Lezione C#: Argomenti Intermedi e Avanzati

Agenda degli Argomenti

- **Tipi di Dato:** differenza tra value type e reference type; shallow copy vs deep copy
- **Collections:** uso di List, Dictionary, Queue, Stack, HashSet e quando utilizzarle
- **LINQ (metodi):** utilizzo di Where, Select, FirstOrDefault; best practice e valori di ritorno
- **Delegati ed Eventi:** utilizzo dei delegati, eventi e dei delegati predefiniti `Action` e `Func`; quando usarli
- **Async e Await:** programmazione asincrona in C#, vantaggi rispetto al sincrono e best practice
- **Task e Thread:** differenze tra Task e thread, come e quando usarli, best practice
- **Configurazione .NET Core:** differenze rispetto a .NET Framework; configurare i servizi (Dependency Injection)
- **Architettura a Microservizi:** definizione, vantaggi, svantaggi e best practice nei microservizi

Tipi di Dato: Value Type vs Reference Type

In C# tutto è **tipizzato** in modo forte: ogni variabile ha un tipo specifico e non può cambiare tipo senza conversione esplicita [1]. I tipi semplici più comuni includono `int`, `double`, `bool`, `string`, `char`, oltre a tipi complessi come classi e strutture. È fondamentale comprendere la differenza tra *tipi valore* e *tipi riferimento* nell'allocazione e copiatura dei dati.

I **tipi valore** (ad es. `int`, `double`, `struct`) vengono *copiati* quando assegnati a una nuova variabile, cioè ogni variabile ha la **propria copia indipendente** del dato [2]. I **tipi riferimento** (ad es. `class`, `array`, `string` [^1]) invece conservano un *riferimento* all'istanza in memoria: assegnare una variabile di tipo riferimento ad un'altra non copia i dati, ma fa puntare entrambe alla *stessa* istanza. Capire questa differenza è essenziale per evitare comportamenti inattesi nel codice [2].

[^1]: In C#, `string` è un caso particolare: è un reference type *immutabile*. Pur essendo memorizzato come riferimento, il suo contenuto non può cambiare (ogni modifica crea una nuova stringa).

Esempio: Tipi Valore vs Riferimento

```
int a = 5;
int b = a;
b = 10;
Console.WriteLine(a);  // Output: 5 (b è una copia di a, modificare b non
tocca a)

int[] arr1 = { 1, 2 };
int[] arr2 = arr1;
arr2[0] = 99;
Console.WriteLine(arr1[0]);  // Output: 99 (arr2 riferisce lo stesso array di
arr1)
```

Nell'esempio sopra: `a` e `b` sono tipi valore (`int`): inizialmente `a=5`, poi `b` viene copiato da `a` e impostato a 10; `a` rimane 5 perché la modifica a `b` non influisce su `a`. Invece `arr1` e `arr2` sono array di int (tipo riferimento); assegnando `arr1` a `arr2`, entrambe le variabili puntano allo stesso array. Modificando `arr2[0]`, vediamo l'effetto anche leggendo `arr1[0]` (risulta 99) perché esiste un'unica struttura dati condivisa.

Shallow Copy vs Deep Copy

Quando si lavora con tipi riferimento, è importante distinguere tra **shallow copy** e **deep copy**. Una *copia superficiale* (*shallow copy*) copia solo il riferimento all'oggetto, lasciando che due variabili puntino alla **stessa** istanza (come visto sopra). Una *copia profonda* (*deep copy*) invece crea una **nuova** istanza dell'oggetto copiando il contenuto del primo oggetto, in modo che le due variabili puntino a oggetti distinti in memoria.

In pratica, per ottenere una deep copy bisogna creare manualmente un nuovo oggetto e copiarne i valori interni. Molti tipi in .NET offrono metodi per clonare dati (ad es. il metodo `MemberwiseClone()` in `Object` o copy constructor personalizzati). Una shallow copy è spesso insufficiente se l'obiettivo è modificare la copia senza impattare l'originale; in questi casi è necessaria la deep copy per duplicare anche i dati referenziati [3].

Esempio: Copia Superficiale vs Copia Profonda

```
class Punto { public int X; public int Y; }

Punto p1 = new Punto { X = 1, Y = 2 };
Punto p2 = p1;                 // Shallow copy: copia il riferimento
all'oggetto
p2.X = 5;
Console.WriteLine(p1.X);       // Output: 5 (p1 e p2 puntano allo stesso
oggetto)

Punto p3 = new Punto { X = p1.X, Y = p1.Y }; // Deep copy: nuovo oggetto con
stessi dati
p3.X = 9;
Console.WriteLine(p1.X);       // Output: 5 (p1.X rimane 5, p3 è
indipendente)
```

Nell'esempio: `p2 = p1;` realizza una copia superficiale (shallow) dell'oggetto `Punto`: sia `p1` che `p2` riferiscono lo stesso `Punto` in memoria, quindi modificare `p2.X` modifica anche `p1.X`. Invece creando `p3` come nuova istanza e copiando i valori di `p1`, otteniamo una copia profonda: `p3` è un oggetto separato e cambiando `p3.X` il `p1.X` originale rimane invariato.

Collezioni (Collections) in C

Il framework .NET fornisce diverse **collezioni** pronte all'uso nel namespace `System.Collections.Generic`. Oltre alle array e alle **List**, esistono collezioni specializzate come **Dictionary**, **Queue**, **Stack** e **HashSet** [4]. Ciascuna di queste strutture dati è ottimizzata per scenari d'uso specifici, ed è importante scegliere quella più adatta al problema da risolvere. In generale, le

collection dinamiche permettono di gestire insiemi di oggetti in modo flessibile (inserimento, rimozione, iterazione, ecc.) rispetto agli array statici.

Di seguito esamineremo le principali collezioni generiche: - **List<T>** – lista dinamica indicizzata: ideale per raccolte ordinate di elementi accessibili per indice.
- **Dictionary<TKey, TValue>** – dizionario (mappa) chiave-valore: ideale per look-up rapidi per chiave univoca.
- **Queue<T>** – coda FIFO (First-In First-Out): ideale per gestire sequenze di operazioni in ordine di arrivo.
- **Stack<T>** – pila LIFO (Last-In First-Out): ideale per scenari come gestione di undo/redo, chiamate nidificate, ecc.
- **HashSet<T>** – insieme non ordinato di elementi unici: ideale per collezioni senza duplicati e test di appartenenza rapidi.

Esempio: Utilizzo di List<T>

```
var numeri = new List<int>();   // Creazione di una lista di interi vuota
numeri.Add(42);
numeri.Add(7);
Console.WriteLine(numeri[1]);   // Output: 7 (accesso per indice, indice base
0)
Console.WriteLine(numeri.Count); // Output: 2 (numero di elementi nella
lista)
numeri.Remove(42);              // Rimuove l'elemento 42 dalla lista
Console.WriteLine(numeri.Count); // Output: 1 (ora rimane solo un elemento)
```

Commenti: La **List** è una lista dinamica: cresce o si riduce automaticamente all'aggiunta/rimozione di elementi. Nel codice, aggiungiamo due numeri con `Add`. Possiamo accedere per indice (`numeri[1]`) e rimuovere elementi specifici con `Remove`. La proprietà `Count` indica il numero di elementi correnti.

Esempio: Utilizzo di Dictionary<TKey, TValue>

```
var rubrica = new Dictionary<string, string>();
rubrica["Luca"] = "1234567890";      // Aggiunge una coppia (chiave, valore)
rubrica["Anna"] = "0987654321";      // Aggiunge un'altra coppia
Console.WriteLine(rubrica["Luca"]);  // Output: 1234567890 (ricerca per
chiave)
rubrica["Luca"] = "1111111111";      // Modifica il valore associato a "Luca"
Console.WriteLine(rubrica.Count);    // Output: 2 (due coppie chiave-valore
nella dictionary)
```

Commenti: Il **Dictionary** associa chiavi univoche a valori. Nell'esempio, usiamo stringhe come chiave (nome) e valore (numero di telefono). Possiamo aggiungere assegnando alla chiave (`rubrica["Luca"] = "..."`), ottenere valori per chiave (`rubrica["Luca"]`) e modificare valori riutilizzando la stessa chiave. La ricerca per chiave in un dictionary è molto efficiente (tempo *O(1)* ammortizzato grazie all'hashing delle chiavi).

Esempio: Utilizzo di Queue<T>

```
var coda = new Queue<string>();
coda.Enqueue("Primo");
coda.Enqueue("Secondo");
Console.WriteLine(coda.Peek());   // Output: Primo (legge l'elemento in testa
senza rimuoverlo)
Console.WriteLine(coda.Dequeue()); // Output: Primo (estrae e ritorna
l'elemento in testa)
Console.WriteLine(coda.Dequeue()); // Output: Secondo (estrae il prossimo
elemento)
Console.WriteLine(coda.Count);    // Output: 0 (la coda è ora vuota)
```

Commenti: La **Queue** è una coda FIFO. Il metodo `Enqueue` accoda un elemento in fondo. `Dequeue` preleva l'elemento in testa (il più vecchio inserito) rimuovendolo. `Peek` consente di vedere l'elemento in testa senza estrarlo. Dopo due inserimenti, `Peek()` mostra "Primo"; `Dequeue()` restituisce "Primo" e lo rimuove, quindi un altro `Dequeue()` restituisce "Secondo". La `Count` scende a 0 dopo aver svuotato la coda.

Esempio: Utilizzo di Stack<T>

```
var pila = new Stack<int>();
pila.Push(10);
pila.Push(20);
Console.WriteLine(pila.Peek());  // Output: 20 (cima della pila, ultimo
inserito)
Console.WriteLine(pila.Pop());   // Output: 20 (estrae e ritorna l'ultimo
elemento inserito)
Console.WriteLine(pila.Pop());   // Output: 10 (estrae il successivo
elemento)
Console.WriteLine(pila.Count);   // Output: 0 (pila vuota dopo aver tolto
tutti gli elementi)
```

Commenti: Lo **Stack** è una pila LIFO. `Push` inserisce un elemento in cima alla pila. `Pop` rimuove e restituisce l'elemento in cima (l'ultimo inserito). `Peek` è simile ma senza rimuovere. Nell'esempio, inseriamo 10 poi 20; `Peek()` vede 20 in cima; `Pop()` restituisce 20 e lo rimuove; un altro `Pop()` restituisce 10. Infine la pila è vuota.

Esempio: Utilizzo di HashSet<T>

```
var insieme = new HashSet<string>();
insieme.Add("rosso");
insieme.Add("blu");
insieme.Add("rosso");            // "rosso" è già presente, l'Add restituisce
false
Console.WriteLine(insieme.Count); // Output: 2 (duplicati ignorati)
Console.WriteLine(insieme.Contains("verde")); // Output: False (controllo di
appartenenza rapido)
insieme.Remove("blu");
```

```
Console.WriteLine(insieme.Contains("blu"));   // Output: False (elemento
rimosso)
```

Commenti: L'**HashSet** rappresenta un insieme di elementi unici non ordinati. Aggiungendo "rosso" due volte, il secondo inserimento viene ignorato (un HashSet non può contenere duplicati). Il metodo `Contains` verifica molto velocemente se un elemento è presente (grazie all'hash, operazione *O(1)* in media). `Remove` elimina un elemento specificato. Gli HashSet sono ideali quando serve assicurare l'unicità degli elementi e rapide operazioni di ricerca/inserimento, senza preoccuparsi dell'ordine.

Quando usare le diverse collezioni

- **List<T>:** scelta generale per insiemi ordinati di dimensione variabile, quando serve accesso casuale per indice o si aggiungono/rimuovono elementi in coda.
- **Dictionary<TKey, TValue>:** quando occorre mappare chiavi univoche a valori (ad es. una rubrica, cache). Fornisce lookup per chiave molto veloce.
- **Queue<T>:** adatta per strutture **FIFO**, ad esempio gestione di task in ordine cronologico (buffer di messaggi, job queue).
- **Stack<T>:** adatta per strutture **LIFO**, ad esempio navigazione (back/forward), algoritmi di backtracking, calcolo di espressioni (stack operand).
- **HashSet<T>:** ideale per collezioni **senza duplicati** e per controlli di appartenenza frequenti (ad es. insiemi matematici, eliminazione di doppi in una lista).
- In generale, valutare la complessità delle operazioni più frequenti: ad esempio, cercare un elemento specifico è *O(n)* in una List ma *O(1)* in un HashSet o Dictionary; viceversa, l'accesso per indice diretto è immediato in una List ma non applicabile a HashSet/Dictionary. Scegliere la collection che offre le operazioni chiave con le migliori performance per il caso d'uso.

LINQ con Metodi (Where, Select, FirstOrDefault)

LINQ (Language Integrated Query) è una componente di C# che permette di effettuare query su collezioni di dati (e altre fonti) con una sintassi concisa e dichiarativa. Dopo aver introdotto la sintassi di query comprehension, è importante padroneggiare l'uso di LINQ tramite **metodi di estensione** su raccolte, che è la forma più comune nel codice C# moderno [5]. I metodi LINQ come `Where`, `Select`, `OrderBy`, `FirstOrDefault` (e molti altri) sono disponibili su qualunque oggetto che implementi `IEnumerable<T>` o `IQueryable<T>`.

Usare LINQ con i metodi offre un codice leggibile e fluente: ad esempio, `Where` filtra gli elementi secondo un predicato logico, `Select` proietta ogni elemento in un nuovo formato, `FirstOrDefault` estrae il primo elemento che soddisfa una condizione (o un default se nessuno trovato). Questi metodi non modificano la collezione originale ma restituiscono nuove sequenze (tipicamente di tipo `IEnumerable<T>`), spesso **valutate in modo pigro** (*lazy evaluation*): l'esecuzione avviene solo quando si itera o si forza il risultato (ad esempio con `ToList()`). È buona pratica quindi evitare di enumerare più volte la stessa query LINQ per non ripetere calcoli, oppure usare ad esempio `ToList()` per materializzare il risultato se serve riutilizzarlo più volte.

Esempio: Uso di Where e Select

```
var numeri = new List<int> { 1, 2, 3, 4, 5, 6 };
var pari = numeri.Where(n => n % 2 == 0);    // Filtra solo numeri pari
var quadratiPari = pari.Select(x => x * x);  // Calcola il quadrato di
```

```
ciascun numero pari
foreach (int q in quadratiPari) {
    Console.WriteLine(q);
}
// Output: 4, 16, 36  (rispettivamente 2^2, 4^2, 6^2)
```

Spiegazione: Abbiamo una lista di interi. Con `Where(n => n % 2 == 0)` otteniamo una sequenza filtrata contenente solo i numeri pari. Su questo risultato (che è ancora *lazy*, non materializzato finché non iterato) applichiamo `Select(x => x * x)` per ottenere i quadrati di quei numeri. Il `foreach` finale forza l'esecuzione della query LINQ, stampando i quadrati dei numeri pari (4, 16, 36). Notare come la sintassi a metodi rende il codice leggibile in pipeline. In alternativa, la stessa operazione avrebbe potuto essere scritta in sintassi di query comprehension: `from n in numeri where n % 2 == 0 select n * n`, ottenendo lo stesso risultato.

Esempio: Uso di FirstOrDefault

```
var nomi = new List<string> { "Alice", "Bob", "Charlie" };
string iniziaConB = nomi.FirstOrDefault(nome => nome.StartsWith("B"));
Console.WriteLine(iniziaConB);  // Output: Bob

string iniziaConZ = nomi.FirstOrDefault(nome => nome.StartsWith("Z"));
Console.WriteLine(iniziaConZ ==
null);  // Output: True (nessun nome con 'Z', restituisce null)
```

Spiegazione: Il metodo LINQ `FirstOrDefault` trova il primo elemento che soddisfa un predicato dato. Nel primo caso cerchiamo il primo nome che inizia per "B": otteniamo `"Bob"` (se esiste almeno un elemento corrispondente, lo restituisce immediatamente). Nel secondo caso cerchiamo un nome che inizi per "Z": non essendoci risultati, `FirstOrDefault` restituisce `null` (default per il tipo `string`). Questo metodo è utile perché evita eccezioni nel caso la collezione sia vuota o nessun elemento soddisfi la condizione (a differenza di `First()` che lancerebbe un'eccezione se nulla viene trovato). Dopo aver chiamato `FirstOrDefault`, è buona norma controllare se il risultato è `null` (o il default del tipo nel caso di tipi valore) prima di usarlo.

Delegati, Eventi, Action e Func

Delegati ed **eventi** sono strumenti fondamentali della programmazione C# per implementare il *decoupling* e il passaggio di comportamenti come parametri. Un **delegato** è essentially un tipo (reference type) che rappresenta un riferimento a uno o più metodi con una certa firma [6]. In altre parole, una variabile delegato può *puntare* a un metodo e invocarlo, permettendo il paradigma *callback* e la programmazione orientata agli eventi. I delegati sono alla base degli eventi (un evento utilizza internamente un delegato) e delle *lambda expression* (che sono sintassi semplificata per definire delegati al volo).

C# fornisce già delegati generici predefiniti per semplificare il codice: `Action` per metodi che non restituiscono valore (void), `Func<TResult>` (e varianti con parametri) per metodi che restituiscono un valore [7]. Questi delegati pronti all'uso evitano di dover dichiarare tipi delegato personalizzati per i casi più comuni. Gli **eventi**, infine, sono un meccanismo basato sui delegati che consente a un oggetto (publisher) di notificare uno o più altri oggetti (subscribers) quando accade qualcosa. Tramite gli eventi

si implementa il pattern *publisher/subscriber*: chi emette l'evento non conosce chi lo riceverà, riducendo l'accoppiamento [8]. È possibile iscriversi (*subscribe*) o disiscriversi (*unsubscribe*) a eventi usando rispettivamente gli operatori `+=` e `-=` su istanze di delegati/eventi.

Esempio: Dichiarazione e Utilizzo di un Delegato

```
// Definizione di un delegato che accetta una stringa e non ritorna nulla
delegate void Messaggio(string testo);

void Saluta(string nome) {
    Console.WriteLine($"Ciao {nome}!");
}

// Uso del delegato
Messaggio handler = Saluta;   // Assegna il metodo Saluta al delegato
handler("Marco");            // Invoca Saluta tramite il delegato -> Output:
Ciao Marco!
handler?.Invoke("Anna");     // Invocazione equivalente con null-check ->
Output: Ciao Anna!
```

Nell'esempio: definiamo un delegato `Messaggio` che corrisponde a metodi che prendono una stringa e ritornano void. Abbiamo un metodo `Saluta(string)` compatibile con quel delegato. Assegniamo `Saluta` a una variabile di tipo `Messaggio` (il delegato diventa un riferimento a quel metodo). Chiamando `handler("Marco")` in realtà verrà eseguito `Saluta("Marco")`. L'invocazione alternativa `handler?.Invoke("Anna")` fa la stessa cosa verificando prima che `handler` non sia null (una buona pratica per evitare NullReferenceException nel caso di delegati/eventi non inizializzati). I delegati possono riferire anche metodi anonimi o lambda expression, non solo metodi nominati.

Esempio: Uso di Action e Func

```
Action<string> stampaSaluto = nome => Console.WriteLine($"Ciao, {nome}");
stampaSaluto("Lucia");   // Output: Ciao, Lucia

Func<int, int, int> somma = (a, b) => a + b;
int risultato = somma(3, 4);
Console.WriteLine(risultato);  // Output: 7
```

Commento: Invece di dichiarare un delegato personalizzato, possiamo usare `Action` e `Func` forniti dal framework. `Action<string>` rappresenta un delegato che accetta una stringa e non restituisce nulla; assegniamo ad esso una lambda `nome => Console.WriteLine(...)` che stampa un saluto usando il nome fornito. Chiamando `stampaSaluto("Lucia")` invoca la lambda e stampa il messaggio. `Func<int, int, int>` rappresenta un delegato che accetta due interi e restituisce un intero (qui usato per una funzione somma). Definiamo la lambda `(a, b) => a + b` e la eseguiamo ottenendo il risultato 7. Questi delegati generici riducono la necessità di definire tipi delegate manualmente e rendono il codice più conciso. In generale, **preferisci** `Action` / `Func` per casi comuni di delegati; usa delegati custom solo se le varianti predefinite non coprono la firma desiderata.

Esempio: Dichiarazione e Sottoscrizione di un Evento

```
class Pulsante {
    public event Action? Premuto;    // Evento (senza parametri) basato sul
delegato Action
    public void Premi() {
        Console.WriteLine("** click **");

Premuto?.Invoke();          // Solleva l'evento notificando tutti gli
iscritti
    }
}

var btn = new Pulsante();
// Sottoscrizione all'evento con una lambda (observer)
btn.Premuto += () => Console.WriteLine("Il pulsante è stato premuto!");
btn.Premi();
// Output:
// ** click **
// Il pulsante è stato premuto!
```

Spiegazione: Nella classe `Pulsante` definiamo un evento `Premuto` di tipo `Action` (quindi senza argomenti). Il metodo `Premi()` simula la pressione del pulsante: stampa un messaggio e invoca l'evento con `Premuto?.Invoke()`. All'esterno, creiamo un `Pulsante` e ci **iscriviamo** al suo evento `Premuto` usando `+=` con una lambda che stampa un messaggio. Quando chiamiamo `btn.Premi()`, la classe Pulsante invoca tutti i gestori iscritti a `Premuto` (in questo caso la nostra lambda), producendo l'output mostrato. L'uso di `?.Invoke` verifica che ci sia almeno un sottoscrittore prima di invocare, per evitare errori se l'evento non ha handler associati. È buona norma rimuovere le sottoscrizioni con `-=` quando non servono più (soprattutto per eventi di oggetti a lunga vita) per evitare memory leak. In generale, gli eventi sono utili per notificare cambiamenti di stato o azioni occorse senza dipendenze dirette tra le classi (ad esempio GUI, notifiche di sistema, ecc.) [8].

Async e Await in C

La programmazione **asincrona** consente di eseguire operazioni lunghe (come accesso a file, rete o attese temporali) senza bloccare il thread che le avvia. In C# questo è realizzato con le parole chiave `async` e `await`, che rendono molto semplice scrivere codice non bloccante [9]. Un metodo dichiarato `async` può usare `await` per sospendere la propria esecuzione in attesa di un risultato (ad esempio l'attesa di una risposta da un server, o un ritardo temporale), **senza bloccare** il thread corrente durante l'attesa. Questo permette, ad esempio, a un'applicazione GUI di rimanere reattiva durante operazioni I/O, o a un server di gestire altre richieste mentre una è in attesa di I/O.

Quando preferire async/await al codice sincrono? Quando abbiamo operazioni *I/O-bound* o *compute-bound* molto lunghe che altrimenti bloccherebbero il flusso principale. Ad esempio, chiamate a servizi esterni, query su database, lettura/scrittura file, o qualsiasi operazione che impiega tempo: in tali casi è conveniente usare async/await per liberare il thread chiamante finché l'operazione non è completata. I vantaggi principali sono la migliore *responsiveness* (nel caso di applicazioni desktop/web) e la capacità di scalare a più operazioni concorrenti usando meno thread (sfruttando in modo efficiente il *thread pool*). Dal punto di vista del codice, `async/await` consente di scrivere in stile sequenziale logiche che sotto il cofano sono gestite in modo asincrono, migliorando la **leggibilità** rispetto a usare direttamente callback o thread manuali.

Esempio: Metodo Asincrono con Await

```
// Metodo asincrono che simula il download di dati con un ritardo
private static async Task ScaricaDatiAsync() {
    Console.WriteLine("Inizio download...");
    await Task.Delay(2000);   // simula un'operazione I/O che dura 2 secondi
    Console.WriteLine("Download completato");
}

// Utilizzo dell'async method
await ScaricaDatiAsync();
Console.WriteLine("Procedo con altre operazioni...");
// Output:
// Inizio download...
// (attesa di 2 secondi senza bloccare il thread corrente)
// Download completato
// Procedo con altre operazioni...
```

Spiegazione: `ScaricaDatiAsync` è un metodo dichiarato `async` che ritorna un `Task`. All'interno, utilizza `await Task.Delay(2000)` per attendere in modo non bloccante 2 secondi (simulando ad esempio un download). Quando viene chiamato con `await ScaricaDatiAsync()`, il controllo torna immediatamente al chiamante fino al completamento del `Task.Delay`. Durante l'attesa di 2 secondi, il thread principale **non è bloccato** e potrebbe svolgere altro lavoro (nel nostro esempio stampa il messaggio finale dopo il download). Questo mostra come `async/await` permetta di scrivere codice apparentemente sequenziale che in realtà libera il thread durante le attese I/O. Da notare che per chiamare un metodo `async` è necessario essere in un contesto asincrono (ad esempio dentro un altro metodo `async` oppure in un entry point di applicazione console con `.GetAwaiter().GetResult()` per le demo).

Best Practice per Async/Await

- **Evita di bloccare manualmente**: non chiamare metodi asincroni usando `.Wait()` o `.Result`, poiché rischi deadlock o blocchi del thread. Usa sempre `await` per ottenere il risultato.
- **Usa async** fino in fondo****: se una funzione chiama operazioni asincrone, dichiarala `async` e usa `await` internamente invece di mescolare codice sincrono e asincrono. Idealmente l'asincronicità si propaga verso l'alto fino ai livelli più esterni.
- **Gestione degli errori**: racchiudi le `await` in blocchi `try/catch` se vuoi gestire eccezioni da operazioni asincrone. Le eccezioni in metodi `async` propagano nel `Task` e vengono rilanciate al punto di await [10].
- **Evita async void**: a meno che tu stia implementando un event handler in UI, i metodi `async` dovrebbero restituire `Task` / `Task<T>` così che il chiamante possa attenderli e catturare eventuali eccezioni. `async void` è difficile da gestire in caso di errori.
- **Sincronizzazione del contesto**: per librerie o operazioni CPU-bound, considera `ConfigureAwait(false)` per evitare di catturare il contesto di sincronizzazione, migliorando le performance in applicazioni console o library (non necessario in ASP.NET Core).
- **Operazioni CPU-bound**: se devi eseguire calcoli intensivi senza bloccare il UI thread, puoi usare `Task.Run` per spostarli su un thread del thread pool e comunque `await` sul risultato (vedi sezione Task/Thread).

Task e Thread in C

C# offre due astrazioni correlate per il lavoro concorrente: i **thread** e i **task**. Un **thread** rappresenta un flusso di esecuzione a basso livello gestito dal sistema operativo: creare un nuovo thread significa avviare in parallelo una parte di codice su una nuova schedulazione CPU. Un **Task**, introdotto con la Task Parallel Library (TPL), è un'astrazione di più alto livello che rappresenta un'operazione asincrona o in parallelo che verrà eseguita, tipicamente, su thread provenienti da un **thread pool** gestito dal runtime [11]. In pratica, usare un Task ti permette di delegare al runtime la gestione efficiente dei thread, riducendo la necessità di crearne e gestirne manualmente.

Differenze chiave: creare e gestire direttamente i `Thread` offre più controllo (priorità, affinità CPU, etc.) ma è raramente necessario nelle applicazioni di alto livello; inoltre creare troppi thread può degradare performance e consumare molte risorse. I `Task` utilizzano thread del pool già esistenti, riusati per più operazioni, e forniscono metodi di alto livello (come `ContinueWith`, `Task.WhenAll`, ecc.) e naturalmente l'integrazione con `async/await`. Nella stragrande maggioranza dei casi moderni, **si preferisce usare i Task** (o costrutti ancora più alti come `Parallel.ForEach`, `async`/`await`) invece di gestire thread esplicitamente. Un caso in cui potrebbe servire un thread dedicato è per operazioni a *lunga durata e bassa priorità* che non vuoi interferiscano col thread pool, oppure per interfacciarsi con API legacy che richiedono thread separati.

Esempio: Creazione di un Thread vs avvio di un Task

```
using System.Threading;
using System.Threading.Tasks;

Thread t = new Thread(() => Console.WriteLine("Eseguito in un nuovo
thread!"));
t.Start();
t.Join();  // attende il termine del thread

Task task = Task.Run(() => Console.WriteLine("Eseguito in un Task del thread
pool!"));
task.Wait();  // attende il completamento del task
```

Spiegazione: Nel primo caso creiamo manualmente un `Thread`, passando un `ThreadStart` come lambda che stampa un messaggio. Chiamiamo `t.Start()` per avviare il thread separato, e `t.Join()` per aspettare che termini (il join blocca il thread chiamante finché `t` non finisce). Nel secondo caso utilizziamo `Task.Run` con una lambda: questo prende un thread dal pool e vi esegue il codice, restituendo un `Task` che rappresenta l'operazione in corso. Chiamiamo `task.Wait()` per attendere il completamento (equivalente ad usare `await task` in un contesto asincrono). Entrambi i messaggi saranno stampati da thread separati rispetto al thread principale. Come si nota, il codice con `Task` è più conciso e delega la gestione del thread al runtime. Inoltre, il `Task` potrebbe essere direttamente *awaitato* invece di `Wait()` per non bloccare il thread chiamante.

Best Practice per Task e Thread

- **Usa Task per operazioni concorrenti**: preferisci `Task.Run` o metodi asincroni invece di creare direttamente thread, a meno di esigenze specifiche. I Task permettono al thread pool di ottimizzare l'uso dei thread disponibili e si integrano con `async/await`.

- **Evita di sovra-creare thread**: ogni thread consuma memoria (stack) e ha un overhead di scheduling. Crearne molti (centinaia) può peggiorare le performance. Meglio suddividere il lavoro in pochi thread o usare Task che riciclano thread esistenti.
- **Sincronizzazione**: se accedi a risorse condivise da più thread/Task, utilizza meccanismi di sincronizzazione appropriati (`lock`, `Mutex`, `SemaphoreSlim`, collezioni thread-safe, etc.) per evitare condizioni di gara. Le operazioni su oggetti *thread-unsafe* (es. lista non sincronizzata) vanno protette.
- **CPU-bound vs I/O-bound**: per compiti *CPU-bound* intensi che devono scalare su più core, considera l'uso di *Parallel LINQ* (`Parallel.For`, `Parallel LINQ`) o il partizionamento del lavoro in più Task. Per compiti *I/O-bound*, l'uso di Task asincroni con await è ideale per non bloccare thread inutilmente.
- **Thread per operazioni speciali**: Se usi thread dedicati (es. un thread in background per polling continuo), assegna magari una *Lower Thread Priority* se appropriato per non sottrarre troppe risorse ai thread principali. In contesti UI (WinForms/WPF), ricorda che solo il thread UI principale può aggiornare i controlli UI: usa `SynchronizationContext` o metodi come `Control.Invoke` per marshallare le chiamate di ritorno sul thread UI se usi thread separati.

Configurazione di .NET Core e Dependency Injection

Una delle differenze significative tra .NET Framework e .NET Core (o .NET 5/6+) è come viene gestita la **configurazione dell'applicazione e dei servizi**. In .NET Framework tradizionale, molte configurazioni risiedevano in file come *App.config* o *Web.config* (XML) e non esisteva un contenitore di *dependency injection (DI)* predefinito – gli sviluppatori spesso integravano librerie di terze parti per gestire l'iniezione delle dipendenze. In .NET Core, invece, il supporto a DI è integrato nativamente nel framework: esiste un container predefinito in cui registrare i servizi e i relativi *lifetime*, e la configurazione applicativa tipicamente risiede in *appsettings.json* e viene caricata tramite l'API di configuration **Options**.

In .NET Core (così come in .NET 5/6), la configurazione dei servizi avviene costruendo un oggetto `IServiceCollection` su cui si registrano le mapping *interfaccia -> implementazione* e altre dipendenze necessarie [12]. Queste registrazioni definiscono quali oggetti il framework creerà e fornirà automaticamente ai punti appropriati (ad esempio ai costruttori dei controller in un'app web, o altrove nel codice tramite injection). I *lifetime* disponibili per i servizi sono tipicamente tre: **Transient** (una nuova istanza ad ogni richiesta di quel servizio), **Scoped** (una istanza per ogni *scope* definito, ad es. per ogni richiesta HTTP in ASP.NET Core) e **Singleton** (una singola istanza condivisa per tutta l'applicazione) [13]. La differenza col passato è netta: in .NET Core la DI e la configurazione sono parte del modello di progettazione standard, incoraggiando una architettura più pulita e testabile (ad es. facilitando il pattern IoC/DI e il testing tramite *mock*).

Esempio: Registrare e Risolvere Servizi in .NET Core

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();
// Registro un'interfaccia e la sua implementazione concreta
services.AddTransient<INotifica, NotificaEmail>();
// Registro un servizio concreto (con eventuali dipendenze iniettate)
services.AddTransient<Studente>();  // Studente dipende da INotifica nel
costruttore

// Costruisco il ServiceProvider (container DI) e richiedo un servizio
```

```
var provider = services.BuildServiceProvider();
Studente stud = provider.GetRequiredService<Studente>();
```

Nell'esempio: creiamo un contenitore dei servizi (`ServiceCollection`) e registriamo due servizi: un'interfaccia `INotifica` associata all'implementazione `NotificaEmail` (injection di interfaccia), e la classe `Studente`. Si assume che `Studente` nel suo costruttore richieda un `INotifica` – il container risolverà automaticamente `NotificaEmail` da fornire a `Studente`. Dopo aver chiamato `BuildServiceProvider()`, otteniamo un `ServiceProvider` attraverso cui richiedere istanze concrete. Con `GetRequiredService<Studente>()` otteniamo un oggetto `Studente` già costruito con la dipendenza `INotifica` soddisfatta dal container. Questa è la base della **Dependency Injection** integrata: il framework (o il nostro codice) *inietta* automaticamente le dipendenze necessarie. In un'app ASP.NET Core, questa registrazione avviene tipicamente in `Startup.cs` (o nel Program in .NET 6 minimal API) e l'infrastruttura fornisce i servizi ai controller, middleware, ecc.

Best Practice per DI e Configurazione in .NET Core

- **Registra per interfaccia**: preferisci registrare i servizi usando le loro interfacce (o classi base) piuttosto che implementazioni concrete. Questo rende facile sostituire l'implementazione (ad es. con una finta in fase di test) e segue il principio DIP (Dependency Inversion).
- **Scegli il lifetime appropriato**: usa *Transient* per servizi senza stato o leggeri, *Scoped* per servizi legati al contesto di una singola operazione (es. una richiesta web), *Singleton* per servizi pesanti da creare o che mantengono stato condiviso (ad es. cache in-memory) [13]. Evita di usare troppi singleton contenenti stato mutevole perché possono introdurre dipendenze nascoste tra parti dell'app.
- **Organizza la configurazione**: sfrutta i file di configurazione JSON (appsettings.json) e il binding a oggetti di configurazione tramite l'API Options (es. `IOptions<T>`). Ciò permette di mantenere separata la configurazione dal codice e facilitare modifiche (o override tramite variabili d'ambiente) senza ricompilare.
- **Configuration e Secrets**: in .NET Core, usa `IConfiguration` fornito dal framework per accedere a configurazioni tipate. Per dati sensibili, approfitta dei *secret* in sviluppo (Secret Manager) e delle variabili d'ambiente/KeyVault in produzione invece di metterli in chiaro nei file.
- **Inizializzazione e servizi esterni**: se hai bisogno di inizializzare risorse esterne (es. connessioni DB, client HTTP) alla partenza dell'app, registrali come singleton nel DI e considera l'uso di `IHostedService` o della sezione `Configure` in Startup per avviare processi in background.
- **Manutenibilità**: centralizza le registrazioni DI (tipicamente tutte nel `Startup` o in moduli di estensione dedicati) per avere un colpo d'occhio di tutte le dipendenze configurate. Mantieni coerenza nei nomi (es. per interfacce preferisci `INomeServizio`).

Architettura a Microservizi

L'**architettura a microservizi** consiste nel strutturare un'applicazione come un insieme di piccoli servizi indipendenti, ciascuno focalizzato su una specifica funzionalità di business [14]. Ogni microservizio è un'applicazione a sé stante, con la propria logica, database (preferibilmente separato) e può essere sviluppato e distribuito in modo indipendente dagli altri. I microservizi comunicano tra loro attraverso la rete, spesso via **API REST** (HTTP) o sistemi di messaging (es. code come RabbitMQ), anziché tramite chiamate in-process. Questo stile architetturale contrasta con il classico approccio monolitico, dove l'applicazione è un unico blocco deployabile.

Dal punto di vista organizzativo, i microservizi permettono a team diversi di lavorare in parallelo su servizi diversi, anche con tecnologie o linguaggi differenti, purché tutti espongano interfacce

compatibili (ad esempio API web). Ogni servizio può essere scalato individualmente in base al carico: ad esempio, il servizio “Ordini” può scalare su più istanze senza dover scalare anche il servizio “Catalogo” se quest’ultimo non lo richiede. Questa indipendenza è potente ma introduce nuove sfide di gestione e complessità distribuita.

Vantaggi dei Microservizi

- **Scalabilità indipendente:** ogni microservizio può essere scalato (aggiungendo risorse o istanze) in autonomia in base alle proprie esigenze di carico, senza dover scalare l'intera applicazione [15]. Ciò rende l’uso delle risorse più efficiente e mirato.
- **Distribuzione continua e indipendente:** si possono distribuire nuove versioni di un microservizio senza dover ridistribuire l’intero sistema. Questo facilita *deploy* frequenti e isolamento dei cambiamenti (minor impatto sul resto).
- **Manutenibilità e isolamento dei guasti:** il codice è suddiviso per dominio funzionale, rendendo più facile la comprensione e manutenzione di ciascun servizio. Un problema in un microservizio (bug, memory leak, ecc.) tendenzialmente non fa collassare l’intero sistema ma solo la funzione relativa, poiché i servizi sono isolati (potenzialmente con circuit breaker per evitare effetto domino).
- **Eterogeneità tecnologica:** ogni microservizio può essere implementato con lo stack tecnologico più adatto (linguaggio, database, librerie) senza vincoli imposti da altri moduli. Questo permette di sfruttare tecnologie ottimali per ciascun componente (pur aumentando l’eterogeneità complessiva).
- **Organizzazione per team e dominio:** i team possono essere allineati ai microservizi (ad es. team “Ordini”, team “Clienti”), lavorando con maggiore autonomia e su cicli di sviluppo differenti [15]. Questo favorisce il *DevOps*: team responsabili end-to-end del servizio (dallo sviluppo al deploy).

Svantaggi dei Microservizi

- **Maggiore complessità distribuita:** passare da un monolite a decine di servizi comporta complessità aggiuntiva in termini di comunicazione di rete, latenza, coerenza dei dati e gestione transazionale distribuita. Coordinare modifiche tra servizi può risultare complicato e richiede design attento.
- **Overhead di comunicazione:** le chiamate tra microservizi avvengono su rete (HTTP/gRPC/ messaging) e sono molto più lente delle chiamate in-process. Bisogna considerare la resilienza delle comunicazioni (reti possono fallire) e implementare meccanismi di retry, timeout e circuit breaker per evitare cascata di fallimenti.
- **Infrastruttura più pesante:** servono strumenti aggiuntivi per orchestrare e monitorare tanti servizi: logging centralizzato, monitoring delle metriche, distributed tracing per seguire chiamate attraverso servizi [16]. È necessario un investimento in piattaforme come Kubernetes, service mesh, registri di servizio, ecc., che aumentano il carico cognitivo e i costi operativi.
- **Gestione di consistenza e transazioni:** operazioni che in un monolite erano transazionali (es. un singolo commit DB) in un sistema distribuito richiedono strategie come *saga pattern* o compensazioni manuali per mantenere la consistenza tra servizi. Questo aggiunge complessità applicativa.
- **Deployment e CI/CD più complessi:** gestire versioni multiple di servizi, compatibilità tra API, e orchestrare il deployment (magari con decine di container) richiede pipeline CI/CD ben progettate e ambienti di testing integrati per evitare incongruenze. L'automazione diventa obbligatoria.

Best Practice per Microservizi

- **Definizione chiara dei boundary (bounded context):** Progetta ogni microservizio attorno a una singola responsabilità o area di business (ad es. *ordini*, *pagamenti*, *catalogo*). Evita overlap di funzionalità tra servizi e accoppiamenti eccessivi. Un buon *domain modeling* iniziale (DDD) aiuta a individuare i confini.
- **Interfacce ben definite e versionate:** I microservizi comunicano tramite API. Progetta API RESTful chiare o utilizza protocolli binari come gRPC quando serve performance. Versiona le API (ad esempio via URL o header) per poter evolvere i servizi senza rompere i client, e documentale (OpenAPI/Swagger). Mantieni i servizi *loose coupling*: ogni servizio conosce il meno possibile degli altri, solo contratti pubblici.
- **Comunicazione e resilienza:** Scegli il meccanismo di comunicazione adatto: richieste sincrone (HTTP/gRPC) vs asincrone (message broker) in base al dominio. Implementa pattern di resilienza: timeout, ritentativi esponenziali, circuit breakers (ad es. con librerie come Polly) per evitare che il fallimento di un servizio propaghi il disservizio a catena [16].
- **API Gateway:** Considera l'utilizzo di un **API Gateway** come ingresso unico per le chiamate esterne [17]. L'API Gateway può gestire autenticazione, routing verso i microservizi interni, aggregazione di risposte e policy di sicurezza/rate limiting. In questo modo semplifichi i client e centralizzi cross-cutting concerns.
- **Osservabilità:** Prepara una solida infrastruttura di **logging, monitoring e tracing distribuito** [18]. Ogni microservizio dovrebbe loggare in modo consistente e inviando i log a un sistema centralizzato (ELK stack, Azure Monitor, etc.). Raccogli metriche (CPU, memoria, latenza delle API, throughput) per ogni servizio e imposta allarmi. Implementa tracing (ad es. con OpenTelemetry) per tracciare una richiesta end-to-end attraverso servizi e identificare rapidamente colli di bottiglia o errori.
- **Automazione e container:** Containerizza i microservizi (Docker) per garantire ambienti uniformi tra sviluppo e produzione [19]. Usa orchestratori come Kubernetes per gestire il deployment, scaling automatico e resilienza (riavvio automatico di container falliti, rolling update). L'infrastruttura as code e pipeline CI/CD automatizzate sono fondamentali per gestire decine di servizi in modo affidabile.
- **Gestione dei dati e consistenza:** Mantieni i database dei microservizi separati per evitare accoppiamento a livello di storage. Per consistenza tra servizi, utilizza approcci come *eventual consistency* e, dove necessario, implementa il **Saga pattern** per orchestrare transazioni distribuite senza blocchi centralizzati [20]. Questo richiede un cambio di mentalità rispetto alle transazioni monolitiche: disegna flussi che tollerino latenze e possibili incoerenze temporanee, risolvendo con eventi di compensazione se necessario.

Conclusione: L'architettura a microservizi, con tutti i suoi vantaggi in termini di scalabilità e flessibilità, introduce complessità che vanno affrontate con un solido design e supporto infrastrutturale. Per un pubblico con conoscenza base-intermedia di C#, l'obiettivo è ora chiaro: dai fondamenti del linguaggio (tipi, collezioni, LINQ, delegati) si passa ad aspetti avanzati come la programmazione asincrona e l'architettura dei sistemi, gettando le basi per costruire applicazioni **pulite, scalabili e mantenibili** [21]. Buon proseguimento nello studio di C# e delle moderne tecnologie di sviluppo!

---

[1] [2] C# Propedeutico 1.pdf
file://file-SAkbnN1mdXNMsiRSVKJmuJ

[3] Come creare una copia profonda di un array bidimensionale? : r ...
https://www.reddit.com/r/csharp/comments/mo4gra/how_to_create_a_deep_copy_of_a_2d_array/?tl=it

4 C# - 2.pdf
file://file-TnN4hTJEuwzqT7kUs9TNcU

5 7 9 10 11 21 C# - 3.pdf
file://file-2m7C9VbgsohWmzf7E97qrF

6 8 C# Propedeutico 2.pdf
file://file-1nGPU47T3Jb4Ue2i4nwq2L

12 13 14 15 16 17 18 19 20 C# Propedeutico 3.pdf
file://file-Rf89AxzfnPgkvVfuZFRQ7t