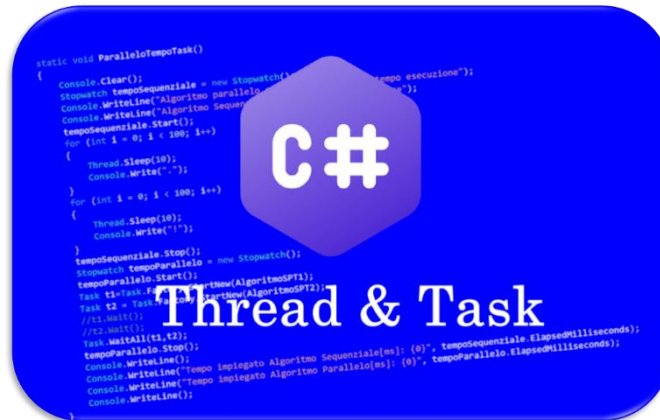
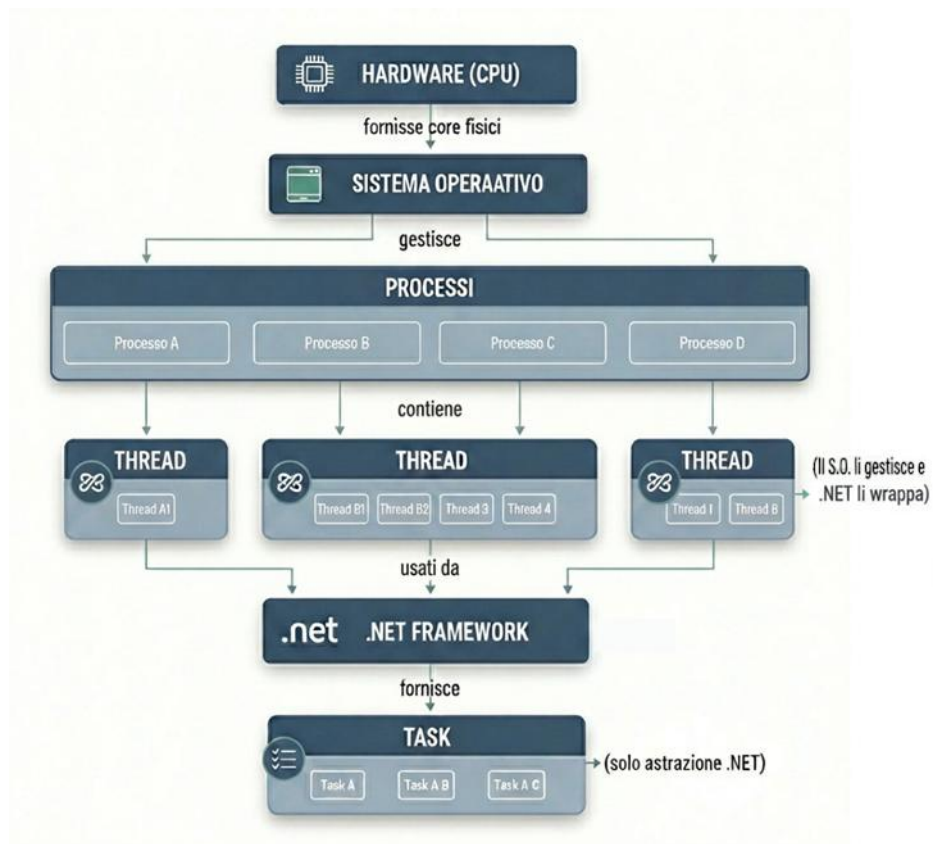

MULTITHREADING IN C#



Sommario

PROCESSI -THREAD -TASK.....	1
Il Multithreading in C#.....	2
Thread	2
thread-safe.....	3
Creazione e gestione dei Thread in C#.....	5
Passaggio dati e comunicazione tra thread.....	10
Thread con metodi statici o di istanza senza parametri	10
Thread con metodi statici o di istanza con parametri.....	11
Thread con parametri tramite classe Helper.....	12
Thread con espressione lambda con e senza parametri	13
BEST PRACTICES	16
Sincronizzazione tra Thread:	17
Lock, Monitor, Mutex e Semaphore	17
Esercizi pratici con codice	27
Task Parallel Library (TPL) e Task	29
task	29
Il Meccanismo Asincrono (async / await).....	33
Cancellazione Cooperativa dei Thread/Task	36
Tabella riassuntiva Thread, Task e Parallel.For.....	38

PROCESSI -THREAD -TASK



Il Multithreading in C#

Thread

Un **thread** è il più piccolo flusso di esecuzione indipendente all'interno di un programma. In pratica sono porzioni di codice di un processo che **possono essere eseguite in parallelo** dal sistema operativo, condividendo le risorse del processo.

Caratteristiche principali dei thread:

- **Sono gestiti dal sistema operativo**, che ne cura scheduling e preemption (meccanismo con cui il **sistema operativo interrompe forzatamente un thread/processo in esecuzione** per assegnare la CPU a un altro). C# fornisce invece le classi e i meccanismi necessari per la loro gestione logica e per il coordinamento dell'esecuzione.
- **Dispongono di risorse proprie**, come stack, registri della CPU e contesto di esecuzione.
- Sebbene siano più leggeri dei processi, **sono considerati risorse costose**: la loro creazione richiede memoria (circa 1MB di stack), chiamate al kernel ed un tempo di circa 50-200ms per la creazione.
- **Operano in modo sincrono (bloccante)**: durante un'attesa (I/O, lock, sleep) il thread resta occupato e **non può sospendersi in modo efficiente** per liberare risorse da destinare ad altri thread.

In C#, la gestione dei thread è affidata al namespace `System.Threading`.

- **Classe principale: `Thread`**. Rappresenta l'oggetto fisico che il Sistema Operativo eseguirà.
- **Costruttore: `new Thread(metodo)`**. Inizializza una nuova istanza della classe, associandola a un metodo da eseguire.
- **Metodo di avvio: `Start()`**. È il comando che comunica al Sistema Operativo di inserire il thread nella coda di esecuzione.

Il **multithreading** è la capacità di un programma di eseguire più thread contemporaneamente, consentendo l'esecuzione parallela o concorrente di più operazioni all'interno dello stesso processo, sfruttando soprattutto le CPU multi-core.

Tuttavia, il *modo* in cui i thread avanzano dipende dall'hardware sottostante:

1. **Concorrenza (Single Core - "L'illusione della simultaneità")**
Se la CPU ha **1 solo Core**, il parallelismo è simulato dal Sistema Operativo.
Come funziona: Il processore dedica rapidissimi intervalli di tempo (*time slice*) a ciascun thread, passando da uno all'altro migliaia di volte al secondo (*Context Switch*).
Risultato: All'utente sembra che tutto accada insieme, ma in realtà la CPU esegue **un'istruzione alla volta**.
2. **Parallelismo (Multi Core - "La vera simultaneità")**
Se la CPU ha **più Core**, i thread vengono eseguiti fisicamente nello stesso istante.

Come funziona: Ogni Core si prende carico di un thread diverso.

Risultato: La velocità di esecuzione aumenta drasticamente, perché c'è più "forza lavoro" reale.

Vantaggi del multithreading:

- Migliora la reattività: mantiene l'UI sempre attiva mentre i thread secondari eseguono operazioni lunghe.
- **Sfrutta la potenza delle CPU multi-core** per eseguire attività in parallelo.
- **Migliora la scalabilità** in applicazioni server e cloud.

Considerazione, anche se la creazione di thread porta a vantaggi in termini di performance, bisogna sempre ragionare in termini di efficienza e rapportare il numero dei thread creati in base al numero di core fisici a disposizione.

Ad esempio, se creassimo 100 thread su una CPU con 4 Core, solo 4 lavorerebbero in **Parallelo**. Gli altri 96 lavorerebbero in **Concorrenza**, costringendo la CPU a perdere tempo per gestire i turni (Context Switch) invece di calcolare.

Molte CPU moderne implementano il sistema **Hyper-Threading** (Intel) o Simultaneous Multithreading (**SMT** AMD). Questa tecnologia hardware permette a un singolo **core fisico** di presentarsi al Sistema Operativo come due **processori logici**.

Questo "sdoppiamento" è reso possibile sfruttando i **tempi morti** (chiamati tecnicamente *pipeline stalls*) che le CPU incontrano inevitabilmente durante la loro attività computazionale.

thread-safe

Il codice si definisce **thread-safe** quando è realizzato per funzionare correttamente anche se più thread lo eseguono contemporaneamente. Il suo scopo principale è proteggere le informazioni condivise, garantendo che solo un thread alla volta possa modificarle, mentre gli altri attendono ordinatamente il proprio turno. Questo previene la corruzione dei dati e risultati imprevedibili.

I principali problemi di concorrenza che si possono verificare quando si utilizzano i thread sono:

- **Race conditions:** si verifica quando due o più thread accedono e modificano simultaneamente dati condivisi senza un'adeguata sincronizzazione, causando risultati imprevedibili.
- **Deadlock:** situazione di stallo in cui due (o più) thread restano bloccati, ciascuno in attesa di una risorsa detenuta dall'altro.

C# mette a disposizione diversi meccanismi di sincronizzazione che permettono di implementare la thread-safety, come **lock**, **Monitor**, **Mutex** e i **semafori**. Questi strumenti consentono di controllare l'accesso concorrente alle risorse condivise e di gestire correttamente errori ed eccezioni all'interno dei thread.

Obiettivo del **thread-safety**: garantire che più thread possano accedere contemporaneamente alle stesse risorse senza compromettere la coerenza dei dati, la correttezza dell'esecuzione e la stabilità del programma.

Esercizi pratici con codice

Esercizio 1: confronto tra esecuzione sequenziale e parallela

Scrivere due versioni di un programma che stampa due sequenze di caratteri sullo schermo ('A' e 'B'):

- Versione sequenziale: stampa prima tutti gli 'A', poi tutti i 'B'.
- Versione parallela: usa un thread aggiuntivo per stampare 'A' mentre il main stampa 'B'.

```
// Versione sequenziale
static void Main(string[] args) {
    for (int i = 0; i < 100; i++) Console.Write("A");
    for (int i = 0; i < 100; i++) Console.Write("B");
}

// Versione parallela (con Thread)
static void Main(string[] args) {
    Thread t = new Thread(StampaA);
    t.Start();
    for (int i = 0; i < 100; i++) Console.Write("B");
}

static void StampaA() {
    for (int i = 0; i < 100; i++) Console.Write("A");
}
```

Creazione e gestione dei Thread in C#

Il modello base per la gestione dei thread in C# passa attraverso la creazione di oggetti **Thread** che fanno riferimento a metodi (statici o di istanza) e che vengono avviati tramite il metodo **Start()**.

Principali Metodi della classe **thread**:

- **Start()**: Avvia il thread. Comunica al sistema operativo che il thread è pronto per essere eseguito.
- **Join()**: Blocca il thread chiamante (spesso il Main) finché il thread su cui è chiamato non termina. È fondamentale per sincronizzare la fine dei lavori.
- **Thread.Sleep(int milliseconds)**: (Metodo statico) Mette in pausa il thread corrente per un tempo determinato. Utile per simulare attese o alleggerire il carico sulla CPU.

Principali Proprietà della classe **thread**:

- **Name**: Permette di assegnare una stringa descrittiva al thread. Si consiglia di usarlo per facilitare il debugging.
- **IsAlive**: Restituisce un valore booleano che indica se il thread è attualmente in esecuzione o attivo.
- **ManagedThreadId**: Restituisce un identificativo numerico univoco per il thread corrente. Utilizzato per distinguere, capire, quale thread sta eseguendo il codice (spesso utilizzato per creare log `Console.WriteLine("[Thread-{0}] Messaggio: {1}", Thread.CurrentThread.ManagedThreadId, Messaggio);`
- **ThreadState**: Indica lo stato esatto del thread (**Unstarted**, **Running**, **WaitSleepJoin**, **Stopped**).
- **Priority**: Permette di impostare l'importanza del thread (**Lowest**, **BelowNormal**, **Normal**, **AboveNormal**, **Highest**). Questa proprietà è solo un suggerimento allo scheduler ma non garantisce che l'ordine di esecuzione venga modificato. Il Sistema Operativo proverà a dare più tempo di CPU ai thread con priorità alta.
- **IsBackground**: Impostata per controllare il comportamento dell'applicazione.
 - **Foreground (false)**: Il processo (l'applicazione) resta in vita finché **almeno uno** di questi thread è attivo. Se il **Main** finisce ma un thread Foreground è ancora in esecuzione, l'applicazione appare chiusa ma il processo è ancora visibile in "Gestione Attività" (Task Manager).
 - **Background (true)**: Questi thread sono "subordinati". Il Sistema Operativo li interrompe bruscamente nel momento in cui l'ultimo thread Foreground del programma termina.

Quando Usarli?

- **Foreground (Default)**: si usano per operazioni critiche che **devono** essere completate prima che l'applicazione si chiuda (es. salvare un file, inviare dati importanti a un server).
- **Background**: si usano per operazioni non essenziali o periodiche che possono essere interrotte senza problemi (es. controllare aggiornamenti, indicizzare file in background, logging).

Creazione di un thread

Costruttori

- **Thread(ThreadStart)**
Inizializza una nuova istanza della classe Thread, specificando un delegato senza parametri.
- **Thread(ParameterizedThreadStart)**
Inizializza una nuova istanza della classe Thread, specificando un delegato che consente di passare un oggetto al thread quando quest'ultimo viene avviato.

Esempio base che crea due thread, il primo (t1) esegue un metodo statico ed il secondo (t2) che esegue il metodo pubblico di una classe

```
using System;
using System.Threading;
public class Info
{
    public void Stampa(object Messaggio)
    {
        Console.WriteLine((string)Messaggio);
    }
}
class Program
{
    static void Main()
    {
        Info a = new Info();
        Thread t1 = new Thread(StaticPrint);
        t1.Start();
        Thread t2 = new Thread(new ParameterizedThreadStart(a.Stampa));
        // Thread t2 = new Thread(a.Stampa); uguale perché C# usa la Delegate Inference
        // (Inferenza del delegato)
        t2.Start("Thread su metodo di istanza");
        Console.WriteLine("\nStato del Thread t1: {0}\nStato del Thread t2: {1}", t1.ThreadState,
t2.ThreadState);
        if(t1.IsAlive)
            Console.WriteLine("Il Thread t1 è VIVO");
        else
            Console.WriteLine("Il Thread t1 ha TERMINATO Il suo ciclo di VITA oppure è stato
INTERROTTO");
        t1.Join();
        t2.Join();
        if (t2.IsAlive)
            Console.WriteLine("Il Thread t2 è VIVO");
        else
            Console.WriteLine("Il Thread t2 ha TERMINATO Il suo ciclo di VITA oppure è stato
INTERROTTO");
        Console.WriteLine("\nFine");
        Console.ReadLine();
    }
    static void StaticPrint()
    {
        Console.WriteLine("Thread su metodo statico");
        Thread.Sleep(100);
    }
}
```

ThreadState può restituire i seguenti valori:

Stato	Significato
Unstarted	Il thread è stato creato ma non ancora avviato (non è stato chiamato Start()).
Running	Il thread è attualmente in esecuzione o pronto per essere eseguito.
WaitSleepJoin	Il thread è sospeso temporaneamente: in attesa (Wait()), addormentato (Sleep()), o in attesa di un altro thread (Join()).
Stopped	Il thread ha terminato l'esecuzione.
AbortRequested / Aborted	È stata richiesta o completata la terminazione forzata del thread.
Background	Il thread è impostato come thread in background (IsBackground = true), e quindi termina automaticamente quando il processo termina.

```
using System;
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("=====");
        Console.WriteLine("    DEMO STATI PRINCIPALI DEI THREAD    ");
        Console.WriteLine("=====\\n");

        Thread worker = new Thread(MetodoThread);
        worker.Name = "Thread Demo";
        Console.WriteLine("Thread creato ma non avviato");
        Console.WriteLine("    ThreadState: {0}\\n", worker.ThreadState);
        Thread.Sleep(1000);

        worker.Start();
        Console.WriteLine("Thread avviato");
        Console.WriteLine("    ThreadState: {0}\\n", worker.ThreadState);
        Thread.Sleep(2000);

        Console.WriteLine("Thread in esecuzione");
        Console.WriteLine("    ThreadState: {0}", worker.ThreadState);
        Thread.Sleep(2000);

        worker.Join();

        Console.WriteLine("Thread terminato");
        Console.WriteLine("    ThreadState: {0}\\n", worker.ThreadState);
        Console.WriteLine("Premi un tasto per uscire...");
        Console.ReadKey();
    }

    static void MetodoThread()
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("    [Thread-{0}] Vado in sleep per 3 secondi...\\n", threadId);
        Thread.Sleep(3000);
        Console.WriteLine("    [Thread-{0}] Svegliato! Termino...\\n", threadId);
    }
}
```


Esempio base in cui vengono creati due thread utilizzando due metodi statici del programma (classe Program) che simulano operazioni varie

```
using System;
using System.Threading;

public class Program
{
    public static void NetworkListener(){
        Console.WriteLine("Il thread '{0}' ha iniziato il suo lavoro.", Thread.CurrentThread.Name);
        Thread.Sleep(5000); // Simula l'ascolto sulla rete
        Console.WriteLine("Il thread '{0}' ha terminato.", Thread.CurrentThread.Name);
    }
    public static void FileLogger(){
        Console.WriteLine("Il thread '{0}' ha iniziato il suo lavoro.", Thread.CurrentThread.Name);
        Thread.Sleep(3000); // Simula la scrittura su file
        Console.WriteLine("Il thread '{0}' ha terminato.", Thread.CurrentThread.Name);
    }
    public static void Main()
    {
        Thread networkThread = new Thread(NetworkListener);
        // Assegno un nome descrittivo
        networkThread.Name = "NetworkRequestListener";

        Thread loggerThread = new Thread(FileLogger);
        // Assegno un nome descrittivo
        loggerThread.Name = "FileLogWriter";

        networkThread.Start();
        loggerThread.Start();

        networkThread.Join();
        loggerThread.Join();
        Console.WriteLine("Fine...\nPremi un tasto per chiudere il programma");
        Console.ReadLine();
    }
}
```

Esercizi

Esercizio 1: Creazione Base e Sincronizzazione con `Join`

Obiettivo: Comprendere come creare, avviare e attendere la terminazione di un thread.

Richieste:

Scrivere un programma che crea due thread

- Il primo thread deve eseguire un metodo che stampa i numeri da 1 a 5, con una pausa di 1 secondo tra ogni numero.
 - Il secondo thread deve eseguire un metodo che stampa le lettere dalla 'A' alla 'E', con una pausa di 500 millisecondi tra ogni lettera.
 - Il thread principale (`Main`) deve avviare entrambi i thread e **poi attendere che entrambi abbiano terminato** il loro lavoro prima di stampare un messaggio finale ("Lavoro completato.") e chiudersi.
 - Utilizza il metodo `Join()` per garantire che il `Main` aspetti correttamente.
-

Esercizio 2: Monitoraggio dello Stato di un Thread

Obiettivo: Imparare a controllare lo stato di un thread durante la sua esecuzione.

Richieste:

Scrivere un programma che crea tre thread

- Assegna un nome ad ogni thread (es. Primo, Secondo, Terzo)
 - Ogni thread stampa il proprio nome ed esegue un'operazione lunga che dura 10 secondi.
 - Il main stampa lo stato di ogni thread e
 - se non è "vivo" lo avvia thread.
 - Attende 3 secondo.
 - Stampa di nuovo lo stato di ogni thread e se è "vivo"
 - Usa `Join()` per attendere la sua terminazione.
 - Stampa per l'ultima volta lo stato dei thread.
 - Analizza e commenta nel codice l'output che ti aspetti di vedere per ogni `Console.WriteLine`.
-

Esercizio 3: Foreground vs. Background

Obiettivo: Comprendere la differenza fondamentale tra thread in foreground e in background e il loro impatto sulla terminazione dell'applicazione.

Richieste:

Crea due thread, entrambi con un ciclo di lavoro che dura 5 secondi.

- Imposta il **primo thread come background**.
 - Lascia il **secondo thread come foreground**.
 - Il thread principale (`Main`) deve avviare entrambi i thread e poi stampare un messaggio ("Il Main ha finito, l'applicazione si chiuderà?") e terminare immediatamente, **senza usare `Join()`**.
 - Esegui il programma e osserva il comportamento. Spiega perché l'applicazione rimane aperta e cosa viene stampato a schermo.
 - **Modifica:** Commenta la riga che avvia il thread in foreground. Esegui di nuovo il programma e descrivi la differenza nel comportamento di chiusura dell'applicazione.
-

Esercizio 4: Combinazione di Concetti (Manager e Operaio)

Obiettivo: Simulare uno scenario più complesso in cui un thread "manager" monitora un thread "operaio".

Richieste:

1. Creare un thread **operaio** (in **foreground**) che simuli un lavoro di 10 secondi.
 2. Creare un thread **supervisore** (in **background**) che, ogni 2 secondi, controlli lo stato del thread operaio e stampi un messaggio a console, come: `""[Supervisore]: Lo stato dell'operaio è: Running"`.
 3. Il thread principale (`Main`) ha il ruolo di "manager":
 - Avvia sia il thread operaio che quello supervisore.
 - Attende la terminazione **solo** del thread operaio usando `Join()`.
 - Una volta che l'operaio ha finito, il `Main` stampa un messaggio finale ("L'operaio ha finito, il manager chiude la giornata.") e termina.
 4. Spiega perché il thread supervisore si arresta automaticamente senza bisogno di un `Join()` specifico.
-

Passaggio dati e comunicazione tra thread

Nella programmazione multithread, spesso è necessario passare informazioni specifiche ai thread. I principali approcci sono:

- **Usare il costruttore di Thread** con delegate senza parametri (**ThreadStart**) o con parametro **object** (**ParameterizedThreadStart**).
- **Utilizzare lambda expressions** per accedere direttamente alle variabili già definite nel programma, mantenendo il loro tipo.

Attenzione: **ParameterizedThreadStart** accetta solo **object**, quindi richiede cast e attenzione ai tipi.

Sintassi

Di seguito sono riportate tutte le principali sintassi per istanziare e avviare un thread in C#.

Esistono tre tipi principale di istanziazione di oggetti thread:

- thread con metodi statici o di istanza senza parametri
- thread con metodi statici o di istanza con parametri
- thread con espressione lambda con e senza parametri

Thread con metodi statici o di istanza senza parametri

```
using System;
using System.Threading;
{
    public class MiaClasse {
        public void Stampa() {
            Console.WriteLine("Thread su metodo di istanza in esecuzione...");
        }
    }
}
class Program
{
    static void Main()
    {
        // istanziazione con metodo statico
        Thread t = new Thread(new ThreadStart(MioMetodo));
        t.Start(); // avvio del thread
        MiaClasse mc=new MiaClasse();
        // istanziazione con metodo di istanza
        Thread t1 = new Thread(new ThreadStart(mc.Stampa));
        t1.Start(); // avvio del thread
        /* In entrambe i casi è possibile istanziare direttamente il thread senza l'istanziazione esplicita
        dell'oggetto tramite new ThreadStart
        Thread t = new Thread(MioMetodo);
        Thread t1 = new Thread(mc.Saluta);
        sistema più moderno e da preferire */
        Console.ReadLine();
    }
    static void MioMetodo(){
        Console.WriteLine("Thread su metodo statico in esecuzione...");
    }
}
```

Thread con metodi statici o di istanza con parametri

```
using System;
using System.Threading;
{
    public class MiaClasse {
        public void Stampa(object Messaggio) {
            Console.WriteLine((string)Messaggio);
        }
    }
    class Program
    {
        static void Main()
        {
            // istanziazione con metodo statico
            Thread t = new Thread(new ParameterizedThreadStart(MioMetodo));
            t.Start("Thread su metodo statico in esecuzione..."); // avvio del thread
            MiaClasse mc=new MiaClasse();
            // istanziazione con metodo di istanza
            Thread t1 = new Thread(new ParameterizedThreadStart(mc.Stampa));
            t1.Start("Thread su metodo di istanza in esecuzione..."); // avvio del thread
            /* In entrambe i casi è possibile istanziare direttamente il thread senza l'istanziazione esplicita
            dell'oggetto tramite new ParameterizedThreadStart
            Thread t = new Thread(MioMetodo);
            Thread t1 = new Thread(mc.Saluta);
            sistema più moderno e da preferire */
            Console.ReadLine();
        }
        static void MioMetodo(object Messaggio)
        {
            Console.WriteLine((string)Messaggio);
        }
    }
}
```

l'argomento del metodo deve essere di tipo object, quindi serve un cast se si vuole usarlo come tipo specifico.

Se dovessero servire più parametri, conviene creare una classe helper, in cui inserire i parametri, da passare al metodo del thread, come proprietà della classe e nel metodo.

Thread con parametri tramite classe Helper

```
using System;
using System.Threading;
{
    class Program
    {
        public class paramtriHelper
        {
            public string Messaggio;
            public int Valore;
            public DateTime Data;
        }
        static void Main()
        {
            // istanziiazione con metodo di istanza con parametri in classe Helper
            paramtriHelper ph = new paramtriHelper();
            ph.Messaggio = "Thread su metodo di istanza con parameti in classe Helper in
esecuzione...";
            ph.Valore = 100;
            ph.Data = new DateTime(2025, 01, 01);
            Thread t = new Thread( () =>{
                StampaValori(ph);
            }
            );
            t.Start();
            t.Join();
            Console.ReadLine();
        }
        static void StampaValori(object Valori)
        {
            paramtriHelper valori = (paramtriHelper)Valori;
            Console.WriteLine(valori.Messaggio + "\n" + valori.Valore + "\n" +
valori.Data.ToShortDateString());
        }
    }
}
```

Thread con espressione lambda con e senza parametri

```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        // istanziazione con metodo lambda
        Thread t = new Thread(() =>
        {
            Console.WriteLine("Thread su metodo lambda in esecuzione...");
        });
        t.Start();
        Thread t1 = new Thread(Messaggio =>
        {
            Console.WriteLine(Messaggio);
        });
        t1.Start("Thread su metodo lambda con singolo parametro in esecuzione...");
        string MSG = "Thread su metodo lambda con variabili locali in esecuzione...";
        int Numero = 100;
        Thread t2 = new Thread(()=>
        {
            Console.WriteLine(MSG + " Secondo parametro: "+Numero);
        });
        t2.Start();
        t.Join();
        t1.Join();
        t2.Join();
        Console.ReadLine();
    }
}
```

Esercizi

Esercizio 2: Creare thread con parametri

Scrivere un programma in C# che facendo uso dei thread visualizzi i numeri multipli di un valore intero passato come parametro al metodo del thread.

Utilizzare sia il sistema con `ParameterizedThreadStart` che con `lambda`.

Il programma deve terminare quando tutti i thread sono conclusi.

Esercizio 3: Passaggio di parametri tramite classe helper

Eseguire l'esercizio precedente specificando anche i limiti di inizio e fine della generazione dei multipli. Si chiede di utilizzare una classe helper e le lambda per definire il codice le thread

Esercizio 4: Passaggio di parametri tramite lambda expression con metodo statico e metodo istanza.

```
using System;
using System.Threading;

class Calcolatrice
{
    public void Moltiplica(int a, int b)
    {
        Console.WriteLine("Moltiplicazione: {0}", a * b);
    }
}

class Program {
    static void Main() {
        int x = 5;
        int y = 7;
        Thread t = new Thread(() => StampaSomma(x, y));
        t.Start();
        t.Join();

        Calcolatrice calc = new Calcolatrice();
        Thread t1 = new Thread(() => calc.Moltiplica(x, y));
        t1.Start();
        t1.Join();
        Console.WriteLine("Fine ..... \nPremi un tasto per chiudere il programma");
        Console.ReadLine();
    }
    static void StampaSomma(int a, int b){
        Console.WriteLine("Somma: {0}", a + b);
    }
}
```

Esercizio 5: esempio completo con thread multipli e restituzione di uno o più valori attraverso un delegato callback.

```
using System;
using System.Threading;

public delegate void CallBackLavoro(int NumCicli);
class lavoro
{
    public void LavoraBase(){
        Console.WriteLine("Lavoro....!");
    }
    public string Chi { get; set; }
    public int Quanto { get; set; }
    private CallBackLavoro callback;
    public lavoro() { }
    public lavoro(string nome,int q){
        this.Chi = nome;
        this.Quanto = q;
    }
    public lavoro(string nome, int q, CallBackLavoro ret){
        this.Chi = nome;
        this.Quanto = q;
        callback = ret;
    }
    public void Lavora(){
        for (int i = 1; i < Quanto; i++){
            Console.WriteLine(Chi + " => {0}/{1}", i, Quanto);
            Thread.Sleep(50);
        }
        if (callback != null)
            callback(Quanto);
    }
}
class Program
{
    static void StampaRisultato(int ris){
        Console.WriteLine("Il ciclo è stato eseguito {0} volte",ris);
    }
    static void Main(string[] args){
        /*****
        * Esecuzione di due thread semplici
        * *****/
        lavoro l=new lavoro();
        Thread th = new Thread(l.LavoraBase);           //Sintassi moderna con delegate inference
        Thread th1 = new Thread(new ThreadStart(l.LavoraBase)); //Sintassi esplicita (vecchio stile)
        th.Start();
        th1.Start();
        th.Join();
        th1.Join();
        Console.ReadLine();
        /*****
        * Esecuzione di due thread con parametri attraverso le proprietà della classe
        * *****/
        lavoro lP1 = new lavoro("Primo", 10);
        lavoro lP2 = new lavoro("Secondo",20);
        Thread thP = new Thread(lP1.Lavora);
        Thread thP1 = new Thread(new ThreadStart(lP2.Lavora));
        thP.Start();
    }
}
```



```

        thP1.Start();
        thP.Join();
        thP1.Join();
        Console.ReadLine();
    }
}

/*****
* Esecuzione di due thread con parametri attraverso le proprietà della classe
* e restituzione di uno o più valori attraverso un delegato callback
* *****/
    lavoro lPC1 = new lavoro("Primo", 15, StampaRisultato);
    lavoro lPC2 = new lavoro("Secondo", 25, StampaRisultato);
    Thread thPC = new Thread(lPC1.Lavora);
    Thread thPC1 = new Thread(new ThreadStart(lPC2.Lavora));
    thPC.Start();
    thPC1.Start();
    thPC.Join();
    thPC1.Join();
    Console.ReadLine();
}
}

```

BEST PRACTICES

1. **Usare la sintassi breve** (inference):
`Thread t = new Thread(metodo);`
 È più pulita e leggibile.
2. **Usare le Lambda** per logica semplice o quando si devono passare variabili:
`int valore = 42;`
`Thread t = new Thread(() => {`
 `Console.WriteLine("Valore: {0}", valore);`
 `});`
3. **Evitare ParameterizedThreadStart quando possibile:**
 Invece di:
`Thread t = new Thread(new ParameterizedThreadStart(Metodo));`
`t.Start(parametro);`
 Meglio utilizzare:
`Thread t = new Thread(() => Metodo(parametro));`
`t.Start();`
La lambda è type-safe! (il compilatore controlla i **tipi** di dato dei parametri ed i valori di ritorno al momento della compilazione, garantendo che siano corretti e prevenendo errori a runtime)

Vantaggi

- Più leggibile e organizzato
- Facile da estendere (aggiungi altri campi senza toccare la logica del thread)
- Evita cast e errori di tipo
- È la tecnica ufficiale consigliata da Microsoft

Sincronizzazione tra Thread:

Lock, Monitor, Mutex e Semaphore

Quando più thread accedono a una risorsa condivisa (come una variabile o una collezione), occorre garantire la **mutua esclusione** per evitare risultati inattesi.

In C# esistono diversi strumenti che permettono di garantire la mutua esclusione:

- **lock(object):** blocca una sezione di codice permettendo l'accesso a un solo thread per volta (basato su Monitor).
- **Monitor:** fornisce metodi Enter/Exit per il locking esplicito e più controllo sulla sincronizzazione.
- **Mutex:** simile a lock, ma può essere usato tra processi distinti.
- **Semaphore/SemaphoreSlim:** limita il numero di thread che possono accedere contemporaneamente a una risorsa.

Questi costrutti garantiscono che il codice sia *thread-safe*, ossia che possa essere eseguito correttamente da più thread contemporaneamente senza causare errori, perdita di dati o risultati imprevedibili.

In altre parole, un metodo o un oggetto è *thread-safe* quando continua a funzionare in modo corretto anche se più thread lo utilizzano nello stesso momento.

lock

lock è lo statement che implementa il meccanismo di sincronizzazione più semplice e comune in C#.

Impedisce che **più thread eseguano contemporaneamente una stessa sezione critica di codice**.

Una **sezione critica** è una porzione di codice che accede a **risorse condivise**, come variabili o strutture dati, e che quindi deve essere eseguita da **un solo thread alla volta** per evitare risultati incoerenti.

lock utilizza un oggetto come riferimento per il controllo dell'accesso. Quando un thread entra nel blocco lock, acquisisce il controllo sull'oggetto specificato ed esegue il codice contenuto nel blocco.

Se un altro thread tenta di entrare in un blocco lock che utilizza lo stesso oggetto, viene sospeso finché il thread che possiede il lock non termina l'esecuzione del blocco e rilascia il controllo sull'oggetto.

In modo più tecnico, si può immaginare che l'oggetto utilizzato dal lock sia **un lucchetto**.

Quando un thread entra in un blocco lock:

- **il lucchetto viene chiuso**
- il codice contenuto nel blocco diventa **accessibile solo a quel thread**
- gli altri thread che tentano di usare **lo stesso lucchetto** vengono **sospesi in attesa che il lucchetto venga liberato dal thread che lo detiene**.

Quando il thread termina l'esecuzione del blocco lock, il **lucchetto viene aperto** e un altro thread può usarlo. L'ordine in cui i thread acquisiscono il lock non è deterministico, ma dipende dallo scheduling del runtime, non dall'ordine in cui sono stati creati, ne tantomeno dal momento in cui sono arrivati a chiedere il lock su quell'oggetto.

Questo meccanismo, se correttamente utilizzato, permette di **gestire variabili condivise tra più thread**, garantendo la **mutua esclusione** ed evitando problemi di **race condition**.

In **Microsoft .NET**, l'istruzione lock è in realtà una sintassi semplificata che utilizza internamente la classe **Monitor**, la quale si occupa di gestire l'acquisizione e il rilascio del blocco.

La scelta dell'oggetto per il lock

L'oggetto usato come "lucchetto" deve essere **privato e di tipo riferimento** (reference type), per questo scopo si utilizza come standard il tipo `object`. Se si usasse un oggetto pubblico, altre parti di codice potrebbero usarlo per i propri lock, causando un "blocco totale" (deadlock) involontario dell'applicazione.

In **c#** di solito si utilizza: `private static readonly object lock1 = new object();`

Sintassi di base

```
object lockObj = new object();

lock (lockObj)
{
    // SEZIONE CRITICA
    // Solo UN thread alla volta può essere qui
}
```

Tecnicamente lo statement `lock` viene trasformato dal compilatore da

```
object lockObj = new object();
lock (lockObj)
{
    // SEZIONE CRITICA
}
```

in

```
object lockObj = new object();
bool lockTaken = false;
try
{
    Monitor.Enter(_lockObj, ref lockTaken);
    // sezione critica
}
finally
{
    if (lockTaken)
        Monitor.Exit(_lockObj);
}
```

Regole generali sull'uso dei lock

1. Usare un oggetto privato per il lock

```
private static readonly object lockObj = new object();
lock (lockObj)
{
    // ...
}
```

2. Usare il lock dove è strettamente necessario e mantenerlo per il più breve tempo possibile

ERRATO, metodi lenti nel lock che prolungano il blocco per troppo tempo

```
lock (lockObj)
{
    LeggiDatabase();    // Lento!
    ComplicaCalcoli();  // Lento!
    ScriviFile();       // Lento!
}
CORRETTO - Lock solo dove necessario
var dati = LeggiDatabase();
var risultato = ComplicaCalcoli();
lock (lockObj)
{
    variabileCondivisa = risultato; // Veloce!
}
ScriviFile();
}
```

3. Evitare DEADLOCK

Situazione in cui due o più thread si bloccano a vicenda aspettando risorse che non si libereranno mai.

```
object lock1 = new object();
object lock2 = new object();
Thread1:
lock (lock1)
{
    Thread.Sleep(10);
    lock (lock2) // Aspetta lock2
    {
        // ...
    }
}
Thread2:
lock (lock2)
{
    Thread.Sleep(10);
    lock (lock1) // Aspetta lock1
    {
        // ...
    }
}
// Thread1 ha lock1, aspetta lock2
// Thread2 ha lock2, aspetta lock1
// DEADLOCK!
```

Soluzione: Acquisire sempre i lock **nello stesso ordine!**

```
object lock1 = new object();
object lock2 = new object();
Thread1:
lock (lock1)
```

```

{
    Thread.Sleep(10);
    lock (lock2) // Aspetta lock2
    {
        // ...
    }
}
Thread2:
lock (lock1)
{
    Thread.Sleep(10);
    lock (lock2) // Aspetta lock2
    {
        // ...
    }
}
}

```

Esempio dell'uso di **lock** per evitare l'accesso a più thread nella **sezione critica** in cui viene incrementato il contatore nel metodo incrementCounter

```

using System;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("DEMO DI SINCRONIZZAZIONE TRA THREAD IN C#");
        Console.WriteLine("-----\n");
        Console.WriteLine("1. Mutua esclusione tramite Lock");
        LockExample.Run();
    }
}

class LockExample
{
    private static readonly object _lockObject = new object();
    private static int _counter = 0;
    public static void Run()
    {
        // Creiamo 5 task che incrementano il contatore
        Thread[] threads = new Thread[5];
        for (int i = 0; i < threads.Length; i++)
        {
            threads[i] = new Thread(() => IncrementCounter());
            threads[i].Name = "Thread n° " + threads[i].ManagedThreadId;
            threads[i].Start();
        }
        for (int i = 0; i < threads.Length; i++)
        {
            threads[i].Join();
        }
        Console.WriteLine("Valore finale contatore: {0}", _counter);
    }
    private static void IncrementCounter()
    {
        for (int i = 0; i < 1000; i++)
        {
            lock (_lockObject)

```

```

    {
        // Utilizziamo lock per garantire che solo un thread alla volta
        // possa accedere alla sezione critica
        _counter++;
        {
            Console.WriteLine("{0}: contatore = {1}", Thread.CurrentThread.Name, _counter);
        }
    }
}
Console.WriteLine("{0} completato", Thread.CurrentThread.Name);
}
}

```

Monitor

Monitor è la versione più a basso livello di **lock** che si usa quando si ha bisogno di un controllo più avanzato sulla sincronizzazione, cosa che la sintassi semplificata di **lock** non offre.

La differenza sostanziale è che **lock** è bloccante per il thread e potrebbe portare al **deadlock**, mentre **Monitor** ha i metodi necessari per gestire eventuali situazioni di criticità di accesso.

In realtà quando si usa lock, dietro le quinte si sta usando Monitor.

Principali Metodi:

- **Monitor.TryEnter()** Prova ad acquisire, per la quantità di tempo specificata, un blocco esclusivo sull'oggetto specificato e imposta un valore che indica se il blocco è stato ottenuto.
- **Monitor.Wait(lockObject)** Metodo che permette ad un thread che ha il lock **lockObject** di **rilasciarlo temporaneamente** e di mettersi in attesa in modo efficiente, senza consumare CPU (si torna al lock con il metodo **Monitor.Pulse(lockObject)**).
- **Monitor.Pulse(lockObject)** Metodo che permette di **risvegliare un thread in attesa** sull'oggetto **lockObject**, che ha chiamato **Monitor.Wait(lockObject)**, cioè **notifica** al thread che si era messo in attesa che la condizione su cui attendeva potrebbe ora essere vera.
- **Monitor.PulseAll(lockObject)** Sveglia **tutti** i thread in attesa.
- **Monitor.Exit(lockObject)** E' il metodo che **rilascia** il lock su un oggetto..

In sintesi

Monitor rispetto a lock permette di:

1. Impostare un **Timeout** (TryEnter(timeout))
2. Implementare la **Coordinazione** tra lo stesso lock in porzioni di codice diverso attraverso i metodi statici Wait() / Pulse()
3. **Gestire manualmente il controllo di tutti gli aspetti del lock**

Regola: Usa lock di default, passa a Monitor solo se serve un controllo più dettagliato

Esempi dei vari aspetti peculiari di Monitor

```
// 1. TIMEOUT
bool lockTaken = false;
Monitor.TryEnter(obj, TimeSpan.FromSeconds(5), ref lockTaken);
if (!lockTaken) return; // Timeout!

// 2. COORDINAZIONE
lock (obj)
{
    while (coda.Count == 0)
        Monitor.Wait(obj);    // Aspetta
    // Consuma...
    Monitor.Pulse(obj);        // Notifica
}

// 3. GESTIONE MANUALE
if (lockTaken)
{
    Monitor.Exit(obj);          // Rilascio quando decido io
    lockTaken = false;
}
```

Esempio

Produttore/Consumatore con Monitor.Wait/Pulse

Implementare un programma in C# che implementi il paradigma Produttore/Consumatore usando i seguenti parametri:

- **Buffer:** Queue<int>, max 5 elementi
- **Produttore:** Produce 10 numeri (1-10), 300ms/elemento, aspetta se pieno
- **Consumatore:** Consuma 10 numeri, 500ms/elemento, aspetta se vuoto
- **Sincronizzazione:** Monitor.Wait() per aspettare, Monitor.Pulse() per notificare
- **Output:** Stampa produzione/consumo, attese, conteggio buffer
- **Main:** Crea, avvia e aspetta (Join()) entrambi i thread

```
class ProducerConsumer
{
    static Queue<int> buffer = new Queue<int>();
    static object lockObj = new object();
    static int maxSize = 5;

    static void Main(string[] args)
    {
        Console.WriteLine("=== Produttore/Consumatore con l'utilizzo di Monitor.Wait/Pulse ===\n");
        Thread produttore = new Thread(Producer);
        Thread consumatore = new Thread(Consumer);
        produttore.Start();
        consumatore.Start();
        produttore.Join();
        consumatore.Join();
    }
}
```



```

        Console.ReadLine();
    }

    static void Producer()
    {
        for (int i = 1; i <= 10; i++)
        {
            lock (lockObj)
            {
                while (buffer.Count >= maxSize) // Aspetta se il buffer è pieno
                {
                    Console.WriteLine("[Produttore] Buffer pieno, ASPETTO...");
                    Monitor.Wait(lockObj); // Si sospende e rilascia il lock
                }
                // Produce
                buffer.Enqueue(i);
                Console.WriteLine("[Produttore] Prodotto: {0} (buffer: {1})", i, buffer.Count);
                Monitor.Pulse(lockObj); // Notifica/Sveglia il consumatore
            }
            Thread.Sleep(300); // Simula tempo di produzione
        }
        Console.WriteLine("[Produttore] Completato!");
    }

    static void Consumer()
    {
        for (int i = 1; i <= 10; i++)
        {
            lock (lockObj)
            {
                while (buffer.Count == 0) // Aspetta se il buffer è vuoto
                {
                    Console.WriteLine("[Consumatore] Buffer vuoto, ASPETTO...");
                    Monitor.Wait(lockObj); // Si sospende e rilascia il lock
                }
                // Consuma
                int item = buffer.Dequeue();
                Console.WriteLine("[Consumatore] Consumato: {0} (buffer: {1})", item, buffer.Count);
                Monitor.Pulse(lockObj); // Notifica/Sveglia il produttore
            }
            Thread.Sleep(500); // Simula tempo di consumo
        }
        Console.WriteLine("[Consumatore] Completato!");
    }
}

```

Mutex

MUTEX = MUTual EXclusion (Mutua Esclusione)

È un meccanismo di sincronizzazione che permette a **UN SOLO thread** (o processo) di accedere a una risorsa alla volta. La logica di funzionamento è identica al lock, ma agisce a livello di sistema, permettendo di bloccare l'accesso ad una risorsa a processi diversi.

Sintassi

```
// Crea un mutex
Mutex mutex = new Mutex();
// Acquisisce il mutex
mutex.WaitOne();
try
{
    // SEZIONE CRITICA
    // Solo un thread alla volta
}
finally
{
    // SEMPRE rilascia!
    mutex.ReleaseMutex();
}
// Crea un mutex GLOBALE con un nome
Mutex mutex = new Mutex(false, "Global\\MioMutexUnico");

// Altri processi possono aprire lo STESSO mutex usando lo stesso nome!
```

Esempio

Il seguente codice utilizza i Mutex per simulare il comando di stampa diretta ad uno specifico dispositivo che non ha coda di stampa e che permette di ricevere solo un processo di stampa alla volta. In questo caso si usano i Mutex, perché il comando di stampa può provenire da istanze diverse del programma (processi diversi) e la risorsa sarà accessibile se nessun altro processo l'ha impegnata. Per rendere interprocesso il blocco durante l'istanziamento dell'oggetto abbiamo utilizzato **"Global\\CodaDiStampa"**.

4. Creare l'eseguibile del programma > Mutex_CodadiStampa.exe
5. Copiare l'eseguibile in C:\temp\ Mutex_CodadiStampa.exe
6. Aprire 4 console di windows
7. Avviare contemporaneamente in ogni console il programma
c:\temp\Mutex_CodadiStampa.exe {Nome del file da inviare alla stampante
VIRTUALE}

Sorgente da inserire nel progetto Mutex_CodadiStampa

```
using System;
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        Thread.CurrentThread.Name = args[0];
        Console.WriteLine("Coda di stampa\nThread {0} => Tentativo di inviare un documento alla stampante...", Thread.CurrentThread.Name);
        using (Mutex mutex = new Mutex(false, "Global\\CodaDiStampa"))
        {
            if (mutex.WaitOne())
            {

```

```

        try
        {
            Console.WriteLine("Stampante in uso. Stampando documento...");
            Thread.Sleep(8000); // Simula stampa
            //Console.ReadLine();
            Console.WriteLine("Stampa completata.");
        }
        finally
        {
            mutex.ReleaseMutex();
        }
    }
}
}
}
}

```

Semaphore/SemaphoreSlim

Un **semaforo** è un meccanismo di **sincronizzazione dei thread** che **limita il numero di thread** che possono accedere contemporaneamente a una risorsa o sezione critica.

Semaphore si appoggia al sistema operativo e ha una caratteristica unica: può essere usato per la sincronizzazione **tra processi diversi** (es. tra due file .exe in esecuzione).

Permette solo blocchi sincroni.

SemaphoreSlim è una **versione più leggera e veloce** di Semaphore, è **gestita interamente in memoria** (non usa handle di sistema), quindi è più efficiente quando serve solo la sincronizzazione tra thread nello **stesso processo e permette il blocco asincrono**.

Quando Usare Uno o l'Altro?

- Si usa **SemaphoreSlim nel 99% dei casi**. È la scelta predefinita e migliore per qualsiasi scenario di sincronizzazione all'interno delle applicazioni. Ad esempio, per limitare il numero di accessi concorrenti a una risorsa o per limitare il numero di task che possono essere eseguiti in parallelo.
- Si usa **Semaphore solo in casi rari**, cioè quando si ha bisogno di coordinare **applicazioni completamente separate** (.exe) in esecuzione sulla stessa macchina. L'esempio classico è un sistema di licenze che permette di eseguire al massimo, ad esempio, 3 istanze della tua applicazione contemporaneamente sul computer di un utente.

Componenti Chiave

1. **Costruttore**: `new Semaphore(initialCount, maximumCount).`
`// Semaforo: max 3 thread contemporanei, inizialmente 0 disponibili`
`private static Semaphore _pool = new Semaphore(0, 3);`
1. **.WaitOne()**: Acquisisce uno slot se disponibile. Se i ticket disponibili sono 0, il thread si blocca.
2. **.Release()**: Incrementa il conteggio dei ticket disponibili, liberando una risorsa.
3. **SemaphoreSlim**: Una versione più leggera (e solitamente consigliata per la programmazione asincrona `async/await`) rispetto a **Semaphore**.

```

// COSTRUTTORI
Semaphore sem = new Semaphore(initialCount, maximumCount)

SemaphoreSlim sem = new SemaphoreSlim(maxCount);
// initialCount = maxCount= 3

SemaphoreSlim sem = new SemaphoreSlim(2, 5);
// initialCount = 2 (slot disponibili ora)
// maxCount = 5 (massimo slot possibili)

// ACQUISISCE uno slot (attesa fino alla disponibilità di uno slot)
sem.WaitOne();

// ACQUISISCE con timeout
bool acquisito = sem.WaitOne(5000); // millisecondi dop o di che ritorna FALSE
bool acquisito = sem.WaitOne(TimeSpan.FromSeconds(5));

// RILASCIAMO uno slot
sem.Release();

// RILASCIAMO N slot
sem.Release(3);

// DISPOSE
sem.Dispose();
// oppure: using (Semaphore sem = new Semaphore(2,2)) { }
// servono a rilasciare le risorse allocate per il semaforo

```

Esercizi pratici con codice

Esercizio con SemaphoreSlim; simulazione del download di 10 file, ma per non sovraccaricare la rete, si limita al massimo a 3 download contemporaneamente.

```

using System;
using System.Threading;

class Program
{
    static Semaphore sem = new Semaphore(2, 2);

    static void Main(string[] args)
    {
        Console.WriteLine("=== SEMAPHORE BASE ===");
        Console.WriteLine("Max 2 thread contemporanei\n");
        Thread[] threads = new Thread[5];
        for (int i = 0; i < 5; i++)
        {
            int id = i + 1;
            threads[i] = new Thread(() => Lavoro(id));
            threads[i].Start();
        }
        foreach (Thread t in threads)
        {
            t.Join();
        }
        Console.WriteLine("\nTutti i thread completati!");
        Console.ReadLine();
    }

    static void Lavoro(int id)
    {
        Console.WriteLine("[Thread {0}] In attesa...", id);
        sem.WaitOne(); // Acquisisce uno slot
        try
        {
            Console.WriteLine("[Thread {0}] ENTRA (slot occupato)", id);
            Thread.Sleep(2000); // Simula lavoro
            Console.WriteLine("[Thread {0}] ESCE (slot liberato)", id);
        }
        finally
        {
            sem.Release();
        }
    }
}

```

```

    }
    finally
    {
        sem.Release(); // Rilascia lo slot
    }
}
}

```

Esercizio: limitazione di istanze del programma. Il programma gestisce l'avvio sulla macchina locale di **massimo 3 istanze**, altrimenti, avvisa l'utente e si chiude.

```

using System;
using System.Threading;

class Program
{
    // Definiamo il numero massimo di istanze e un nome univoco per il semaforo.
    private const int MAX_ISTANZE = 3;
    private const string NOME_SEMAFORO = "MyAppInstanceLimiterSemaphore";

    static void Main(string[] args)
    {
        // Tenta di creare un nuovo semaforo di sistema o di aprire uno esistente.
        // 'createdNew' sarà 'true' solo per la prima istanza che lo crea.
        bool createdNew;
        Semaphore semaphore = new Semaphore(
            MAX_ISTANZE,      // Conteggio iniziale (tutti i posti sono liberi all'inizio)
            MAX_ISTANZE,      // Conteggio massimo
            NOME_SEMAFORO,    // Nome a livello di sistema
            out createdNew);  // Parametro di output
        Console.WriteLine("Avvio dell'applicazione...");
        // Tenta di acquisire uno slot dal semaforo senza aspettare.
        // WaitOne(0) prova ad acquisire il semaforo con un timeout di 0 millisecondi.
        if (semaphore.WaitOne(0))
        {
            // -- SLOT ACQUISITO CON SUCCESSO --
            Console.WriteLine("Slot acquisito. L'applicazione può essere eseguita");
            Console.WriteLine("Premi Invio per rilasciare lo slot e chiudere l'applicazione.");
            // Tieni l'applicazione in esecuzione finché l'utente non preme Invio
            Console.ReadLine();

            // Rilascia lo slot del semaforo prima di uscire.
            // Questo permette a un'altra istanza in attesa di avviarsi.
            semaphore.Release();
            Console.WriteLine("Slot rilasciato.");
        }
        else
        {
            // -- NESSUNO SLOT DISPONIBILE --
            Console.WriteLine("Errore: troppe istanze dell'applicazione già in esecuzione.");
            Console.WriteLine("Chiusura in corso...");
            Thread.Sleep(3000); // Pausa per permettere all'utente di leggere il messaggio
        }
        // Chiude l'handle del semaforo
        semaphore.Close();
    }
}

```

Task Parallel Library (TPL) e Task

La **Task Parallel Library (TPL)** è una libreria di .NET progettata per semplificare drasticamente il processo di aggiunta di parallelismo e concorrenza nelle applicazioni C#.

Elementi del Task Parallel Library:

1. **Il ThreadPool** è un insieme di Thread pre-creati e pronti all'uso, gestiti dal runtime di .NET.
2. **Il Task**: rappresenta un'**operazione sincrona o asincrona** che può essere eseguita. I Task vengono eseguiti in modo efficiente dal ThreadPool ottimizzando l'uso delle risorse.
3. **Il TaskScheduler**: È l'algoritmo che decide quale **Task** assegnare a quale **Thread** del ThreadPool.

Componente	Ruolo	Analogia
Task	Il Lavoro	L'ordine di lavoro da completare.
ThreadPool	Gli Operai	La squadra di thread che eseguono il lavoro.
TaskScheduler	Il Manager	Colui che decide quale operaio esegue quale lavoro.

task

I **Task** sono astrazioni dei thread, la classe base dei task appartiene allo spazio dei nomi **System.Threading.Tasks.Task**.

Le Caratteristiche Salienti dei Task

1. **Astrazione di Alto Livello ("Cosa", non "Come")** Mentre un Thread rappresenta un lavoratore fisico, un Task rappresenta un **lavoro da svolgere** (un'operazione asincrona).
 - Non ci si deve preoccupare di creare o distruggere il thread.
 - Basta definire il Task (il compito), e il Task Scheduler di .NET decide quale thread usare e quando.
2. **Uso Intelligente delle Risorse (ThreadPool)** I Task utilizzano automaticamente il **ThreadPool** (pool di thread).
 - Invece di creare un thread nuovo per ogni operazione (costoso in termini di memoria e CPU), i Task "riciclano" i thread esistenti.
 - Appena un Task finisce, il thread che lo eseguiva viene pulito e assegnato a un nuovo Task in coda.
 - Quando un Task asincrono è in attesa di un evento esterno (come una risposta dal database o un file dal disco), esso **rilascia il Thread**. In quel momento, il Task esiste solo come "promessa" in memoria, ma **non occupa nessun processore fisico**. Il Thread viene liberato e torna nel Pool per eseguire altri lavori nel frattempo.
3. **Valore di Ritorno (Task<TResult>)** A differenza dei Thread che sono void, i Task sono progettati per poter restituire un risultato.

- La classe generica Task<T> espone la proprietà .Result.
 - Questo elimina la necessità di usare variabili esterne condivise o array di appoggio per recuperare i dati calcolati.
4. **Gestione Unificata delle Eccezioni** Se un Thread classico va in crash (eccezione non gestita), spesso fa terminare l'intero processo.
 - Nei Task, le eccezioni vengono "catturate" e inglobate in un oggetto speciale chiamato AggregateException.
 - L'eccezione viene rilanciata solo quando il codice principale chiede il risultato del Task (o usa await), permettendo di gestirla con un normale try-catch nel thread principale.
 5. **Supporto alla Cancellazione (CancellationToken)** I Task offrono un meccanismo standard per essere interrotti in modo sicuro: il **CancellationToken**.
 - Invece di "uccidere" brutalmente un thread (operazione pericolosa), si passa al Task un "gettone".
 - Il Task controlla periodicamente il gettone: se è stato annullato, il Task smette di lavorare e pulisce le sue risorse ordinatamente.
 6. **Continuazione e Concatenazione (ContinueWith / await)** I Task possono essere incatenati. È possibile definire cosa fare *dopo* che un Task è finito.
 - *Esempio*: "Scarica il file (Task A) -> E POI -> Elabora i dati (Task B) -> E POI -> Salva su disco (Task C)".
 - Con async/await, questo flusso diventa leggibile come se fosse codice sequenziale.
 7. **Stato dell'Esecuzione** Un oggetto Task possiede una proprietà .Status che permette di sapere in ogni istante cosa sta succedendo.
Gli stati principali sono:
 - *Created* (Creato, non ancora avviato)
 - *Running* (In esecuzione)
 - *RanToCompletion* (Finito con successo)
 - *Faulted* (Finito con errore/eccezione)
 - *Canceled* (Annullato)

Il Flusso di Lavoro: Dalla Creazione all'Esecuzione

Quando si avvia un nuovo Task (ad esempio con Task.Run(...)), non lo si sta assegnando direttamente a un Thread, ma si avvia il processo di seguito descritto:

1. **Creazione del Task**: si crea il contenitore che eseguirà l'operazione.
2. **Consegna al TaskScheduler**: Il Task viene consegnato al TaskScheduler che gestisce il ThreadPool.
3. **Messa in Coda**: Il TaskScheduler non esegue subito il Task ma lo mette in una coda di attesa per l'esecuzione.
4. **Assegnazione a un Thread**: Il TaskScheduler monitora costantemente il ThreadPool. Appena un Thread si libera, il TaskScheduler prende il prossimo Task dalla coda e glielo assegna per l'esecuzione.
5. **Riciclo del Thread**: Una volta che il Thread ha completato il Task, non viene dismissed ma torna a essere disponibile nel ThreadPool, pronto per un nuovo lavoro che gli assegnerà il TaskScheduler.

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("=== Esempio sull'uso dei TASK ===\n");
        // Crea e avvia un task
        Task task = Task.Run(() =>
        {
            Console.WriteLine("Task avviato!");
            Thread.Sleep(2000); // Simula lavoro
            Console.WriteLine("Task completato!");
        });
        Console.WriteLine("Main: Continuo mentre il task lavora...");
        // Aspetta che il task finisca
        task.Wait(); // Blocca finché non finisce
        Console.WriteLine("Main: Task terminato, proseguo");
        Console.ReadLine();
    }
}

```

Sintassi, Metodi e Proprietà principali:

- `Task task = new Task([metodo][lambda])`
Crea l'oggetto task assegnando un metodo o un lambda da utilizzare come elemento da eseguire
- `Task task = Task.Run([metodo][lambda])`
Crea l'oggetto task assegnando un metodo o un lambda da utilizzare come elemento da eseguire e lo avvia
- `Task<T> task = new Task([metodo][lambda])`
`Task<T> task = new Task.Run([metodo][lambda])`
Crea l'oggetto task, lo esegue con Task.Run e restituisce un valore
- `task.Wait([int millisecondsTimeout]);`
attende la fine del task, opzionalmente si possono specificare il numero di millisecondi di attesa.
Se il task termina prima: il metodo ritorna **true** ed il thread chiamante riprende l'esecuzione normalmente
Altrimenti: il metodo ritorna **false** ed il thread chiamante smette di aspettare e continua la sua esecuzione. **Il task continua comunque a lavorare in background**
- `task.Result;`
blocca l'esecuzione per ottenere il valore. Si applica solo a task che sono creati in modo da restituire un valore.
- `await task;`
aspetta il termine di un task asincrono senza bloccare il thread chiamante restituisce il valore solo per i task asincronici che restituiscono un valore
- `Task.WaitAll(Task[] tasks)`
`Task.WaitAll(Task[] tasks, int millisecondsTimeout);`
Aspetta che TUTTI i task passati come parametro finiscano. Nel secondo caso si aspetta al massimo millisecondsTimeout
- `Task.WaitAny();`

- `Task.WaitAny();`
come `Task.WaitAll` ma funziona al contrario: **sblocca il thread chiamante appena uno dei task termina**, non tutti.
- `Task.WhenAll();`
- `Task.WhenAny();`

Operazione	Sintassi
Creare	<code>Task.Run(() => {...})</code>
Creare con ritorno	<code>Task<int>.Run(() => 42)</code>
Aspettare	<code>task.Wait()</code>
Ottenere risultato	<code>task.Result</code>
Aspettare tutti	<code>Task.WaitAll(t1, t2, t3)</code>
Aspettare primo	<code>Task.WaitAny(t1, t2, t3)</code>
Async aspettare	<code>await task</code>
Async tutti	<code>await Task.WhenAll(tasks)</code>
Delay	<code>await Task.Delay(1000)</code>
Continuazione	<code>task.ContinueWith(t => {...})</code>

```
using System;
using System.Threading;
using System.Threading.Tasks;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Inizio programma\n");
        // Avvio il metodo asincrono e attendo la sua conclusione
        EseguiOperazioniAsync().Wait();
        Console.WriteLine("\nTutte le operazioni completate!");
        Console.ReadKey();
    }
    static async Task EseguiOperazioniAsync()
    {
        // Avvio due operazioni in parallelo
        Task t1 = OperazioneAsync("Operazione 1", 3000);
        Task t2 = OperazioneAsync("Operazione 2", 2000);
        // Attendo il completamento di entrambe
        await Task.WhenAll(t1, t2);
    }
    static async Task OperazioneAsync(string nome, int durata)
    {
        Console.WriteLine("{0} avviata...", nome);
        await Task.Delay(durata); // Simula un'operazione lunga (es. I/O)
        Console.WriteLine("{0} completata dopo {1} secondi.", nome, durata / 1000);
    }
}
```

Il Meccanismo Asincrono (async / await)

La programmazione asincrona serve a **non bloccare** il programma mentre attende che qualcun altro (il disco, il database, la rete) finisca un lavoro lento.

Immagina il tuo programma come un **Cameriere al ristorante**:

- **Sincrono (Bloccante)**: Il cameriere prende l'ordine, va in cucina e *aspetta immobile davanti al cuoco* finché il piatto non è pronto. Nel frattempo, nessun altro cliente può ordinare. (Pessimo servizio).
- **Asincrono (Non bloccante)**: Il cameriere porta l'ordine in cucina, *torna subito in sala* a servire altri clienti, e *viene richiamato* (callback) solo quando il piatto è pronto per portarlo al tavolo. (Servizio eccellente).

Come funziona in C#

Il meccanismo si basa su tre parole chiave:

1. **Task**: È la "comanda" del ristorante. Rappresenta una **promessa**: *"Ti darò un risultato in futuro, intanto tieni questa ricevuta"*.
2. **async**: È un'etichetta che metti sul metodo per dire al compilatore: *"Attenzione, qui dentro userò i superpoteri dell'attesa non bloccante"*.
3. **await**: È il punto magico. Quando il codice incontra `await operazioneLenta()`:
 - Il metodo **si mette in pausa**.
 - Il Thread che lo stava eseguendo viene **liberato immediatamente** e può tornare a fare altro (es. mantenere reattiva l'interfaccia grafica o gestire altre richieste web).
 - Quando l'operazione lenta finisce, il sistema recupera un thread libero e riprende l'esecuzione dalla riga successiva.

Quando Utilizzarlo?

Bisogna usare **async/await** quando il collo di bottiglia **NON è la CPU**, ma una risorsa esterna. Questo scenario si chiama **I/O Bound** (Input/Output Bound).

Usa ASYNC quando:

1. **Accesso al Disco**: Leggere/scrivere file di grandi dimensioni.
2. **Database**: Eseguire query SQL complesse.
3. **Rete / Web**: Scaricare file da internet, chiamare API REST, inviare email.
4. **UI Responsiveness**: Nelle app grafiche (WPF, Windows Forms), per evitare che l'interfaccia si "conghi" e diventi bianca mentre carichi i dati.

NON usare (solo) ASYNC quando:

- Devi fare calcoli matematici pesanti (CPU Bound). In quel caso, l'asincronia da sola non basta: serve il **Parallelismo** (Parallel.For o Task.Run) per usare più core.

Differenza tra Parallelismo e Asincronia:

- **Parallelismo:** Usa più operai (Core) per fare un lavoro pesante più in fretta (es. `Parallel.For` sui numeri primi).
- **Asincronia:** Usa lo stesso operaio (Thread) in modo più intelligente, permettendogli di fare altro mentre aspetta che una macchina esterna finisca il lavoro.

Esempi

Avviare ed attendere Task Scrivere codice che avvia due task paralleli che effettuano due lavori diversi (per esempio, generano diversi numeri o calcolano valori).

```
Task t1 = Task.Run(() => {
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("Task1: " + i);
        Thread.Sleep(100);
    }
});

Task t2 = Task.Run(() => {
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("Task2: " + i);
        Thread.Sleep(80);
    }
});

Task.WaitAll(t1, t2);
Console.WriteLine("Entrambi i Task hanno terminato.");
```

Download asincrono

```
public async Task ScaricaDatiAsync()
{
    HttpClient client = new HttpClient();
    string dati = await client.GetStringAsync("https://www.example.com");
    Console.WriteLine(dati);
}
```

`ScaricaDatiAsync` è un metodo asincrono che:

- utilizza `HttpClient`, una classe che supporta nativamente l'asincronicità
- chiama `GetStringAsync()`, che effettua una richiesta HTTP senza bloccare il thread
- usa `await` per sospendere il metodo finché la risposta non arriva
- riprende l'esecuzione quando il download è completato

Simulare lavoro asincrono

```
public async Task<int> CalcoloAsync ()
{
    int risultato = await Task.Run(() => {
        Thread.Sleep(9000);
        return 42;
    });
    return risultato;
}
```

`CalcoloAsync` è un metodo asincrono che:

- utilizza un task asincrono che si sospende per 9 secondi, nei quali il thread che eseguiva il task viene liberato

- quando il tempo di attesa è terminato restituisce il valore 42 e termina.

Esercizio: Scrivere codice che avvia più task e gestisce la terminazione non appena il primo di essi completa (WaitAny) o tutti completano (WaitAll).

Esercizio: Differenza tra Task.Wait e await

Scrivere un programma C# che esegue un'operazione lenta (ad esempio una pausa di 5 secondi) all'interno di un Task. Il programma deve mostrare cosa accade quando:

1. si attende il completamento del Task usando un Task sincrono
2. si attende il completamento del Task usando un Task asincrono

L'obiettivo è osservare come:

- il Task sincrono **blocca il thread chiamante**, impedendo l'esecuzione di altre istruzioni
- Il Task asincrono **non blocca il thread**, permettendo al programma di continuare a rispondere mentre il Task è in esecuzione

Cancellazione Cooperativa dei Thread/Task

In C#, "uccidere" brutalmente un thread (come si faceva un tempo con `Thread.Abort`) è una pratica sconsigliata e pericolosa, perché può lasciare i dati in uno stato incoerente. Il modo corretto è implementare la **Cancellazione Cooperativa**.

Il concetto: Il "capo" (Thread Principale) non spegne forzatamente l'operaio (Task), ma gli invia una **richiesta gentile** di smettere. L'operaio, che è "educato" (cooperativo), controlla periodicamente questa richiesta e, se la trova, pulisce la sua postazione e smette di lavorare.

I due protagonisti (CancellationToken)

Il pattern standard in .NET separa la responsabilità in due oggetti distinti:

1. CancellationTokenSource (Il Telecomando)

- È l'oggetto che **genera** il segnale di stop.
- Rimane in mano al thread principale (il Main).
- Possiede il metodo `.Cancel()`.

2. CancellationToken (Il Ricevitore)

- È un oggetto leggero (struct) che viene **passato** ai Task operai.
- Funziona in sola lettura: il Task può guardarlo per sapere se deve fermarsi, ma non può usarlo per fermare altri.
- Possiede la proprietà `.IsCancellationRequested`.

Esempio Pratico

Questo programma avvia tre task che lavorano in parallelo. Dopo 3 secondi, il thread principale invia una richiesta di cancellazione a tutti e attende la loro terminazione pulita.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Avvio dei thread operai...");
        // Crea il "telecomando" per inviare la richiesta di stop.
        var cts = new CancellationTokensource();
        // Ottieni il "sensore" da passare ai thread.
        CancellationToken token = cts.Token;

        var tasks = new List<Task>();
        for (int i = 1; i <= 3; i++)
        {
            int taskId = i;
            tasks.Add(Task.Run(() => LavoroDelThread(taskId, token))); // 3. Passa il 'token' a ogni task.
        }
        Console.WriteLine("Tutti i thread sono partiti. Attendo 3 secondi prima di inviare lo stop.\n");
        Thread.Sleep(3000);
        Console.WriteLine("INVIO DELLA RICHIESTA DI CANCELLAZIONE A TUTTI I THREAD!\n");
        // Invia il segnale di "stop" a tutti i thread che hanno il token.
        cts.Cancel();
    }
}
```

```

        try
        {
            // Aspetta che tutti i task terminino.
            // Si usa un try-catch perché se i task lanciano OperationCanceledException,
            // WhenAll rilancerà un'eccezione aggregata.
            Task.WhenAll(tasks.ToArray()).Wait();
        }
        catch (AggregateException ex)
        {
            // Filtra e gestisce le eccezioni di cancellazione, che sono previste
            ex.Handle(e => e is OperationCanceledException);
            Console.WriteLine("Tutti i thread hanno confermato la cancellazione.");
        }
        Console.WriteLine("\nProgramma principale terminato.");
        Console.ReadKey();
    }
    // Il metodo eseguito da ogni thread, da notare il token per ricevere la cancellazione.
    static void LavoroDelThread(int id, CancellationToken token)
    {
        Console.WriteLine("Thread {0} avviato.", id);
        int contatore = 0;
        while (true)
        {
            // CONTROLLO COOPERATIVO: il thread controlla se è arrivata la richiesta.
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("--> Thread {0}: Richiesta di cancellazione ricevuta. Eseguo pulizia e
termino.", id);
                // Qui si potrebbe inserire codice di pulizia (es. chiudere file).
                return; // Esce dal metodo e termina il task.
            }
            // Se non c'è richiesta, il thread continua il suo lavoro.
            Console.WriteLine("Thread {0}: Lavoro in corso... ({1})", id, contatore++);
            Thread.Sleep(500); // Simula lavoro
        }
    }
}

```

In Sintesi, i Passaggi Sono:

1. **Istanziare:** Il Main crea un `CancellationTokenSource` (`cts`).
2. **Distribuire:** Il Main prende il `cts.Token` (`token`) e lo passa come parametro a tutti i Task che vuole controllare.
3. **Controllare:** Dentro il ciclo `while` del Task, si inserisce il controllo che se attivato dal Main permette di attivare la cancellazione del Task.
4. **Cancellare:** Quando serve, il Main chiama `cts.Cancel()`, che attiva il processo di cancellazione
5. **Attesa e Sincronizzazione:** Dopo aver premuto il pulsante di stop (`cts.Cancel()`), il programma principale **non deve terminare immediatamente**. I Task hanno bisogno di tempo fisico (millisecondi o secondi) per accorgersi della richiesta, chiudere le risorse (file, connessioni) e uscire dal ciclo.
 - `Task.WhenAll(...)`: È il comando che dice al Main: *"Non chiudere l'applicazione finché l'ultimo dei Task non ha finito le sue operazioni di pulizia"*. Senza questo, il programma si chiuderebbe mentre i thread stanno ancora lavorando.
 - Gestione eccezioni `AggregateException`: Poiché si stanno gestendo più Task insieme, se uno o più di essi dovessero terminare con un errore o con una cancellazione tramite eccezione, .NET raggruppa tutti gli errori in un unico contenitore chiamato `AggregateException`. Il blocco `try-catch` serve a catturare questo contenitore e gestire la chiusura in modo ordinato, evitando crash improvvisi.

Tabella riassuntiva Thread, Task e Parallel.For

Caratteristica	Thread (System.Threading)	Task (System.Threading.Tasks)	Parallel.For (System.Threading.Tasks)
Tipo	Classe (Oggetto "fisico" OS)	Classe (Astrazione "Logica")	Metodo Statico (Ciclo parallelo)
Parametri (Input)	Delegato ThreadStart (void) o ParameterizedThreadStart (accetta object).	Delegato Action (nessun ritorno) o Func<T> (con ritorno).	Indice Inizio, Indice Fine, Delegato Action o Lambda.
Valore Restituito (Output)	Nessuno (void) . Il metodo eseguito non può restituire valori direttamente.	Un oggetto Task<T> . Questo oggetto conterrà il risultato futuro.	Struttura ParallelLoopResult . Indica solo lo stato del ciclo (es. completato), non il risultato del calcolo.
Recupero del Risultato	Manuale e Indiretto . Bisogna scrivere su variabili condivise esterne (es. array).	Diretto . Si legge la proprietà. Result.	Manuale e Indiretto . Si scrive su variabili condivise (richiede lock o Interlocked).
Avvio ed Esecuzione	Esplicito con .Start().	Implicito (parte spesso alla creazione con Task.Run).	Immediato e Bloccante . Il codice non prosegue finché il ciclo non finisce.
Gestione Risorse	Costosa . Crea uno stack di memoria dedicato (1MB) per ogni istanza.	Efficiente . Usa il <i>ThreadPool</i> per riciclare i thread esistenti.	Ottimizzata . Decide dinamicamente quanti thread usare in base alla CPU.