventArgs e)
        {
            Dispatcher.Invoke(UpdateUI);
        }

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

        }

        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.

```
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 considerrae nel conteggio
       public CancellationTokenSource Cts { get; set; }
        public int Max { get; private set; }
        public int Limit { get; private set; }

        //costruttore
        public Worker(int max, int limit, CancellationTokenSource cts)
        {
            Limit = limit;
            Max = max;
            Cts = cts;
        }
        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)
        {
            for (int i = 0; i < max; i++)
            {
                Thread.Sleep(delay);
                if (Cts.IsCancellationRequested)
                    break;
            }
        }
    }
```

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

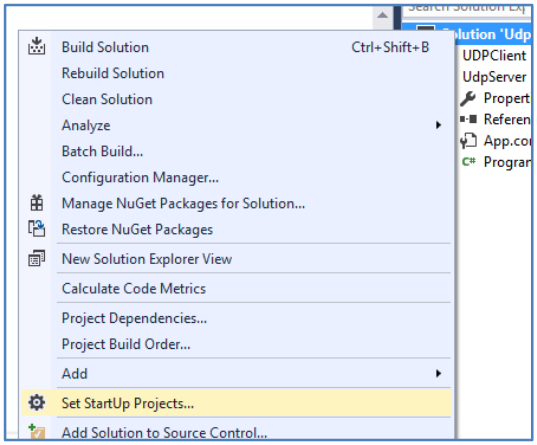
```
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        CancellationTokenSource cts;
        private async void btnStartTaskNoPar_Click(object sender, RoutedEventArgs e)
        {
            //creo token per fermarlo
            cts = new CancellationTokenSource();
            //creo worker
            Worker wrk = new Worker(5, 1000, cts);
           await wrk.StartAsync();
            //Osserviamo che in questo caso il Thread principale ha atteso la conclusione
del secondario!
            MessageBox.Show("Ho aspettato il Thread secondario e ora riprendo il mio
lavoro!","Messaggio dal Thread pincipale");
        }
```

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

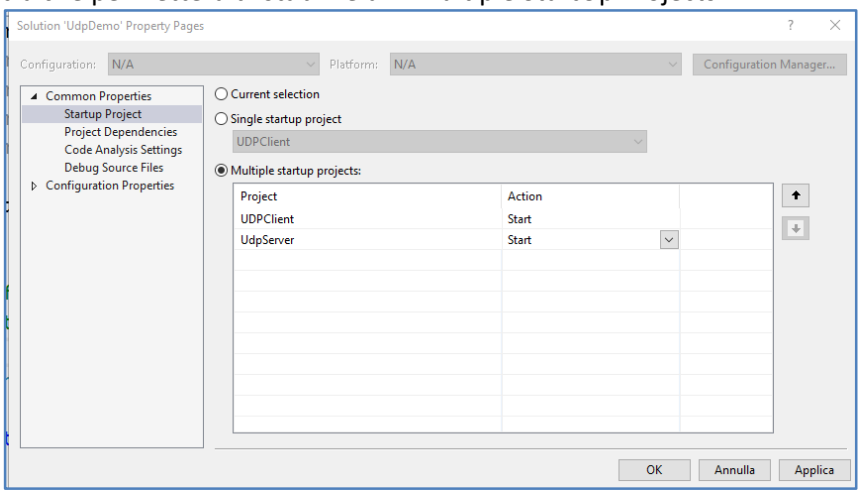
Chat Multithreading

UDP in .NET

Per creare una Chat tramite UDP abbiamo bisogno di aprire due progetti: uno per il Client ed una per il server.
I due progetti devono essere avviati contemporaneamente. Per far ciò basta cliccare con il tasto desttro sulla solution e selezionare Set StartUp Projects.



Si aprirà una finestra che permetterà di stabilire un “Multiple StartUp Projects”.



A questo punto è possibile procedere con la scrittura del codice.
Pr prima cosa è necessario utilizzare i seguenti namespace:

```
using System.Net;
using System.Net.Sockets;
```

Affinchè due processi possano comunicare è necessario creare un EndPoint, anzi due, uno per il client

```
IPEndPoint epLocal = new IPEndPoint(IPAddress.Any, 4000);
```

ed uno per il server

```
IPEndPoint epLocal = new IPEndPoint(IPAddress.Any, 5000);.
```

Nel costruttore si specificano indirizzi ip e numeri di porta.
Un EndPoint è rappresentato da un indirizzo ip che identifica un host della rete e da una porta che identifica un processo di un’applicazione.

Il protocollo di trasporto UDP è implementato dalla classe UDP Client.
E' quindi necessario creare un oggetto UDPClient, specificando nel nel costruttore l'EndPoint

```
UdpClient cli = new UdpClient(epLocal);
```

Analogamente per il server

```
UdpClient serv = new UdpClient(epLocal);
```

Il client successivamente si connette all'indirizzo IP del server (nel nostro esempio client e server risideono sulla setta machina, pertanto l'indirizzo è localhost).

```
cli.Connect("127.0.0.1",5000);
Il messaggio viene trasformato un vettore di byte e codificato:
byte[] dati = Encoding.ASCII.GetBytes(messaggio);
```

Successivamente viene inviato insieme alla sua lunghezza.

```
cli.Send(dati, dati.Length);.
```

Per permettere più invii è sufficiente utilizzare un ciclo while. Ecco il codice completo.

```
        static void Main(string[] args)
        {
            Console.WriteLine("Client");
            //creare un oggetto UDPClient
            IPEndPoint epLocal = new IPEndPoint(IPAddress.Any, 4000);
            UdpClient cli = new UdpClient(epLocal);

            //connection-less: non è garantita la connessione
            // il client si vuole collegare al server

            cli.Connect("127.0.0.1",5000);

            //inserisco un ciclo while per inviare più messaggi

            while (true)
            {
                string messaggio = Console.ReadLine();
                //non posso inviare stringhe, devo codificarla
                //creo un vettore di byte
                byte[] dati = Encoding.ASCII.GetBytes(messaggio);

                //una volta codificati posso inviarli col metodo send
                //siccome invia un vettore,
                //devo passargli come parametro i dati e la langhezza del vettore

                cli.Send(dati, dati.Length);
                //se mando in esecuzione, il client non si accorge se è il server in ascolto

            }

            Console.ReadLine();
        } }
```

Passiamo al server che riceverà il vettore di byte e lo decodificherà.

```
byte[] dati = serv.Receive(ref ipe);
string s = Encoding.ASCII.GetString(dati);
```

Per ricevere è sufficiente inserire un ciclo while.
Ecco il codice completo:

```
    class Program
    {
        static void Main(string[] args)
        {

            Console.WriteLine("Server");
            IPEndPoint epLocal = new IPEndPoint(IPAddress.Any, 5000);
```

```
            UdpClient serv = new UdpClient(epLocal);
            //ipEndPoint ipe rappresenta il mittente
            IPEndPoint ipe= null;
            while(true)
            {
            byte[] dati = serv.Receive(ref ipe);
                //il Receive è un metodo bloccante
                //posso inviare solo dopo aver ricevuto
                //analogamente nel client ricevo dopo aver mandato
            string s = Encoding.ASCII.GetString(dati);
            Console.WriteLine(s);
            }

            Console.ReadLine();
        }
    }
```

TCP in .NET

Come per UDP creiamo una soluzione vuota alla quale aggiungiamo due progetto TCPClient e TCPServer (Ricordarsi di effettuare la procedura “Set StartUp Projects.” per permettere ad entrambi i progetti di avviarsi contemporaneamente).

Procediamo con la scrittura del codice per il Client

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Client partito...premere un tasto per collegarsi");
            Console.ReadKey();
            //E' necessario un oggetto TcpClient
            //Devo connettermi al server
            // Mi serve il flusso di dati per comunicare
            TcpClient cli = new TcpClient();
            try
            {

                cli.Connect("127.0.0.1", 8080);
                Console.WriteLine("Collegato al Server, scrivi un messaggio");
                NetworkStream ns = cli.GetStream();
                StreamWriter sw = new StreamWriter(ns);
                sw.AutoFlush = true;//lo invia subito senza attendere che il buffer si pieni

                string msg=Console.ReadLine();
                //Nel caso di UDP effettuavo la codifica
                //byte[] msgDaMandare = Encoding.ASCII.GetBytes(msg);
                //Vettore di byte, parto da 0, invio la lunghezza
                //ns.Write(msgDaMandare, 0, msgDaMandare.Length);
                sw.WriteLine(msg);
            }
            catch (Exception ex)
            { Console.WriteLine("Sever non partito"); }

        }
    }
```

E procediamo con la scrittura del codice per il server

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net;
using System.Net.Sockets;
using System.IO;

namespace TcpServer
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Server Partito");
            // dichiaro un oggetto di tipo TcpListner che sta in ascolto sulla porta
            //un computer può avere più indirizzi IP ecco perchè ci sta IPAddress.Any e
quindi si mette in ascoltos u tutti gli i dirizzi IP
            TcpListener listner = new TcpListener(IPAddress.Any, 8080);
            listner.Start(); //parte l'ascolto delle connessioni, il ciclo è infinito perché
il server deve ascoltare sempre

            while (true)
            {

                Console.WriteLine("Server in attesa...");

                //il metodo che segue è bloccante
                //blocca in attesa del client

                TcpClient tcpClient = listner.AcceptTcpClient();
                //avvia comunicazione con il client
                //e crea un oggetto TCPClient
                Console.WriteLine("Connessione effettuata al client");
                NetworkStream ns = tcpClient.GetStream();
                StreamReader sr = new StreamReader(ns);
                Console.WriteLine("ricevuto");
                //inizia a leggere e poi si mette ad ascoltare un'altra
                string msg = sr.ReadLine();
                //quindi almeno nel server è necessario creare thread separati
                Console.WriteLine(msg);

            }
            //listner.Stop(); // termina l'ascolto
        }
    }
}
```

Il sistema di comunicazione deve essere bloccante.
Un oggetto TcpClient è in esecuzione sul Client. Si connette ad un Server.
Un oggetto TcpListner è in esecuzione solo sul Server. Sta in ascolto su una determinata port, **è in attesa** (l'azione è bloccante), rileva un tentativo di connessione, crea un oggetto TcpClient (in esecuzione sul server) utilizzato per la comunicazione (si tratta di un thread secondario). Così il server può rimettersi in ascolto per un'altra connessione.
Una volta stabilita la connessione, un oggetto di tipo TcpClient è in esecuzione sia sul Client che sul Server. La comunicazione si identifica mediante un flusso di byte Stream. La stringa va convertita, codificata in Unicode e poi inviata. E' necessario un meccanismo per separare i messaggi tra di loro. Il meccanismo più semplice è \r\n.

Si utilizza quindi un StreamReader ed uno StreamWriter perché gestiscono in automatico \n e \r

Questa parte di codice andrebbe gestita in thread separati perché se un secondo client si volesse connettere al server, quest'ultimo non può ascoltare essendo occupato con la comunicazione con il primo client. La conversazione del server va gestita con un Thread secondario.

```
                NetworkStream ns = tcpClient.GetStream();
                 StreamReader sr = new StreamReader(ns);
                 Console.WriteLine("ricevuto");
                 //inizia a leggere e poi si mette ad ascoltare un'altra
                 string msg = sr.ReadLine();
                 //quindi almeno nel server è necessario creare thread separati
```

Iniziamo con l'incapsulare il codice in un metodo DoWork().

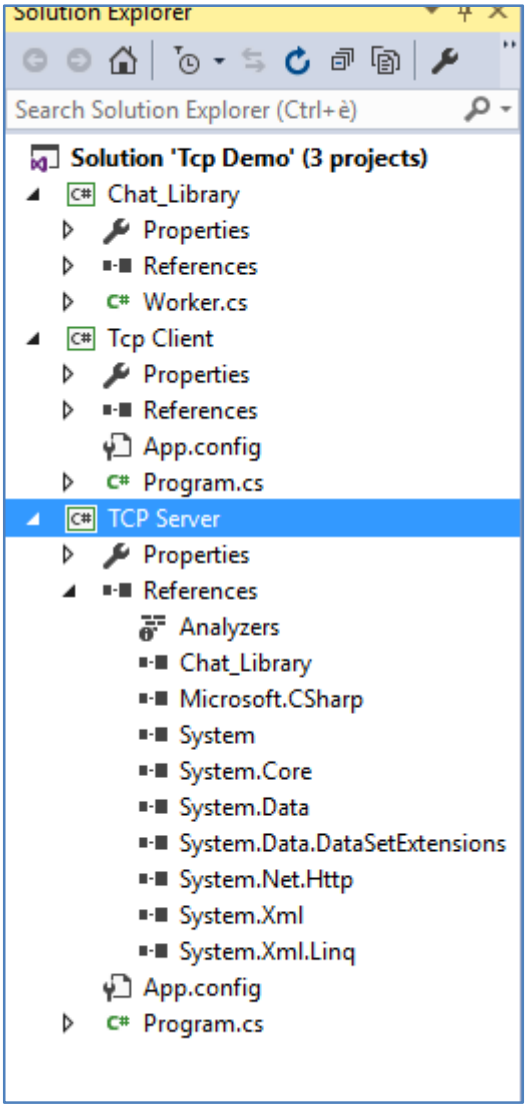
```
namespace TcpServer
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Server Partito");
            // dichiaro un oggetto di tipo TcpListner che sta in ascolto sulla porta
            //un computer può avere più indirizzi IP ecco perchè ci sta IPAddress.Any
            TcpListener listner = new TcpListener(IPAddress.Any, 8080);
            listner.Start(); //parte l'ascolto delle connessioni

            while (true)
            {

                Console.WriteLine("Server in attesa...");
                //il metodo che segue è bloccante
                //blocca in attesa del client

                TcpClient tcpClient = listner.AcceptTcpClient();
                //avvia comunicazione con il client
                //e crea un oggetto TCPClient
                Console.WriteLine("Connessione effettuata al client");
                DoWork(tcpClient);

            }
            listner.Stop(); // termina l'ascolto
        }
        static void DoWork(TcpClient tcpClient)
        {
            NetworkStream ns = tcpClient.GetStream();
            StreamReader sr = new StreamReader(ns);
            Console.WriteLine("ricevuto");
            //inizia a leggere e poi si mette ad ascoltare un'altra
            string msg = sr.ReadLine();
            //quindi almeno nel server è necessario creare thread separati
            Console.WriteLine(msg);
        }
    }
}
```

Per rendere la chat multithread è necassario implementare il pattern **Fire&Forget**.

Applicazione di Fire&Forget alla chat.

La Chat fin qui implementata non permette al server di accettare più connsessioni da più client. Questo perché il server se è impegnato nella comunicazione col primo client non può rimettersi in ascolto. Pertanto è necessario creare dei thread secpondari per gestire la comunicazione una volta accettata la connessione in modo da permettere al thread principale di continuare nell'ascolto.

Pertanto procediamo con la creazione di una libreria, che nel nostro esempio chiameremo Chat_Library. Ricordiamo che è necessario aggiungere al progetto TCpClient la reference alla nuova libreria e la using.



Creiamo una classe Worker con un metodo Start che permette di avviare il Task secondario ed il metodo DoWork che contiene tutte le istruzioni relative alla comunicazione.
Ovviamente la classe avrà un attributo di tipo TcpClient ed costruttore che ne permetterà l'inizializzazione.

Vediamo il codice nel dettaglio:

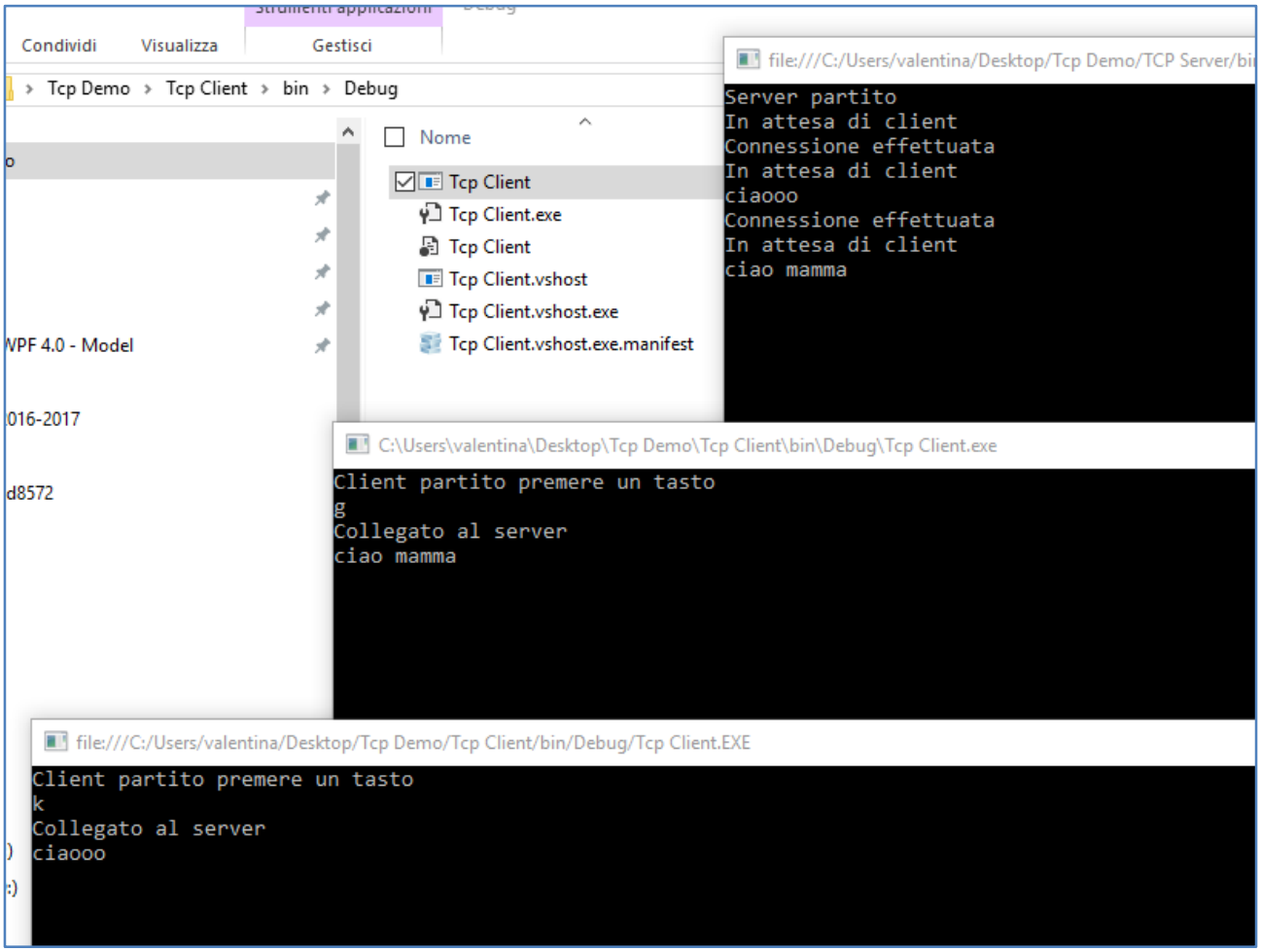
```
namespace Chat_Library
{


    public class Worker
  {
    TcpClient _tcpClient;

    public Worker(TcpClient tcpClient)
        { _tcpClient = tcpClient; }

    public void Start()
        { Task.Factory.StartNew((() => DoWork(_tcpClient))); }

    void DoWork(TcpClient tcpClient)
        {
            while (true)
            {
                NetworkStream ns = tcpClient.GetStream();
                StreamReader sr = new StreamReader(ns);

                string msg = sr.ReadLine();
                Console.WriteLine(msg);
            }
        }
    }
}
```

La parte di codice relativa al server si riduce, in quanto ora è sufficiente creare l'oggetto e invocare il metodo Start.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;
using Chat_Library;

namespace TCP_Server
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Server partito");
            TcpListener listener = new TcpListener(IPAddress.Any, 8080); // ascolta sulla
porta 8080
            listener.Start();
            while (true)
            {
                Console.WriteLine("In attesa di client");
                TcpClient tcpClient = listener.AcceptTcpClient(); // blocca in attesa del
client

                // ... avvia comunicazione con il client
                Console.WriteLine("Connessione effettuata");
```

```
                Worker wrk = new Worker(tcpClient);
                wrk.Start();
            }
            listener.Stop(); // ferma listener (con questa implementazione è inutile)
        }
    }
}
```

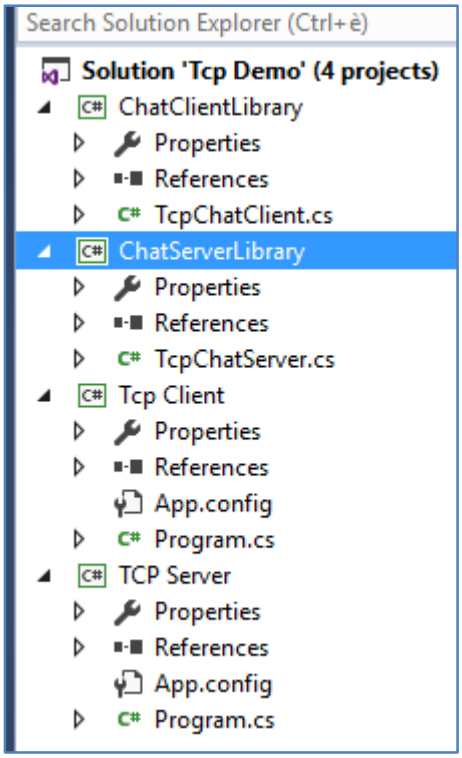
Con queste modifiche è possibile mandare in esecuzione più client ed il server riuscirà a gestire tutte le comunicazioni!



Inviare un messaggio a tutti i client della chat.

Vogliamo creare un meccanismo di "Dispatch" cioè un messaggio inviato al thread del server deve essere distribuito a tutti gli altri client.
Il TCPListner è a conoscenza di tutti i client che si sono connessi. Quindi il messaggio verrà passato al TCPListner per inviare tutti i messaggi ai client che fanno parte della lista. Invece di far comunicare i processi (TcpClient che comunica con TCPListner) tra di loro si utilizza una struttura dati condivisa.

Per migliorare il progetto, invece di utilizzare un'unica libreria, andiamo a creare due distinte librerie, una che conterrà tutta la logica implementativa del server ed una del client.



Quindi nella classe TcpClassServer andiamo ad inserire tutto il vecchio codice della classe Worker visto al paragrafo precedente.

Per poter inoltrare un messaggio di un client a tutti i client connessi al server, il server deve disporre di una lista di client che viene popolata d ogni connessione accettata.
Vediamo quindi le modifiche da fare alla classe TcpClassServer. In particolare va creata una lista nel costruttore, che poi andrà popolata attraverso il metodo AddClient. Lo stesso metodo per ora farà partire il Thread secondario responsabile della comunicazione effettiva tra client e server.

```
namespace ChatServerLibrary
{
    public class TcpChatServer
    {

        private TcpClient _tcpClient;
        private List<TcpClient> _clientsList;

        public TcpChatServer()
        {
            _clientsList = new List<TcpClient>();
        }

        public void AddClient(TcpClient tcpClient)
        {
            _clientsList.Add(tcpClient);
            _tcpClient = tcpClient;
            Task.Factory.StartNew(() => DoWork(_tcpClient));
        }
        public void RemoveClient(TcpClient tcpClient)
        {
            _clientsList.Remove(tcpClient);
        }
        //public void Start()
```

```
        //{
        //    Task.Factory.StartNew(() => DoWork(_tcpClient));
        //}
        void DoWork(TcpClient tcpClient)
        {
            try
            {
                while (true)
                {
                    NetworkStream ns = tcpClient.GetStream();
                    StreamReader sr = new StreamReader(ns);
                    string msg = sr.ReadLine();
                    Console.WriteLine(msg);
                }
            }
            catch (Exception)
            {
                Console.WriteLine("Connessione chiusa");
            }
        }
    }
}
```

Al momento della creazione dell'oggetto, nel Main, non verrà invocato il metodo Start, ma il metodo AddClient.

```
namespace TCP_Server
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Server partito");
            TcpListener listener = new TcpListener(IPAddress.Any, 8088); // ascolta sulla
porta 8080
            listener.Start();
            while (true)
            {
                Console.WriteLine("In attesa di client");
                TcpClient tcpClient = listener.AcceptTcpClient(); // blocca in attesa del
client

                // ... avvia comunicazione con il client
                Console.WriteLine("Connessione effettuata");
                TcpChatServer wrk = new TcpChatServer();
                wrk.AddClient(tcpClient);
            }
            listener.Stop(); // ferma listener (con questa implementazione è inutile)
        }
    }
}
```

Per reinviare il messaggio a tutti i client è necessario riprendere il codice che usa il Client per l'invio ed inserirlo nella classe che gestisce la logica di implementazione del server.

In particolare aggiungiamo un metodo SentToAll che verrà poi invocato nel DoWork. Il metodo scorre la lista di client per reinviare il messaggio a tutti.
Vediamo il codice

```
void DoWork(TcpClient tcpClient)
{
    try
    {
        while (true)
        {
            NetworkStream ns = tcpClient.GetStream();
            StreamReader sr = new StreamReader(ns);
            string msg = sr.ReadLine();
            Console.WriteLine(msg);
            SendToAll(msg);
        }
    }
    catch (Exception)
    {
        Console.WriteLine("Connessione chiusa");
    }
}

private void SendToAll(string msg)
{
    foreach (var tcpClient in _clientsList)
    {

        NetworkStream ns = tcpClient.GetStream();
        StreamWriter sw = new StreamWriter(ns);
        sw.AutoFlush = true;
        sw.WriteLine(msg);
    }
}
```

Si può migliorare il codice prevedendo anche un metodo SendToOne che potrà essere utilizzato per inviare un messaggio ad un singolo utente. Il metodo verrà utilizzato nel ciclo di SendToAll in modo da utilizzare l'invio singolo a tutti i client della lista.

```
void DoWork(TcpClient tcpClient)
{
    try
    {
        while (true)
        {
            NetworkStream ns = tcpClient.GetStream();
            StreamReader sr = new StreamReader(ns);
            string msg = sr.ReadLine();
            Console.WriteLine(msg);
            SendToAll(msg);
        }
    }
    catch (Exception)
    {
        Console.WriteLine("Connessione chiusa");
    }
}
```

```
        private void SendToAll(string msg)
        {
            foreach (var tcpClient in _clientsList)
            {
                SendToOne(msg, tcpClient);
            }
        }

        private static void SendToOne(string msg, TcpClient tcpClient)
        {
            NetworkStream ns = tcpClient.GetStream();
            StreamWriter sw = new StreamWriter(ns);
            sw.AutoFlush = true;
            sw.WriteLine(msg);
        }
    }
```

Torniamo al Main.
Il codice relativo all'accettazione della connessione da parte del server può essere incapsulato all'interno della classe TCpChatServer.

```
namespace TCP_Server
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Server partito");
            TcpListener listener = new TcpListener(IPAddress.Any, 8088);
            listener.Start();
            while (true)
            {
                Console.WriteLine("In attesa di client");
                TcpClient tcpClient = listener.AcceptTcpClient();

                Console.WriteLine("Connessione effettuata");
                TcpChatServer wrk = new TcpChatServer();
                wrk.AddClient(tcpClient);
            }
            listener.Stop();
        }
    }
}
```

Quindi dal Main, trasportiamo la logica nella classe. Osserviamo che facciamo partire un thread secondario anche per il lavoro di ascolto del Server.

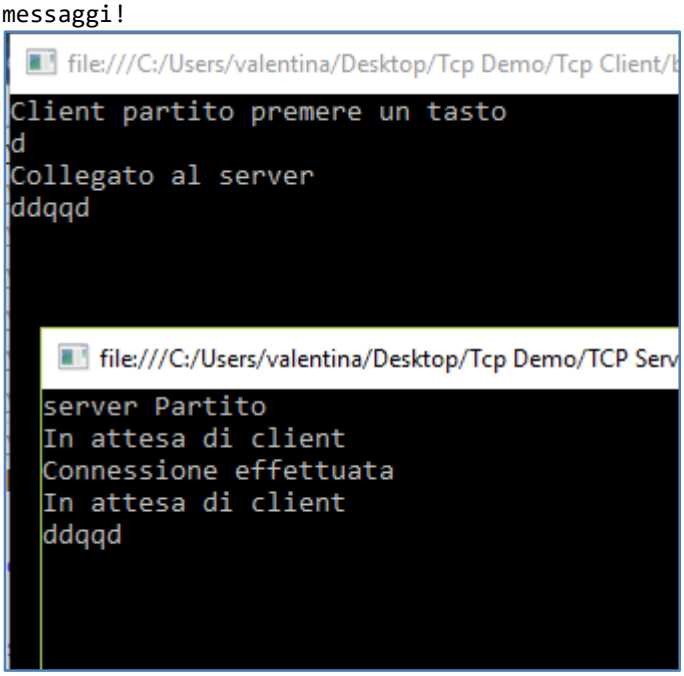
```
namespace ChatServerLibrary
{
    public class TcpChatServer
    {

        private TcpClient _tcpClient;
        private List<TcpClient> _clientsList;

        public TcpChatServer()
        {
```

```
            _clientsList = new List<TcpClient>();
        }
        public void AddClient(TcpClient tcpClient)
        {//codice già visto        }
        public void RemoveClient(TcpClient tcpClient)
        {
            //codice già visto
        }
        void DoWork(TcpClient tcpClient)
        {
          //codice già visto
        }
        private void SendToAll(string msg)
        {
          //codice già visto
        }

        private static void SendToOne(string msg, TcpClient tcpClient)
        {
          //codice già visto
        }

        public void Start()
        {
            Task.Factory.StartNew(ServerWork);
        }
        private void ServerWork()
        {

            TcpListener listener = new TcpListener(IPAddress.Any, 8088);
            listener.Start();
            while (true)
            {
                Console.WriteLine("In attesa di client");
                TcpClient tcpClient = listener.AcceptTcpClient();
                Console.WriteLine("Connessione effettuata");
                AddClient(tcpClient);
            }
            listener.Stop(); // ferma listener (con questa implementazione è inutile)}
        }
    }
}
```

Il codice del Main relativo al server si riduce a 3 semplici righe di codice.

```
namespace TCP_Server
{
    class Program
    {
        static void Main(string[] args)
        {

                TcpChatServer wrk = new TcpChatServer();
                wrk.Start();
            Console.WriteLine("server Partito");
            Console.ReadLine();
            }

        }
    }
```

La chat continua a funzionare, con la differenza che adesso il server è pronto per inviare i messaggi!



La parte server è terminata, ora è necessario modificare la parte client per permettere a ciascun utente connesso di leggere i messaggi.

Uso delle sezioni critiche.

Siccome siamo in modello a memoria condivisa, è necessario stabilire dei meccanismi di sincronizzazione.
La sezione critica va messa in tutti i posti in cui si utilizza una lista.
In tutti i linguaggi ci sta un costrutto che permette di stabilire dove inizia e dove finisce la sezione critica e che cosa bloccare.
In .NET l'inizio dela fine della sezione critica va messa tra parentesi graffe e poi si usa la parola chiave lock(risorsa da bloccare).

```
        public void AddClient(TcpClient tcpClient)
        {lock (_tcpClient)
            {
                _clientsList.Add(tcpClient);
            }
            _tcpClient = tcpClient;
            Task.Factory.StartNew((() => DoWork(_tcpClient)));
        }

        public void RemoveClient(TcpClient tcpClient)
        {
            lock (_tcpClient)
            {
                _clientsList.Remove(tcpClient);

            }
        }


private void SendToAll(string msg)
```

```
        {lock(_clientsList)
            {
                foreach (var tcpClient in _clientsList)
                {
                    SendToOne(msg);

                }
            }
        }
```

Ora abbiamo un problema da migliorare.
In TcpChatServer vi è un metodo che con istruzioni che fanno riferimento all'interfaccia grafica

```
private void ServerWork()
        {

            TcpListener listener = new TcpListener(IPAddress.Any, 8088);
            listener.Start();
            while (true)
            {
                Console.WriteLine("In attesa di client");
                TcpClient tcpClient = listener.AcceptTcpClient();
                Console.WriteLine("Connessione effettuata");
                AddClient(tcpClient);
            }
            listener.Stop(); // ferma listener (con questa implementazione è inutile)}
        }
```

Si tratta di trasformare il Fire & Forget in Progress Notification.
Per far ciò è necessario un EventHandler per notificare l'arrivo del messaggio e di un progress per notificare il messaggio, prima al Listner e poi alla UI. Inoltre si dovrebbe anche notificare chi ha inviato il messaggio. Pertanto, siccome serve trasportare due stringe, serve un oggetto di tipo DTO.

Uso di Progress Notification

Riprendiamo il codice della classe TcpChatServer.
Tutti i metodi che riguardano la gestione della lista e l'invio del messaggio possono essere separati in un Worker e pertanto in un thread separato.

```
namespace ChatServer
{
    public class TCPChatServer
    {
        private TcpClient _tcpClient;
        private List<TcpClient> _clientsList;
        public TCPChatServer()
        {
            _clientsList = new List<TcpClient>();
        }

        public void Start()
```

```
        {
            Task.Factory.StartNew(ServerWork);
        }

        private void ServerWork()
        {
            TcpListener listener = new TcpListener(IPAddress.Any, 8080);
            listener.Start();
            while (true)
            {
                Console.WriteLine("In attesa di client");
                TcpClient tcpClient = listener.AcceptTcpClient();
                AddClient(tcpClient);
            }
            listener.Stop();

        }
        public void AddClient(TcpClient tcpClient)
        {
            lock (_clientsList)
            {
                _clientsList.Add(tcpClient);
            }
            _tcpClient = tcpClient;
            Task.Factory.StartNew(() => DoWork(_tcpClient));
        }

        public void RemoveClient(TcpClient tcpClient)
        {
            lock (_clientsList)
            {
                _clientsList.Remove(tcpClient);
            }
        }

        void DoWork(TcpClient tcpClient)
        {
            try
            {
                NetworkStream ns = tcpClient.GetStream();
                StreamReader sr = new StreamReader(ns);

                while (true)
                {
                    string msg = sr.ReadLine();
                    Console.WriteLine(msg);
                    SendToAll(msg);
                }

            }
            catch (Exception)
            {
                RemoveClient(tcpClient);
            }
        }

        private void SendToAll(string msg)
        {
            lock (_clientsList)
            {
                foreach (var tcpClient in _clientsList)
```

```
                {
                    SendToOne(tcpClient, msg);
                }

            }
        }

        private void SendToOne(TcpClient tcpClient, string msg)
        {
            NetworkStream ns = tcpClient.GetStream();
            StreamWriter sw = new StreamWriter(ns);
            sw.AutoFlush = true;
            sw.WriteLine(msg);
        }
    }
}
```

Pertanto andiamo a creare una classe ClientWorker che conterrà tutti i metodi del codice precedente evidenziati in giallo. Ovviamente sarà necessario creare un costruttore ed un metodo Start. La lista clientsList potrebbe essere fatta passare dal costruttore o dal metodo Start ma siccome la lista serve all'oggetto è preferibile passarla al costruttore. In tal caso è necessario anche creare una variabile.

```
        private List<TcpClient> _clientsList;

        public ClientWorker(List<TcpClient> clientsList)
        {
            _clientsList = clientsList;
        }
```

Al metodo Start invece si passa la connessione (tcpClient), la scelta è dovuta dal fatto che la connsessione è specifica del lavoro da svolgere e non dell'oggetto. In tal caso non è necessario creare una specifica variabile.

```
        public void Start(TcpClient tcpClient)
        {
            Task.Factory.StartNew(() => DoWork(tcpClient));
        }
```

Il ClientWorker dovrà notificare il messaggio e le informazioni del mittente. Pertanto è necessario un DTO.

```
namespace ChatServer
{
    public class ReceivedMessage
    {
        public string Message { get; set; }
        public string Sender { get; set; }
    }
}
```

A questo punto è possibile creare un evento OnClientReceived.

```
public event EventHandler<ReceivedMessage> OnClientMessageReceived;
```

In questo modo possiamo modificare il codice DoWork cancellando ogni riferimento al codice dell'interfaccia (Cinsole.Writeline) ed inserendo la notifica.

```
        void DoWork(TcpClient tcpClient)
        {
            try
            {
                NetworkStream ns = tcpClient.GetStream();
                StreamReader sr = new StreamReader(ns);

                while (true)
                {
                    string msg = sr.ReadLine();
                    //scompare la Console.WriteLine (il messaggio all'inizio arriva qui!)
                    //la notifica chiede se ci sta qualcuno in ascolto
                    NotifyMessageReceived(msg, tcpClient);
                    SendToAll(msg);
                }

            }
            catch (Exception)
            {
                RemoveClient(tcpClient);
            }
        }
```

Quando arriva il messaggio il codice chiede se ci sta qualcuno in ascolto

```
        private void NotifyMessageReceived(string message, TcpClient tcpClient)
        {
            if (OnClientMessageReceived != null)
            {
                var msg = new ReceivedMessage();
                msg.Message = message;
                msg.Sender = tcpClient.Client.LocalEndPoint.ToString();
                // si implementa la notifica
                OnClientMessageReceived(this, msg);
            }
        }
```

Ecco il codice definitivo di ClientWWorker:

```
namespace ChatServer
{
    public class ClientWorker
    {

        public event EventHandler<ReceivedMessage> OnClientMessageReceived;

        private List<TcpClient> _clientsList;

        public ClientWorker(List<TcpClient> clientsList)
        {
            _clientsList = clientsList;
        }


        public void Start(TcpClient tcpClient)
        {
            Task.Factory.StartNew(() => DoWork(tcpClient));
        }
        void DoWork(TcpClient tcpClient)
        {
            try
```

```
            {
                NetworkStream ns = tcpClient.GetStream();
                StreamReader sr = new StreamReader(ns);

                while (true)
                {
                    string msg = sr.ReadLine();
                    //scompare la Console.WriteLine (il messaggio all'inizio arriva qui!)
                    //la notifica chiede se ci sta qualcuno in ascolto
                    NotifyMessageReceived(msg, tcpClient);
                    SendToAll(msg);
                }

            }
            catch (Exception)
            {
                RemoveClient(tcpClient);
            }
        }

        public void RemoveClient(TcpClient tcpClient)
        {
            lock (_clientsList)
            {
                _clientsList.Remove(tcpClient);
            }
        }

        private void NotifyMessageReceived(string message, TcpClient tcpClient)
        {
            if (OnClientMessageReceived != null)
            {
                var msg = new ReceivedMessage();
                msg.Message = message;
                msg.Sender = tcpClient.Client.LocalEndPoint.ToString();
                //devo implementare la notifica
                OnClientMessageReceived(this, msg);
            }
        }

        private void SendToAll(string msg)
        {
            lock (_clientsList)
            {
                foreach (var tcpClient in _clientsList)
                {
                    SendToOne(tcpClient, msg);
                }

            }
        }

        private void SendToOne(TcpClient tcpClient, string msg)
        {
            NetworkStream ns = tcpClient.GetStream();
            StreamWriter sw = new StreamWriter(ns);
            sw.AutoFlush = true;
            sw.WriteLine(msg);
        }

    }
```

```
}
```

In ascolto della notifica vi è il metodo AddClient di TcpChatServer che genera l'oggetto Worker, si sosttoscrive all'evento e avvia il task. In questo modo si è trasformato il precedente Fire&Forget in ProgressNotification.

```
        public void AddClient(TcpClient tcpClient)
        {
            lock (_clientsList)
            {
                _clientsList.Add(tcpClient);
            }

            ClientWorker cli = new ClientWorker(_clientsList);
            cli.OnClientMessageReceived += OnClientMessageReceived;
            cli.Start(tcpClient);
        }
```

Ora anche il TCPChatServer deve notificare i dati ricevuti alla UI e pertanto va implementato anche in questa classe il sistema di notifiche. Vediamo il codice completo.

```
    public class TCPChatServer
    {

        public event EventHandler<ReceivedMessage> OnMessageReceivedFromClient;

        private TcpClient _tcpClient;
        private List<TcpClient> _clientsList;
        public TCPChatServer()
        {
            _clientsList = new List<TcpClient>();
        }

        public async void Start()
        {
            TcpListener listener = new TcpListener(IPAddress.Any, 8080);
            listener.Start();
            while (true)
            {
                Console.WriteLine("In attesa di client");
                TcpClient tcpClient = listener.AcceptTcpClient();
                AddClient(tcpClient);
            }
            listener.Stop();
        }


        public void AddClient(TcpClient tcpClient)
        {
            lock (_clientsList)
            {
                _clientsList.Add(tcpClient);
            }

            ClientWorker cli = new ClientWorker(_clientsList);
            cli.OnClientMessageReceived += OnClientMessageReceived;
            cli.Start(tcpClient);
        }
```

```
        // ora va notificato il messaggio al Server
        private void OnClientMessageReceived(object sender, ReceivedMessage e)
        {
            if (OnMessageReceivedFromClient != null)
                OnMessageReceivedFromClient(this, e);
        }

    }
```

Pertanto anche il codice relativo alla UI viene modificato.
Si crea l'oggetto TcpChatServer, prima di avviarlo si sottoscrive all'evento in modo tale che al momento della ricezione dei dati verrà visualizzato tutto dall'interfaccia.

```
namespace TCP_Server
{
    class Program
    {
        static void Main(string[] args)
        {
            TCPChatServer worker = new TCPChatServer();
            // adesso posso attivare il sistema di notifica al TcpChatServer
            worker.OnMessageReceivedFromClient += OnMessageReceivedFromClient;
            worker.Start();
            Console.WriteLine("Server partito");

            Console.ReadLine();

        }

        private static void OnMessageReceivedFromClient(object sender, ReceivedMessage e)
        {
           Console.WriteLine($"Messaggio:{e.Message} - Ricevuto da:{e.Sender}");
        }
    }
}
```

Osservazione:
In TCPChatServer si potrebbe creare anche un altro task secondario che gestisce l'arrivo di un nuovo client.

Chat lato Client

Vediamo il codice per la parte Client.
Anche in questo caso è necessario creare un sistema di notifiche e pertanto

Una classe Worker che chiameremo TCpChatClient.

```
namespace ChatClient
{
    public class TCPChatClient
    {
        public event EventHandler<string> OnMessageReceived;
        StreamReader _sr;
```

```
        public TCPChatClient(StreamReader sr)
        {
            _sr = sr;
        }
        public void Start()
        {
            Task.Factory.StartNew(() => DoWork(_sr));
        }
        void DoWork(StreamReader sr)
        {
            string msgRicevuto = sr.ReadLine();
            NotifiMessageReceived(msgRicevuto);
        }

        private void NotifiMessageReceived(string msgRicevuto)
        {
            if (OnMessageReceived != null)
                OnMessageReceived(this, msgRicevuto);
        }
    }
}
```

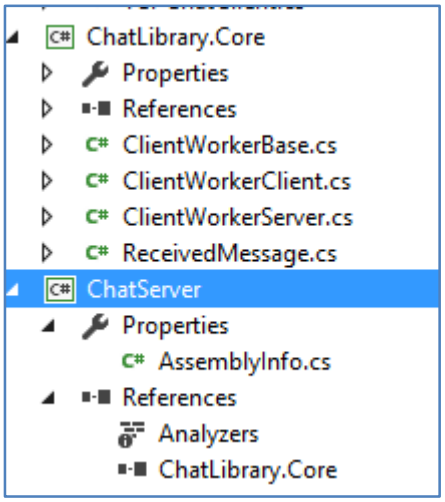
Il codice relativo all'interfaccia sarà così modificato:

```
namespace Tcp_Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Client partito premere un tasto");
            Console.ReadLine();

            TcpClient cli = new TcpClient();
            try
            {
                cli.Connect("127.0.0.1", 8081);
                NetworkStream ns = cli.GetStream();
                StreamWriter sw = new StreamWriter(ns);
                StreamReader sr = new StreamReader(ns);
                sw.AutoFlush = true;

                TCPChatClient chatClient = new TCPChatClient(sr);
                chatClient.OnMessageReceived += ChatClient_OnMessageReceived;
                chatClient.Start();

                Console.WriteLine("Collegato al server");
                while (true)
                {
                    string msg = Console.ReadLine();
                    sw.WriteLine(msg);
                    //Task.Factory.StartNew(()=> DoWork(sr));
                }

            }
            catch (Exception ex)
            {

                Console.WriteLine("Server non trovato");
            }
            Console.ReadLine();
        }
```

```
        private static void ChatClient_OnMessageReceived(object sender, string e)
        {
            Console.WriteLine(e);
        }

        //static void DoWork(StreamReader sr)
        //{
        //    string msgRicevuto = sr.ReadLine();
        //    Console.WriteLine(msgRicevuto);
        //}
    }
}
```

Ereditarietà nella Chat

Vediamo come possiamo ottimizzare il codice con ereditarietà e classi astratte.
Osserviamo che il codice relativo al WorkerClient del Client e del Server sono molto simili salvo piccole modifiche. E' possibile dunque creare una nuova Library, denominata ChatLibraryCore, con le classi in comune al Client ed al Server. Ovviamente nelle Library ChatServer e ChatClient necessario afggiungervi la reference al nuovo progetto.
La struttura della solution sarà come rappresentato in figura:



La Classe ReceivedMessage era comune a tuttee edue i progetti (sia Client, sia server) pertanto non viene modificata.

```
namespace ChatLibrary.Core
{
    public class ReceivedMessage
    {
        public string Message { get; set; }
        public string Sender { get; set; }
    }
}
```

Passiamo al codice relativo alla classe astratta WorkerBase.cs

Sottilineiamo in giallo le parole chiave tipiche necessarie per implementare l'ereditarietà.

```
namespace ChatLibrary.Core

{
    public abstract class ClientWorkerBase
    {
        public event EventHandler<ReceivedMessage> OnClientMessageReceived;
        public void Start(TcpClient tcpClient)
        {
            Task.Factory.StartNew(() => DoWork(tcpClient));
        }
        void DoWork(TcpClient tcpClient)
        {
            try
            {
                NetworkStream ns = tcpClient.GetStream();
                StreamReader sr = new StreamReader(ns);

                while (true)
                {
                    string msg = sr.ReadLine();
                    NotifyMessageReceived(msg, tcpClient);
                }

            }
            catch (Exception)
            {
                RemoveClient(tcpClient);
            }

        }
        private void NotifyMessageReceived(string message, TcpClient tcpClient)
        {
            if (OnClientMessageReceived != null)
            {
                var msg = new ReceivedMessage();
                msg.Message = message;
                msg.Sender = tcpClient.Client.LocalEndPoint.ToString();
                OnClientMessageReceived(this, msg);
            }
        }

        public virtual void RemoveClient(TcpClient tcpClient)
        {

        }
    }
}
```

La Classe ClientWorkerClient eredita la classe base, senza necessità di modifiche (a parte il costruttore).

```
namespace ChatLibrary.Core
{
    public class ClientWorkerClient : ClientWorkerBase
    {
```

```
        public ClientWorkerClient()
        {

        }
    }
}
```

La classe ClientWorkerServer invece effettua l'overraide del metodo RemoveClient e utilizza la lista di client.

```
namespace ChatLibrary.Core
{
    public class ClientWorkerServer : ClientWorkerBase
    {
        private List<TcpClient> _clientsList;

        public ClientWorkerServer(List<TcpClient> clientsList)
        {
            _clientsList = clientsList;
        }
        public override void RemoveClient(TcpClient tcpClient)
        {
            lock (_clientsList)
            {
                _clientsList.Remove(tcpClient);
            }
        }
    }
}
```