Async & Await

Programmazione Asincrona in C# — async e await

1. Perché servono

Quando un'applicazione esegue operazioni lente (download, accesso al disco, query al database), il thread principale rischia di **bloccarsi**.

Con `async` e `await`, possiamo **rilasciare il thread** durante l'attesa, rendendo il programma reattivo e non bloccante.

---

2. Parole chiave fondamentali

- **`async`** → indica che un metodo è asincrono e può usare `await`.
- **`await`** → sospende l'esecuzione del metodo fino al completamento del `Task` atteso, senza bloccare il thread.

---

3. Esempio base

```
async Task ScaricaDati()
{
    await Task.Delay(1000); // simula un download
    Console.WriteLine("Download completato");
}
```

`Task.Delay(1000)` crea un'attesa asincrona di 1 secondo.

L'esecuzione del metodo viene "messa in pausa", ma **il thread è libero** di fare altro nel frattempo.

---

4. Task e Thread

Un `Task` rappresenta un'operazione asincrona.

È simile a un thread, ma **più leggero e gestito dal runtime**.

```
Task.Run(() => Console.WriteLine("In esecuzione su thread separato"));
```

> Task.Run avvia un nuovo task in un thread del thread pool.

---

5. Metodi async con ritorno

I metodi `async` possono restituire:

- `Task` → se non serve restituire nulla
- `Task<T>` → se vogliamo restituire un valore

```
async Task<int> SommaAsync(int a, int b)
{
   await Task.Delay(100); // simula calcolo
   return a + b;
}

// Uso
int risultato = await SommaAsync(2, 3);
```

---

6. Gestione delle eccezioni asincrone

Le eccezioni nei metodi `async` vengono incapsulate nel `Task`.

Si catturano con un normale `try-catch` attorno a `await`:

```
try
{
   await ScaricaDati();
}
catch (Exception e)
{
   Console.WriteLine($"Errore: {e.Message}");
}
```

---

7. Esempio pratico completo

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
```

```
class Program
{
    static async Task Main()
    {
        Console.WriteLine("Download in corso...");
        string contenuto = await ScaricaPaginaAsync("https://example.com");
        Console.WriteLine(contenuto.Substring(0, 100)); // mostra le prime rig
he
    }

    static async Task<string> ScaricaPaginaAsync(string url)
    {
        using var client = new HttpClient();
        string contenuto = await client.GetStringAsync(url);
        return contenuto;
    }
}
```

Qui `await` sospende l'esecuzione finché `GetStringAsync` non termina, **senza bloccare** la UI o il thread principale.

---

8. Differenza fra Sincrono e Asincrono

Tipo	Comportamento	Thread bloccato?
Sincrono	Attende finché l'operazione non termina	Sì
Asincrono (`await`)	Libera il thread durante l'attesa	No

---

9. Buone pratiche

- Non usare `async void` (tranne per gestori di eventi).
- Evita `Task.Wait()` o `.Result`, che bloccano comunque il thread.
- Usa `ConfigureAwait(false)` nelle librerie per evitare deadlock.
- Componi metodi asincroni con `await` anziché concatenare `ContinueWith`.

1️⃣ ESEMPIO BASE — Il blocco del thread

Sincrono (bloccante)

```
void Scarica()
{
    Thread.Sleep(3000); // blocca il thread per 3 secondi
    Console.WriteLine("Download completato!");
}

void Main()
{
    Console.WriteLine("Inizio");
    Scarica();
    Console.WriteLine("Fine"); // viene eseguito solo dopo 3 secondi
}
```

Output:

```
Inizio
(3 secondi di blocco)
Download completato!
Fine
```

> Il thread principale è fermo durante Thread.Sleep.

2️⃣ ESEMPIO CON ASYNC/AWAIT — Non blocca il thread

```
async Task ScaricaAsync()
{
    await Task.Delay(3000); // attesa "non bloccante"
    Console.WriteLine("Download completato!");
}

async Task Main()
{
    Console.WriteLine("Inizio");
    await ScaricaAsync();
```

```
    Console.WriteLine("Fine"); // viene eseguito solo dopo che l'attesa è finit
a
}
```

Output:

```
Inizio
Download completato!
Fine
```

Differenza:

`Task.Delay` libera il thread, quindi l'app non "si congela".

3️⃣ ESEMPIO CON PIÙ TASK — Esecuzione in parallelo

```
async Task OperazioneAsync(string nome, int secondi)
{
    Console.WriteLine($"{nome} iniziata...");
    await Task.Delay(secondi * 1000);
    Console.WriteLine($"{nome} completata dopo {secondi}s");
}

async Task Main()
{
    Task t1 = OperazioneAsync("Task 1", 2);
    Task t2 = OperazioneAsync("Task 2", 3);

    await Task.WhenAll(t1, t2); // attende entrambe le operazioni
    Console.WriteLine("Tutti i task completati");
}
```

Output (i tempi si sovrappongono):

```
Task 1 iniziata...
Task 2 iniziata...
Task 1 completata dopo 2s
```

```
Task 2 completata dopo 3s
Tutti i task completati
```

> WhenAll serve ad attendere più operazioni in parallelo.

4️⃣ ESEMPIO CON VALORI DI RITORNO

```
async Task<int> CalcolaDoppioAsync(int n)
{
    await Task.Delay(1000); // simula calcolo lento
    return n * 2;
}

async Task Main()
{
    int risultato = await CalcolaDoppioAsync(5);
    Console.WriteLine($"Risultato: {risultato}");
}
```

Output:

```
Risultato: 10
```

> Il metodo async può restituire un valore con Task<T>.

5️⃣ ESEMPIO CON `Task.Run` — Lavoro su thread separato

```
async Task EseguiLavoroPesante()
{
    Console.WriteLine("Lavoro iniziato...");
    await Task.Run(() ⇒ // delega il lavoro a un thread del pool
    {
        Thread.Sleep(2000); // simula calcolo intensivo
        Console.WriteLine("Lavoro completato nel thread in background");
    });
```

```
    Console.WriteLine("Tornato al thread principale");
}
```

> Task.Run serve per spostare lavori CPU-bound (pesanti) fuori dal thread principale.

6️⃣ ESEMPIO CON ERRORE ED ECCEZIONE

```
async Task ScaricaAsync()
{
    await Task.Delay(500);
    throw new Exception("Errore di rete");
}

async Task Main()
{
    try
    {
        await ScaricaAsync();
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Catturata eccezione: {ex.Message}");
    }
}
```

Output:

```
Catturata eccezione: Errore di rete
```

> Le eccezioni propagate da un metodo async vengono catturate normalmente con try-catch.

7️⃣ ESEMPIO — `async void` (solo per eventi)

```
async void OnClick(object sender, EventArgs e)
{
    await Task.Delay(1000);
    Console.WriteLine("Bottone cliccato");
}
```

> async void non restituisce un Task, quindi non può essere “atteso”.
>
> Usalo **solo per gestori di eventi GUI**.
>
> Mai in codice di libreria o business logic.

8️⃣ ESEMPIO REALE — Download da Internet

```
using System.Net.Http;

async Task ScaricaPagina(string url)
{
    using HttpClient client = new HttpClient();
    Console.WriteLine("Download in corso...");
    string contenuto = await client.GetStringAsync(url);
    Console.WriteLine($"Contenuto ricevuto: {contenuto.Length} caratteri");
}

await ScaricaPagina("https://example.com");
```

> HttpClient.GetStringAsync è già asincrono, quindi non blocca l’app mentre scarica.

9️⃣ ESEMPIO DI PARALLELISMO CONTROLLATO

```
async Task<int> CalcoloAsync(int n)
{
    await Task.Delay(1000);
    return n * n;
}
```

```
async Task Main()
{
   var tasks = Enumerable.Range(1, 5)
                     .Select(CalcoloAsync)
                     .ToList();

   int[] risultati = await Task.WhenAll(tasks);
   Console.WriteLine($"Somma totale: {risultati.Sum()}");
}
```

> Tutti i calcoli partono insieme e vengono attesi contemporaneamente.

1️⃣0️⃣ ESEMPIO — Esecuzione sequenziale forzata

```
for (int i = 1; i <= 3; i++)
{
   await Task.Delay(1000);
   Console.WriteLine($"Passo {i}");
}
```

> L’await dentro il for garantisce che ogni iterazione avvenga in sequenza, non in parallelo.

Slide 1 — Il problema del codice sincrono

Scenario:

Un’app legge un file e poi aggiorna la UI.

```
LeggiFile();
AggiornaUI();
```

Timeline:

```
[Thread Principale]
```

```
|----LeggiFile (5s)----|----AggiornaUI----|
```

👉 Durante `LeggiFile`, tutto è **bloccato**: l'utente non può fare nulla.

Slide 2 — Introduzione di async/await

```
await LeggiFileAsync();
AggiornaUI();
```

Timeline:

```
[Thread Principale]
|--avvia LeggiFileAsync⟶ (Thread I/O)
        ↓
    [thread libero] → può aggiornare UI
        ↓
    ⟵——riprende quando finito---
```

✅ `await` *sospende* il metodo, ma **non blocca** il thread principale.

Quando il file è letto, l'esecuzione riprende.

Slide 3 — Cosa fa il compilatore

Un metodo marcato `async` viene *riscritto* dal compilatore in una **macchina a stati**.

```
async Task EsempioAsync()
{
   await Task.Delay(1000);
   Console.WriteLine("Ripreso!");
}
```

Diventa (semplificando):

```
Task.Delay(1000)
   .ContinueWith(_ ⇒ Console.WriteLine("Ripreso!"));
```

Quindi `await` non blocca: registra “cosa fare dopo” e ritorna immediatamente un `Task`.

Slide 4 — Esecuzione parallela di più Task

```
var t1 = OperazioneAsync("A");
var t2 = OperazioneAsync("B");
await Task.WhenAll(t1, t2);
```

Timeline:

```
[Main]
|--A start--|--B start--|
  ↓          ↓
 [Thread A]   [Thread B]
   ↘          ↙
    -——→ completano insieme -——→
```

💡 `Task.WhenAll` attende che *tutti* i Task abbiano finito,

non in ordine, ma in parallelo.

Slide 5 — Ritorno dei risultati

```
int a = await SommaAsync(2, 3);
int b = await SommaAsync(4, 5);
```

Timeline:

```
T1: SommaAsync(2,3) -——→ [attesa]
    ↳ restituisce 5
T2: SommaAsync(4,5) -——→ [attesa]
    ↳ restituisce 9
```

`await` riceve il valore del `Task<T>` come se fosse sincrono,

ma senza bloccare il thread.

Slide 6 — Riassunto visivo

Concetto	Significato
`async`	Segnala che il metodo contiene `await` e restituisce `Task`
`await`	Sospende il metodo fino al completamento del `Task`
`Task`	Oggetto che rappresenta un'operazione asincrona
Thread principale	Rimane libero durante l'attesa
`WhenAll`	Attende più `Task` in parallelo
`Task.Run`	Sposta il lavoro su un thread del pool