

Appunti di informatica

a.s. 2024-2025

classe 3N

prof.ssa Chiara Fusaroli

ATTENZIONE: Versione in fase di sviluppo, i contenuti potranno subire variazioni

Ricordo a tutti gli studenti che come previsto dal regolamento di istituto (<https://www.ispascalcomandini.it/wp-content/uploads/2020/12/INTEGRAZIONE-AL-REGOLAMENTO-DI-ISTITUTO-PER-LA-DIDATTICA-A-DISTANZA.pdf>)

"L'utilizzo del materiale audiovisivo è riservato esclusivamente agli alunni della classe

ed è perciò consentito soltanto un uso privato da parte degli stessi allievi per fini

didattici. Il materiale didattico è protetto dalle vigenti normativa in materia di tutela

del diritto d'autore (Legge n. 633/1941 e ss. mm. e ii.) nonché dalla normativa in tema

di tutela dei dati personali (D.lgs. n 196/2003 ess.mm. e ii. e Regolamento UE n

679/2016 – GDPR), pertanto è assolutamente vietato divulgarlo a terzi in qualsiasi

forma, ivi compresa la sua riproduzione, pubblicazione e/o condivisione su social

media, piattaforme web (come ad esempio YouTube), applicazioni di messaggistica (come ad es. Whatsapp).

Ogni utilizzazione indebita e/o violazione sarà perseguita a termini di legge."

è quindi vietato riprodurre i materiali creati da docenti e/o compagni, senza il consenso degli autori.

Potete quindi utilizzare tutti i contenuti caricati sulla Classroom per studiare, potete scaricarli sui vostri dispositivi, stamparli, ma non potete condividerli con utenti esterni alla Classroom o divulgarli a terzi.

INDICE

INDICE.....	1
Introduzione.....	7
La programmazione.....	7
Algoritmi, Strutture Dati e gestione della memoria.....	7
Sintassi e Semantica.....	7
Controllo del Flusso.....	7
Modularità e Riutilizzabilità.....	8
Errori e Gestione delle Eccezioni.....	8
Testing e Debugging.....	8
Version Control e Collaborazione.....	8
Documentazione.....	9
Concorrenza e Multithreading.....	9
Linguaggi e paradigmi di programmazione.....	10
I paradigmi di programmazione.....	10
Linguaggi di Basso Livello.....	10
Linguaggi di Alto Livello.....	11
Esempio.....	11
linguaggi compilati e linguaggi interpretati.....	12
Principi di creazione del software.....	13
KISS (Keep It Simple, Stupid).....	13
DRY (Don't Repeat Yourself).....	13
YAGNI (You Ain't Gonna Need It).....	13
Separation of Concerns (SoC).....	14
Single Responsibility Principle (SRP).....	14
Modularità.....	14
Testabilità.....	14
Open/Closed Principle.....	14
Principi della OOP.....	15
Incapsulamento.....	15
ESEMPIO.....	15
Ereditarietà.....	16
Polimorfismo.....	17
Astrazione.....	17
Principi SOLID.....	17
Classi e Oggetti.....	18
Esempio:.....	18
Caratteristiche dell'Oggetto:.....	18
Messaggistica.....	19

INTRODUZIONE ALLA PROGRAMMAZIONE AD OGGETTI CON IL LINGUAGGIO C#. 20

.NET.....	21
Caratteristiche chiave del framework .NET.....	21
.NET 8.....	21
Code Convention.....	23
Nomenclatura (Naming).....	23
Spaziatura e indentazione.....	23
Dichiarazione di variabili.....	23
Commenti.....	24
Gestione delle eccezioni.....	24
Struttura delle classi.....	25
Uso delle proprietà.....	25
Struttura dei metodi.....	25
Linq e Lambda Expressions.....	25
Costrutti condizionali.....	25
Uso di async/await.....	26
Pattern Matching.....	26
Evitare "Magic Numbers".....	26
Gestione della nullabilità.....	26
Usare IDisposable e using.....	27
Documentazione automatica del codice.....	27
Diagramma delle classi.....	29
Elementi Principali di un Diagramma delle Classi.....	29
Esempio di Diagramma delle Classi.....	30
Utilizzo dei Diagrammi delle Classi.....	31
Creazione di una Classe in C#.....	34
Esempio.....	34
Creazione di un Oggetto (Istanza).....	34
Attributi.....	35
Proprietà.....	35
Tipi Di Proprietà.....	36
Proprietà Automatiche (Auto-implemented Properties).....	36
Proprietà con logica personalizzata (Full Properties).....	36
Proprietà con solo get o set.....	37
Proprietà con espressioni lambda (Expression-bodied Properties).....	38
Proprietà con valore predefinito.....	39
Proprietà Readonly con inizializzazione nel costruttore.....	39
Proprietà Indexer.....	40
Proprietà con modificatori di accesso diversi.....	40
Proprietà statiche.....	41
RIFERIMENTO ALLA GUIDA MICROSOFT.....	41
Metodi.....	46

Esempio:.....	46
Caratteristiche di un metodo:.....	47
Costruttore.....	47
Caratteristiche del costruttore.....	47
Esempio:.....	48
Cosa Fare Prima di Scrivere Codice.....	50
Linguaggio C#.....	51
Tipi di dati.....	51
Tipi di dati primitivi:.....	51
Tipi di dati composti:.....	51
Tipi di dati con notazione speciale:.....	51
Tipi di dati per dati nullable:.....	52
Tipi di dati per collezioni:.....	52
Altri tipi di dati specializzati:.....	52
Istruzioni in C#.....	52
Regole di sintassi.....	52
Costrutti di Selezione.....	54
IF.....	54
Operatore Ternario.....	55
Esempio.....	55
SWITCH.....	56
esempio.....	56
esempio:.....	58
Operatori Booleani.....	58
AND (&&).....	58
OR ().....	59
NOT (!).....	59
Costrutti Iterativi.....	60
DO-WHILE (ciclo in coda).....	60
Esempio:.....	60
WHILE (ciclo in testa).....	61
FOR.....	61
Esempio:.....	61
FOREACH.....	63
Gestione delle Eccezioni.....	64
Esempio 1.....	65
Esempio 2:.....	66
Esempio 3.....	67
CREAZIONE DELLA PRIMA CLASSE CON TEST - PROGETTO COFFE MACHINE...	68
ESEMPI.....	73
EQUALS E TOSTRING.....	73
SOVRASCRITTURA DELL'OPERATORE ==.....	75

GESTIONE DELLA MEMORIA IN C#.....	77
Stack.....	77
Heap.....	78
ESEMPIO DI GESTIONE DELLA MEMORIA.....	79
ESEMPIO 1:.....	79
ESEMPIO 2:.....	80
ESEMPIO 3:.....	80
MODALITA' DI PASSAGGIO DEI PARAMETRI AD UN METODO.....	84
Passaggio per valore.....	84
Passaggio per riferimento con ref.....	86
Passaggio per riferimento con out.....	87
ESEMPIO PROGETTO CON COMPOSIZIONE, AGGREGAZIONE.....	89
Dati ENUMERATIVI.....	93
Esempio Utilizzo di TryParse e IsDefined con Enumerativi.....	99
STRUTTURE DATI.....	102
ARRAY.....	102
LISTE (ARRAY DINAMICI).....	105
ARRAY E LISTE COME PARAMETRI.....	107
RICERCHE.....	109
RICERCA SEQUENZIALE.....	109
RICERCA DICOTOMICA (O BINARIA).....	110
Esempio:.....	111
ORDINAMENTI.....	111
SELECTION SORT.....	111
BUBBLE SORT.....	113
BUBBLE SORT CON SENTINELLA.....	113
INSERTION SORT.....	114
ARRAY SORT.....	115
CONFRONTI SUGLI ALGORITMI.....	117
ARRAY BIDIMENSIONALI.....	119
Definizione di un array bidimensionale.....	120
Accesso agli elementi della matrice.....	120
Le Matrici in memoria.....	121
ARRAY JAGGED.....	121
Esempio.....	121
ARRAY JAGGED VS MATRICI.....	121
STRING.....	123
PERSISTENZA DEI DATI.....	124
I/O da file di testo.....	124
LEGGERE DATI DA UN FILE DI TESTO.....	124
SCRIVERE DATI SU UN FILE DI TESTO.....	126
AGGIUNGERE TESTO AD UN FILE GIÀ ESISTENTE.....	127

I/O da file JSON.....	127
LEGGERE DATI DA UN FILE JSON.....	128
SCRIVERE DATI SU UN FILE JSON.....	128
APPENDICE.....	130
Classe Random.....	130
Complessità Computazionale.....	131
O(1): Complessità Costante.....	131
Principali Notazioni per il calcolo della complessità computazionale.....	131
GENERIC.....	132
CLASSI STATIC.....	133
string reversed = StringHelper.Reverse("ciao"); // reversed conterrà "oaiC".....	134
CLASSI PARTIAL.....	134
INTERFACCE.....	135
Le interfacce di sistema per gli ordinamenti.....	136
IComparer<T>.....	136
IComparable<T>.....	138
DEPENDENCY INJECTION.....	140
Esempi Progetti con uso di Interfacce:.....	143
Gioco Indovina il numero.....	143
WPF.....	144
Perché utilizzare WPF.....	144
WPF: Ancora Attuale o Superato.....	144
Alternative a WPF.....	145
UX e UI.....	146
MARKUP.....	146
CODE-BEHIND.....	147
Controlli WPF per funzione.....	148
Finestre di dialogo.....	148
Layout.....	149
Data binding.....	149
Esercitazione:.....	150
.NET MAUI.....	151
Creare la prima APP.....	151
Esercitazione guidata.....	152
Creare una APP .Net Maui con codice multiplatforma.....	152
Aggiungere i concetti MVVM.....	152
Creare APP per dispositivi mobili e desktop.....	153
INotifyPropertyChanged.....	154
ObservableObject (CommunityToolkit.Mvvm).....	155
ObservableCollection.....	156
IObservable<T>.....	156
Esempio progetto WPF con MVVM:.....	158

Clean Architecture.....	161
Struttura di un progetto Clean.....	162

Introduzione

La programmazione

Alcuni punti cardine della **programmazione** che riflettono i concetti, le metodologie e le tecniche utilizzate per scrivere codice sono riassumibili nelle seguenti categorie.

Algoritmi, Strutture Dati e gestione della memoria

- **Algoritmi:** Sequenze di passi per risolvere un problema o eseguire un compito specifico. Devono essere efficienti in termini di tempo (complessità temporale) e spazio (complessità spaziale).
- **Strutture Dati:** Modi di organizzare e gestire i dati in un programma. Le strutture dati che abbiamo visto in Kotlin o che vedremo nel corso dello studio della disciplina di Informatica sono principalmente:
 - Array
 - Liste
 - Code e Pile
 - Alberi
 - Grafi
 - Hash table
- **Allocazione e Deallocazione della Memoria:** In molti linguaggi di programmazione (come C++) è fondamentale comprendere come viene gestita la memoria per evitare sprechi (memory leaks) o errori di segmentazione.
- **Garbage Collection:** Molti linguaggi (come Java e C#) includono meccanismi automatici per gestire la memoria non più utilizzata.

Sintassi e Semantica

- **Sintassi:** Le regole grammaticali di un linguaggio di programmazione, ovvero come scrivere correttamente il codice.
- **Semantica:** Il significato dietro il codice. Anche se la sintassi è corretta, non è detto che la semantica lo sia; il programma deve avere un senso logico ed essere in grado di eseguire correttamente ciò per cui è stato progettato.

Controllo del Flusso

- **Condizioni:** Istruzioni come if, else, switch, che consentono di eseguire diverse azioni in base a determinate condizioni.
- **Loop:** Cicli come for, while, do-while, foreach, che permettono di ripetere istruzioni più volte.
- **Salti:** Come break, continue o return, che controllano quando uscire da un ciclo o una funzione. **I salti diminuiscono la leggibilità del codice e la loro manutenibilità nel tempo,**

per questo motivo, dobbiamo assolutamente evitare di inserire salti all'interno dei cicli. Se stiamo inserendo un salto all'interno di un ciclo for probabilmente sarebbe stato meglio inserire un ciclo di altro tipo

Modularità e Riutilizzabilità

- **Funzioni e Metodi:** Suddividere il codice in unità più piccole e riutilizzabili (moduli, funzioni o metodi) permette di mantenere il codice ordinato, leggibile e facile da mantenere.
- **Modularità:** Separare il codice in componenti distinte che possono essere sviluppate, testate e mantenute in modo indipendente permette di mantenere il codice ordinato, leggibile e facile da mantenere e permette di inserire nuove funzionalità con un impiego di tempo minore.

Errori e Gestione delle Eccezioni

- **Gestione degli Errori:** Prevedere e gestire situazioni inattese come divisioni per zero, errori sui valori dei parametri, situazioni di inconsistenza, ecc.
- **Eccezioni:** Utilizzare il meccanismo di gestione delle eccezioni (**try-catch**) per intercettare e gestire errori in fase di esecuzione.

Testing e Debugging

- **Test Unitari:** Verificare che piccole porzioni di codice funzionino correttamente.
- **Debugging:** Identificare e risolvere errori o bug nel codice, utilizzando strumenti di debug o inserendo istruzioni di log.
- **Test di Integrazione e Sistemi:** Testare l'interazione tra componenti diverse di un sistema complesso.
- i test dovrebbero essere pensati e codificati prima della stesura del codice
 - test di **accettabilità** per verificare che il software non permetta di introdurre dati errati
 - test di **controllo** per verificare che le funzionalità implementate si comportino correttamente
- tutti i test devono essere eseguiti al termine della stesura del codice, prima di rilasciare la versione finale. se risultano errori dall'esecuzione del test e' necessario apportare le modifiche opportune al codice e rieseguire tutti i test (anche quelli che prima non davano errori)
- ogni test deve avere **uno** scopo ben preciso

Version Control e Collaborazione

- **Sistemi di Versionamento:** Strumenti come Git per tracciare le modifiche del codice e facilitare il lavoro di squadra.

- **Collaborazione:** Lavorare in team di sviluppo richiede spesso di condividere e gestire il codice in modo efficace e organizzato.

Documentazione

- Scrivere codice ben documentato è essenziale per garantire che sia comprensibile anche a lungo termine da altri sviluppatori o dallo stesso autore.

Concorrenza e Multithreading

- **Processi e Thread:** La capacità di eseguire più istruzioni contemporaneamente (parallelo o concorrente).
- **Sincronizzazione:** Garantire che i thread che accedono a risorse condivise lo facciano in modo sicuro senza causare condizioni di corsa.

Linguaggi e paradigmi di programmazione

- specifici d un dominio (esempio sql)
- general purpose (c#, kotlin, java, python,...)
- di sistema (c++)

I paradigmi di programmazione

- **Programmazione Procedurale:** Organizza il codice in sequenze di istruzioni e funzioni.
- **Programmazione Orientata agli Oggetti (OOP):** Modella i dati come oggetti che interagiscono tra loro. Si basa su concetti come:
 - **Classe e Oggetto**
 - **Ereditarietà**
 - **Incapsulamento**
 - **Polimorfismo**
- **Programmazione Funzionale:** Focalizzata su funzioni pure e immutabilità, evitando cambiamenti di stato.
- **Programmazione Logica:** Basata su regole logiche per risolvere problemi (es. Prolog).

Linguaggi di Basso Livello

I **linguaggi di programmazione** possono essere classificati in **alto livello** e **basso livello** in base alla loro vicinanza al linguaggio macchina (il linguaggio che un computer comprende direttamente) e alla facilità con cui possono essere compresi dagli esseri umani.

I linguaggi di basso livello sono molto vicini all'hardware e al funzionamento interno di un computer. Offrono poco o nessun livello di astrazione rispetto al linguaggio macchina, quindi richiedono una conoscenza dettagliata dell'architettura del sistema.

- **Linguaggio Macchina:** È il linguaggio che un processore esegue direttamente. Consiste in una serie di numeri binari (0 e 1) che rappresentano istruzioni. È estremamente difficile da leggere e scrivere per gli esseri umani.
- **Assembly:** È un linguaggio simbolico che rappresenta il linguaggio macchina utilizzando abbreviazioni o codici mnemonici per le istruzioni. Per esempio, al posto di un'istruzione in codice binario, in assembly si usa una sigla come **MOV** per "muovere" un dato da un registro all'altro. Anche se più leggibile del linguaggio macchina, rimane strettamente legato all'architettura specifica del processore e richiede una conoscenza approfondita dell'hardware.

Caratteristiche dei linguaggi di basso livello:

- **Alta efficienza:** Possono essere molto veloci e ottimizzati, dato che il controllo sul processore è diretto.
- **Difficoltà:** Sono complessi da scrivere e mantenere.

- **Portabilità ridotta:** I programmi scritti in assembly o linguaggio macchina funzionano solo su una specifica architettura di processore.

Linguaggi di Alto Livello

I linguaggi di alto livello sono progettati per essere più vicini al modo di pensare umano e più astratti rispetto all'hardware. Sono più facili da leggere, scrivere e mantenere perché usano istruzioni simili al linguaggio naturale (come l'inglese) o simboli matematici.

Sono ad esempio linguaggi di alto livello:

- Python
- C#
- C++
- JavaScript
- Kotlin
-

Caratteristiche dei linguaggi di alto livello:

- **Astrazione dall'hardware:** Gli sviluppatori non devono preoccuparsi di dettagli come la gestione della memoria o i registri della CPU, perché queste operazioni sono gestite automaticamente.
- **Facilità di utilizzo:** Sono progettati per essere intuitivi e leggibili, con sintassi più vicina al linguaggio naturale.
- **Portabilità:** Il codice scritto in un linguaggio di alto livello è generalmente portabile su diverse piattaforme e architetture, grazie a compilatori o interpreti che lo convertono nel linguaggio macchina della specifica piattaforma.
- **Prestazioni:** Anche se meno efficienti dei linguaggi di basso livello, grazie agli ottimizzatori dei compilatori e degli interpreti, i linguaggi di alto livello possono essere sufficientemente veloci per la maggior parte delle applicazioni.

Esempio

Se vuoi sommare due numeri in un linguaggio di basso livello (come assembly), potresti scrivere:

```
MOV AX, 2      ; Muove il valore 2 nel registro AX
```

```
ADD AX, 3      ; Aggiunge 3 al valore nel registro AX
```

Mentre in un linguaggio di alto livello potrei fare la stessa cosa in modo molto più semplice:

```
somma = 2 + 3
```

linguaggi compilati e linguaggi interpretati

- linguaggi compilati (c#)
 - sorgente → fase di compilazione → eseguibile
il compilatore compila tutto il sorgente generando un file direttamente eseguibile dal sistema
 - gli errori di sintassi sono **tutti** rilevati prima della generazione dell'eseguibile
- linguaggi interpretati (javascript)
 - il codice sorgente è tradotto in codice eseguibile dall'interprete nel momento in cui deve essere eseguito (istruzione per istruzione)
 - potrebbero esserci errori di sintassi in porzioni di codice non eseguito (questi errori non saranno quindi rilevati se non quando quelle istruzioni andranno in esecuzione)

Principi di creazione del software

Il software che produciamo, affinché possa essere considerato un **software professionale** deve raggiungere dei livelli di qualità interne (rilevate dagli sviluppatori es: leggibilità del codice, riusabilità, robustezza,..) ed esterne (rilevate dagli utilizzatori es: facilità di utilizzo, velocità, ..)

Le qualità indispensabili che dobbiamo raggiungere per ogni software che generiamo sono l'**affidabilità** in termini di **correttezza** (il software deve risolvere correttamente i problemi per il quale è stato creato) e **robustezza** (il software deve essere sempre in grado di rispondere, eventualmente generando un errore ma senza andare in crash) e la **modularità** in termini di **estensibilità** (deve essere semplice poter aggiungere nuove funzionalità al software, senza dover modificare o riscrivere il codice già presente) e **riusabilità** (non dobbiamo ricreare gli stessi componenti se li abbiamo già creati o se già esistono)

Il software dovrebbe poi raggiungere anche altre qualità come:

- compatibilità
- efficienza (minimizzare / trovare il migliore equilibrio nell'uso delle risorse (tempo e memoria))
- portabilità
- facilità di utilizzo (user friendly)

I seguenti principi di creazione del software ci forniscono le linee guida per lo sviluppo di un software robusto, manutenibile nel tempo e scalabile.

Questi principi si applicano in generale a qualsiasi approccio di sviluppo software, indipendentemente dal paradigma di programmazione utilizzato (OOP, funzionale, procedurale, ecc.).

KISS (Keep It Simple, Stupid)

- Il codice dovrebbe essere il più semplice possibile, evitando complessità inutili. Un software più semplice è più facile da capire, mantenere e testare.
- **Obiettivo:** ridurre al minimo le funzionalità non necessarie e concentrarsi sulla soluzione del problema in modo chiaro e diretto.

DRY (Don't Repeat Yourself)

- Evitare la duplicazione del codice. Se lo stesso blocco di codice è ripetuto in più punti, significa che può essere astratto in una funzione, modulo o classe.
- **Obiettivo:** Ogni porzione di conoscenza nel sistema dovrebbe avere una singola rappresentazione. Questo riduce la possibilità di errori e facilita la manutenzione.

YAGNI (You Ain't Gonna Need It)

- Non implementare funzionalità di cui non hai immediatamente bisogno. Anticipare funzionalità non necessarie può complicare il codice e introdurre bug.
- **Obiettivo:** Sviluppare solo ciò che è richiesto dal progetto in quel momento. Se una nuova funzionalità diventa necessaria in seguito, sarà aggiunta quando serve.

Separation of Concerns (SoC)

- Suddividere un programma in parti indipendenti, ognuna delle quali gestisce un aspetto specifico del problema. Questo favorisce la modularità e la comprensibilità del codice.
- **Obiettivo:** Utilizzare funzioni, moduli o classi che gestiscano solo un ambito limitato di responsabilità, riducendo le interdipendenze tra parti del sistema.

Single Responsibility Principle (SRP)

- Ogni modulo o componente dovrebbe avere una singola responsabilità. Ogni componente deve fare una sola cosa e farla bene.
- **Obiettivo:** Questo principio migliora la manutenibilità e la chiarezza del codice, poiché ogni componente può essere modificato o migliorato senza influenzare altre parti del sistema.

Modularità

- Il software dovrebbe essere diviso in moduli autonomi che interagiscono tra loro. Ogni modulo dovrebbe avere una chiara interfaccia.
- **Obiettivo:** La modularità permette di sviluppare, testare e mantenere diverse parti del software in modo indipendente.

Testabilità

- Il codice dovrebbe essere progettato per essere facilmente testato. Test automatizzati come unit test o test d'integrazione dovrebbero essere scritti per verificare il corretto funzionamento del software.
- **Obiettivo:** Scrivere funzioni piccole e modulari facilita il testing. Inoltre, separare la logica applicativa da altre preoccupazioni (ad esempio, l'interfaccia utente o l'accesso ai dati) permette di testare in isolamento le diverse parti.

Open/Closed Principle

- Le entità software (classi, moduli, funzioni) dovrebbero essere aperte per estensione ma chiuse per modifiche.
- **Obiettivo:** Dovrebbe essere possibile estendere il comportamento di un componente senza modificarne il codice sorgente, ad esempio, attraverso l'uso di classi derivate, interfacce o funzioni.

Principi della OOP

I principi OOP si concentrano su come organizzare e strutturare il codice in termini di oggetti, per modellare il comportamento e le proprietà di entità nel mondo reale nel sistema.

Incapsulamento

- Nascondere i dettagli interni di un oggetto e consentire l'accesso e la modifica solo attraverso metodi pubblici definiti (getter e setter).

L'incapsulamento separa le responsabilità e protegge i dati.

L'incapsulamento aiuta a proteggere l'integrità dei dati e a garantire che le operazioni sugli oggetti siano coerenti. L'incapsulamento assicura che sia possibile accedere e modificare dati (attributi) e comportamenti (metodi) solo in modi che rispettino le regole definite dalla classe, garantendo così la coerenza e la sicurezza del sistema.

Gli utilizzatori di un oggetto possono interagire con esso solo attraverso l'interfaccia pubblica, senza dover preoccuparsi dei dettagli di implementazione. Questo migliora la manutenibilità del codice e riduce la complessità.

- **Obiettivo:** Usare attributi privati o protetti in una classe e fornire metodi pubblici per interagire con questi attributi. Ciò garantisce il controllo su come i dati vengono letti e modificati.

ESEMPIO

```
class Prodotto
{
    public string Nome { get; }
    public int Quantita { get; private set; }

    public Prodotto(string nome, int quantita)
    {
        Nome = nome;
        Quantita = quantita;
    }

    public void AggiungiProdotto(int quantitaAggiunta)
```



```

    {
        if (quantitaAggiunta <= 0)
        {
            throw new ArgumentException("La quantità da aggiungere
deve essere maggiore di zero.");
        }

        Quantita += quantitaAggiunta;
    }

    public void RimuoviProdotto(int quantitaRimossa)
    {
        if (quantitaRimossa <= 0 || quantitaRimossa > Quantita)
        {
            throw new ArgumentException("La quantità da rimuovere
deve essere maggiore di zero e non superare la quantità
disponibile.");
        }

        Quantita -= quantitaRimossa;
    }
}

```

Ereditarietà

- L'ereditarietà permette a una classe (classe derivata) di ereditare attributi e metodi da un'altra classe (classe base o superclasse), promuovendo il riutilizzo del codice. L'ereditarietà consente di creare nuove classi basate su classi esistenti. questa nuova classe eredita attributi e metodi della classe genitore, consentendo il riutilizzo del codice. è possibile estendere o sovrascrivere i metodi ereditati per adattarli alle esigenze specifiche della nuova classe.
- **Applicazione:** Utilizzare classi genitoriali per implementare comportamenti comuni e far derivare classi figlie per aggiungere o estendere funzionalità specifiche. Ad esempio, una

classe Animale può avere sottoclassi come Cane o Gatto che condividono comportamenti comuni.

Polimorfismo

- Il polimorfismo consente agli oggetti di diverse classi di essere trattati allo stesso modo attraverso un'interfaccia comune. Si divide in due forme:
 - Polimorfismo statico (**overloading**): Diverse versioni di un metodo possono avere lo stesso nome ma parametri differenti.
 - Polimorfismo dinamico (**overriding**): Una sottoclasse può fornire la propria implementazione di un metodo definito nella classe base.
- **Obiettivo**: Implementare metodi con lo stesso nome ma con comportamenti diversi a seconda del contesto o della classe dell'oggetto.

Astrazione

- L'astrazione implica nascondere i dettagli complessi e mostrare solo le funzionalità essenziali. In OOP, l'astrazione si realizza attraverso l'uso di **classi astratte o interfacce**. Più in generale questo principio consente agli sviluppatori di creare modelli concettuali di oggetti del mondo reale e di interagire con essi nel software. L'astrazione si basa sull'idea di nascondere i dettagli complessi e irrilevanti di un oggetto, concentrandosi solo su quelli rilevanti per uno scopo specifico. (es. persona per registro elettronico o per servizio anagrafe)
- **Obiettivo**: Creare interfacce o classi astratte che definiscono i metodi base senza implementazione dettagliata, lasciando alle classi concrete la responsabilità di fornire implementazioni specifiche.

Principi SOLID

I principi SOLID sono un insieme di linee guida per rendere il codice OOP più manutenibile e flessibile:

- Single Responsibility Principle (SRP): Ogni classe dovrebbe avere una singola responsabilità.
- Open/Closed Principle (OCP): Le classi dovrebbero essere aperte all'estensione ma chiuse alla modifica.
- Liskov Substitution Principle (LSP): Le classi derivate dovrebbero poter essere utilizzate al posto delle loro classi base senza rompere l'applicazione.
- Interface Segregation Principle (ISP): Le interfacce dovrebbero essere piccole e specifiche, piuttosto che grandi e generiche.
- Dependency Inversion Principle (DIP): Le classi dovrebbero dipendere da astrazioni (interfacce), non da classi concrete.

Classi e Oggetti

Nella **programmazione orientata agli oggetti (OOP)**, **classi** e **oggetti** sono i concetti centrali che modellano il software in termini di entità reali o astratte.

Una classe è un modello o un prototipo da cui vengono creati gli oggetti, che sono le istanze specifiche di quella classe.

Una **classe** è un modello che definisce le caratteristiche e i comportamenti di un gruppo di oggetti simili. La classe specifica quali **attributi** (dati, proprietà) e quali **metodi** (funzionalità, comportamenti) avranno gli oggetti creati da essa.

Esempio:

Se consideriamo la classe **Automobile**, potrebbe avere attributi come:

- **marca**
- **modello**
- **colore**

E metodi come:

- **accendi()**: che avvia l'automobile.
- **ferma()**: che ferma l'automobile.

Un **oggetto** è un'**istanza di una classe**. Quando si crea un oggetto, si sta effettivamente creando un'unità concreta che segue il modello definito dalla classe.

Ogni oggetto ha i propri dati e può eseguire i metodi definiti nella classe.

Caratteristiche dell'Oggetto:

1. **Stato**: Ogni oggetto ha un insieme di valori unici per gli attributi.
2. **Comportamento**: Gli oggetti possono eseguire i metodi della classe. Anche se **accendi()** è definito nella classe **Automobile**, quando lo chiami su un oggetto come **auto1**, l'azione viene eseguita specificamente per quell'oggetto.

Potremmo immaginare che la classe sia come un **progetto architettonico** per una casa. Il progetto definisce le caratteristiche (numero di stanze, altezza del soffitto, ecc.) e le funzioni (come aprire le porte o accendere le luci). Tuttavia, non è una casa reale. La casa reale che costruisci a partire dal progetto è l'**oggetto**: ha un numero specifico di stanze e un soffitto di un'altezza precisa, che puoi interagire con azioni reali (aprire porte, accendere luci).

Messaggistica

Nella **programmazione orientata agli oggetti (OOP)**, gli **oggetti** comunicano tra loro attraverso lo scambio di **messaggi** o **invocazione di metodi**. Questo processo è fondamentale per il funzionamento del software basato su oggetti, in quanto consente di modellare interazioni tra entità diverse del sistema.

Gli oggetti comunicano principalmente chiamando i metodi di altri oggetti. Quando un oggetto vuole interagire con un altro, invia un messaggio (chiamata di metodo) che di solito include dati sotto forma di **argomenti**.

- L'oggetto chiamante (mittente) invoca un metodo sull'oggetto ricevente.
- L'oggetto ricevente esegue il metodo specifico, eventualmente restituendo un risultato al mittente.

INTRODUZIONE ALLA PROGRAMMAZIONE AD OGGETTI CON IL LINGUAGGIO C#

<https://learn.microsoft.com/it-it/dotnet/csharp/>

C# è un linguaggio di programmazione sviluppato da Microsoft. È parte del framework .NET e viene ampiamente utilizzato per lo sviluppo di applicazioni desktop, web e mobile su piattaforme Windows.

Caratteristiche:

Orientato agli oggetti: C# è un linguaggio di programmazione orientato agli oggetti (OOP). Supporta concetti OOP come classi, oggetti, incapsulamento, ereditarietà e polimorfismo.

Sicurezza del tipo: C# è un linguaggio fortemente tipizzato, il che significa che è in grado di catturare errori di tipo a tempo di compilazione, contribuendo a rendere il codice più sicuro e meno soggetto a errori.

Interoperabilità: C# è stato progettato per essere altamente interoperabile con altre tecnologie Microsoft. Può essere utilizzato in combinazione con librerie C/C++ tramite P/Invoke e può chiamare librerie di Windows direttamente.

Sintassi chiara e leggibile: C# offre una sintassi chiara e leggibile, che lo rende facile da apprendere e scrivere. Questo contribuisce a migliorare la produttività degli sviluppatori.

Gestione della memoria: **C# è un linguaggio con un sistema di gestione automatica della memoria.** Utilizza il Garbage Collector per liberare automaticamente la memoria dai riferimenti non più utilizzati, semplificando la gestione della memoria.

Framework .NET: C# è spesso utilizzato in combinazione con il framework .NET, che fornisce una vasta gamma di librerie e strumenti per lo sviluppo di applicazioni Windows, web e mobile.

Supporto multiplatforma: Con l'introduzione di **.NET Core** (ora noto come .NET 5 e successivi), C# è diventato un linguaggio multiplatforma che può essere utilizzato per sviluppare applicazioni su Windows, Linux e macOS.

C# è ampiamente utilizzato per lo sviluppo di applicazioni desktop (tramite Windows Forms o WPF), applicazioni web (tramite ASP.NET), applicazioni mobili (tramite Maui), giochi (tramite Unity), servizi Windows, e molto altro.

Strumenti di sviluppo: Microsoft fornisce un'ampia gamma di strumenti di sviluppo per C#, tra cui l'IDE Visual Studio e l'editor di codice open source Visual Studio Code.

.NET

Il framework .NET è un ambiente di sviluppo software ampio e flessibile sviluppato da Microsoft. Fornisce una piattaforma per la creazione, la distribuzione e l'esecuzione di applicazioni software. Il framework .NET supporta una varietà di linguaggi di programmazione, tra cui C#, F#, Visual Basic, e altri, consentendo agli sviluppatori di scrivere applicazioni per diverse piattaforme e scopi.

Caratteristiche chiave del framework .NET

Architettura modulare: Il framework .NET è composto da diversi componenti modulari che possono essere utilizzati in modo indipendente. Alcuni di questi componenti includono il Common Language Runtime (CLR), il Framework Class Library (FCL), il compilatore JIT (Just-In-Time), e altri.

Common Language Runtime (CLR): Il CLR è il motore di esecuzione delle applicazioni .NET. Esso fornisce servizi come la gestione della memoria, il garbage collection, la sicurezza del tipo e altro. Il codice sorgente scritto in linguaggi .NET viene compilato in codice IL (Intermediate Language), che viene poi eseguito dal CLR.

Framework Class Library (FCL): La FCL è una libreria di classi che fornisce un ampio insieme di funzionalità standard per la programmazione. Include classi per lavorare con collezioni, input/output, comunicazione di rete, grafica, sicurezza e altro. Semplifica lo sviluppo delle applicazioni, poiché molti compiti comuni sono già implementati in queste classi.

Linguaggi di programmazione: Il framework .NET supporta diversi linguaggi di programmazione, inclusi C#, F#, Visual Basic e altri. Gli sviluppatori possono scegliere il linguaggio più adatto per il loro progetto.

Sistemi operativi: Il framework .NET è multiplatforma e può essere eseguito su Windows, Linux e macOS. Questo consente agli sviluppatori di scrivere applicazioni che possono essere eseguite su diverse piattaforme senza dover riscrivere il codice da zero.

Ambiente di sviluppo: Microsoft offre strumenti di sviluppo come Visual Studio e Visual Studio Code per semplificare lo sviluppo di applicazioni .NET. Inoltre, è possibile sviluppare applicazioni .NET in molti altri editor di testo e ambienti di sviluppo.

Con l'introduzione di .NET 5 e versioni successive, il framework .NET è diventato sempre più potente e multiplatforma.

.NET 8

.NET 8 è l'ultima versione principale della piattaforma .NET, rilasciata a novembre 2023. Fa parte della piattaforma unificata di Microsoft, offrendo strumenti per la creazione di un'ampia gamma di

applicazioni, inclusi web, desktop, mobile e cloud. Questa versione introduce varie novità e miglioramenti in termini di prestazioni, supporto per nuove funzionalità di linguaggio, e strumenti di sviluppo avanzati.

Alcune delle novità principali in .NET 8 sono:

1. **Miglioramenti delle prestazioni:** riduzione dei tempi di avvio delle applicazioni e ottimizzazione dell'uso della memoria.
2. **Aggiornamenti a ASP.NET Core:** Sono state introdotte nuove funzionalità per lo sviluppo di applicazioni web moderne, tra cui miglioramenti nella gestione delle API e supporto per nuove tecnologie web.
3. **Blazor full-stack:** La tecnologia Blazor consente di sviluppare interfacce utente web utilizzando C# anziché JavaScript.
In .NET 8, è stato migliorato il supporto per lo sviluppo di applicazioni web interamente con C#.
4. **Miglioramenti al supporto del cloud:** .NET 8 integra nuovi strumenti per semplificare la creazione di applicazioni cloud-native e il supporto a tecnologie come Kubernetes.
5. **Supporto avanzato per IA:** .NET 8 migliora il supporto per integrazioni con servizi di intelligenza artificiale, aiutando gli sviluppatori a costruire applicazioni che sfruttano AI/ML.

Code Convention

Regole e linee guida che definiscono come dovrebbe essere formattato il codice sorgente scritto in un linguaggio di programmazione specifico.

Le code convention aiutano a mantenere il codice più leggibile, uniforme e facilmente comprensibile da parte dei membri del team di sviluppo.

Principali convenzioni di codifica per C#:

Nomenclatura (Naming)

- **PascalCase** per i nomi di classi, metodi e proprietà pubbliche.
 - Esempio: `public class MyClass { }, public void CalculateTotal() { }`
- **camelCase** per variabili locali e campi privati.
 - Esempio: `private int itemCount;, string customerName;`
- **CONSTANT_UPPERCASE** per le costanti.
 - Esempio: `const int MAX_SIZE = 100;`
- **Prefisso underscore _** per i campi privati, se necessario per distinguerli da parametri.
 - Esempio: `_myField, _totalCount`

Spaziatura e indentazione

- **Indentazione a 4 spazi** (impostare le tabulazioni dell'ambiente per uniformarsi)
- Le **parentesi graffe {}** devono essere posizionate su una nuova linea.

Esempio:

```
if (isValid)

{

    DoSomething();

}
```

- **Riga vuota** tra metodi e blocchi di codice logici per migliorare la leggibilità.

Dichiarazione di variabili

- Dichiarare una variabile **il più vicino possibile** al punto in cui viene utilizzata.
- Usare `var` solo quando il tipo è **ovvio dal contesto** o quando il tipo è complesso e lungo.

Esempio:

```
var person = new Person(); // Ok, il tipo è chiaro
```

```
int age = 30; // Meglio specificare il tipo in casi semplici
```

Commenti

- Usare commenti solo quando è necessario **spiegare il "perché"**, non il "come" (che dovrebbe essere chiaro dal codice).
- **Commenti XML (///)** per documentare i membri pubblici delle classi.

Esempio:

```
/// <summary>  
/// Metodo che calcola il totale.  
/// </summary>
```

```
public void CalculateTotal()  
{  
  
    // Logica di calcolo  
  
}
```

L'uso di un set di linee guida di codifica coerenti è particolarmente importante in progetti di sviluppo collaborativo, in quanto contribuisce a mantenere il codice uniforme e facilmente comprensibile per tutti i membri del team.

Gestione delle eccezioni

- Usare try-catch per gestire le eccezioni ma evitare di catturare eccezioni generiche (catch (Exception)), a meno che non sia strettamente necessario.
- Fornire messaggi di errore significativi o logging nel blocco catch.

Esempio:

csharp

Copia codice

```
try  
{  
  
    // Codice che potrebbe lanciare un'eccezione  
  
}  
  
catch (SpecificException ex)
```

```
{  
  
    // Gestione specifica dell'eccezione  
  
    LogError(ex.Message);  
  
}
```

Struttura delle classi

- Le classi dovrebbero essere **piccole e focalizzate** su una singola responsabilità (principio SRP - Single Responsibility Principle).
- Ordinare i membri della classe con questo schema: **campi privati, costruttore, proprietà, metodi pubblici, metodi privati**.

Uso delle proprietà

- Usare le **proprietà automatiche** per semplificare la dichiarazione dei campi.

Esempio:

```
public string Name { get; set; }
```

- Evitare i campi pubblici; usare proprietà pubbliche per l'incapsulamento.

Struttura dei metodi

- Un metodo dovrebbe avere un **solo scopo** e una dimensione ragionevole (meno di 30 righe di codice, se possibile).
- Evitare metodi con troppi parametri; se necessario, considerare di incapsulare i parametri in un oggetto.

Linq e Lambda Expressions

- Usare LINQ quando rende il codice più leggibile e conciso, ma senza abusarne al punto da compromettere la chiarezza.

Esempio:

```
var adultNames = people.Where(p => p.Age >= 18).Select(p => p.Name);
```

Costrutti condizionali

- Utilizzare **operatore ternario** per assegnazioni semplici, ma evitare di usarlo per condizioni complesse.

Esempio:

```
var status = isActive ? "Active" : "Inactive";
```

Uso di async/await

- Seguire la convenzione di aggiungere il suffisso Async ai metodi asincroni.

Esempio:

```
public async Task FetchDataAsync()
{
    await GetDataFromApi();
}
```

Pattern Matching

- Usare il pattern matching per scrivere codice più conciso, specialmente nei blocchi switch o nelle dichiarazioni if.

Esempio:

```
if (obj is Person person)
{
    Console.WriteLine(person.Name);
}
```

Evitare "Magic Numbers"

- Non utilizzare valori numerici direttamente nel codice. Definire costanti significative.

Esempio:

```
const int MaxRetries = 3;

for (int i = 0; i < MaxRetries; i++) { }
```

Gestione della nullabilità

- Utilizzare l'operatore nullo (??, ?.) per prevenire eccezioni NullReferenceException.

Esempio:

```
var name = person?.Name ?? "Default Name";
```

Usare IDisposable e using

- Usare la dichiarazione using per assicurarsi che gli oggetti che implementano IDisposable vengano liberati correttamente.

Esempio:

```
using (var stream = new FileStream("file.txt", FileMode.Open))

{

    // Operazioni sul file

}
```

Queste convenzioni aiutano a scrivere codice che sia coerente, leggibile e più facile da mantenere nel tempo. Molti team di sviluppo definiscono inoltre proprie regole personalizzate, basandosi su queste linee guida generali.

Documentazione automatica del codice

La documentazione automatica, spesso chiamata "XML doc comments", può essere generata in Visual Studio utilizzando commenti speciali nel codice sorgente. Questi commenti sono formattati in modo specifico e possono essere successivamente elaborati dagli strumenti di generazione della documentazione.

Inserimento dei commenti XML :

Per generare documentazione automatica, devi inserire commenti XML sopra le classi, i metodi, i campi, le proprietà, i parametri, i valori restituiti e altro nel tuo codice.

(in visual studio è sufficiente inserire la tripla barra /// ed i tag xml vengono autogenerati)

Esempio:

```
/// <summary>

/// Questo è un esempio di commento XML per una classe.

/// </summary>

public class MiaClasse

{

    /// <summary>

    /// Questo è un esempio di commento XML per un metodo.
```

```
/// </summary>

/// <param name="parametro">Descrizione del parametro.</param>

/// <returns>Descrizione del valore restituito.</returns>

public int MioMetodo(int parametro)

{

    // Il tuo codice qui.

}

}
```

Diagramma delle classi

Un **diagramma delle classi** è uno dei principali tipi di diagrammi utilizzati nella modellazione ad oggetti e nell'**UML (Unified Modeling Language)**. Viene utilizzato per rappresentare la struttura statica di un sistema software, mostrando le classi, le loro proprietà (attributi), i metodi (operazioni), e le relazioni tra le varie classi.

Elementi Principali di un Diagramma delle Classi

Classi:


- Rappresentano le entità principali del sistema.
- Viene rappresentata da un rettangolo diviso in tre sezioni:
 - La sezione superiore contiene il nome della classe.
 - La sezione centrale contiene gli **attributi** (proprietà).
 - La sezione inferiore contiene i **metodi** (operazioni o comportamenti).

Esempio di rappresentazione di una classe:


```
-----  
|      Persona      | ← Nome della classe  
-----  
| nome: string      | ← Attributi (proprietà)  
| età: int          |  
-----  
| parlare(): void    | ← Metodi (operazioni)  
| camminare(): void  |  
-----
```

Relazioni:

- **Associazioni:** Indicano una **relazione generica tra due classi**. Viene rappresentata con una **linea semplice** tra le classi. Può avere una **molteplicità** (1 a 1, 1 a molti, molti a molti) che descrive il numero di oggetti coinvolti.
 - Esempio: Una Persona può possedere più Automobili → 1 a molti.
 - Notazione: Una linea con una cardinalità accanto (1, 0..1, *, ecc.).

- **Ereditarietà (Generalizzazione):** Rappresenta il fatto che **una classe è una sottoclasse o derivata di un'altra classe** (concetto di ereditarietà in OOP). Viene rappresentata con una **linea con una freccia vuota** che punta alla classe "madre".
 - Esempio: Studente è una sottoclasse di Persona.
 - Notazione: Studente —————▶ Persona
- **Composizione:** Indica una **relazione "parte-tutto" forte**. Se la parte viene distrutta, il tutto viene distrutto anch'esso. È rappresentata da una **linea con un rombo nero vicino alla classe che rappresenta il "tutto"**.
 - Esempio: Un Motore è parte di un Automobile (se l'automobile viene distrutta, anche il motore viene distrutto).
 - Notazione: Automobile  ————— Motore

esempio segmento composto da due punti (inizio/fine)

- **Aggregazione:** Rappresenta una **relazione "parte-tutto" più debole** rispetto alla composizione. È rappresentata da un rombo vuoto.
 - Esempio: Una Biblioteca ha molti Libri, ma i libri possono esistere indipendentemente dalla biblioteca.
 - Notazione: Biblioteca  ————— Libro

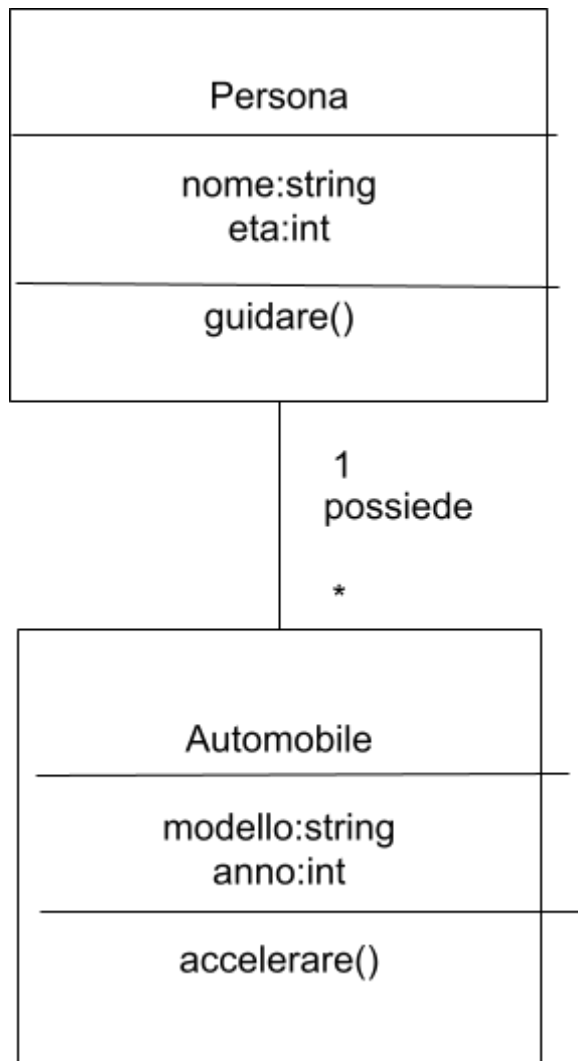
Visibilità:

- La visibilità di attributi e metodi può essere rappresentata con simboli:
 - + pubblico (public) → visibile a tutti.
 - - privato (private) → visibile solo all'interno della classe.
 - # protetto (protected) → visibile alla classe e alle sue sottoclassi.

Esempio di Diagramma delle Classi

Supponiamo di avere un semplice sistema che rappresenta persone, veicoli e relazioni tra loro. Potrebbe esserci una relazione di possesso tra una persona e una o più automobili.

Il diagramma delle classi potrebbe apparire così:



Da questo diagramma capiamo che:

- Persona ha due attributi, nome e età, e un metodo guidare().
- Automobile ha due attributi, modello e anno, e un metodo accelerare().
- C'è una relazione di associazione tra Persona e Automobile (una persona può possedere più automobili).
- La relazione è una molteplicità uno-a-molti (1 persona possiede * automobili).

Utilizzo dei Diagrammi delle Classi

1. **Progettazione:** È utilizzato nella fase di progettazione di software per descrivere come le classi si relazionano e interagiscono tra loro. Aiuta a capire la struttura del sistema prima di implementarlo.
2. **Documentazione:** Serve come documentazione per il sistema, rendendo più facile per gli sviluppatori comprendere come il codice è organizzato.

3. **Manutenzione:** Durante lo sviluppo o la manutenzione di un'applicazione, i diagrammi delle classi aiutano a visualizzare rapidamente le relazioni tra le classi e come un cambiamento in una classe può impattare altre classi.

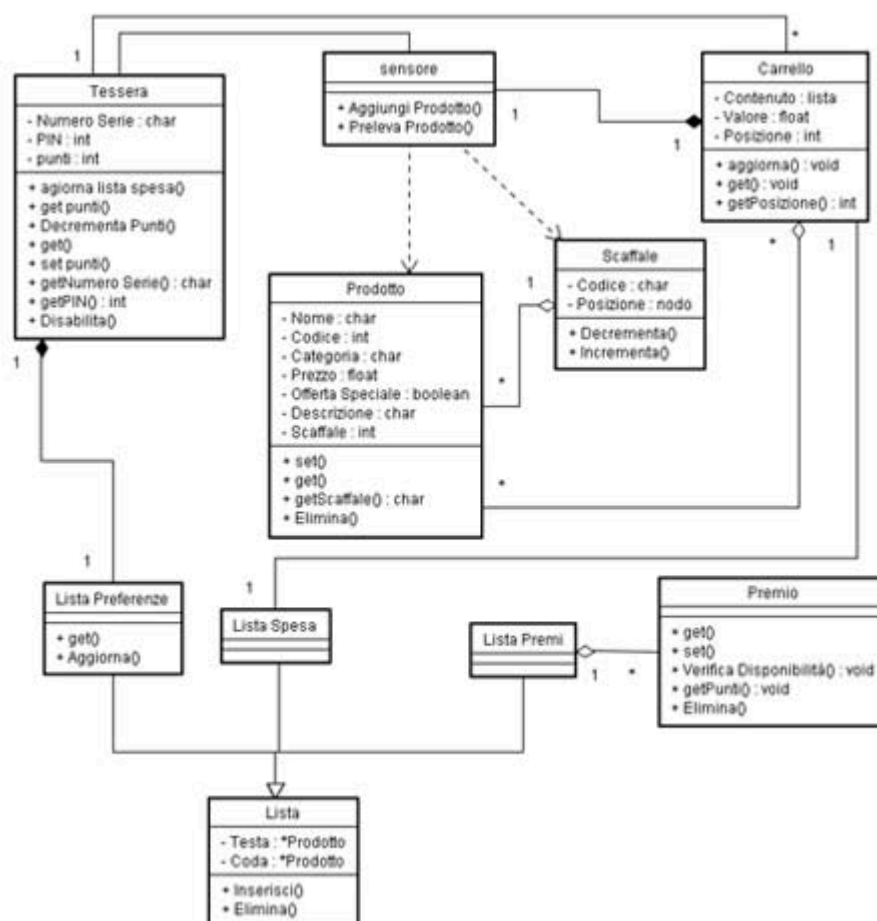
Un diagramma delle classi è quindi una rappresentazione visuale dell'architettura di un sistema software che mostra le classi, i loro attributi, i metodi, e le relazioni tra di esse. E' uno strumento fondamentale per comprendere, progettare e comunicare le strutture di un sistema orientato agli oggetti.

Abbiamo a disposizione vari strumenti per generare un diagramma delle classi a partire dal codice o viceversa generare codice a partire da un diagramma delle classi; di seguito alcuni link utili:

<https://learn.microsoft.com/it-it/visualstudio/ide/class-designer/designing-and-viewing-classes-and-types?view=vs-2022>

<https://support.microsoft.com/it-it/office/creare-un-diagramma-di-classe-uml-de6be927-8a7b-4a79-ae63-90da8f1a8a6b>

<https://learn.microsoft.com/it-it/visualstudio/modeling/understanding-models-classes-and-relationships?view=vs-2022>



Creazione di una Classe in C#

Come abbiamo detto una **classe** è un concetto fondamentale nella programmazione orientata agli oggetti. È un **modello** che definisce le caratteristiche (proprietà) e i comportamenti (metodi) di un tipo di oggetto.

Esempio

Immaginiamo di voler rappresentare una Persona in un programma. Possiamo creare una classe chiamata **Persona** che descrive le proprietà di una persona e i suoi comportamenti.

```
public class Persona
{
    // Attributi (proprietà)

    public string Nome { get; set; }

    public int Eta { get; set; }

    // Metodo per aggiornare l'età

    public void Invecchia()
    {
        Eta += 1; // Aumenta l'età di 1
    }

    // Metodo per ottenere una descrizione della persona

    public string Descrizione()
    {
        return $"Nome: {Nome}, Età: {Eta}";
    }
}
```

Creazione di un Oggetto (Istanza)

Una volta definita la classe, possiamo creare **oggetti** da essa. Questi oggetti sono **istanze** della classe, cioè copie reali basate su quel modello.

Esempio di creazione di un oggetto **Persona** in C#:

```
// Creiamo un'istanza della classe Persona
Persona personal = new Persona();

// Assegniamo i valori agli attributi
personal.Nome = "Mario";
personal.Eta = 30;

// Chiamiamo il metodo Descrizione
string descrizione = personal.Descrizione();
```

Attributi

Un **attributo** è una variabile che viene definita all'interno di una **classe** e descrive le proprietà o caratteristiche degli oggetti che derivano da quella classe.

Gli attributi possono essere di vario tipo, ad esempio numerici, booleani, stringhe o oggetti complessi. Gli attributi sono spesso dichiarati come variabili all'interno della classe e rappresentano lo stato dell'oggetto.

Ad esempio, in una classe "Persona," gli attributi potrebbero includere "nome," "età," "sesso," ecc. Gli **attributi** descrivono **cosa** è un oggetto (nome, età, ecc.).

I **metodi** descrivono **cosa può fare** un oggetto (camminare, parlare, ecc.).

Ogni attributo ha un nome che deve essere significativo, un tipo che rappresenta quali dati può contenere ed una visibilità.

PRINCIPIO GENERALE: l'attributo è privato perché è pericoloso lasciare che dall'esterno si possa cambiare il suo valore senza controllarne la correttezza ed accettabilità

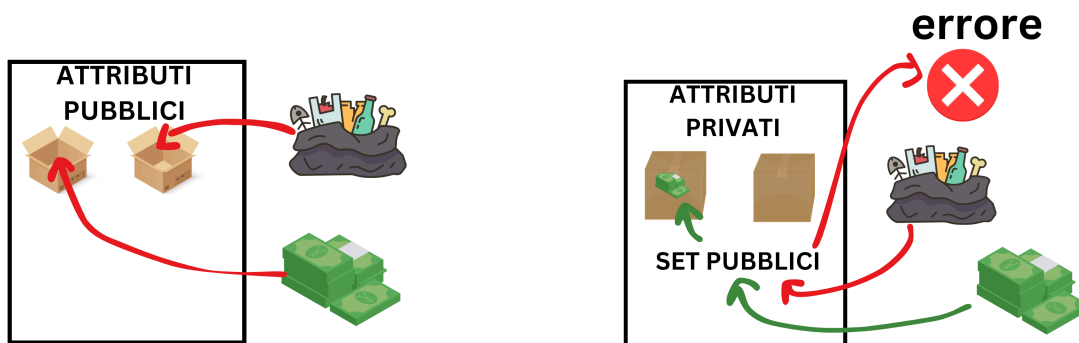
Proprietà

Le **proprietà** in C# sono un costrutto molto importante per il controllo e l'incapsulamento dei campi di una classe. Offrono un modo per esporre dati agli utenti della classe mantenendo il controllo

sull'accesso e la modifica di tali dati. Le proprietà forniscono un'interfaccia per accedere e modificare gli attributi (campi) di un oggetto in modo controllato e incapsulato

Set e Get solitamente sono pubblici, ma possono anche essere privati. Ad esempio il set è privato se il dato deve essere di sola lettura.

Il set conterrà la logica di correttezza e di robustezza del dato, verranno quindi inseriti al suo interno i controlli che impediscono di trovarci in situazioni di inconsistenza.



Tipi Di Proprietà

Proprietà Automatiche (Auto-implemented Properties)

Le proprietà automatiche sono una sintassi semplificata che permette di dichiarare una proprietà senza dover definire un campo di supporto manuale. Il compilatore si occupa di creare un campo privato nascosto per te.

Esempio:

```
public class Person
{
    public string Name { get; set; } // Proprietà automatica
}
```

In questo esempio, C# crea automaticamente un campo nascosto che memorizza il valore di **Name**. Questa sintassi è utile quando non hai bisogno di aggiungere logica personalizzata per leggere o scrivere il valore della proprietà.

Proprietà con logica personalizzata (Full Properties)

Se hai bisogno di aggiungere logica personalizzata quando un valore viene letto o scritto, puoi definire una **proprietà completa** con metodi **get** e **set** espliciti.

Esempio:

```
public class Person
{
    private string _name;

    public string Name
    {
        get
        {
            return _name;
        }

        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
        }
    }
}
```

Qui il metodo **set** verifica che il valore assegnato non sia vuoto o nullo prima di impostarlo. Questo è un vantaggio delle proprietà rispetto ai campi pubblici, poiché consente di controllare l'accesso e la modifica dei dati.

Proprietà con solo get o set

Puoi creare proprietà che siano **sola lettura** o **sola scrittura**.

- **Proprietà di sola lettura:** permettono solo la lettura del valore, ma non la sua modifica dall'esterno della classe. Spesso vengono usate per valori che devono essere impostati solo internamente alla classe (es. nel costruttore) e non devono essere modificati successivamente.

Esempio:

```
public class Person
{
    public string Name { get; private set; } // La proprietà può
    essere impostata solo dall'interno della classe

    public Person(string name)
    {
        Name = name; // Imposta il valore tramite il costruttore
    }
}
```

- **Proprietà di sola scrittura:** sono meno comuni, ma si possono creare nel caso in cui vuoi limitare la lettura del valore dall'esterno.

Esempio:

```
private string _password;

public string Password
{
    set { _password = HashPassword(value); } // Solo scrittura
}
```

Proprietà con espressioni lambda (Expression-bodied Properties)

C# permette una sintassi più concisa per le proprietà che eseguono operazioni semplici usando le **expression-bodied members**. Questo è utile per proprietà che non richiedono un campo di supporto complesso.

Esempio:

```
public class Circle
{
    public double Radius { get; set; }
```

```
        public double Area => Math.PI * Radius * Radius;    // Proprietà
di sola lettura
    }
```

Qui, la proprietà **Area** è una proprietà calcolata che restituisce il valore dell'area del cerchio. Non è necessario un campo privato, perché il valore è sempre calcolato dinamicamente quando viene chiamato il getter.

Proprietà con valore predefinito

In C#, è possibile assegnare un **valore predefinito** a una proprietà automatica. Questo consente di inizializzare una proprietà al momento della dichiarazione.

Esempio:

```
public class Car
{
    public string Model { get; set; } = "Unknown";    // Valore
predefinito
}
```

In questo caso, se il valore della proprietà Model non viene impostato esplicitamente, assumerà il valore "Unknown".

Proprietà Readonly con inizializzazione nel costruttore

Una proprietà può essere dichiarata **readonly**, nel senso che può essere impostata solo durante la costruzione di un oggetto o nell'inizializzazione.

Esempio:

```
public class Person
{
    public string Name { get; }

    public Person(string name)
    {
        Name = name;    // Il valore è impostato nel costruttore e
non può essere modificato successivamente
    }
}
```



```
    }  
}
```

In questo caso, la proprietà Name può essere impostata solo una volta, nel costruttore, e non può essere modificata in seguito.

Proprietà Indexer

Le proprietà **indexer** permettono di accedere agli oggetti di una classe come se fossero array, usando gli indici.

Esempio:

```
public class MyCollection  
{  
    private int[] numbers = new int[10];  
  
    public int this[int index]  
    {  
        get { return numbers[index]; }  
        set { numbers[index] = value; }  
    }  
}
```

In questo modo è possibile accedere agli elementi di MyCollection come faresti con un array:

```
MyCollection collection = new MyCollection();  
collection[0] = 10; // chiama il set sull'indice 0 di numbers  
Console.WriteLine(collection[0]); // Output: 10
```

Proprietà con modificatori di accesso diversi

È possibile avere un **modificatore di accesso diverso per i metodi get e set**. Ad esempio, potresti voler esporre la proprietà in sola lettura all'esterno della classe, ma consentire la modifica internamente.

Esempio:

```
public class BankAccount
{
    public decimal Balance { get; private set; } // 'get'
    pubblico, 'set' privato

    public void Deposit(decimal amount)
    {
        Balance += amount; // Il saldo può essere modificato solo
        tramite metodi interni
    }
}
```

In questo caso, l'esterno della classe può leggere il saldo (Balance), ma non può modificarlo direttamente. Solo i metodi della classe stessa possono cambiare il valore della proprietà.

Proprietà statiche

Le proprietà possono essere **statiche**, il che significa che appartengono alla classe anziché a un'istanza specifica.

Esempio:

```
public class Configuration
{
    public static string ApplicationName { get; set; } = "MyApp";
}
```

In questo caso, puoi accedere alla proprietà ApplicationName senza istanziare un oggetto della classe Configuration.

Le proprietà sono essenziali per gestire l'accesso ai dati in modo sicuro e flessibile, mantenendo la classe aderente ai principi dell'OOP (incapsulamento).

RIFERIMENTO ALLA GUIDA MICROSOFT:

<https://learn.microsoft.com/it-it/dotnet/csharp/programming-guide/classes-and-structs/properties>
qui sotto un estratto di quanto riportato nella guida:

Una proprietà è un membro che fornisce un meccanismo flessibile per leggere, scrivere o calcolare il valore di un campo privato. Le proprietà possono essere usate come se fossero

membri dati pubblici, ma sono metodi speciali denominati *funzioni di accesso*. Questa funzionalità consente l'accesso ai dati facilmente e contribuisce comunque a promuovere la sicurezza e la flessibilità dei metodi.

- Le proprietà consentono a una classe di esporre un modo pubblico per ottenere e impostare i valori, nascondendo però il codice di implementazione o di verifica.
- Una funzione di accesso della proprietà `get` viene usata per restituire il valore della proprietà, mentre una funzione di accesso della proprietà `set` viene usata per assegnare un nuovo valore. In C# 9 e versioni successive viene usata una funzione di accesso alle proprietà `init` per assegnare un nuovo valore solo durante la costruzione di oggetti.
- La parola chiave `value` viene usata per definire il valore assegnato dalla `set` funzione di accesso o `init`.
- Le proprietà possono essere di *lettura/scrittura* con entrambe le funzione di accesso `get` e `set`, di *sola lettura* con la funzione di accesso `get` e senza la funzione di accesso `set` o di *sola scrittura* con la funzione di accesso `set` e senza la funzione di accesso `get`. Le proprietà di sola scrittura sono rare e vengono in genere usate per limitare l'accesso ai dati sensibili.
- Le proprietà semplici che non richiedono alcun codice di funzione di accesso personalizzata possono essere implementate come definizioni del corpo dell'espressione o come [proprietà implementate automaticamente](#).

Un modello di base per l'implementazione di una proprietà prevede l'uso di un campo sottostante privato per l'impostazione e il recupero del valore della proprietà. La funzione di accesso `get` restituisce il valore del campo privato e la funzione di accesso `set` può eseguire una convalida dei dati prima di assegnare un valore al campo privato. Entrambe le funzioni di accesso possono anche eseguire alcune conversioni o calcoli sui dati prima che vengano archiviate o restituite.

L'esempio seguente illustra il modello. Nell'esempio la classe `TimePeriod` rappresenta un intervallo di tempo. Internamente la classe archivia l'intervallo di tempo in secondi in un campo privato denominato `_seconds`. Una proprietà di lettura/scrittura denominata `Hours` consente al cliente di specificare l'intervallo di tempo in ore. Entrambe le funzioni di accesso `get` e `set` eseguono la conversione necessaria tra ore e secondi. Inoltre, la funzione di accesso `set` convalida i dati e genera [ArgumentOutOfRangeException](#) se il numero di ore non è valido.

```
public class TimePeriod
{
```

```

private double _seconds;

public double Hours
{
    get { return _seconds / 3600; }

    set
    {
        if (value < 0 || value > 24)
            throw new ArgumentOutOfRangeException(nameof(value),
                "The valid range is between 0 and 24.");

        _seconds = value * 3600;
    }
}

```

È possibile accedere alle proprietà per ottenere e impostare il valore come illustrato nell'esempio seguente:

```

TimePeriod t = new TimePeriod();

// The property assignment causes the 'set' accessor to be called.
t.Hours = 24;

// Retrieving the property causes the 'get' accessor to be called.
Console.WriteLine($"Time in hours: {t.Hours}");

// The example displays the following output:
//  Time in hours: 24

```

Sia la `get` funzione di accesso che la `set` funzione di accesso possono essere implementate come membri con corpo di espressione. In questo caso, è necessario che siano presenti le parole chiave `get` e `set`. L'esempio seguente illustra l'uso di definizioni del corpo dell'espressione per entrambe le funzioni di accesso. La `return` parola chiave non viene usata con la `get` funzione di accesso.

```
public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}
```

```
}  
  
}
```

In alcuni casi, le funzioni di accesso e `set` proprietà `get` assegnano semplicemente un valore a o recuperano un valore da un campo sottostante senza includere alcuna logica aggiuntiva. Usando le proprietà implementate automaticamente è possibile semplificare il codice e fare in modo che il compilatore C# specifichi automaticamente in modo trasparente il campo sottostante.

Se una proprietà ha sia una `getset` funzione di accesso (o e) `init` che una `get` funzione di accesso , entrambi devono essere implementati automaticamente. È possibile definire una proprietà implementata automaticamente usando le parole chiave `get` e `set` senza specificare alcuna implementazione. L'esempio seguente ripete l'esempio precedente, ad eccezione del fatto che `Name` e `Price` sono proprietà implementate automaticamente. Nell'esempio viene rimosso anche il costruttore con parametri, in modo che `SaleItem` gli oggetti vengano ora inizializzati con una chiamata al costruttore senza parametri e a un [inizializzatore di oggetti](#).

```
public class SaleItem  
{  
  
    public string Name  
    { get; set; }  
  
    public decimal Price  
    { get; set; }  
}
```

A partire da C# 11, è possibile aggiungere il `required` membro per imporre al codice client di inizializzare qualsiasi proprietà o campo:

```
public class SaleItem  
{  
  
    public required string Name  
    { get; set; }  
}
```

```
public required decimal Price

{ get; set; }

}
```

Per creare un `SaleItem` oggetto , è necessario impostare le `Name` proprietà e `Price` usando gli [inizializzatori di oggetto](#), come illustrato nel codice seguente:

```
SaleItem item = new SaleItem { Name = "Shoes", Price = 19.95m };
```

```
Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
```

Metodi

Un **metodo** è una funzione definita all'interno di una classe che descrive un comportamento o un'azione che gli oggetti di quella classe possono eseguire.

Un metodo rappresenta **cosa può fare un oggetto**. È un insieme di istruzioni che permette a un oggetto di compiere determinate operazioni o azioni.

Esempio:

```
public class Persona
{
    // Attributi della classe Persona

    public string Nome { get; set; }

    public int Eta { get; set; }


    // Metodo per far invecchiare la persona
    public void Invecchia()
    {
        Eta += 1; // Aumenta l'età di 1
    }
}
```

Caratteristiche di un metodo:

- **Comportamento:** definisce un'azione che un oggetto può compiere.
- Operazioni sugli attributi: può modificare lo stato interno di un oggetto, ad esempio, il metodo `Invecchia()` aumenta l'età della persona modificando l'attributo `Eta`.
- Ritorno di valori: può restituire un valore. Se un metodo non restituisce nulla, è dichiarato come `void`, ma può anche restituire un tipo specifico di dato, come `int`, `string`, ecc.
- **Parametri:** I metodi possono ricevere parametri che permettono di personalizzare il loro comportamento.
Ad esempio:

```
public void Boost(int aumento)

{

    velocità += aumento

}
```

- Polimorfismo: i metodi possono essere sovrascritti (**overriding**) o sovraccaricati (**overloading**). Metodi con lo stesso nome possono comportarsi in modo diverso a seconda dei parametri o della classe da cui sono chiamati.

Ogni metodo ha un nome che deve essere significativo, un tipo di ritorno che rappresenta quale tipo di dato restituisce (`void` se non restituisce nulla), una lista di parametri (che può anche essere vuota) che rappresentano i dati da passare come input al metodo affinché questo possa svolgere il suo compito ed una visibilità.

I metodi sono uno dei principali strumenti di incapsulamento in OOP, poiché consentono di definire il comportamento di un oggetto o di una classe, nascondendo i dettagli di implementazione ai clienti dell'oggetto.

I metodi forniscono anche un modo per garantire che le operazioni siano eseguite in modo coerente e prevedibile per gli oggetti della classe.

Costruttore

Un "costruttore" è un metodo speciale all'interno di una classe che viene chiamato automaticamente quando si crea un nuovo oggetto di una classe. Il costruttore è utilizzato per inizializzare l'oggetto, impostare i valori iniziali dei suoi attributi ed eseguire le operazioni necessarie per preparare l'oggetto all'uso.

Caratteristiche del costruttore

- Il costruttore ha lo stesso nome della classe in cui è definito.

- I costruttori non hanno un tipo di ritorno (nemmeno **void**).
- Viene chiamato automaticamente quando si crea un'istanza della classe. Non è necessario chiamarlo esplicitamente.
- Viene utilizzato per inizializzare gli attributi dell'oggetto e per eseguire altre operazioni necessarie all'istanziamento dell'oggetto.
- I costruttori possono accettare parametri che vengono utilizzati per inizializzare gli attributi dell'oggetto
- Se non viene definito ne esiste uno di default (senza parametri), il cui compito è costruire l'oggetto ed assegnare ai suoi attributi i valori di default previsti per i tipi di dato (esempio i dati numerici vengono impostati a 0).
- Se viene definito uno o più costruttori non è più possibile utilizzare il costruttore di default (se non ridefinendolo esplicitamente)
- Il costruttore è sempre **public** (per ovvie ragioni, altrimenti non sarebbe possibile istanziare un oggetto)

Possono essere definiti più costruttori attraverso l'**overload dei parametri**, ovvero attraverso la definizione di un costruttore con diversa lista di parametri formali (**attenzione** quello che differenzia una lista di parametri non è il nome dei parametri ma il loro tipo quindi per esempio:

```
public Aula(int lunghezza, int larghezza)
```

e

```
public Aula(int lunghezza, int altezza)
```

dal compilatore vengono visti come due costruttori uguali perchè hanno la stessa lista di parametri, entrambi si aspettano due dati interi, non importa come li chiamo.)

Esempio:

```
public class Prodotto
{
    public string Nome { get; set; }
    public double Prezzo { get; set; }

    // Costruttore 1: Senza parametri
    public Prodotto()
    {
```

```

        Nome = "Prodotto Sconosciuto";

        Prezzo = 0.0;
    }

    // Costruttore 2: Con parametri per nome e prezzo
    public Prodotto(string nome, double prezzo)
    {
        Nome = nome;

        Prezzo = prezzo;
    }
}

```

Il primo costruttore è senza parametri e viene utilizzato quando si crea un oggetto Prodotto senza specificare un nome o un prezzo. In questo caso, il prodotto viene inizializzato con un nome predefinito ("Prodotto Sconosciuto") e un prezzo predefinito (0.0).

```
Prodotto prodotto1 = new Prodotto(); // Usa il costruttore senza parametri
```

Il secondo costruttore accetta due parametri, nome e prezzo, che vengono utilizzati per inizializzare il nome e il prezzo del prodotto quando si crea un oggetto.

Questo consente di specificare i dettagli del prodotto al momento della creazione.

```
Prodotto prodotto2 = new Prodotto("Prodotto A", 19.99); // Usa il costruttore con parametri
```

RICORDA:

se ho il metodo

```
public void metodo(int a, int b)
```

questi sono esempi di **overload**:

- public void metodo(double a, double b) (cambia il tipo dei parametri)
- public void metodo(int a, int b, int c) (cambia il numero di parametri)

questi **NON** sono **overload**

- public int metodo(int a, int b) NON è overload perché cambia solo il tipo di ritorno
- public void metodo(int c, int b) NON è overload perché cambia solo il nome e non il tipo del parametro

Cosa Fare Prima di Scrivere Codice

Scrivere un software è un processo non banale che richiede attenzione ed un approccio ragionato e non casuale che procede per passi.

1. Definizione dei requisiti:

Prima di iniziare a sviluppare è necessario assicurarsi di aver compreso chiaramente i requisiti del software e le esigenze degli utilizzatori. Risulta importante confrontarsi con gli stakeholder per definire e documentare i requisiti in modo completo e preciso prima di iniziare il processo di sviluppo al fine di evitare di sprecare lavoro.

2. Progettazione:

Prima di scrivere codice è necessario fare una fase di progettazione in cui definire l'architettura, la struttura e le interfacce del sistema. In questa fase è importante suddividere il software in moduli o componenti in modo da facilitare la manutenzione e la scalabilità.

3. Codifica:

Scrivere il codice in modo modulare, diviso in funzioni e/o classi, ciascuna con una responsabilità specifica.

Applicare il principio "Single Responsibility Principle" (SRP) in modo che ogni classe abbia una sola responsabilità.

Seguire uno stile di codifica coerente e che rispetti le code convention.

Documentare in modo opportuno il codice nel mentre che lo si sta creando.

Utilizza un sistema di controllo del codice sorgente (come Git) per tenere traccia delle modifiche e collaborare con altri sviluppatori. Usa il branching e il merging per gestire le modifiche e le nuove funzionalità in modo strutturato.

Tratta gli errori in modo adeguato, fornendo messaggi di errore significativi agli utenti e registrando gli errori nei log.

Implementa meccanismi di gestione delle eccezioni per garantire che l'applicazione non si blocchi in caso di errori.

4. Test

Scrivere test unitari e test di integrazione per verificare che il software funzioni correttamente.

Eseguire i test dopo le modifiche per verificare che le modifiche non abbiano compromesso comportamenti corretti preesistenti.

Linguaggio C#

Tipi di dati

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/language-specification/types>

Principali tipi di dati in C#:

Tipi di dati primitivi:

int: Interi con segno a 32 bit.

long: Interi con segno a 64 bit.

float: Numeri in virgola mobile a 32 bit.

double: Numeri in virgola mobile a 64 bit (doppia precisione).

decimal: Numeri decimali ad alta precisione.

char: Carattere Unicode a 16 bit.

bool: Tipo booleano (vero o falso).

byte: Byte non con segno a 8 bit.

sbyte: Byte con segno a 8 bit.

Tipi di dati composti:

string: Stringhe di testo.

object: Tipo di dati di base da cui derivano tutti gli altri tipi (tipo di dati universale).

enum: Enumerazioni, una serie di costanti denominate.

struct: Strutture, simili alle classi ma con tipo valore.

class: Classi, tipi di dati di riferimento utilizzati per definire oggetti.

interface: Interfacce, contratti che le classi possono implementare.

Tipi di dati con notazione speciale:

var: Utilizzato per dichiarare variabili con tipo dedotto automaticamente dal compilatore (conoscenza a partire da C# 3.0).

dynamic: Utilizzato per dichiarare variabili con tipizzazione dinamica (conoscenza a partire da C# 4.0).

Tipi di dati per dati nullable:

int?: Tipi di dati nullable (ad esempio, int? può contenere un intero o un valore nullo).

Tipi di dati per collezioni:

List<T>: Una lista generica.

Dictionary<TKey, TValue>: Un dizionario generico.

Array: Un array unidimensionale o multidimensionale.

Altri tipi di dati specializzati:

DateTime: Data e ora.

TimeSpan: Intervallo di tempo.

Guid: Identificatore univoco globale.

Questi sono alcuni dei tipi di dati fondamentali in C#.

Ogni tipo di dato ha le proprie caratteristiche e utilizzi specifici.

La scelta del tipo di dati dipenderà dal contesto e dai requisiti specifici del tuo programma.

Istruzioni in C#

Regole di sintassi

Le regole di base di sintassi in C# sono fondamentali per scrivere codice C# correttamente.

Principali regole di sintassi in C#:

Nomi delle variabili e delle classi:

I nomi di variabili e classi sono sensibili alle maiuscole e minuscole.

I nomi possono contenere lettere, cifre e il carattere di sottolineatura (_).

Un nome deve iniziare con una lettera o il carattere di sottolineatura (_), non con una cifra.

Punto e virgola (;):

Ogni dichiarazione e istruzione C# deve terminare con un punto e virgola (;).

Parentesi graffe {}:

Le parentesi graffe vengono utilizzate per definire blocchi di codice, come il corpo di una classe, metodo o istruzione condizionale.

I blocchi di codice devono essere aperti con { e chiusi con }.

Commenti:

I commenti sono preceduti da // per i commenti in linea o racchiusi tra /* e */ per i commenti su più righe.

I commenti sono utilizzati per documentare il codice e rendere più leggibile il programma.

Dichiarazione di variabili:

Le variabili devono essere dichiarate prima di essere utilizzate.

La dichiarazione di variabili segue la sintassi tipo nomeVariabile;

Tipi di dati:

I tipi di dati specificano il tipo di valore che una variabile può contenere (es. int, string, double, ecc.).

I tipi di dati sono sensibili alle maiuscole e minuscole.

Assegnazione di variabili:

Per assegnare un valore a una variabile, usa l'operatore di assegnazione = (es. variabile = valore;).

Stringhe:

Le stringhe sono racchiuse tra virgolette doppie (es. "questa è una stringa").

Operatori:

C# supporta vari operatori come +, -, *, /, ==, !=, % ed altri.

Condizioni:

Per le istruzioni condizionali, come if e switch, usa parentesi tonde () per definire le condizioni.

Per i cicli, come for, while, e do-while, usa parentesi tonde () per definire le condizioni del ciclo e le parentesi graffe {} per il corpo del ciclo.

Array:

Per dichiarare e utilizzare array, usa parentesi quadre [] (es. `int[] numeri = new int[5];`).

Metodi e funzioni:

I metodi devono essere dichiarati con una sintassi specifica:

`tipoRitorno NomeMetodo(parametri) { /* corpo del metodo */ }`.

Classe:

La dichiarazione di una classe inizia con la parola chiave `class`, seguita dal nome della classe e le parentesi graffe contenenti il corpo della classe.

Costruttori:

I costruttori di una classe hanno lo stesso nome della classe e vengono utilizzati per inizializzare gli oggetti.

Modificatori di accesso:

Usa modificatori di accesso come `public`, `private`, `protected`, `internal`, ecc., per controllare l'accesso alle variabili e ai metodi.

Case sensitivity:

C# è sensibile alle maiuscole e minuscole, quindi `variabile` e `Variabile` sono considerate diverse.

Parole chiave:

Alcune parole chiave, come `class`, `int`, `if`, ecc., hanno significati specifici in C# e non possono essere utilizzate come nomi di variabili o metodi.

Costrutti di Selezione

IF

ogni volta che ho necessità di **verificare una condizione ed eseguire operazioni diverse a seconda che la condizione sia vera o falsa** ho bisogno di una selezione

`if(<condizione da verificare>)`

`{`

`//blocco di istruzioni da eseguire se la condizione è vera`

`}`

```
else{  
  
//il blocco di istruzioni da eseguire se la condizione è falsa  
  
}
```

esempio:

```
if (numero > 5)  
{  
    Console.WriteLine("Il numero è maggiore di 5.");  
}
```

esempio:

```
if (numero > 5)  
{  
    Console.WriteLine("Il numero è maggiore di 5.");  
}  
else  
{  
    Console.WriteLine("Il numero non è maggiore di 5.");  
}
```

RICORDA:

- la parte di else è facoltativa
- di if ne posso avere più di uno anche uno dopo l'altro o uno dentro l'altro
- con l'operatore = assegno un valore con l'operatore == confronto due valori

Operatore Ternario

L'**operatore ternario** è una forma concisa per una istruzione if-else che permette di scegliere tra due valori a seconda di una condizione.

La sintassi dell'operatore ternario è:

condizione ? valore_se_vero : valore_se_falso;

- condizione: l'espressione booleana che viene valutata.
- valore_se_vero: Il valore da restituire se la condizione è vera.
- valore_se_falso: Il valore da restituire se la condizione è falsa.

Esempio

Supponiamo di voler determinare se una persona è maggiorenne o meno in base alla sua età. Possiamo usare l'operatore ternario per semplificare il codice:


```

if(eta >= 18)
    stato = "Maggiorenne"
else
    stato = "Minorenne"

string stato = eta >= 18 ? "Maggiorenne" : "Minorenne";

```

Condizione: `eta >= 18` verifica se l'età è maggiore o uguale a 18.

Valore se vero: "Maggiorenne" viene scelto se la condizione è vera.

Valore se falso: "Minorenne" viene scelto se la condizione è falsa.

L'operatore ternario consente di assegnare direttamente il valore stato basato sulla condizione in un'unica riga, rendendo il codice più compatto e leggibile rispetto all'uso di un costrutto if-else tradizionale. L'operatore ternario è particolarmente utile quando la logica condizionale è semplice e il codice risulta più chiaro in forma compatta.

SWITCH

(<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/statements/selection-statements>)

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/language-specification/statements#1383-the-switch-statement>

)

un'istruzione che mi permette di scrivere in modo più veloce una serie di if che dipendono tutti dal test sullo stesso valore

esempio

se il valore del dato **n** è 1 fai questa cosa

se il valore del dato **n** è 2 fai questa cosa

se il valore del dato **n** è 5 fai questa cosa

se il valore del dato **n** è 12 fai questa cosa

```

if (espressione == valore1)
{
    // Codice da eseguire se l'espressione è uguale a valore1
}

```

```

}
else if (espressione == valore2)
{
    // Codice da eseguire se l'espressione è uguale a valore2
}
// Altri casi...
else
{
    // Codice da eseguire se nessun caso corrisponde
}

```

posso scriverlo in modo più compatto in questo modo

```

switch (espressione)
{
    case valore1:
        // Codice da eseguire se l'espressione è uguale a valore1
        break;
    case valore2:
        // Codice da eseguire se l'espressione è uguale a valore2
        break;
    // Altri casi...
    default:
        // Codice da eseguire se nessun caso corrisponde
        break;
}

```

L'espressione tra parentesi tonde (espressione) viene valutata.

Il valore risultante dell'espressione viene confrontato con ciascun caso (case valore1, case valore2, ecc.).

Se il valore dell'espressione corrisponde a uno dei casi, il blocco di codice corrispondente a quel caso viene eseguito.

L'istruzione break viene utilizzata per uscire dal blocco switch dopo l'esecuzione del caso corrispondente.

Se nessun caso corrisponde al valore dell'espressione, viene eseguito il blocco **default**, se presente.

il default è opzionale e deve sempre essere l'ultima opzione

esempio:

```
switch (scelta) {  
  
    case 1: Console.WriteLine("Hai scelto la prima opzione.");  
        break;  
  
    case 2: Console.WriteLine("Hai scelto la seconda opzione.");  
        break;  
  
    case 3: Console.WriteLine("Hai scelto la terza opzione.");  
        break;  
  
    default: Console.WriteLine("Opzione non valida.");  
}  

```

Operatori Booleani

AND (&&)

due condizioni legate dall'operatore and risultano vere se e solo se entrambe sono vere

```
if(a>b && b>c)  
{  
    //questo blocco viene eseguito solo se entrambe sono vere  
    (p.e a= 5, b=3, c=1)  
}  
  
else{
```

```

        //questo blocco viene eseguito se almeno una delle condizioni
è falsa
        (p.e b=1, c=2) (la seconda condizione è falsa)

        (p.e a= 0, b=1, c=0) (la prima condizione è falsa e la
seconda è vera)

        (p.e a= 0, b=1, c=2) (entrambe le condizioni sono false)
}

```

OR (||)

due condizioni legate dall'operatore or risultano vere se almeno una delle due è vera

```

if(a>b || b>c)
{
    //questo blocco viene eseguito se almeno una è vera

    (p.e a= 5, b=3) (la prima è vera)

    (p.e a= 5, b=8, c = 1) (la prima è falsa ma la seconda è
vera)

    (p.e a= 9, b=8, c = 2) (entrambe sono vere)
}

else{

    //questo blocco viene eseguito se entrambe le condizioni sono
false

    (p.e a= 0, b=1, c=0) (entrambe le condizioni sono false)
}

```

NOT (!)

verifica se una condizione è false

(spesso è utilizzato in combinazione con l'uguale != per formare il simbolo diverso)

```

if(a!=b)
{

```

```

        //eseguo questo codice solo se a e b hanno valori diversi
    }

    if(! (a==b && c!= d))
    {
        //eseguo questo codice se il risultato di questa espressione
        (a==b && c!= d) è false
    }

```

Costrutti Iterativi

(<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements>)

DO-WHILE (ciclo in coda)

lo utilizzo ogni volta che devo ripetere delle istruzioni e non so quante volte devo ripetere, ma sono certo di ripetere almeno una volta e comunque prima voglio eseguire le istruzioni e poi verificare se ripetere le istruzioni

Sintassi:

```

do{

    //istruzioni da ripetere finché la condizione è vera

}while(<condizione>);

```

Esempio:

ripeto la chiamata a Lancio() finché non mi viene restituito il valore 1

- **almeno 1 volta devo fare il lancio**

- **non posso sapere per quante volte devo ripetere**

```

do
{
    //lancia il dado
    risultatoLancio = dado1.Lancio();

} while (risultatoLancio != 1);

```

WHILE (ciclo in testa)

Utilizzo il ciclo while ogni volta che devo ripetere delle istruzioni finché una condizione è vera, ma non so quante volte devo ripetere.

Potrei anche non ripetere mai perchè prima eseguire le istruzioni mi pongo la condizione, quindi se la condizione è subito falsa non eseguo mai le istruzioni.

```
while (condizione)
{
    // Blocco di codice da eseguire
}
```

condizione: Un'espressione booleana che viene valutata prima di ogni iterazione. Se è vera, il blocco di codice viene eseguito. Se è falsa, il ciclo termina.

- **non so quante volte devo ripetere**
- **potrei anche non dovere eseguire mai il codice nel ciclo**

```
int n = rnd.Next(); //numero random
int contatore=0;
// Ciclo while si esegue finché n è minore o uguale a 5
while(n <= 5)
{
    // Incrementa il contatore
    contatore++;
}
```

FOR

Utilizzo il ciclo for **se devo ripetere delle istruzioni per un numero determinato di volte**

Esempio:

```
//voglio lanciare il dado 5 volte
int somma = 0

for(int i = 0; i<5; i+=1)//ripetere 5 volte
{
    //lancio il dado e sommo i valori
    somma += dado1.Lancio();
}
```

il for si aspetta 3 valori:

```
for(int i = 0; i<5; i++)
```

`int i = 0;` → definizione ed inizializzazione dell'iteratore (viene eseguito solo la prima volta che entro nel ciclo)

`i<5;` → condizione di permanenza nel ciclo (finché la condizione è vera ripeto)

`i++` → passo (l'operazione che viene eseguita ad ogni iterazione)

Se ho più inizializzazioni o più passi da eseguire questi vengono separati da **virgola**

Esempio

```
int estrazione = 0;
```

```
    for (int i = 0, conta5=0; i < 5; i++)//ripetere 5 volte
    {
        estrazione = dado1.Lancio();
        if(estrazione == 5)
        {
            //aggiungo 1 al valore di conta5
            conta5+=1;

            /* modi equivalenti di scrivere la stessa istruzione
            conta5 ++;
            conta5 = conta5+1;
            */
        }
        Console.WriteLine(estrazione);
        Console.WriteLine("il valore 5 al momento è uscito " + conta5 + " volte");
    }
```

Se ho più condizioni da testare vengono unite dagli **operatori booleani** (&& o || o ..)

Esempio

```
int estrazione = 0;

for (int i = 0; i < 5 && estrazione != 1; i++)//ripetere 5 volte
{
    estrazione = dado1.Lancio();
    Console.WriteLine(estrazione);
}
```

FOREACH

Il ciclo foreach fornisce un modo semplice per scorrere gli elementi di una raccolta o di un array di elementi.

Ovviamente prima di utilizzare il ciclo foreach dobbiamo dichiarare l'array (o le raccolte).

```
char[]myLetter = {'a','b','a','c','c','a','b','a','A','f','a'}

//per ogni elemento element di tipo char contenuto nell'array myLetter

foreach(char element in myLetter) //ad ogni iterazione element conterrà l'i-simo valore di myLetter
{
    Console.Write(element);
}
```

ad ogni iterazione element conterrà il valore di una delle posizioni dell'array (ovviamente partendo dalla posizione 0 andando avanti di una in una)

Gestione delle Eccezioni

per rendere robusti i programmi abbiamo necessità di catturare eventuali errori “lanciati” dalle chiamate ai vari metodi, per fare questo si utilizzano i blocchi try catch

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/statements/exception-handling-statements>

<https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/using-standard-exception-types>

I blocchi try-catch in C# sono utilizzati per gestire eccezioni o errori durante l'esecuzione di un programma. Consentono di scrivere codice che può generare eccezioni, e poi catturare e gestire queste eccezioni in modo controllato, evitando che il programma si arresti in modo anomalo.

Come funzionano:

Blocco Try: All'interno di un blocco try, inserisci il codice che potrebbe generare un'eccezione. Quando si verifica un'eccezione all'interno del blocco try, il flusso di controllo viene trasferito al blocco catch corrispondente.

```
try
{
    // Codice che potrebbe generare un'eccezione
}
```

Blocco Catch: Il blocco catch è responsabile della gestione delle eccezioni. Contiene il codice che verrà eseguito quando si verifica un'eccezione all'interno del blocco try. È possibile specificare il tipo di eccezione che si desidera gestire all'interno delle parentesi tonde del blocco catch. Inoltre, puoi avere più blocchi catch per gestire diversi tipi di eccezioni.

```
catch (TipoEccezione specifica)
{
    // Gestione dell'eccezione specifica
}

catch (AltroTipoEccezione specifica)
{
    // Gestione dell'altro tipo di eccezione specifica
}
```

Rilancio di eccezioni: Si può anche decidere di rilanciare un'eccezione dopo averla gestita. Questo è utile quando desideri registrare l'errore o effettuare ulteriori azioni prima di propagare l'eccezione a un livello superiore.

```
catch (TipoEccezione specifica)
{
    // Gestione dell'eccezione specifica
    throw; // Rilancia l'eccezione
}
```

Blocco Finally (opzionale): È possibile utilizzare un blocco finally per contenere il codice che deve essere eseguito in ogni caso, indipendentemente da se si è verificata o meno un'eccezione.

Questo è utile per la pulizia delle risorse o per garantire che determinate azioni siano eseguite indipendentemente dalle eccezioni.

```
finally
{
    // Codice da eseguire sempre, ad esempio, la chiusura di un
    file o una connessione al database.
}
```

Esempio 1

```
try
{
    int numero = 10;
    int divisore = 0;

    int risultato = numero / divisore; // Questo genererà
    un'eccezione di divisione per zero
}

catch (DivideByZeroException ex)
{
    Console.WriteLine("Errore: " + ex.Message);
}
```

Esempio 2:

```
static void Main(string[] args)
{
    try
    {
        // qui dentro inserisco tutto il codice che voglio chiamare e
        // che potrebbe generare un errore

        CoffeeMachine machine = new CoffeeMachine(0.60);

        //tutto il codice dentro al blocco try viene
        //eseguito finchè non c'è un errore

        //se c'è un errore l'esecuzione si sposta al
        //blocco catch saltando tutte le istruzioni dall'errore in poi
    }

    catch (ArgumentException e)
    {
        //il codice che è qui viene eseguito solo se
        //l'errore è di tipo ArgumentException

        Console.WriteLine(e);
    }

    catch (Exception e)
    {
        //il codice che è qui viene eseguito solo se
        //l'errore è di tipo Exception (tutti gli errori sono anche
        //Exception)

        Console.WriteLine(e);
    }

    //il codice scritto qui viene sempre eseguito perché
    //fuori dal blocco try-catch

}
```

Esempio 3

```
bool error;

Player p1, p2;

Dice dice;

//blocco di input con controlli
do {

    error = false;

    try

    {

        Console.WriteLine("inserire in nome del primo giocatore");

        string nome = Console.ReadLine();

        p1 = new Player(nome);

        Console.WriteLine("inserire in nome del secondo giocatore");

        nome = Console.ReadLine();

        p2 = new Player(nome);

        Console.WriteLine("inserire in tipo di dado (numero di facce)");

        int facce = int.Parse(Console.ReadLine());

        dice = new Dice(facce);

    } catch (Exception e) {

        error = true;

    }

}while(error);
```

CREAZIONE DELLA PRIMA CLASSE CON TEST - PROGETTO COFFE MACHINE

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

namespace CoffeMachine
{
    /*
    * la macchina del caffè eroga solo caffè e non da resto
    * il resto rimane a credito
    * dobbiamo conoscere:
    * - il costo del caffè
    * - i soldi inseriti
    * - il credito residuo
    * - se la macchina è accesa (attaccata alla corrente)
    * - se la macchina è pronta (il caffè e l'acqua sono sufficienti)
    * (attenzione non deve essere possibile trovarsi nella situazione
    * in cui la macchina risulta spenta e pronta!!!!)
    */
    //accessibilità(internal) parola chiave class nome della classe (Machine)
    public class Machine
    {
        private int _priceInCent; //private int _priceInCent --> accessibilità (private), tipo (int) nome
        (_priceInCent)
        private int _insertedInCent;
        private int _creditInCent;
        private bool _on;
        private bool _ready;

        /*
        * fun nomefunzione(parametro:tipo):tipo di ritorno
        visibilità tipo_di_ritorno nomefunzione (tipo parametro,...)
        quindi questa fun
        fun somma(n1:Int, n2:Int):Int
        diventa così
        int somma (int n1, int n2)

        se ho un fun che non restituisce nulla
        fun aggiungi(qta:Int)
        void aggiungi(int qta)
        il tipo di ritorno void significa che il metodo non restituisce nulla
        */
    }
}
```

```

/*
 * il costruttore è l'unico metodo che NON ha il tipo di ritorno
 * è sempre pubblico
 * ha lo stesso nome della classe
 * viene invocato ogni volta che devo costruire un oggetto
 *
 * Esiste sempre un costruttore di default (che è il costruttore senza parametri)
 * questo costruttore non può essere chiamato nel caso in cui sia definito un costruttore con
parametri e non esplicitamente un costruttore senza
 */
public Machine(int cost)
{
    //uso la proprietà e non l'attributo perchè così viene richiamato il set
    PriceInCent = cost;
    InsertedInCent = 0;
    CreditInCent = 0;
    _on = true;
    _ready = true;
}

//chiamiamo l'altro costruttore e qui scriviamo solo il codice di interesse di questo caso
//:this(cost) --> la chiamata al costrutto Machine che si aspetta come parametro un intero
public Machine(int cost, int amount):this(cost)
{
    //@todo controllo sul parametro
    _insertedInCent = amount;
}

/*
 * abbiamo applicato il polimorfismo l'overload dei parametri al metodo costruttore
 * ora abbiamo 2 modi di costruire un oggetto Machine
 * 1) costruire l'oggetto con il costo del caffè
 * 2) costruire l'oggetto con il costo del caffè e il credito residuo
 */

//proprietà
/*
/// PriceInCent proprietà
/// con get pubblico con comportamento di default
/// con set privato che controlla che sia stato effettivamente cambiato il valore e che il valore sia
accettabile
--> solo all'interno della classe posso modificare il valore
 */
public int PriceInCent
{
    get { return _priceInCent; }
    private set {
        if (_priceInCent != value && value >=0)

```

```

        {
            _priceInCent = value;
        }
    }
}

public int InsertedInCent
{
    get { return _insertedInCent; }
    private set
    {
        if ( value >=0)
        {
            _insertedInCent = value;
        }
        else
        {
            throw new ArgumentOutOfRangeException("valore inserito errato");
        }
    }
}

public int CreditInCent
{
    get { return _creditInCent; }
    private set
    {
        if(value >=0)
        {
            _creditInCent = value;
        }
    }
}

//campo calcolato
public bool ReadyForCoffe
{
    get
    {
        if (_ready && !_on) throw new Exception("stato macchina non corretto");
        if (_on && _ready) return true;
        return false;
    }
}

/// <summary>
/// aggiungere amount ai centesimi inseriti
/// </summary>
/// <param name="amount"></param>

```

```

public void AddInsertedCent(int amount)
{
    InsertedInCent += amount;
}

/// <summary>
/// impostare a 0 i centesimi inseriti
/// </summary>
public void ResetInsertedCent()
{
    InsertedInCent = 0;
}

public void OffMachine()
{
    _on = false;
}

private void CheckMachine()
{
    //TODO: qui ci va il codice che verifica se la macchinetta è ready ovvero se c'è acqua, caffè,
    temperatura...
    _ready = true;
}

public bool MakeCoffe()
{
    CheckMachine();
    bool result = false;
    //verifico che la macchinetta sia pronta per fare il caffè
    if(ReadyForCoffe)
    {
        //verifico che ci sia credito suff
        if(InsertedInCent+CreditInCent >= PriceInCent)
        {
            //posso fare il caffè
            result = true;

            //calcolare il credito residuo
            CreditInCent = InsertedInCent + CreditInCent - PriceInCent;
            ResetInsertedCent();
        }
    }

    return result;
}
}

```



```

}

using CoffeMachine;
using System.Security.Principal;

namespace TestCoffeMachine
{
    [TestClass]
    public class MachineTest
    {
        //AddInsertedCent_WithValidPrice_UpdatesInsertednCent
        //AddInsertedCent --> su cosa lavoriamo
        //WithValidValue --> quale caso ci aspettiamo di testare
        //UpdatesInsertedInCent --> cosa verifichiamo al termine del test
        [TestMethod]
        public void AddInsertedCent_WithValidValue_UpdatesInsertedInCent()
        {
            //creo un oggetto Machine
            Machine machine = new Machine(100);

            //modifico InsertedInCent attraverso l'add con un valore valido
            machine.AddInsertedCent(100);

            //verifico che il valore attuale dei soldi inseriti
            //sia uguale al valore atteso --> 100
            int expected = 100;
            int actual = machine.InsertedInCent;

            Assert.AreEqual(expected, actual);
        }

        //AddInsertedCent_WithValidPrice_UpdatesInsertednCent
        //AddInsertedCent --> su cosa lavoriamo
        //WithNotValidValue --> quale caso ci aspettiamo di testare
        //ShouldThrowException --> cosa verifichiamo al termine del test
        [TestMethod]
        public void AddInsertedCent_WithNotValidValue_ShouldThrowException()
        {
            //creo un oggetto Machine
            Machine machine = new Machine(100);

            //modifico InsertedInCent attraverso l'add con un valore non valido e verifico che ci sia
            //l'exception
            Assert.ThrowsException<System.ArgumentOutOfRangeException>(() =>
            machine.AddInsertedCent(-1));
        }
    }
}

```

```

}
namespace CoffeMachine
{
    internal class Program
    {
        static void Main(string[] args)
        {
            //Machine myMachine --> definisco myMachine come oggetto di tipo Machine
            //myMachine = new Machine(70); --> istanzio un oggetto di tipo Machine richiamando il
            costruttore con un parametro intero
            Machine myMachine = new Machine(70);

            //int price = myMachine.PriceInCent;
            //definisco un dato di tipo intero price ed al suo interno
            //salvo il valore restituito dal get della proprietà PriceInCent
            int price = myMachine.PriceInCent;

            //Console.WriteLine --> è un metodo che mi permette di scrivere output in console
            Console.WriteLine($"il caffè costa {price} centesimi");

            //simulo l'inserimento di 1 euro
            myMachine.AddInsertedCent(100);

            bool coffe = myMachine.MakeCoffe();
            Console.WriteLine(coffe); //visualizza true

            coffe = myMachine.MakeCoffe();
            Console.WriteLine(coffe); //visualizza false perchè non bastano i soldi

        }
    }
}

```

ESEMPI

vedi esercitazioni su Classroom

EQUALS E TOSTRING

<https://learn.microsoft.com/en-us/dotnet/api/system.object.equals?view=net-7.0>

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/operators/equality-operators#equality-operator->

Ogni oggetto ha a disposizione i metodi resi disponibili dalla classe Object, in particolare abbiamo a disposizione

- Equals che è il metodo utilizzato per il confronto tra oggetti (il comportamento di default return true se gli oggetti hanno lo stesso indirizzo di memoria, false altrimenti)

- ToString che è il metodo che ci restituisce una stringa che rappresenta l'oggetto, di default viene restituito il nome del namespace e della classe.

Questi metodo possono essere **sovrascritti** per ridefinire i comportamenti nel modo si ritiene più opportuno e corretto per gli oggetti della classe di riferimento

Esempio:

```
class Punto{
...
//vogliamo che due punti siano considerati uguali quando hanno gli
stessi valori per la coordinata X e per la coordinata Y
public override bool Equals(object? obj)
{
    //conversione di tipo da obj a Punto
    //Punto p = (Punto)obj ;
    if(obj == null) return false;
    //verifica se il tipo di obj è Punto
    if (obj is Punto)
    {
        Punto p = obj as Punto;

        if(X==p.X && Y==p.Y)
            return true;
    }
    return false;
}

//vogliamo che il ToString restituisca le coordinate del punto
nella forma (3,6)

public override String ToString()
{
    return $"({X}, {Y})";
}

}

static void Main(string[] args)
{
    Punto punto1 = new Punto(-8, 3.5);
    Punto punto2 = new Punto(-8, 3.5);

    Console.WriteLine(punto1); → visualizza in console il
messaggio (-8, 3.5)
```

if(punto1==punto2) → risponde false perché confronta i riferimenti

```
{
    ...
}
else
{
    ..
}
```

if (punto1.Equals(punto2)) → risponde true perché richiama il metodo

```
{
    ...
}
else
{
    ...
}
```

SOVRASCRITTURA DELL'OPERATORE ==

== fa l'uguaglianza dei riferimenti quindi se voglio che si comporti come Equals devo **sovrascrivere** l'operatore == ed anche !=

```
public static bool operator ==(Punto p1, Punto p2)
{
    return p1.Equals(p2);
}
public static bool operator !=(Punto p1, Punto p2)
{
    return !(p1.Equals(p2));
}
```

fatto questo nel main

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

```
    Punto punto1 = new Punto(-8, 3.5);
    Punto punto2 = new Punto(-8, 3.5);
```

if(punto1==punto2) → risponde true perché richiama == di

Punto

```
{
    ...
}
```

```
else
{
    ..
}
```

if (punto1.Equals(punto2)) → risponde true perché richiama il metodo

```
{
    ...
}
else
{
    ...
}
```

GESTIONE DELLA MEMORIA IN C#

In C#, la memoria è suddivisa principalmente in due aree: **lo stack e l'heap**. Queste due aree sono gestite in modo differente e contengono dati con scopi diversi.

In C#, i tipi valore (come int, char, bool, ecc.) vengono allocati nello stack, mentre i tipi riferimento (come oggetti delle classi) vengono allocati nell'heap.

Le variabili che contengono tipi valore memorizzano direttamente il valore, mentre le variabili che contengono tipi riferimento memorizzano un riferimento all'oggetto nell'heap.

La gestione della memoria nello heap è automatizzata attraverso la raccolta dei rifiuti (**garbage collection**), che rileva e dealloca gli oggetti che non sono più raggiungibili. Questo consente di evitare perdite di memoria e semplifica la gestione delle risorse di sistema.

Questa è solo una rappresentazione concettuale e semplificata. La gestione della memoria in C# è più complessa

Stack

Lo stack è una struttura dati a pila che segue un modello LIFO (Last-In-First-Out), il che significa che l'ultimo elemento inserito nello stack è il primo a essere rimosso.

Gli oggetti nello stack vengono allocati e deallocati molto rapidamente. Questo è particolarmente utile per la gestione delle variabili locali e dei parametri delle funzioni.

Gli oggetti nello stack hanno dimensioni fisse e sono generalmente noti a tempo di compilazione (esempio un int il compilatore sa che nello stack occuperà 4Byte e questa dimensione non cambia durante l'esecuzione del programma).

Gli oggetti nello stack esistono solo all'interno dell'ambito in cui sono dichiarati. Quando una funzione termina, le variabili nello stack vengono automaticamente deallocate (Poiché le dimensioni degli oggetti nello stack sono note a tempo di compilazione, il compilatore può generare istruzioni di allocazione e deallocazione di memoria direttamente nel codice eseguibile. Ciò significa che l'allocazione e la deallocazione delle variabili nello stack sono molto efficienti e veloci).

Si può immaginare lo "stack" come una pila di blocchi di memoria.

Ogni blocco di memoria nello stack rappresenta un contesto o un'invocazione di funzione.

Esempio:

Se abbiamo una funzione A che chiama una funzione B, avremo due blocchi nello stack: uno per A e uno per B.

Questi blocchi sono disposti in modo che il più recente (l'invocazione più recente) sia in cima alla pila, e il più antico (l'invocazione iniziale) è in fondo.

[Blocco di A]

[Blocco di B]

[... altri blocchi ...]

All'interno di ciascun blocco, sono presenti variabili locali e parametri della funzione. Queste variabili vengono allocate e deallocate rapidamente quando il blocco viene creato e distrutto.

Heap

L'heap è una zona di memoria più ampia, ed è utilizzata per la gestione di oggetti con durata di vita dinamica. Gli oggetti nell'heap possono essere allocati e deallocati in modo più flessibile rispetto allo stack, è infatti possibile allocare memoria in modo dinamico durante l'esecuzione del programma. Ciò significa che puoi creare oggetti nell'heap quando ne hai bisogno e liberare la memoria quando non ne hai più bisogno. Questo è particolarmente utile quando non sai in anticipo quanti oggetti devi gestire o quando la dimensione degli oggetti può variare.

L'allocazione di oggetti nell'heap è più lenta rispetto allo stack perché richiede la ricerca di spazio libero sufficiente.

Gli oggetti nell'heap possono avere dimensioni variabili ed essere gestiti in modo dinamico.

Gli oggetti nell'heap esistono finché vengono mantenute delle referenze ad essi. Se non ci sono più riferimenti a un oggetto nell'heap, diventa candidato per la garbage collection. La gestione della memoria nell'heap è più complessa e coinvolge la raccolta automatica dei rifiuti (garbage collection) per deallocare gli oggetti che non sono più raggiungibili. Anche se l'allocazione e la deallocazione nell'heap offrono maggiore flessibilità, la gestione della memoria nell'heap è più complessa rispetto allo stack. Richiede un meccanismo di garbage collection per identificare e deallocare automaticamente gli oggetti non più utilizzati. Questo processo aggiunge una certa complessità al runtime dell'applicazione.

In sintesi, la flessibilità nell'allocazione e nella deallocazione degli oggetti nell'heap è utile quando hai bisogno di oggetti con durata di vita dinamica o quando non puoi prevedere in anticipo il numero o la dimensione degli oggetti da gestire.

Lo heap è una zona di memoria più ampia e dinamica rispetto allo stack, quindi è più difficile da rappresentare visivamente.

Gli oggetti nell'heap **non** sono organizzati in uno stack lineare ma possono essere sparsi in memoria.

Gli oggetti nel heap sono allocati dinamicamente e gestiti dalla garbage collection. Ci sono oggetti di dimensioni diverse con durate di vita variabili nell'heap.

L'idea generale è che nello stack hai una pila di contesti di funzioni, ognuno con le sue variabili locali, mentre nell'heap hai oggetti che esistono in modo più flessibile e possono essere referenziati da variabili in diversi stack frames.

Ora dovrebbe essere chiaro perchè il comportamento di default del metodo Equals confronta gli indirizzi di memoria per restituire true o false

ESEMPIO DI GESTIONE DELLA MEMORIA

ESEMPIO 1:

```
using System;

class Studente
{
    public string Nome { get; set; }
    public int Età { get; set; }

    public Studente(string nome, int età)
    {
        Nome = nome;
        Età = età;
    }
}

class Program
{
    static void Main()
    {
        // Variabile nello stack
        int x = 10;

        // Variabile nello stack
        string nome = "Mario";

        // Creazione di un oggetto Studente nell'heap
        Studente studente = new Studente("Luigi", 20);

    }
}
```

La variabile x è allocata nello stack. Contiene un valore intero (int) e la sua durata di vita è limitata al contesto Main.

La variabile nome è anch'essa allocata nello stack e contiene una stringa (string) con il nome "Mario". Anche questa variabile ha una durata di vita limitata al contesto Main.

Viene creata un'istanza della classe Studente tramite `new Studente("Luigi", 20)`. Questo oggetto studente è allocato nell'heap. L'oggetto studente contiene i dati del nome e dell'età di uno studente. Il GarbageCollector gestirà l'eliminazione dell'oggetto studente nell'heap quando non sarà più referenziato.

Quindi:

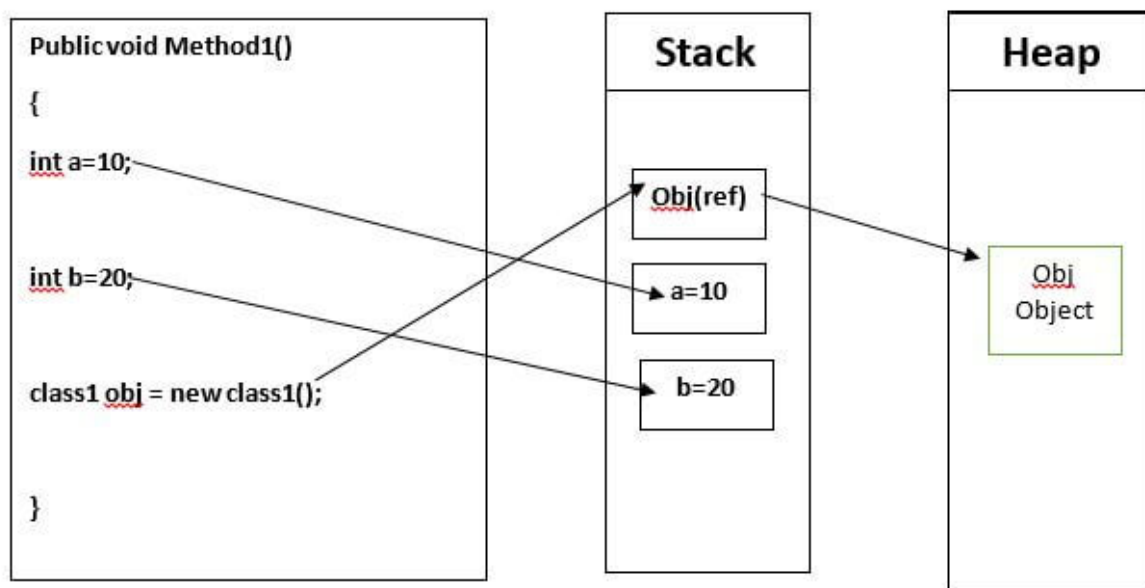
Quando crei un oggetto di una classe, vengono allocati nello heap spazio per tutti gli attributi di quella classe.

Ogni oggetto di quella classe condividerà gli stessi membri (attributi), ma avrà valori diversi per questi attributi.

Le variabili nello stack che contengono riferimenti agli oggetti della classe puntano a queste istanze allocate nell'heap. Quindi, la variabile nello stack contiene solo il riferimento all'oggetto nell'heap, non l'oggetto stesso.

Gli attributi all'interno dell'oggetto nell'heap sono accessibili tramite il riferimento dalla variabile nello stack.

ESEMPIO 2:



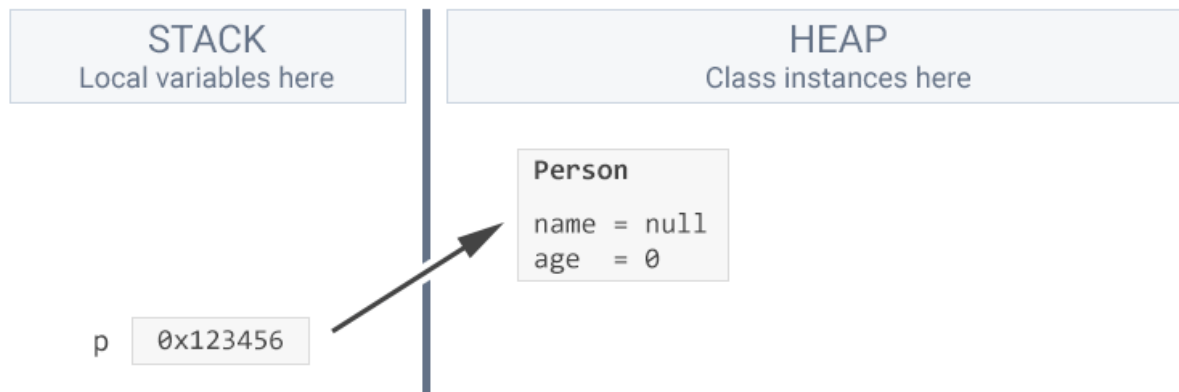
ESEMPIO 3:

(da <https://www.koderhq.com/tutorial/csharp/stack-v-heap/>)

```
class Person
{
    public string name;
    public int age;
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
    }
}
```

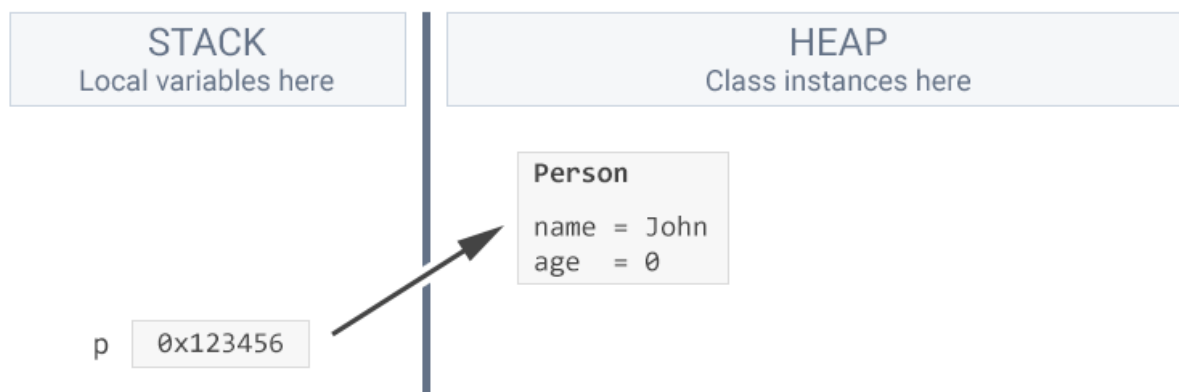
IN QUESTO CASO IN MEMORIA SI AVRÀ:



```
class Person
{
    public string name;
    public int age;
}

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        p.name = "John";
    }
}
```

IN QUESTO CASO IN MEMORIA SI AVRÀ:



```
class Person
{
    public string name;
    public int age;
}

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

```

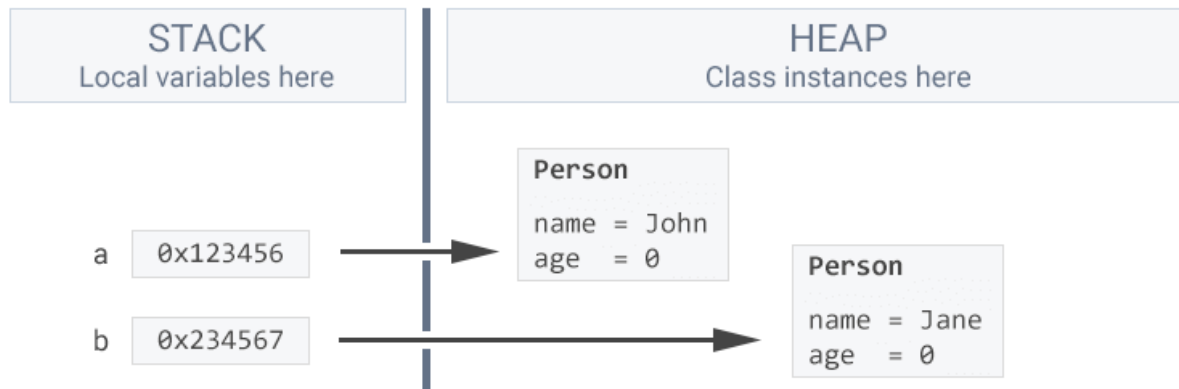
```

    Person a = new Person();
    a.name = "John";

    Person b = new Person();
    b.name = "Jane";

```

}
 }
 IN QUESTO CASO IN MEMORIA SI AVRÀ:



```

class Person
{
    public string name;
    public int age;
}

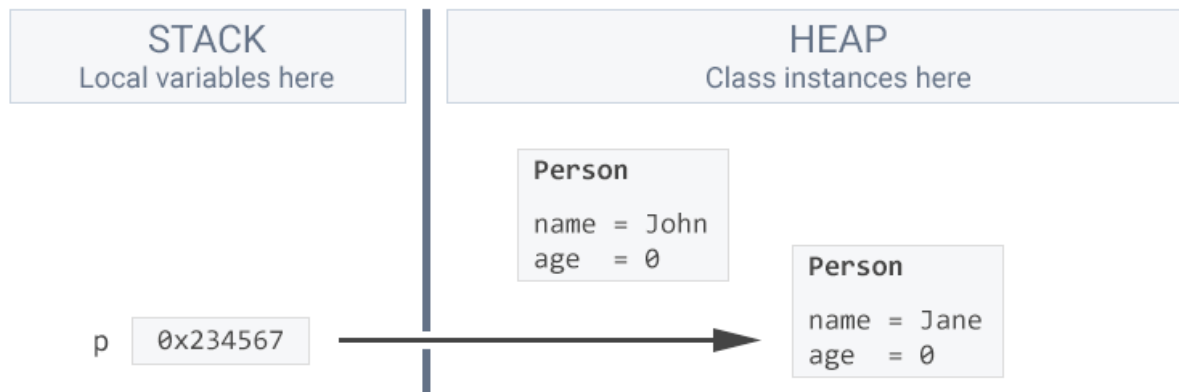
class Program
{
    static void Main(string[] args)
    {
        Person p = new Person();
        p.name = "John";

        p = new Person();
        p.name = "Jane";
    }
}

```

la seconda chiamata a `p = new Person();` fa perdere il riferimento all'oggetto iniziale che quindi sarà poi eliminato dal garbage collector

IN QUESTO CASO IN MEMORIA SI AVRÀ:

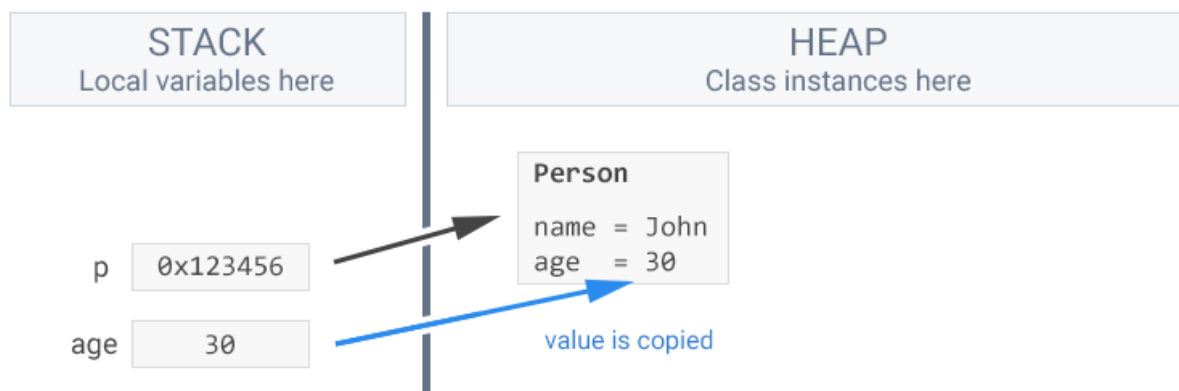


```
class Person
{
    public string name;
    public int age;
}

class Program
{
    static void Main(string[] args)
    {
        int age = 30;

        Person p = new Person();
        p.name = "John";
        p.age = age;
    }
}
```

IN QUESTO CASO IN MEMORIA SI AVRÀ:



il valore di age è stato copiato in age di Person. ATTENZIONE: non esiste nessuna connessione tra age del main ed age di Persona, semplicemente il valore di age è stato copiato dentro al campo age. Se modifico age del main o age di Person, la modifica non influenzerà l'altro valore

MODALITA' DI PASSAGGIO DEI PARAMETRI AD UN METODO

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/keywords/method-parameters>

In C#, il passaggio di parametri a un metodo può avvenire in tre modalità principali: **per valore**, **per riferimento con ref**, e **per riferimento con out**. Ogni modalità ha un diverso impatto su come i parametri vengono gestiti in memoria e quali modifiche possono essere apportate ai parametri stessi all'interno del metodo.

RICORDA:

parametro formale → il parametro nella firma del metodo

parametro attuale → il parametro nella chiamata al metodo

Passaggio per valore

Quando un parametro viene passato **per valore** (questo è il comportamento di default), il metodo riceve una **copia del valore**. Questo significa che qualsiasi modifica fatta al parametro formale all'interno del metodo **non** influenzerà il parametro attuale.

ATTENZIONE: se il parametro è un **'oggetto'**, anche se facciamo un passaggio per valore, **il tipo è riferimento** quindi in questo caso le modifiche fatte sullo stato del parametro formale sono applicate anche al parametro attuale avendo ricevuto copia del riferimento quindi lavorando sulla stessa memoria.

ESEMPI:

```
public class DemoPassaggioPerValore
{
    public void ChangeIntValueWithRandom(int number)
    {
        Random rnd = new Random()
        number = rnd.Next();
    }

    public void ChangePersonName(Person person)
    {
        person.Name = "Updated Inside"; } //qui modifico un
    attributo del parametro formale, che essendo un tipo riferimento
```

porterà alla modifica anche del parametro attuale (lavoro sulla stessa memoria)

```
public void ChangePersonNameNew(Person person)
{
    person = new Person("Updated Inside"); // Crea un nuovo
    oggetto, ovvero cambio il riferimento al parametro formale, questo
    non modifica l'oggetto passato (il parametro attuale)
}

public class Person
{
    public string Name { get; set; }

    public Person(string name) {Name = name;}
}

public class Program
{
    public static void Main()
    {
        DemoPassaggioPerValore demo = new
        DemoPassaggioPerValore();

        // Passaggio per valore di un parametro di tipo int
        int originalNumber = 50;
        demo.ChangeIntValue(originalNumber);

        Console.WriteLine($"{originalNumber}"); //output del
        numero 50 perchè il numero random generato all'interno del metodo
        non ha effetti sul parametro attuale

        // Passaggio per valore di un oggetto (ovvero di un
        riferimento)

        Person originalPerson = new Person("Original Name");
        demo.ChangePersonNameNew(originalPerson);
    }
}
```

```
        Console.WriteLine($"{originalPerson.Name}"); //output di
Original Name perchè all'interno del metodo ho fatto un new quindi
il parametro formale non è stato modificato
```

```
        demo.ChangePersonName(originalPerson);
        Console.WriteLine($"{originalPerson.Name}"); //output di
UpdateInside perchè all'interno del metodo ho modificato il valore
dell'attributo del parametro di cui avevo il riferimento

    }

}
```

Passaggio per riferimento con ref

Quando un parametro viene passato per riferimento con **ref**, il metodo riceve un **riferimento al parametro attuale e non una copia**. Questo significa che qualsiasi modifica fatta al parametro formale nel metodo influenzerà il parametro attuale. Se il passaggio è fatto con ref, il parametro attuale deve essere inizializzato prima di essere passato al metodo.

ESEMPIO:

```
public class DemoPassaggioPerRef
```

```
{

    public void ChangeIntValue(ref int number)

    {

        number = 200;

    }

    public void ChangePersonName(ref Person person)

    {

        person = new Person("Updated with ref"); // Cambia il
riferimento dell'oggetto del parametro formale e quindi anche
dell'attuale

    }

}

public class Program

{
```

```

public static void Main()
{
    DemoPassaggioPerRef demo = new DemoPassaggioPerRef();

    // Passaggio per riferimento con int

    int originalNumber = 50;
    demo.ChangeIntValue(ref originalNumber); //passo il
riferimento a //output di 200

    Console.WriteLine($"{originalNumber}");//output di 200

    // Passaggio per riferimento con oggetto

    Person originalPerson = new Person("Original Name");

    demo.ChangePersonName(ref originalPerson);

    Console.WriteLine($"{originalPerson.Name}");//output di
Update with ref
}
}

```

Passaggio per riferimento con out

Anche quando un parametro viene passato per riferimento con **out**, il metodo riceve un **riferimento al parametro attuale e non una copia**; ma la differenza rispetto all'uso di **ref** è che il parametro non deve essere inizializzato prima di essere passato ed il metodo è obbligato a inizializzare il parametro prima della fine del metodo.

```

public class DemoPassaggioPerOut

```

```

{
    public void InitializeIntValue(out int number)
    {
        number = 300; // Devo sempre assegnare un valore al
parametro
    }
}

```



```

        public void InitializePerson(out Person person)
        {
            person = new Person("Initialized with out"); // Assegna un
            nuovo oggetto
        }
    }

    public class Program
    {
        public static void Main()
        {
            DemoPassaggioPerOut demo = new DemoPassaggioPerOut();

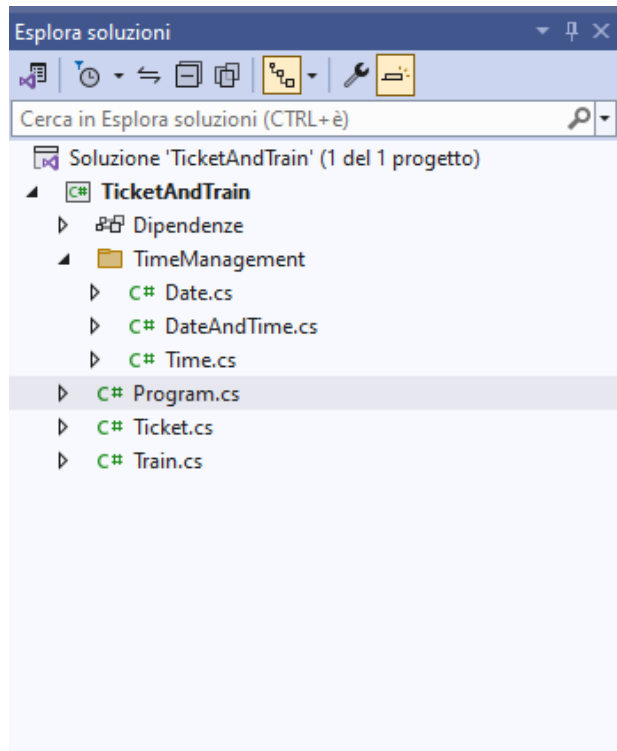
            // Passaggio per out con int
            int uninitializedNumber; // Non inizializzato
            demo.InitializeIntValue(out uninitializedNumber);
            Console.WriteLine($" {uninitializedNumber}");//output di
300

            // Passaggio per out con oggetto
            Person uninitializedPerson; // Non inizializzato
            demo.InitializePerson(out uninitializedPerson);

            Console.WriteLine($"{uninitializedPerson.Name}"); //output
di Initialized with out
        }
    }

```


ESEMPIO PROGETTO CON COMPOSIZIONE, AGGREGAZIONE



IL PROGETTO CONTIENE:

una classe **Orario** che permetta di:

- definire un orario valido all'interno di una giornata.
- restituire solo il dato dell'ora
- restituire se l'orario è AM o PM
- incrementare l'orario di tot ore
- incrementare (spostare in avanti o indietro) l'orario di tot minuti
- visualizzare l'ora nel formato ore:minuti
- far sì che due oggetti Orario risultino uguali se rappresentano lo stesso orario

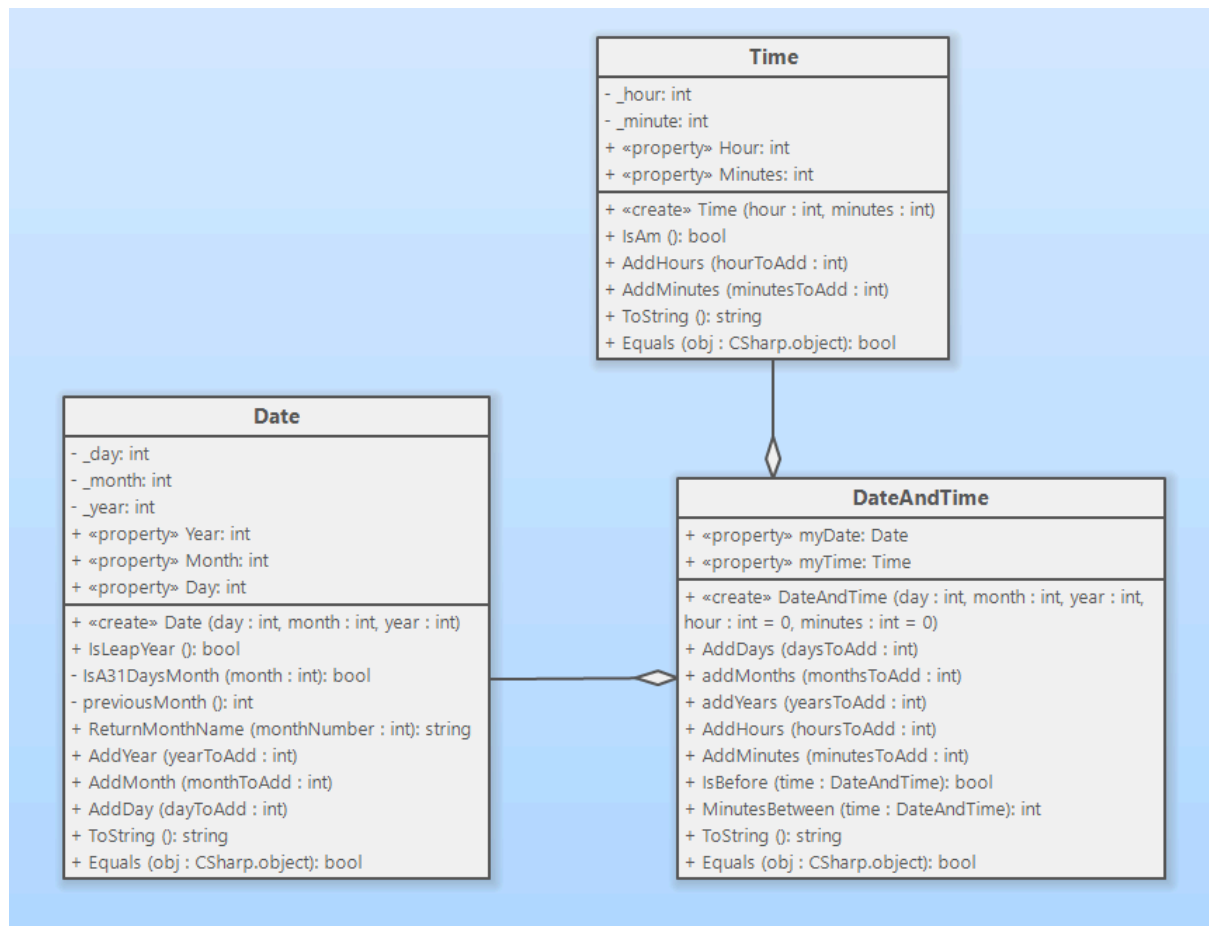
una classe **Data** che permetta di:

- definire una data valida

- restituire il giorno (numero)
- restituire il mese (numero, realizzare anche un metodo che permetta di restituire il mese in formato stringa)
- restituire l'anno (numero)
- verificare se l'anno è bisestile
- incrementare la data di tot giorni
- incrementare la data di tot mesi
- incrementare la data di tot anni
- visualizzare la data nel formato giorno/mese/anno
- far sì che due oggetti Data risultino uguali se rappresentano la stessa data

una classe **DataOra** che permetta di:

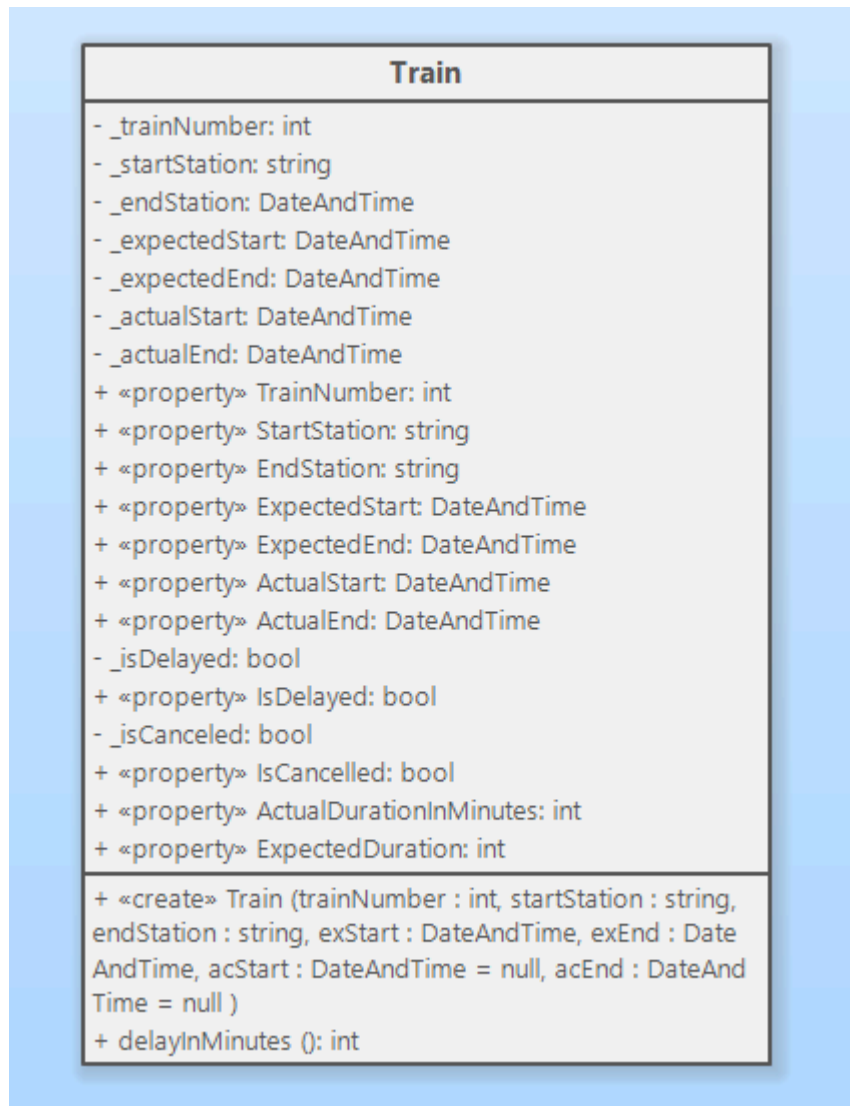
- memorizzare la data (sempre obbligatoria)
- memorizzare l'orario (se non definita impostare l'orario a 0:00)
- incrementare la data di tot giorni
- incrementare la data di tot mesi
- incrementare la data di tot anni
- incrementare l'orario di tot ore
- incrementare l'orario di tot minuti
- visualizzare la data e l'orario nel formato giorno/mese/anno ora:minuti
- far sì che due oggetti Data risultino uguali se rappresentano la stessa data e lo stesso orario



una classe **Treno** che permetta di memorizzare:

- numero del treno
 - stazione di partenza (memorizzare la stringa in lowercase)
 - stazione di arrivo (memorizzare la stringa in lowercase)
 - date e ora di partenza prevista
 - data e ora di arrivo prevista
 - date e ora di partenza effettiva
 - data e ora di arrivo effettivo
- deve essere possibile per ogni treno calcolare
- ritardo in minuti
 - durata prevista del percorso

- durata effettiva del percorso

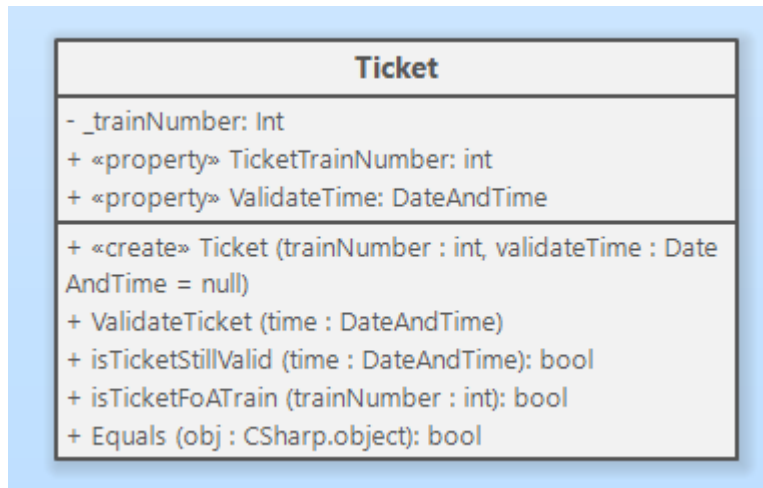


Una la classe **biglietto**

Un biglietto si riferisce ad un treno, ha una data di convalida ed una validità che è pari a 3 ore dal momento della convalida.

deve essere possibile

- verificare se un biglietto è ancora valido
- se un biglietto si riferisce ad un determinato treno
- se due biglietti sono uguali (sono per lo stesso treno e sono convalidati allo stesso momento o non convalidati)



Dati ENUMERATIVI

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/language-specification/enums#196-enum-values-and-operations>

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/builtin-types/enum>

Il tipo enumerazione (detto anche **enum**) rappresenta un modo efficiente per definire un insieme di costanti di tipo intero associate ad un nome e che possono essere assegnate a una variabile.

È utile utilizzare gli enum quando delle variabili assumono dei valori specifici che corrispondono a nomi specifici come accade per esempio per i giorni di una settimana o per i mesi di un anno.

Esempio:

enum **Giorno**

```
{
    Lunedì,
    Martedì,
    Mercoledì,
    Giovedì,
    Venerdì,
    Sabato,
    Domenica
}
```

In questo caso abbiamo definito un nuovo tipo enumerativo che si chiama **Giorno** costituito dai giorni della settimana e automaticamente sono associati i suoi valori a numeri di tipo intero (int) partendo da 0 per lunedì, 1 per martedì e così via.

Volendo possiamo definire un dato di tipo Giorno e poi eseguire dei confronti con il tipo enumerativo rendendo il codice molto più leggibile rispetto all'uso delle costanti intere.

Esempio:

```
Giorno x = Giorno.Lunedì;
```

```
    if (x == Giorno.Lunedì)
```

```
{
```

```
    .....
```

```
}
```

Se è necessario o conveniente è possibile cambiare il numero associato ad un valore oppure è possibile modificare tipo sottostante (quando non definisco nessun tipo di default il tipo è int)

Esempio:

```
enum Giorno: byte
```

```
{
```

```
    Lunedì = 1,
```

```
    Martedì = 2,
```

```
    Mercoledì = 3,
```

```
    Giovedì = 4,
```

```
    Venerdì = 5,
```

```
    Sabato = 6,
```

```
    Domenica = 7
```

```
}
```

o equivalentemente:

```
enum Giorno: byte
```

```
{
```



```

    Lunedì = 1,
    Martedì ,
    Mercoledì,
    Giovedì,
    Venerdì,
    Sabato,
    Domenica
}

```

in questo modo otteniamo che il giorno Lunedì sia associato al valore 1, martedì al valore 2 e via via, ed il dato è di tipo byte (quindi occupa 8 bit e non 32 come un int)

L'utilizzo degli enum rende il nostro codice maggiormente leggibile perché ci permette di sostituire l'uso di valori numerici sparsi per il codice con l'uso di nomi significativi.

Un ulteriore utilizzo molto importante legato ai dati di tipo enumerativo è il fatto che quando vengono utilizzati come parametri ci permettono di limitare i possibili valori assunti dagli stessi. Se ho un parametro intero giorno questo potrà assumere un qualsiasi numero intero memorizzabile, se ho parametro di tipo Giorno, questo occuperà sempre 32 bit in memoria se non specificato diversamente, ma potrà assumere solo i valori definiti dall'enumerativo.

Esempio:

```

void Main(string[] args)
{
    StampaGiorno(Giorno.Lunedì);
}

//metodo che ha come parametro un dato di tipo Giorno
void StampaGiorno(Giorno x)
{
    .....
}

```

Esempio Immobile e tipologie

```
public class Immobile
{
    public enum TipologiaImmobile
    {
        Appartamento,
        Box,
        Villa
    }

    private int _id;

    private int _superficie;

    private TipologiaImmobile _tipologia;

    public int Id
    {
        get { return _id; }

        set { _id = value; }
    }

    public int Superficie
    {
        get { return _superficie; }

        set {
            if(value < 0)throw new ArgumentOutOfRangeException("superficie non accettabile");

            _superficie = value;
        }
    }
}
```

```

    }
}

public TipologiaImmobile Tipologia
{
    get { return _tipologia; }

    set
    {
        //attenzione non è suff questo controllo per assicurarmi che sia uno dei 3 valori definiti
        if (!(value is TipologiaImmobile)) throw new ArgumentOutOfRangeException("tipologia
non accettabile");

        _tipologia = value;
    }
}

public Immobile(int id, int superficie, TipologiaImmobile tipologia)
{
    _id = id;

    Superficie = superficie;

    Tipologia = tipologia;
}

/// <summary>
/// output delle caratteristiche dell'immobile
/// </summary>
/// <returns></returns>

public override string ToString()
{

```

```
        return $"Immobile {Id}\nSuperficie {Superficie}mq\nTipologia {Tipologia}\n\n ";
    }
}
```

```
/// <summary>
/// due immobili sono uguali se hanno lo stessocodce
/// </summary>
/// <param name="obj"></param>
/// <returns></returns>
public override bool Equals(object? obj)
{
    if(obj == null || !(obj is Immobile)) return false;

    Immobile immobile = (Immobile)obj;

    return immobile.Id==Id;
}
}
```

e nel main

```
static void Main(string[] args)
{
    Immobile im1 = new Immobile(0, 100, Immobile.TipologiaImmobile.Villa);

    Console.WriteLine($"{im1}");

    //?questa chiamata genera un errore oppure no?

    Immobile im2 = new Immobile(0, 100, (Immobile.TipologiaImmobile)5);

    Console.WriteLine($"{im2}");

    Random rnd = new Random();
}
```

```

        for (int i = 0; i < 10; i++)
        {
            int id = rnd.Next();

            int sup = rnd.Next();

//Tipologia Random

            Immibile.TipologiaImmibile tipologia =
(Immibile.TipologiaImmibile)(rnd.Next((int)Immibile.TipologiaImmibile.Appartamento,
(int)Immibile.TipologiaImmibile.Villa + 1));

            Immibile immobile = new Immibile(id, sup, tipologia);

//richiam ToString

            Console.WriteLine($"{immobile}");

        }
    }
}

```

Esempio Utilizzo di TryParse e IsDefined con Enumerativi

```

internal class Program
{
    enum Colors { White = 0, Red = 1, Green = 2, Blue = 4 };

    public static void Main()
    {
        int[] colorsTest = { 0,1,2,8 };

        foreach (int color in colorsTest)
        {
            Colors colorValue;

            //Enum.TryParse si aspetta come parametro una stringa e cerca di convertire quella stringa
            in un elemento dell'enum

            //se riesce a fare la conversione il valore convertito viene salvato in colorValue ed il metodo
            restituisce true

```

//(out significa che il valore è passato per riferimento ed il suo valore viene modificato dal metodo)

//se la conversione non riesce il metodo restituisce false

if (Enum.**TryParse**(color.ToString(), out colorValue))

{

 Console.WriteLine(\$"esito positivo di TryParse per {color} --> {colorValue}");

 if (Enum.**IsDefined**(typeof(Colors), colorValue))

 Console.WriteLine(\$"{{color}} viene convertito in {colorValue.ToString()}");

 else

 Console.WriteLine(\$"{{color}} non ha valori associati definiti per l'enumerativo.");

 }

else

 Console.WriteLine(\$"esito negativo di TryParse per {color} non è un membro di Colors.");

}

string[] colorStrings = { "0", "2", "8", "blue", "Blue", "Yellow" };

foreach (string colorString in colorStrings)

{

 Colors colorValue;

 if (Enum.TryParse(colorString, out colorValue))

 {

 Console.WriteLine(\$"esito positivo di TryParse per {colorString} --> {colorValue}");

 if (Enum.IsDefined(typeof(Colors), colorValue))

 Console.WriteLine(\$"{{colorString}} viene convertito in {colorValue.ToString()}");

 else

```
        Console.WriteLine($"{colorString} non ha valori associati definiti per l'enumerativo." );
    }
    else
        Console.WriteLine($"esito negativo di TryParse per {colorString} non è un membro di
Colors.");
    }
}
}
```

STRUTTURE DATI

Una **struttura dati** è un modo per organizzare e gestire i dati in memoria in modo efficiente. Le strutture dati permettono di eseguire operazioni sui dati (come inserimento, ricerca, aggiornamento e cancellazione) in modo ottimizzato per il contesto in cui è utilizzata. Le strutture dati più comuni, e che vedremo nel corso degli studi sono: array, liste, pile, code e alberi.

In C#, un array è una struttura di dati che contiene un numero fisso di elementi dello stesso tipo, indicizzati da un numero intero. Gli array in C# sono di dimensione statica, il che significa che una volta creati, la loro dimensione non può essere cambiata. Gli array dinamici, d'altra parte, possono variare in dimensione durante il tempo di esecuzione e sono generalmente implementati usando le classi `List<T>` o `ArrayList` per consentire aggiunte e rimozioni di elementi.

In C#, le strutture dati come gli array e le liste dinamiche, essendo di tipo reference sono allocati sulla **heap**. Sullo stack vengono memorizzati solo i riferimenti a queste strutture.

ARRAY

<https://learn.microsoft.com/it-it/dotnet/csharp/language-reference/builtin-types/arrays>

Gli Array contengono dati omogenei (ovvero dati tutti dello stesso tipo) e **sono utilizzati quando si conosce in anticipo il numero di elementi da archiviare** e questo non subirà modifiche nel corso dell'esecuzione. Per esempio utilizzerò un array per memorizzare le temperature registrate in una settimana perché a priori posso sapere che saranno 7.

Un array ha una lunghezza o dimensione che rappresenta la quantità di dati che può contenere. Questa lunghezza è predefinita e immutabile.

Dichiarazione di un array composto da 7 elementi di tipo int
`int[] temperature = new int[7];`

0	0	0	0	0	0	0
---	---	---	---	---	---	---

0 1 2 3 4 5 6

L'array viene inizializzato al valore di default previsto per il tipo (es: 0 nel caso di int, null nel caso di reference)

E' possibile accedere direttamente ad una qualsiasi posizione dell'array attraverso il suo indice.

`temperature[2] = 27;`

trasformerà l'array nel seguente modo:

0	0	27	0	0	0	0
0	1	2	3	4	5	6

Gli array sono allocati sulla heap perché gli array in C# sono tipi reference.

L'allocazione richiede **spazio continuo in memoria** per tutti gli elementi dell'array.

L'array viene allocato in un unico blocco ma è statico, ridimensionarlo vuol dire riallocare l'array e copiare i valori che si vogliono mantenere nella nuova allocazione (vedi esempio sotto)

La complessità computazione per l'accesso ad un elemento dell'array è $O(1)$ ovvero ha un tempo costante, ovvero il tempo di questa operazione non varia al variare della dimensione dell'input

Il tempo di ricerca di un elemento dell'array è un tempo lineare, indicato con $O(n)$, ovvero il tempo di questa operazione varia al variare in modo lineare rispetto alla dimensione dell'input.

Semplificando possiamo dire che supponendo di avere un array di $n=10$ elementi la ricerca, nel peggiore dei casi ovvero quando l'elemento cercato non è presente, dovrà effettuare 10 operazioni di confronto per poter terminare. Se l'array ha $n=100$ elementi, la ricerca, nel peggiore dei casi, dovrà effettuare 100 confronti per poter terminare.

Esempio:

```
public class ArrayOperations
{
    private int[] numbers;//definisco come attributo della classe
    un array di interi. In questo momento l'array non è ancora stato
    istanziato, perchè su di esso non è ancora stato richiamato il new

    // Costruttore sia aspetta come paramento la dimensione
    dell'array e lo istanzia

    public ArrayOperations(int n)
    {
        //qui andrebbero inseriti i controlli su n che deve
        essere>0

        numbers = new int[n]; //istanzio l'array ovvero occupo in
        memoria lo spazio contiguo per poter contenere n elementi di tipo
        int
    }

    // Metodo per riempire l'array con numeri casuali tra 1 e 100
```

```

public void FillArray()
{
    Random random = new Random();

    for (int i = 0; i < numbers.Length; i++)
    {
        numbers[i] = random.Next(1, 101);
    }
}

// Metodo per cercare un elemento nell'array

public bool FindElement(int value)
{
    //nel peggiore dei casi, per rispondere false devo aver
    controllato tutti i valori dell'array, quindi la complessità
    computazionale di questa operazione, ovvero il numero di
    istruzioni da eseguire e quindi il tempo di esecuzione dipende
    dalla dimensione dell'array O(n)

    foreach (int number in numbers)
    {
        if (number == value)
            return true;
    }

    return false;
}

// Metodo per ridefinire l'array con una dimensione di 1 in
più di prima

public void ResizeArray()
{
    //ridimensionare l'array vuol dire creare un nuovo array di
    dimensione 1 in più rispetto a prima

    int[] newNumbers = new int[numbers.Length + 1];

```

```
//copiare nel nuovo array tutti i valori dell'array di partenza
for (int i = 0; i < numbers.Length; i++)
{
    newNumbers[i] = numbers[i];
}

numbers = newNumbers; // Sostituzione con il nuovo array
}
```

ATTENZIONE:

numbers = newNumbers non copia i valori di newNumbers nell'array numbers, ma aggiorna il riferimento numbers per puntare allo stesso blocco di memoria di newNumbers.

Ora numbers e newNumbers puntano entrambi allo stesso array sulla heap.

Il precedente array puntato da numbers diventa non referenziato (se non ci sono altri riferimenti che lo puntano). Questo significa che, anche se l'array esiste ancora fisicamente in memoria, non è più accessibile tramite alcun riferimento e verrà considerato garbage.

Alla prossima esecuzione del Garbage Collector il vecchio array non referenziato viene rilevato e la memoria occupata da esso viene recuperata.

LISTE (ARRAY DINAMICI)

<https://learn.microsoft.com/it-it/dotnet/api/system.collections.generic.list-1?view=net-8.0>

```
public class List<T> : System.Collections.Generic ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>,
System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.Generic.IReadOnlyList<T>,
System.Collections.IList
```

Le liste sono degli Array Dinamici (List<T>) e vengono utilizzati quando la quantità di dati può cambiare nel corso dell'esecuzione e non si può conoscere a priori, per esempio utilizzerò una lista per la gestione dei prodotti presenti nel carrello di un'applicazione e-commerce.

Gli Array Dinamici (List<T>) sono allocati sulla heap, come per gli array, ma la gestione della memoria viene gestita da List. Quando una lista supera la sua capacità iniziale, viene creato automaticamente un nuovo array più grande (la dimensione viene generalmente duplicata) e copiato l'array precedente. Questa operazione non è lasciata allo sviluppatore, ma ottimizzata dalla classe, ma ovviamente, il costo della **copia** degli elementi quando la capacità viene superata si sostiene e questo può avere un impatto sulle prestazioni se l'espansione avviene frequentemente.

List ci mette a disposizione una serie di metodi per la gestione delle operazioni principali (ricerca di un elemento, rimozione di un valore, inserimento,...).

Esempio:

```
using System.Collections.Generic;
```

```
public class ListOperations
```

```
{
```

```
    private List<int> numbers; //definisco come attributo della
    classe una lista di interi. In questo momento la lista non è
    ancora stata istanziata, perchè su di essa non è ancora stato
    richiamato il new
```

```
    // Costruttore per inizializzare la lista
```

```
    public ListOperations()
```

```
    {
```

```
    //la lista può essere inizializzata vuota oppure con una dimensione
    iniziale
```

```
        numbers = new List<int>(); // lista vuota
```

```
        //se avessi avuto una dimensione avrei potuto scrivere
        // numbers = new List<int>(n); // Dimensione iniziale di n
    elementi
```

```
    }
```

```
    // Metodo per aggiungere alla lista n valori (numeri casuali
    tra 1 e 100)
```

```
    public void FillList(int n)
```

```
    {
```

```
        Random random = new Random();
```

```
        for (int i = 0; i < n; i++) // Inseriamo esattamente n
    elementi
```

```
        {
```

```
        //Add aggiunge elementi alla lista, nel caso in cui lo spazio
    della lista fosse esaurito si occuperà anche di ridimensionare.
```

Questo vuol dire che il numero di operazioni svolte dall'Add potrà essere diverso a seconda dei casi (aggiungo un elemento oppure ridimensiono, copio, cambio riferimento)

```
        numbers.Add(random.Next(1, 101));
    }
}

// Metodo per cercare un elemento nella lista
public bool FindElement(int value)
{
    //list ha già a disposizione dei metodi per le operazioni
    principali come per esempio la ricerca, la rimozione, l'aggiunta.
    ATTENZIONE dal punto di vista computazione richiamare Contains non
    equivale a fare 1 operazione, ma a fare tutte le operazioni
    previste dal metodo. In questo caso la ricerca, come per gli array
    è O(n)

    return numbers.Contains(value);
}

// Metodo per aggiungere un nuovo elemento alla lista
(espansione automatica)

public void AddElement(int value)
{
    numbers.Add(value);
}
```

ARRAY E LISTE COME PARAMETRI

Quando un array o una List<T> viene passato come parametro a un metodo, bisogna tenere presente che questi sono tipi di riferimento. Questo significa che quando vengono passati a un metodo, **viene passato un riferimento all'oggetto sulla heap**, viene quindi passato una copia del valore dell'indirizzo di memoria e non una copia dello stato dell'oggetto.

Quando un array o una lista è passato come parametro, il metodo riceve un riferimento all'oggetto originale quindi:

- Le modifiche agli elementi dell'array o della lista che avvengono all'interno del metodo influenzeranno l'array o la lista originale (perché nel metodo lavoro sullo stesso indirizzo di memoria).
- Se il riferimento ricevuto come parametro viene riassegnato a un nuovo array o lista all'interno del metodo, questa riassegnazione non si riflette fuori dal metodo tranne nel caso in cui il parametro non sia specificato con il modificatore ref o out.

Esempio con Array come parametro:

```
public void ModifyArray(int[] arr)
{
    // Modifica di un elemento - modifica l'array originale
    if (arr.Length > 0)
        arr[0] = 99;

    // Riassegnazione dell'intero array - NON modifica l'array originale
    arr = new int[] { 1, 2, 3 };
}
```

Se nel main ho il seguente codice:

```
int[] myArray = { 5, 10, 15 };
ModifyArray(myArray);
```

`Console.WriteLine(myArray[0]);` // Output: 99 e non 1 poiché l'operazione `arr[0] = 99` ha influenzato l'array `myArray`, mentre l'operazione `arr = new int[] { 1, 2, 3 }` ha riassegnato `arr` e quindi ha lavorato su altra zona di memoria non influenzando `myArray`

Esempio con List<T> come parametro:

```
public void ModifyList(List<int> list)
{
    // Modifica di un elemento - modifica la lista originale
    if (list.Count > 0)
        list[0] = 99;
```

```

        // Aggiunta di un elemento - modifica la lista originale

        list.Add(100);

        // Riassegnazione della lista - NON modifica la lista originale

        list = new List<int> { 1, 2, 3 };
    }

```

Se nel main ho il seguente codice:

```

List<int> myList = new List<int> { 5, 10, 15 };

ModifyList(myList);

Console.WriteLine(myList.Count); // Output: 4, poiché è stato
aggiunto un elemento attraverso list.Add(100) che ha modificato
anche myList che ora avrà { 0, 10, 15, 100 } come valori

Console.WriteLine(myList[0]); // Output: 99 e non 1 poiché
l'operazione list[0] = 99 ha influenzato la lista myList , mentre
l'operazione list = new List<int> { 1, 2, 3 } ha riassegnato list
e quindi ha lavorato su altra zona di memoria non influenzando
myList

```

RICERCHE

RICERCA SEQUENZIALE

Quando i valori dell'array o della lista non hanno uno specifico ordinamento l'unico modo per ricercare un valore all'interno di essa è quello di cercarlo a partire dalla prima posizione (la posizione 0) ed andare avanti di una posizione alla volta sino al suo ritrovamento o sino al termine dei valori.

Il tempo di ricerca in questo caso è un tempo lineare $O(n)$

```

public bool FindElement(int value)
{
    foreach (int number in numbers)
    {
        if (number == value)
            return true;
    }
}

```

```

        return false;
    }

    public bool FindElement(int value)
    {
        return numbers.Contains(value);
    }

```

RICERCA DICOTOMICA (O BINARIA)

Quando i valori presenti nell'array o nella lista sono ordinati, il modo migliore per cercare un valore al suo interno è la ricerca binaria o dicotomica.

Il funzionamento della ricerca binaria consiste nel valutare l'elemento al centro e verificare se è quello che cerchiamo oppure no. In caso non sia il valore che cerchiamo possiamo scartare tutti valori che si trovano prima o dopo di esso a seconda che il valore risulta minore o maggiore del valore cercato. Se i dati sono ordinati quindi se il valore che cerco è più piccolo sarà sicuramente prima, in caso contrario sarà sicuramente dopo.

Supponiamo di ricercare il valore 10

0	3	5	6	10	15	20
0	1	2	3	4	5	6

il primo controllo verifica se il valore al centro (il 6) è quello che cerco, non essendo quello che cerco, ed essendo minore, so per certo che posso scartare tutti i dati prima di lui, quindi ora la mia ricerca si concentrerà solo sui valori che si trovano dopo.

Ora andremo quindi a valutare il valore che si trova al centro dell'intervallo rimanente.

0	3	5	6	10	15	20
0	1	2	3	4	5	6

il 15 è maggiore del valore che cerco quindi posso scartare tutti valori che si trovano dopo di lui. Ora andremo quindi a valutare il valore che si trova al centro dell'intervallo rimanente.

0	3	5	6	10	15	20
0	1	2	3	4	5	6

il 10 è il valore che cercavo quindi potrò rispondere che è presente. Diversamente, se avessi cercato per esempio il numero 11 che non è presente nei valori dell'array avrei potuto rispondere che il numero non c'era perchè scartando il 10 ho scartato tutti i valori.

Il tempo di ricerca in questo caso è $\log_2 N$

La classe List<T> preve già il metodo BinarySearch

Esempio:

esempio di implementazione di un metodo BinarySearch per la ricerca di un elemento all'interno di un array ordinato di valori interi (ipotizziamo di essere in una classe e di avere come attributo della classe un array di interi int[] array opportunamente istanziato e riempito. La funzione restituisce la posizione in cui si trova l'elemento da cercare oppure null se l'elemento non è presente

```
int? BinarySearch(int element)
{
    int start = 0, end = array.Length - 1, middle= 0;

    while (start <= end)
    {
        middle= (start + end) / 2;
        if (element < array[middle])
        {
            end = middle- 1;
        }
        else
        {
            if (element > array[middle])
                start = middle + 1;
            else
                return middle;
        }
    }
    return null;
}
```

ORDINAMENTI

La classe List<T> preve già il metodo per ordinare gli elementi presenti al suo interno

SELECTION SORT

Il Selection Sort è un algoritmo di ordinamento semplice, ma potenzialmente inefficiente per grandi insiemi di dati. Funziona individuando ripetutamente il minimo elemento in una porzione non ancora ordinata dell'array e scambiandolo con il primo elemento di quella porzione.

Come Funziona:

1. **Trova il minimo:** Si inizia considerando il primo elemento dell'array come il minimo provvisorio. Si scorre poi il resto dell'array alla ricerca di un elemento più piccolo.
2. **Scambia:** Se viene trovato un elemento più piccolo, si scambiano di posto il minimo provvisorio e l'elemento appena trovato.
3. **Ripeti:** Si ripete il processo a partire dal secondo elemento non ancora ordinato, fino a che l'intero array non è ordinato.

Esempio:

Supponiamo di avere l'array: [64, 25, 12, 22, 11].

1. **Passaggio 1:**
 - Il minimo è 64 (il primo elemento).
 - Si scorre l'array: si trova 11 che è più piccolo di 64.
 - Si scambiano 64 e 11. L'array diventa: [11, 25, 12, 22, 64].
2. **Passaggio 2:**
 - Si parte dal secondo elemento (25).
 - Il minimo è 12.
 - Si scambiano 25 e 12. L'array diventa: [11, 12, 25, 22, 64].
3. **Si continua così fino a che l'array è completamente ordinato.**

Il nome deriva dal fatto che ad ogni iterazione si "seleziona" il minimo elemento rimanente e lo si posiziona nella sua posizione corretta.

- **Vantaggi:** Facile da implementare, richiede una memoria minima aggiuntiva.
- **Svantaggi:** Inefficiente per grandi array, il numero di confronti è sempre lo stesso indipendentemente dall'ordine iniziale dei dati.

Ipotizziamo che arr sia un attributo di tipo array di interi presente nella classe

```
public void SelectionSort()
{
    int n = arr.Length;

    for (int i = 0; i < n - 1; i++)
    {
        // cerco il minimo
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Scambio
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
    }
}
```

```

        arr[i] = temp;
    }
}

```

BUBBLE SORT

Il Bubble Sort è un algoritmo di ordinamento semplice da comprendere ma generalmente inefficiente per grandi insiemi di dati.

Come funziona:

Il Bubble Sort confronta a coppie gli elementi adiacenti di un array.

Se un elemento è maggiore di quello successivo, i due elementi vengono scambiati tra di loro.

Questo processo viene ripetuto fino a quando non ci sono più scambi da effettuare, questo indicherà che l'array è ordinato.

- **Confronto a coppie:** Si parte dal primo elemento dell'array e si confronta con il secondo.
- **Scambio (se necessario):** Se il primo elemento è maggiore del secondo, vengono scambiati di posto.
- **Ripeto i confronti sino al termine dell'array:** Si passa al secondo e terzo elemento e si ripete il confronto e lo scambio, se necessario.
- **Ripetere i passaggi:** Si ripetono questi passaggi fino a quando l'intero array è ordinato.

Ipotizziamo che arr sia un attributo di tipo array di interi presente nella classe

```

public void BubbleSort()
{
    for (int i = 0; i < array.Length; i++)
    {
        for (int j = 0; j < array.Length - 1 - i; j++)
        {
            if (array[j] > array[j + 1])
            {
                int temp = array[j + 1];
                array[j + 1] = array[j];
                array[j] = temp;
            }
        }
    }
}

```

BUBBLE SORT CON SENTINELLA

Il Bubble Sort con sentinella è un Bubble Sort ottimizzato, nel senso che si accorge se l'array è ordinato ed in caso affermativo interrompe la sua esecuzione. In questo modo, se l'array è parzialmente ordinato o la distribuzione dei valori è favorevole il Bubble Sort migliora le sue performance. Nel caso peggiore rimane però $O(n^2)$.

```

public void BubbleSortWithSentinel()

```

```

{
    int lastSwap = array.Length-1;
    int timeSwap = array.Length - 1;
    while (lastSwap != 0)
    {
        for (int i = 0; i < lastSwap; i++)
        {
            if (array[i] > array[i + 1])
            {
                int temp = array[i + 1];
                array[i + 1] = array[i];
                array[i] = temp;
                timeSwap = i;
            }
        }
        if (timeSwap == lastSwap) { lastSwap = 0; }
        lastSwap = timeSwap;
    }
}

```

INSERTION SORT

Come funziona:

L'algoritmo ordina l'array procedendo con un inserimento dei dati in ordine, inserendoli ad uno ad uno nel posto corretto rispetto alla parte ordinata dei valori presenti.

L'Insertion Sort funziona in modo simile.

1. **Suddivisione dell'array:** L'array viene considerato come composto da una parte ordinata (inizialmente vuota) e una parte non ordinata.
2. **Inserimento:** Si prende il primo elemento della parte non ordinata e lo si inserisce nella posizione corretta all'interno della parte ordinata, spostando gli elementi più grandi di una posizione verso destra.
3. **Ripetizione:** Si ripete il passo 2 fino a quando tutti gli elementi sono stati inseriti nella parte ordinata.

Ipotizziamo che arr sia un attributo di tipo array di interi presente nella classe

```

public void InsertionSort()
{

```

```

    int n = arr.Length;
    for (int i = 1; i < n; ++i)
    {
        int key = arr[i];
        int j = i - 1;

```

```

        // Sposta gli elementi maggiori di key di una posizione a destra
    }
}

```

```

while (j >= 0 && arr[j] > key)
{
    arr[j + 1] = arr[j];
    j = j - 1;
}
arr[j + 1] = key;
}
}

```

ARRAY SORT

[https://learn.microsoft.com/it-it/dotnet/api/system.array.sort?view=net-8.0#system-array-sort\(system-array\)](https://learn.microsoft.com/it-it/dotnet/api/system.array.sort?view=net-8.0#system-array-sort(system-array))

Ordina gli elementi in un intero [Array](#) unidimensionale usando l'implementazione [Comparable](#) di ogni elemento del [Array](#).

```
public static void Sort (Array array);
```

Esempio di uso:

```
String[] words = { "The", "QUICK", "BROWN", "FOX", "jumps", "over", "the", "lazy", "dog" };
Array.Sort(words); // → ordina le parole in ordine alfabetico
```

(vedi esempio su classroom (Search e Sort su Array) e appendice su Interfacce)

```

namespace MarksAndStudent
{
    public enum Result
    {
        ADMITTED,
        NOTADMITTED,
        SUSPENDED
    }
    public class PascalStudent: IComparable<PascalStudent>
    {
        private int[] _marks;
        public string Name { get; private set; }
        public string Surname { get; private set; }

        public PascalStudent(bool isBiennium, string name, string surname)
        {
            if (string.IsNullOrEmpty(name)) throw new ArgumentNullException("name");
            if (string.IsNullOrEmpty(surname)) throw new ArgumentNullException("surname");

            if (isBiennium)
                _marks = new int[13];
            else
                _marks = new int[10];
        }
    }
}

```

```

    Name = name;
    Surname = surname;
}

```

```

public double average()
{
    int sum = 0;
    int count = 0;
    foreach (int mark in _marks)
    {
        if (mark == 0)
            count++;
        else
            sum += mark;
    }
    if (count == _marks.Length)
        return 0;
    else
        return (double)sum / (_marks.Length - count);
}

```

```

/// <summary>
/// ordinamento degli studenti per media (per primo lo studente con la media più alta) (a parità
di media per cognome)
/// </summary>
/// <param name="other"></param>
/// <returns></returns>
public int CompareTo(PascalStudent? other)
{
    if (other == null || (!(other is PascalStudent))) throw new ArgumentException("tipo di dato
sbagliato");

    if (average() > other.average())
    {
        return -1;
    }
    else
    {
        if (average() == other.average())
        {
            return Surname.CompareTo(other.Surname);
        }
        else
        {
            return 1;
        }
    }
}

```

```

    }
}
}
}

```

CONFRONTI SUGLI ALGORITMI

Algoritmo	Tipo	Migliore	Medio	Peggior	Osservazioni
Ordinamento					
Bubble Sort	Confronto	$O(n)$	$O(n^2)$	$O(n^2)$	Semplice, ma non efficiente per grandi array.
Insertion Sort	Inserimento	$O(n)$	$O(n^2)$	$O(n^2)$	Efficiente per array quasi ordinati.
Selection Sort	Selezione	$O(n^2)$	$O(n^2)$	$O(n^2)$	Semplice, ma non efficiente (in ogni caso).
Merge Sort	Divide et impera	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Efficiente, stabile, richiede memoria aggiuntiva.
Quick Sort	Divide et impera	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Efficiente in media, ma sensibile all'ordine iniziale.
Ricerca					
Ricerca Sequenziale	Confronto	$O(1)$	$O(n/2)$	$O(n)$	Semplice, ma non efficiente per grandi array.
Ricerca Binaria	Divide et impera	$O(1)$	$O(\log n)$	$O(\log n)$	Efficiente ma funziona solo su array ordinati.

Esempio con qualche numero ipotizzando per il calcolo del tempo di essere in grado di eseguire 100000000 di operazioni al secondo

complessità	n =	tempo	n =	tempo	n =	tempo
-------------	-----	-------	-----	-------	-----	-------

computazionale	1000		1000000		1000000000	
ordinamento						
Selection Sort $O(n^2)$	1000000	0.001s	1000000000 000	1000s $\approx 16\text{min}$	10000000000 00000000	≈ 31 anni
Merge Sort $O(n \log_2 n)$	≈ 10000	0,00001 s	≈ 20000000	$\approx 0,02\text{s}$	≈ 3000000000 0	$\approx 30\text{s}$
ricerca						
Ricerca Sequenziale $O(n)$	1000	0.00000 1s	1000000	0.001s	1000000000	1 s
Ricerca Binaria $O(\log_2 n)$	≈ 10	0.00000 001s	≈ 20	0,00000002s	≈ 30	0,00000003s

RICORDA:

$\log_2 n = x \rightarrow 2^x = n$

$\log_2 1000 = \text{circa } 10$

$\log_2 1000000 = \text{circa } 20$

$\log_2 1000000000 = \text{circa } 30$

Domanda:

Conviene ordinare i valori prima di effettuare la ricerca?

se devo fare 1 ricerca non conviene, se devo fare molte ricerche qualche volta potrebbe convenire (guarda bene la tabella sopra per capire!)

Grafici delle complessità

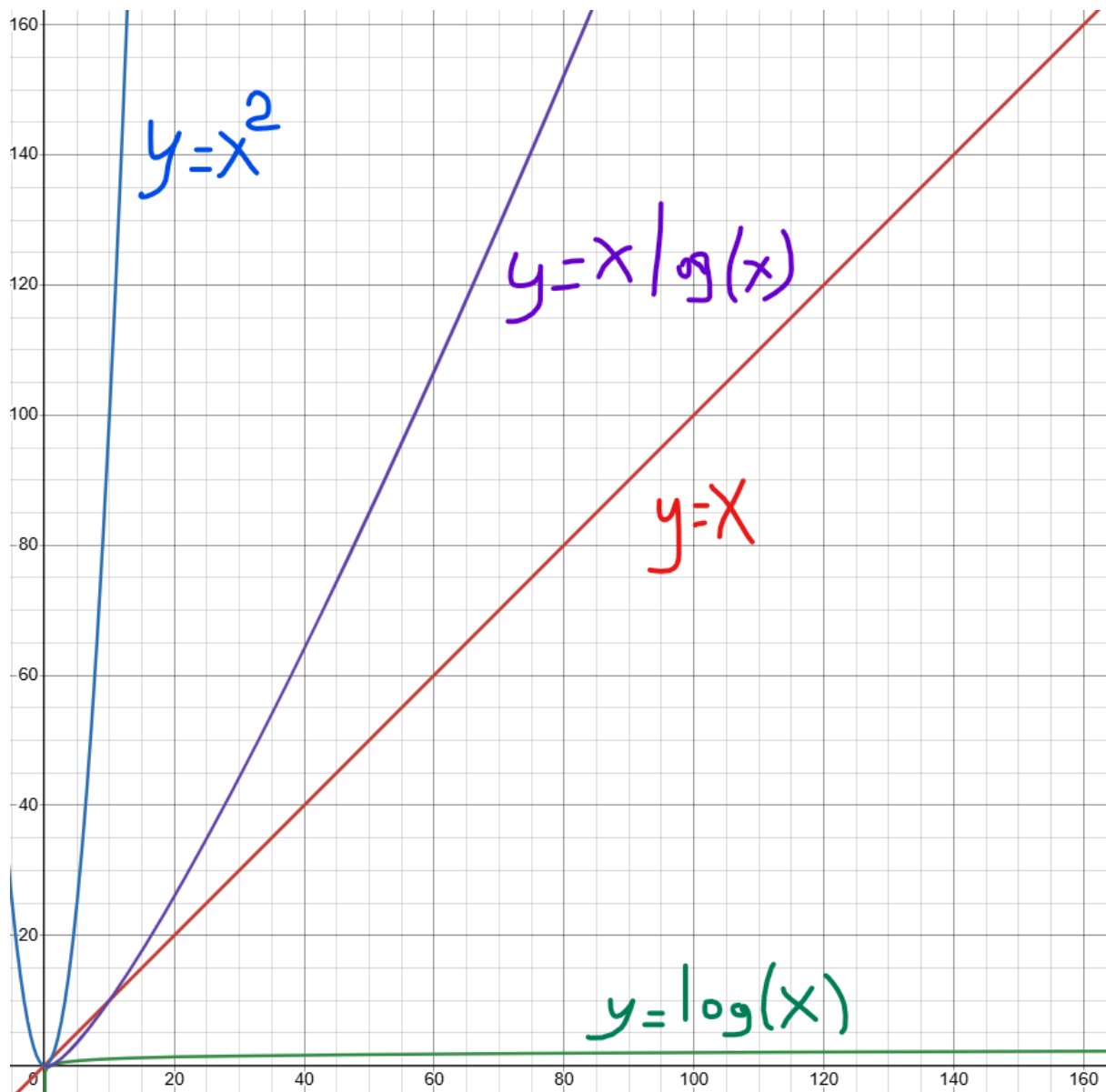
$O(n)$ - Lineare: Il grafico è una linea retta che cresce linearmente con l'aumentare di n . Ciò significa che il tempo di esecuzione dell'algoritmo aumenta proporzionalmente alla dimensione dell'input. (nel grafico l'andamento è quello della retta rossa che rappresenta $y=x$)

$O(n^2)$ - Quadratico: Il grafico è una parabola che cresce molto più rapidamente della linea retta. Il tempo di esecuzione aumenta in modo quadratico rispetto alla dimensione dell'input. (nel grafico l'andamento è quello della retta blu che rappresenta $y=x^2$)

$O(\log n)$ - Logaritmico: Il grafico cresce molto lentamente, quasi piatto. All'aumentare di n , il tempo di esecuzione aumenta molto lentamente. (nel grafico l'andamento è quello della retta verde che rappresenta $y=\log(x)$)

$O(n \log n)$ - Lineare-logaritmico: Il grafico cresce più rapidamente di $O(\log n)$ ma più lentamente di $O(n^2)$. È una sorta di compromesso tra i due. (nel grafico l'andamento è quello della retta viola che rappresenta $y=x \log(x)$)

Naturalmente ipotizzando di avere nell'asse delle x la dimensioni dell'input e nell'asse delle y il tempo di esecuzione, la parte di grafico di interesse è solo il primo quadrante (dove la dimensione dell'input è positiva)



ARRAY BIDIMENSIONALI

Un array bidimensionale o matrice è una struttura dati statica ed omogenea in cui i dati sono organizzati per righe e colonne.

Quando si dichiara una matrice è necessario definire il numero delle righe e delle colonne (il primo valore rappresenta il numero delle righe ed il secondo il numero delle colonne)

Definizione di un array bidimensionale

```
int[,] arrayBidimensionale = new int[3, 4]; // Un array 3x4 di interi
```

Accesso agli elementi della matrice

Per scorrere gli elementi della matrice è necessario muoversi nelle due dimensioni spostandosi per riga oppure per colonna.

Quando ci si riferisce ad un elemento della matrice si hanno due indici, il primo rappresenta l'indice di riga ed il secondo l'indice di colonne.

Essendo array, ovviamente anche gli indici delle matrici sono zero-based

```
// Dichiarazione e inizializzazione di una matrice 3x4
```

```
int[,] matrice = new int[3, 4];
```

```
// Inizializzazione della matrice per riga utilizzando l'espressione i * 10 + j
```

```
for (int i = 0; i < matrice.GetLength(0); i++) // Dimensione della prima dimensione (righe)
```

```
{
```

```
    for (int j = 0; j < matrice.GetLength(1); j++) // Dimensione della seconda dimensione (colonne)
```

```
    {
```

```
        matrice[i, j] = i * 10 + j;
```

```
    }
```

```
}
```

la matrice risultante avrà questi valori:

```
0  1  2  3
```

```
10 11 12 13
```

```
20 21 22 23
```

eseguendo invece questo codice

```
// Dichiarazione e inizializzazione di una matrice 3x4
```

```
int[,] matrice = new int[3, 4];
```

```
// Inizializzazione della matrice per colonna utilizzando l'espressione i * 10 + j
```

```
for (int i = 0; i < matrice.GetLength(1); i++) // Dimensione delle colonne
```

```
{
```

```
    for (int j = 0; j < matrice.GetLength(0); j++) // Dimensione delle righe
```

```
    {
```

```
        matrice[j, i] = i * 10 + j;
```

```
    }
```

```
}
```

la matrice risultante avrà questi valori:

```
0  1  2
```

```
10 11 12
```

20 21 22
30 31 32

Le Matrici in memoria

Le matrici, ed in generale gli array multidimensionali in memoria vengono gestiti come un array, ovvero gli elementi vengono memorizzati in modo sequenziale, poi gli indici vengono interpretati come se i dati fossero memorizzati in forma tabellare.

ARRAY JAGGED

In c# è possibile definire un array di array (array jagged); in questo caso, a differenza delle matrici, ogni sotto-array può avere una dimensione diversa.

Attenzione: gli array jagged (array di array, come int[][]) sono gestiti in modo diverso in memoria rispetto alle matrici; gli array jagged sono effettivamente array di array separati.

Esempio

```
// Dichiarazione e inizializzazione di un array jagged per rappresentare una tavola da gioco
int[][] tavolaDaGioco = new int[][]
{
    new int[] {1, 2, 3},
    new int[] {4, 5},
    new int[] {6, 7, 8, 9}
};

// Stampa della tavola da gioco
Console.WriteLine("Tavola da Gioco:");

for (int i = 0; i < tavolaDaGioco.Length; i++)
{
    for (int j = 0; j < tavolaDaGioco[i].Length; j++)
    {
        Console.Write(tavolaDaGioco[i][j] + "\t");
    }
    Console.WriteLine();
}
```

ARRAY JAGGED VS MATRICI

La scelta tra un array jagged (array di array) e una matrice (array bidimensionale) dipende dalle esigenze specifiche dell'applicazione e dai requisiti di prestazione.

Elementi da considerare per scegliere il tipo di struttura più opportuna:

Array Jagged:

- Dimensioni Diverse per Ogni Sottoarray: potere avere sottogruppi (sottoarray) con dimensioni diverse può essere utile se nel problema che si affronta la dimensione delle "righe" può variare da una all'altra.
- Allocazione Flessibile della Memoria: è possibile aggiungere o rimuovere sottoarray in modo dinamico; questo è utile quando la dimensione delle righe cambia dinamicamente.

Matrice:

- Struttura Tabellare: se tutte le righe e tutte le colonne hanno la stessa dimensione, una matrice può essere più intuitiva.
- Accesso più Efficiente: in memoria la matrice è gestita come un array, è quindi possibile effettuare un accesso diretto in base agli indici. (Negli array jagged l'indirizzamento è indiretto)

Gli array jagged offrono maggiore flessibilità, ma le matrici possono essere più adatte in situazioni in cui la struttura tabellare è chiaramente definita e la prestazione è una priorità.

STRING

<https://learn.microsoft.com/it-it/dotnet/csharp/programming-guide/strings/>

<https://learn.microsoft.com/en-us/dotnet/api/system.string?view=net-9.0>

Una stringa è un oggetto di tipo [String](#) il cui valore è testo. Internamente, il testo è memorizzato come una collezione sequenziale a sola lettura di oggetti [Char](#). La proprietà [Length](#) di una stringa rappresenta il numero di oggetti Char che contiene

In C# la parola chiave string è un alias per [String](#); pertanto, String e string sono equivalenti.

Gli oggetti stringa sono non modificabili: non possono essere modificati dopo la creazione. Tutti i metodi [String](#) e gli operatori C# che sembrano modificare una stringa restituiscono effettivamente i risultati in un nuovo oggetto stringa. Nell'esempio seguente, quando il contenuto di s1 e s2 vengono concatenati per formare una singola stringa, le due stringhe originali non vengono modificate. L'operatore += crea una nuova stringa contenente il contenuto combinato. Tale nuovo oggetto viene assegnato alla variabile s1 e l'oggetto originale che era stato assegnato a s1 viene liberato per la raccolta dei rifiuti perché nessun'altra variabile contiene un riferimento ad esso.

```
string s1 = "A string is more ";  
string s2 = "than the sum of its chars.";  
  
// Concatenate s1 and s2. This actually creates a new  
// string object and stores it in s1, releasing the  
// reference to the original object.  
s1 += s2;  
  
System.Console.WriteLine(s1);  
// Output: A string is more than the sum of its chars.
```

Studia tutto il contenuto della guida al link sopra riportato.

PERSISTENZA DEI DATI

I/O da file di testo

Spesso abbiamo la necessità di salvare dei dati, ad esempio le impostazioni di default di un applicativo, in modo da averle a disposizione per gli usi successivi e non doverle reinserire ad ogni utilizzo del software.

Può quindi essere utile scrivere queste informazioni su un file di testo, in modo che queste possano essere poi lette all'avvio del programma.

Esistono altri tipi di file oltre ai file di testo, ad esempio i file xml o i file Json o, ad oggi quasi non più utilizzati, i file binari.

Per il momento ci concentriamo sui file di testo e vedremo più avanti altri tipi di file.

I file per poter essere letti o scritti devono essere prima aperti ed infine chiusi.

L'utilizzo dello statement **using** ci permette di creare l'istanza di StreamReader o StreamWriter e chiuderla automaticamente al termine dell'utilizzo.

Ricorda che i file di testo sono **sequenziali**.

//qui creo l'istanza ad esempio di un oggetto StreamReader

```
using (StreamReader sr = new StreamReader("TestFile.txt"))
```

```
{
```

```
.....
```

```
}//qui automaticamente viene chiusa
```

StreamReader e StreamWriter utilizzano come codifica predefinita UTF-8. Se serve utilizzare una codifica diversa è possibile specificarla nel costruttore

LEGGERE DATI DA UN FILE DI TESTO

<https://learn.microsoft.com/it-it/dotnet/api/system.io.streamreader?view=net-7.0>

StreamReader è una classe presente nello spazio dei nomi System.IO che ci permette di leggere in modo sequenziale i caratteri da un flusso di byte, interpretandoli come testo secondo una codifica specifica. In altre parole, ci consente di aprire un file di testo e di leggerne il contenuto riga per riga. La classe StreamReader per poter leggere dati da un file di testo dovrà aprire, leggere ed infine chiudere il file di testo.

Se si tenta di aprire in lettura un file di testo che non esiste verrà generata una exception.

Si può passare il percorso di un file di testo al costruttore StreamReader per aprirlo automaticamente.

Metodi per la lettura dei dati:

- **ReadLine():** Legge una riga alla volta. Restituisce null quando si raggiunge la fine del file.
- **ReadToEnd():** Legge tutto il contenuto del file e lo restituisce salvato in una singola stringa.
- **Read():** Legge un singolo carattere.

Il metodo **ReadLine**, quello utilizzato più comunemente quando le informazioni memorizzate all'interno del file sono strutturate a righe (esempio file .csv) legge ogni riga di testo e incrementa il puntatore del file alla riga successiva durante la lettura.

Quando il metodo **ReadLine** raggiunge la fine del file, restituisce un riferimento null.

```
using System;
using System.IO;
```

```
...
```

```
try
{
    // Create an instance of StreamReader to read from a file.
    // The using statement also closes the StreamReader.
    using (StreamReader sr = new StreamReader("TestFile.txt"))
    {
        string line;
        // Read and display lines from the file until the end of
        // the file is reached.
        while ((line = sr.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }
}
catch (Exception e)
{
    // Let the user know what went wrong.
    Console.WriteLine("The file could not be read:");
    Console.WriteLine(e.Message);
}
```

Se volessimo leggere l'intero contenuto del file e salvarlo in una stringa potremmo utilizzare la seguente modalità:

```
string tuttoIlContenuto = reader.ReadToEnd();
```

SCRIVERE DATI SU UN FILE DI TESTO

<https://learn.microsoft.com/it-it/dotnet/api/system.io.streamwriter?view=net-8.0>

La classe **StreamWriter** ci permette di scrivere dati su un file di testo; per fare questo dovrà aprire, scrivere e chiudere il file di testo.

Se si tenta di scrivere un file di testo che non esiste, a differenza della lettura, non viene generato nessun errore, ed il file viene creato.

In modo analogo alla classe StreamReader, è possibile passare il percorso di un file di testo al costruttore StreamWriter per aprire automaticamente il file.

Per scrivere testo nel file possiamo utilizzare i metodi:

- **Write(string)**: Scrive una stringa nel file.
- **WriteLine(string)**: Scrive una stringa nel file e aggiunge un a capo.

```
using System.Text;
```

```
using System.IO;
```

```
...
```

```
// Get the directories currently on the C drive.
```

```
DirectoryInfo[] cDirs = new DirectoryInfo(@"c:\").GetDirectories();
```

```
// Write each directory name to a file.
```

```
using (StreamWriter sw = new StreamWriter("CDriveDirs.txt"))
```

```
{
```

```
    foreach (DirectoryInfo dir in cDirs)
```

```
    {
```

```
        sw.WriteLine(dir.Name);
```

```
    }
```

```
}
```

```
...
```

```
// creo un array di stringhe
```

```
string[] lines = { "First line", "Second line", "Third line" };
```

```
// imposto una variabile al path in cui voglio scrivere il documento.
```

```
string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

```
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

```
...
```



```
//Creo il file di testo e Scrivo ogni elemento dell'array di stringhe in una linea del file di testo che
chiamo "WriteLines.txt".
using (StreamWriter outputFile = new StreamWriter(Path.Combine(docPath, "WriteLines.txt")))
{
    foreach (string line in lines)
        outputFile.WriteLine(line);
}
```

AGGIUNGERE TESTO AD UN FILE GIÀ ESISTENTE

Quando apro un file di testo in modalità scrittura, se il file esiste il suo contenuto viene cancellato

Questo codice ad esempio produrrà un file con contenuto una riga di testo con scritto: "Example text in file"

...

```
string fileName = "test.txt";
string textToAdd = "Example text in file";

using (StreamWriter writer = new StreamWriter(fileName))
{
    writer.Write(textToAdd);
}
```

Se voglio mantenere il contenuto del file ed aggiungere a questo nuove informazioni devo specificarlo nel momento in cui richiamare il costruttore utilizzando il costruttore:

public StreamWriter (string path, bool append);

Questo codice produrrà un file in cui viene aggiunto al contenuto presente una riga con testo: "Example text in file"

...

```
string fileName = "test.txt";
string textToAdd = "Example text in file";

using (StreamWriter writer = new StreamWriter(fileName, true))
{
    writer.Write(textToAdd);
}
```

I/O da file JSON

I file JSON sono un formato di testo leggero e facile da leggere, spesso utilizzato per lo scambio di dati tra sistemi diversi e per memorizzare le configurazioni dei software. Sono particolarmente popolari nelle applicazioni web e mobile.

I file JSON sono organizzati in coppie chiave-valore. Le chiavi sono sempre stringhe, mentre i valori possono essere stringhe, numeri, booleani, array o altri oggetti JSON. La sintassi utilizzata dai file JSON è semplice ed intuitiva, si utilizzano le parentesi graffe per gli oggetti e quadre per gli array.

Esempio di file JSON:

```
{
  "nome": "Mario Rossi",
  "età": 30,
  "città": "Roma",
  "interessi": ["programmazione", "musica", "viaggi"]
}
```

Esistono varie librerie utilizzabili per lavorare con i file Json, principalmente System.Text.Json (libreria di sistema) e Newtonsoft.Json (libreria esterna da installare)

LEGGERE DATI DA UN FILE JSON

```
using System.Text.Json;
```

```
// Caricare il contenuto del file JSON in una stringa
string jsonString = File.ReadAllText("dati.json");

// Deserializzare la stringa JSON in un oggetto C#
// Assumiamo che la struttura del JSON corrisponda alla classe Persona
Persona persona = JsonSerializer.Deserialize<Persona>(jsonString);
```

```
// Utilizzare i dati dell'oggetto
Console.WriteLine($"Nome: {persona.Nome}");
Console.WriteLine($"Età: {persona.Eta}");
```

```
// Classe per rappresentare la struttura del JSON
public class Persona
{
    public string Nome { get; set; }
    public int Eta { get; set; }
}
```

SCRIVERE DATI SU UN FILE JSON

```
using System.Text.Json;
```

```
// Creare un oggetto C#
nome = "Mario Rossi";
eta = 30;
Persona nuovaPersona = new Persona(nome, eta);
```

```
// Serializzare l'oggetto in una stringa JSON
string jsonString = JsonSerializer.Serialize(nuovaPersona);

// Scrivere la stringa JSON in un file
File.WriteAllText("nuova_persona.json", jsonString);
```

APPENDICE

Classe Random

Classe per la generazione di valori pseudo casuali.

La classe fa parte di System quindi non servono using aggiuntivi.

Creando un oggetto di tipo Random ho poi a disposizione i metodi Next (per i quali esistono varie definizioni ovvero varie sovrascritture dei parametri), attraverso i quali posso generare dati casuali di vario tipo.

Esempio

```
Random rand = new Random();  
  
int numero = rand.Next(1,6); → generi un numero tra 1 e 5  
  
int numero = rand.Next(1,7); → generi un numero tra 1 e 6  
  
int numero = rand.Next(1,n); → generi un numero tra 1 e n escluso
```

<https://learn.microsoft.com/it-it/dotnet/api/system.random?view=net-8.0>

<https://learn.microsoft.com/it-it/dotnet/api/system.random.next?view=net-8.0>

Complessità Computazionale

La **complessità computazionale** è una misura del tempo o dello spazio (memoria) necessario a un algoritmo per risolvere un problema, il suo calcolo viene svolto in funzione dell'input ricevuto.

La **notazione O grande** o **Big O** viene utilizzata per descrivere il limite superiore della crescita di una funzione, quindi rappresenta la velocità con cui aumenta il tempo di esecuzione o l'utilizzo di memoria in funzione alla dimensione dell'input, che solitamente viene indicato con la variabile n .

O(1): Complessità Costante

La notazione **O(1)** indica una complessità costante.

Un'operazione ha complessità **O(1)** se il tempo di esecuzione (o lo spazio utilizzato) non cambia al variare della dimensione dell'input, ovvero, indipendentemente da quanto sia grande l'input, l'algoritmo impiega sempre lo stesso tempo per la sua esecuzione.

Questo tipo di complessità è molto efficiente e spesso rappresenta il caso migliore.

Un esempio di algoritmo con complessità **O(1)** è l'accesso ad un elemento di un array

Esempio:

```
int[] array = { 1, 2, 3, 4, 5 };  
  
int firstElement = array[0];
```

Accedere al primo elemento di un array (o in generale all'elemento di posizione i) richiede sempre lo stesso tempo, indipendentemente dalla lunghezza dell'array.

Principali Notazioni per il calcolo della complessità computazionale

- **O(log n)**: Complessità logaritmica. Usata in algoritmi come la ricerca binaria, in cui la dimensione dell'input viene ridotta a metà a ogni passo.
- **O(n)**: Complessità lineare. L'algoritmo cresce proporzionalmente alla dimensione dell'input per esempio la ricerca di un elemento all'interno di un array, che richiede, nel caso peggiore, lo scorrimento di tutti gli elementi di un array ha questo tipo di complessità.
- **O($n \log n$)**: Complessità quasi lineare, tipica di molti algoritmi di ordinamento efficienti come il mergesort.
- **O(n^2)**: Complessità quadratica. Il tempo di esecuzione cresce proporzionalmente al quadrato della dimensione dell'input, come in algoritmi di ordinamento meno efficienti (es. bubblesort).
- **O(2^n)**: Complessità esponenziale. Ogni incremento dell'input raddoppia il tempo di esecuzione, spesso presente in problemi di combinatoria.
- **O($n!$)**: Complessità fattoriale, che cresce estremamente velocemente, come nei problemi di permutazione.

GENERICCS

<https://learn.microsoft.com/it-it/dotnet/csharp/fundamentals/types/generics>

I generics consentono di progettare classi e metodi che rinviando la specifica di uno o più parametri di tipo fino a quando non si usa la classe o il metodo nel codice.

// Declare the generic class.

```
public class GenericList<T>
{
    public void Add(T item) { }
}

public class ExampleClass { }

class TestGenericList
{
    static void Main()
    {
        // Create a list of type int.
        GenericList<int> list1 = new();
        list1.Add(1);

        // Create a list of type string.
        GenericList<string> list2 = new();
        list2.Add("");

        // Create a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new();
        list3.Add(new ExampleClass());
    }
}
```

Le classi e i metodi generici combinano riutilizzabilità, sicurezza dei tipi ed efficienza. I parametri di tipo generico vengono sostituiti con gli argomenti di tipo durante la compilazione. **Nell'esempio precedente il compilatore sostituisce T con int.** I generics sono in genere usati con le raccolte e i metodi che operano su di essi.

CLASSI STATIC

Le classi statiche sono **classi che non possono essere istanziate**, il che significa che non puoi creare oggetti da esse. **Tutti i membri di una classe statica (campi, metodi, proprietà) sono anch'essi statici** e vengono condivisi tra tutte le istanze della classe (che, tecnicamente, non esistono).

Quando è bene utilizzarle:

- **Metodi di utilità:**
 - Le classi statiche sono perfette per **raggruppare metodi di utilità** che non dipendono dallo stato di un oggetto. Ad esempio, una classe statica "Math" potrebbe contenere metodi per eseguire calcoli matematici, o una classe "StringUtil" potrebbe contenere metodi per manipolare stringhe.
- **Dati condivisi:**
 - Le classi statiche possono essere utilizzate per memorizzare dati che devono essere condivisi tra tutte le parti del tuo programma. Ad esempio, potresti avere una classe statica che memorizza le impostazioni di configurazione dell'applicazione. Per noi solitamente però sarà preferibile **memorizzarle in un file e leggerle all'apertura!**

Quando **NON devono essere utilizzate**:

- Se la tua classe ha bisogno di mantenere uno stato (cioè, se i suoi campi cambiano nel tempo), non dovresti usare una classe statica. Le classi statiche non sono progettate per gestire lo stato dell'oggetto.
- Le classi statiche non supportano il polimorfismo, il che significa che non puoi sostituire un metodo statico con un metodo in una classe derivata.
- Le classi statiche possono rendere il tuo codice più difficile da testare, poiché i loro metodi non possono essere facilmente sostituiti con "mock" (oggetti fittizi) durante i test unitari.
- L'uso eccessivo di classi statiche può portare a dipendenze nascoste nel tuo codice, rendendolo più difficile da capire e da mantenere.

Le classi statiche sono uno strumento utile per raggruppare metodi di utilità e funzioni globali. Tuttavia, è **importante utilizzarle con parsimonia e considerare attentamente se sono la scelta giusta** per il tuo caso d'uso.

Ad esempio è corretto utilizzare una classe static per implementare un Helper per la gestione delle stringhe

```
public static class StringHelper
{
    public static string Reverse(string input)
    {
        char[] chars = input.ToCharArray();
        Array.Reverse(chars);
        return new string(chars);
    }
}
```

```
}  
}
```

```
// Utilizzo  
//richiamo il metodo senza istanziare un oggetto
```

```
string reversed = StringHelper.Reverse("ciao"); // reversed conterrà "oaic"
```

CLASSI PARTIAL

In **C#**, una **partial class** è una classe che può essere suddivisa in più file. Questo permette di organizzare meglio il codice, specialmente quando una classe diventa molto grande o quando strumenti come **designer visuali** generano automaticamente una parte della classe.

l'utilizzo di classi di tipo Partial può essere vantaggioso per

1. **Migliorare l'organizzazione:** si può suddividere una classe grande in più file.
2. **Facilitare la gestione del codice generato automaticamente:** ad esempio, nei progetti **WPF** le UI generate dal designer usano **partial class** per separare il codice UI (il file xaml) da quello scritto dallo sviluppatore in c# (il file .xaml.cs).

Le classi generate automaticamente da **WPF**, come **MainWindow**, sono partial perché il codice della finestra è diviso in più file per separare la definizione dell'interfaccia utente (XAML) dalla logica di programmazione (C#).

Quando creiamo una Window in WPF vengono generati 2 file:

Window.xaml → che contiene lo xaml che descrive la struttura della finestra ed i suoi componenti.

Window.xaml.cs → che contiene la logica della finestra. Questa classe è di tipo partial perché il compilatore genera automaticamente un file **Window.g.cs** (visibile nella cartella obj/Debug) il quale collega i controlli definiti nello xaml al file cs che contiene la definizione della classe; è qui infatti che troviamo la definizione del metodo `InitializeComponent` che viene generata a partire dal contenuto dello xaml

Le classi delle finestre in **WPF** sono **partial** perché devono combinare codice generato automaticamente con la logica scritta dallo sviluppatore, mantenendo separati il **layout UI (XAML)** e il **codice C#** in modo organizzato e scalabile.

INTERFACCE

Le interfacce sono uno strumento fondamentale nella programmazione ad oggetti (OOP) e si inseriscono in diverse caratteristiche chiave come astrazione, polimorfismo ed incapsulamento.

Le interfacce contengono al loro interno solo firme di metodi (non contengono attributi o costruttori)

Vengono definite delle interfacce per intercettare, descrivere, comportamenti che sono trasversali rispetto alle gerarchie di classi (esempio: se voglio poter ordinare oggetti, il funzionamento dell'ordinamento è indipendentemente dalla tipologia di classi a cui appartengono gli oggetti)

Una classe può ereditare da una sola classe base ma può implementare più interfacce

Quando una classe implementa un'interfaccia si assume l'obbligo di implementare i metodi che questa definisce.

Possiamo pensare ad un'interfaccia come ad un contratto. Essa infatti definisce un insieme di metodi che una classe si impegna a implementare. In altre parole, un'interfaccia specifica **cosa** una classe deve fare, senza dire **come** farlo.

A cosa servono:

- **Standardizzazione:** Definiscono un comportamento comune che diverse classi possono condividere.
- **Polimorfismo:** Permettono di trattare oggetti di classi diverse in modo uniforme, se queste classi implementano la stessa interfaccia.
- **Decoupling:** Scollegano l'utilizzo di un oggetto dalla sua implementazione concreta.

Le interfacce sono uno strumento molto potente e sono fondamentali per creare codice più flessibile, riutilizzabile e mantenibile.

Perché usare le interfacce:

- **Flessibilità:** Permettono di sostituire un'implementazione con un'altra senza modificare il codice cliente, purché l'interfaccia rimanga la stessa.
- **Riutilizzabilità:** Un'interfaccia può essere utilizzata da molte classi diverse.
- **Test:** Facilita la creazione di test unitari, poiché puoi testare il comportamento di un'interfaccia senza conoscere l'implementazione specifica.
- **Polimorfismo:** Permette di trattare oggetti di tipo diverso in modo uniforme, se implementano la stessa interfaccia.

Quando usare le interfacce:

- Quando vuoi definire un contratto che diverse classi devono rispettare.
- Quando vuoi creare componenti riutilizzabili.
- Quando vuoi isolare l'implementazione dai dettagli di utilizzo.

Le interfacce di sistema per gli ordinamenti

In C#, si possono utilizzare le interfacce di sistema come [IComparer<T>](#) e [IComparable<T>](#) per implementare ordinamenti personalizzati su collezioni di oggetti.

[IComparer<T>](#)

Ipotizziamo di avere una classe `Persona` e di voler ordinare una collezione di oggetti `Persona` in base a parametri diversi a seconda della esigenza. In questo caso possiamo sfruttare l'uso di `IComparer`, definendo classi che implementano l'interfaccia (ovvero che implementano il metodo [Compare\(Object, Object\)](#)) e ne definiscono il comportamento di ordinamento specifico

```
using System;
using System.Collections.Generic;

public class Persona
{
    public string Nome { get; set; }
    public int Età { get; set; }

    public Persona(string nome, int età)
    {
        Nome = nome;
        Età = età;
    }

    public override string ToString()
    {
        return $"{Nome}, {Età} anni";
    }
}

// Comparatore per ordinare per Nome
public class OrdinaPerNome : IComparer<Persona>
{
    public int Compare(Persona x, Persona y)
    {
        if (x == null || y == null)
            throw new ArgumentException("Entrambi gli oggetti
            devono essere diversi da null");

        return string.Compare(x.Nome, y.Nome);
    }
}

// Comparatore per ordinare per Età
```

```

public class OrdinaPerEtà : IComparer<Persona>
{
    public int Compare(Persona x, Persona y)
    {
        if (x == null || y == null)
            throw new ArgumentException("Entrambi gli oggetti
devono essere diversi da null");

        return x.Età.CompareTo(y.Età);
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Creiamo un elenco di persone
        List<Persona> persone = new List<Persona>
        {
            new Persona("Mario", 30),
            new Persona("Giulia", 25),
            new Persona("Alberto", 40),
            new Persona("Lucia", 20)
        };

        // Ordinamento per Nome
        Console.WriteLine("Ordinamento per Nome:");
        persone.Sort(new OrdinaPerNome());
        foreach (var persona in persone)
        {
            Console.WriteLine(persona);
        }

        // Ordinamento per Età
        Console.WriteLine("\nOrdinamento per Età:");
        persone.Sort(new OrdinaPerEtà());
        foreach (var persona in persone)
        {
            Console.WriteLine(persona);
        }
    }
}

```

Classe **OrdinaPerNome**: Implementa l'interfaccia `IComparer<Persona>` per ordinare le persone in base al nome. Utilizza `string.Compare` per confrontare i nomi e realizzare un ordinamento alfabetico.

Classe **OrdinaPerEtà**: Implementa anch'essa `IComparer<Persona>`, ma per ordinare le persone in base all'età, usando il metodo `CompareTo`.

L'Ordinamento sulle collezioni di dati viene generato richiamando il metodo **Sort()** con i diversi comparatori.

L'interfaccia **`IComparer<T>`** è molto utile quando hai bisogno di **più criteri di ordinamento per la stessa classe**. Puoi infatti facilmente implementare più classi “comparatrici” senza modificare la classe principale

`IComparable<T>`

Se ho necessità di un **unico criterio di ordinamento** posso utilizzare l'interfaccia **`IComparable<T>`** la quale consente agli oggetti di confrontarsi tra loro.

Implementando questa interfaccia in una classe, sarà possibile definire la logica di ordinamento predefinita per gli oggetti di quella classe.

L'interfaccia prevede un metodo **`CompareTo`** che confronta l'istanza corrente con un altro oggetto dello stesso tipo e restituisce un valore intero che indica se l'istanza corrente precede, segue o si trova nella stessa posizione dell'altro oggetto all'interno dell'ordinamento; in particolare:

- se viene restituito un valore Minore di zero allora l'istanza corrente precede other nell'ordinamento.
- se viene restituito un valore Zero allora l'istanza corrente si presenta nella stessa posizione di other nell'ordinamento.
- se viene restituito un valore Maggiore di zero allora l'istanza corrente segue other nell'ordinamento.

```
using System;
using System.Collections.Generic;

public class Persona : IComparable<Persona>
{
    public string Nome { get; set; }
    public int Età { get; set; }

    public Persona(string nome, int età)
    {
        Nome = nome;
        Età = età;
    }

    public override string ToString()
    {
        return $"{Nome}, {Età} anni";
    }
}
```

```

        /* Implementazione del metodo CompareTo dell'interfaccia
        IComparable
        Il metodo confronta l'oggetto corrente con un altro oggetto
        Persona. Qui, utilizziamo CompareTo dell'intero Età per
        determinare l'ordinamento. Se l'oggetto other è nullo,
        consideriamo l'oggetto corrente come più grande.*/
        public int CompareTo(Persona other)
        {
            if (other == null) return 1; // Considera gli oggetti
nulli come più piccoli
            return Età.CompareTo(other.Età); // Ordina in base all'età
        }
    }

class Program
{
    static void Main(string[] args)
    {
        // Creiamo un elenco di persone
        List<Persona> persone = new List<Persona>
        {
            new Persona("Mario", 30),
            new Persona("Giulia", 25),
            new Persona("Alberto", 40),
            new Persona("Lucia", 20)
        };

        // Ordinamento delle persone usando il metodo Sort
        persone.Sort();

        // Stampa dell'elenco ordinato
        Console.WriteLine("Persone ordinate per Età:");
        foreach (var persona in persone)
        {
            Console.WriteLine(persona);
        }
    }
}

```

DEPENDENCY INJECTION

La Dependency Injection, (DI) è un pattern di progettazione che permette di separare la creazione di oggetti (dipendenze) dall'utilizzo di questi oggetti. Invece di creare direttamente le dipendenze all'interno di una classe, le ricevi come parametri nel costruttore o tramite proprietà.

La DI offre numerosi vantaggi:

- **Aumenta la testabilità**
- **Migliora l'accoppiamento:** Riduce la dipendenza tra le classi, rendendo il codice più flessibile e riutilizzabile.
- **Facilita la manutenzione:** Semplifica le modifiche al codice, poiché le dipendenze sono gestite in modo centralizzato.
- **Promuove il principio di inversione delle dipendenze:** Le classi di alto livello non dovrebbero dipendere da quelle di basso livello, ma entrambe dovrebbero dipendere da astrazioni.

Vediamo un esempio:

```
public class Order
{
    public Client client {get;set;}
    .....
    .....
    public void ProcessOrder()
    {
        EmailSender emailSender = new EmailSender();
        emailSender.SendEmail(client.Email, "Ordine confermato");
    }
}
```

In questo esempio Order crea direttamente una istanza di EmailSender al suo interno e questo crea un accoppiamento tra le due classi, rendendo difficile la sostituzione di EmailSender con un'implementazione diversa e rendendo difficile i test

Se inseriamo la DI otteniamo questo:

DEFINIAMO UNA INTERFACCIA IEmailSender CHE DEFINISCE IL “contratto” PER INVIALE EMAIL

```
public interface IEmailSender
{
    void SendEmail(string to, string subject);
}
```

DEFINIAMO UNA CLASSE EmailSender CHE IMPLEMENTA IEmailSender E DEFINISCE IL COMPORTAMENTO PER L'INVIO DELLA MAIL

```
public class EmailSender : IEmailSender
{
```

```

// Implementazione del metodo
void SendEmail(string to, string subject)
{
    //corpo del metodo
}

```

DEFINIAMO UNA CLASSE `Order` CHE NEL COSTRUTTORE RICEVE UNA ISTANZA DI `ISender` ED UTILIZZA QUESTA ISTANZA PER L'INVIO DELLA MAIL

```

public class Order
{
    private readonly ISender _sender;
    public Client client {get;set;}

    public Order(ISender sender, Client client)
    {
        _sender = sender;
        this.client = client
    }

    public void ProcessOrder()
    {
        _sender.SendEmail(client.Email, "Ordine confermato");
    }
}

```

In questo modo ottengo i seguenti **vantaggi**:

- `Order` dipende solo dall'interfaccia `ISender`, non dalla classe concreta `Sender`.
- Possiamo facilmente sostituire `Sender` con un'altra implementazione (ad esempio, un simulatore di email per i test) senza modificare `Order`.
- Il codice è più testabile e meno accoppiato.

Possiamo testare il codice nel seguente modo utilizzando i Mock

Un mock è un oggetto fittizio che simula il comportamento di un oggetto reale, consentendoci di verificare le interazioni con esso.

```

// Unit test
[TestClass]
public class OrderTests
{
    [TestMethod]
    public void ProcessOrder_ShouldSendEmail()
    {
        Client client = new Client("test", "cliente@esempio.com");
        // Arrange
        var mockSender = new Mock<ISender>();
        var order = new Order(mockSender.Object, client );
    }
}

```

```

// Act
order.ProcessOrder();

// Assert - Verifichiamo che SendEmail sia stato chiamato una volta con i parametri corretti.

mockEmailSender.Verify(sender => sender.SendEmail("cliente@esempio.com", "Ordine
confermato"), Times.Once);
}
}

```

Oppure possiamo testare il codice definendo una classe di Test che implementa `ISender`

```

public class EmailLogger : ISender
{
    private readonly List<string> _log = new();

    public void SendEmail(string to, string subject, string body)
    {
        _log.Add($"Email sent to {to} with subject {subject}");
    }

    public List<string> GetLog()
    {
        return _log;
    }
}

[TestClass]
public class OrderTests
{
    [TestMethod]
    public void ProcessOrder_ShouldSendEmail()
    {
        Client client = new Client("test", "cliente@esempio.com");
        EmailLogger emailLogger= new EmailLogger()
        var order = new Order(emailLogger, client );

        // Act
        order.ProcessOrder();

        Assert.Single(emailLogger.GetLog());
        Assert.Equal("Email sent to cliente@esempio.com with subject Ordine confermato",
emailLogger.GetLog()[0]);

    }
}

```


Esempi Progetti con uso di Interfacce:

Gioco Indovina il numero

https://github.com/chiarafusaroli/indovina_numero/

Solution con al suo interno

- progetto Libreria di classi con interfacce per la gestione di un generatore di numeri e la gestione dell'I/O
- progetto Console con classi per I/O
- progetto WPF con definizioni dei metodi per i/o
- progetto Unit Test con output in debug console
- Versione Base: versione con una classe game che risulta essere non testabile perchè genera il numero da indovinare in modo random
- Versione Interfacce: creazione di una interfaccia IGenerator e di una classe RandomGenerator utilizzata da Game per gestire la generazione del numero da indovinare Random e creazione di una classe nel progetto di test che implementa l'interfaccia e mi permette di definire il numero da indovinare per testare il corretto comportamento della classe Game
- Versione Architecture: Interfaccia IInputInterface e IOutputInterface per la gestione dell'I/O. Utilizzo di task, async ed await per poter gestire l'input da WPF (I Task servono per eseguire operazioni in modo asincrono, evitando che il programma si blocchi. Sono molto utili in WPF per la gestione dell'input essendo WPF event-driven. Un Task rappresenta un'operazione in esecuzione (o che verrà eseguita) in modo asincrono; Task.Run() esegue il codice su un thread separato, utile per operazioni pesanti. await ci permette di attendere ed ottenere un valore da un task

WPF

<https://learn.microsoft.com/it-it/dotnet/desktop/wpf/overview/?view=netdesktop-9.0>

<https://github.com/microsoft/WPF-Samples>

WPF (Windows Presentation Foundation) è un framework di Microsoft utilizzato per creare **interfacce utente grafiche ricche e personalizzabili** per applicazioni desktop Windows. WPF consente di sviluppare applicazioni usando sia **markup** sia **code-behind**. Il markup XAML viene normalmente usato per implementare l'aspetto di un'applicazione, mentre i linguaggi di programmazione gestiti (code-behind) vengono usati per implementarne il comportamento. La separazione tra aspetto e comportamento offre alcuni vantaggi:

- I costi di sviluppo e gestione risultano ridotti, perché il markup specifico per l'aspetto non è associato strettamente a codice specifico per il comportamento.
- Lo sviluppo è più efficiente, perché i progettisti possono implementare l'aspetto di un'applicazione mentre, contemporaneamente, gli sviluppatori ne implementano il comportamento.

Perché utilizzare WPF

- **Flessibilità:** WPF offre una grande flessibilità nella creazione di interfacce utente personalizzate, consentendo di creare design complessi e interattivi.
- **XAML:** WPF utilizza XAML (Extensible Application Markup Language) per definire l'interfaccia utente in modo dichiarativo, separando la logica di presentazione dalla logica di business.
- **Data binding:** WPF semplifica la gestione dei dati, consentendo di collegare i dati dell'applicazione direttamente all'interfaccia utente.
- **Controlli:** WPF offre una vasta gamma di controlli predefiniti, che possono essere personalizzati e estesi per creare interfacce utente complesse.

WPF: Ancora Attuale o Superato

WPF è un framework Microsoft che ha segnato un'epoca nella creazione di interfacce utente grafiche per applicazioni desktop Windows. **La risposta alla domanda se sia ancora attuale o superato non è netta e dipende da diversi fattori.**

WPF rimane un'ottima scelta per creare applicazioni desktop ricche di funzionalità e personalizzabili, specialmente quando si richiede un alto livello di controllo sull'interfaccia utente. WPF non è la scelta primaria per le applicazioni web, e non è progettato specificamente per le applicazioni mobile.

WPF è un framework maturo e stabile, con una vasta documentazione e una grande community, offre prestazioni elevate, specialmente per applicazioni grafiche intensive, consente di creare

interfacce utente altamente personalizzate e complesse, si integra perfettamente con il resto dell'ecosistema .NET.

Alternative a WPF

- **Avalonia:**
 - Basato su XAML, open-source, supporta Windows, macOS, Linux e WebAssembly. Offre un'esperienza molto simile a WPF, rendendo la migrazione più facile.
 - Cross-platform, community attiva, supporto per le ultime tecnologie .NET.
 - Community più piccola rispetto a WPF, alcune funzionalità potrebbero non essere mature come in WPF.
- **Uno Platform:**
 - Basato su .NET MAUI, permette di creare app native per Windows, macOS, Linux, iOS, Android e WebAssembly, condividendo gran parte del codice.
 - Cross-platform completo, integrazione con .NET MAUI, supporto per le ultime tecnologie .NET.
 - Ancora in fase di sviluppo attivo, alcune funzionalità potrebbero essere limitate.
- **UWP (Universal Windows Platform):**
 - Framework Microsoft per creare app che funzionano su diversi dispositivi Windows 10.
 - Integrato con il sistema operativo Windows, accesso a molte funzionalità di Windows 10.
 - Limitato all'ecosistema Windows, alcune funzionalità potrebbero non essere disponibili su tutte le versioni di Windows.
- **.NET MAUI:** framework di sviluppo .NET open-source che permette di creare app native per Android, iOS, macOS, Windows e persino WebAssembly, condividendo gran parte del codice base. È l'evoluzione di Xamarin.Forms e offre un modo unificato e moderno per sviluppare applicazioni multi-piattaforma. Mentre WPF è focalizzato sullo sviluppo di applicazioni desktop Windows, MAUI è progettato per creare app multi-piattaforma.

Caratteristica	WPF	MAUI
Piattaforme	Principalmente Windows	Android, iOS, macOS, Windows, WebAssembly
Linguaggio	C#	C#
XAML	Sì	Sì (simile a WPF)
Prestazioni	Ottime per applicazioni desktop	Ottime per applicazioni native multi-piattaforma
Community	Matura e vasta	In crescita, ma già solida

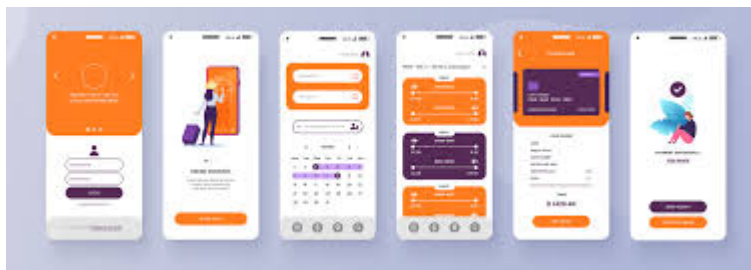
UX e UI

L'esperienza utente, nota come user experience o UX, è un termine utilizzato per definire la relazione tra una persona e un prodotto, un servizio, un sistema

La UI è una delle componenti della UX e rappresenta l'interfaccia utente.



Attenzione:



MARKUP

<https://learn.microsoft.com/it-it/dotnet/desktop/wpf/xaml/?view=netdesktop-9.0>

XAML è un linguaggio di markup basato su XML che implementa l'aspetto di un'applicazione in modo dichiarativo. Viene in genere usato per definire finestre, finestre di dialogo, pagine e controlli utente e per inserire in questi elementi controlli, forme e grafica.

<Window

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
Title="Window with button"
```

```

Width="250" Height="100">

<!-- Add button to window -->
<Button Name="button">Click Me!</Button>

</Window>

```

CODE-BEHIND

Il comportamento principale di un'applicazione consiste nell'implementare la funzionalità che risponde alle interazioni dell'utente, ad esempio la scelta di un menu o di un pulsante e la chiamata alla logica di business e alla logica di accesso ai dati in risposta a tali eventi. In WPF questo comportamento viene in genere implementato in codice associato a markup. Questo tipo di codice è noto come code-behind. L'esempio seguente illustra il markup aggiornato dell'esempio precedente e il code-behind

```

<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.AWindow"
  Title="Window with button"
  Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>

```

```

using System.Windows;

namespace SDKSample
{
    public partial class AWindow : Window
    {
        public AWindow()
        {
            // InitializeComponent call is required to merge the UI
            // that is defined in markup with this class, including
            // setting properties and registering event handlers
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}

```

```
}  
}  
}
```

InitializeComponent viene chiamato dal costruttore della classe code-behind per unire l'interfaccia utente definita nel markup con la classe code-behind.

La combinazione di x:Class e InitializeComponent assicura che l'implementazione venga inizializzata correttamente ogni volta che viene creata.

Si noti che nel markup l'elemento <Button> ha definito un valore di button_Click per l'attributo Click. Con il markup e il code-behind inizializzati e funzionanti insieme, viene eseguito automaticamente il mapping dell'evento Click per il pulsante al metodo button_Click.

Quindi quando si fa clic sul pulsante, viene richiamato il gestore eventi e viene eseguito il codice presente all'interno del metodo **button_Click**.

IMPORTANTE: il metodo richiamato dal bottone non deve contenere la logica del problema, che invece deve essere inserita negli opportuni metodi di una classe. **Ricordatevi che le classi devono essere indipendenti dal tipo di interazione I/O**

Controlli WPF per funzione

I controlli WPF incorporati sono elencati di seguito:

- Pulsanti: [Button](#) e [RepeatButton](#).
- Visualizzazione di dati: [DataGrid](#), [ListView](#) e [TreeView](#).
- Visualizzazione e selezione di date: [Calendar](#) e [DatePicker](#).
- Finestre di dialogo: [OpenFileDialog](#), [PrintDialog](#) e [SaveFileDialog](#).
- Input penna: [InkCanvas](#) e [InkPresenter](#).
- Documenti: [DocumentViewer](#), [FlowDocumentPageViewer](#), [FlowDocumentReader](#), [FlowDocumentScrollViewer](#) e [StickyNoteControl](#).
- Input: [TextBox](#), [RichTextBox](#) e [PasswordBox](#).
- Layout: [Border](#), [BulletDecorator](#), [Canvas](#), [DockPanel](#), [Expander](#), [Grid](#), [GridView](#), [GridSplitter](#), [GroupBox](#), [Panel](#), [ResizeGrip](#), [Separator](#), [ScrollBar](#), [ScrollViewer](#), [StackPanel](#), [Thumb](#), [Viewbox](#), [VirtualizingStackPanel](#), [Window](#) e [WrapPanel](#).
- Supporti: [Image](#), [MediaElement](#) e [SoundPlayerAction](#).
- Menu: [ContextMenu](#), [Menu](#) e [ToolBar](#).
- Navigazione: [Frame](#), [Hyperlink](#), [Page](#), [NavigationWindow](#) e [TabControl](#).
- Selezione: [CheckBox](#), [ComboBox](#), [ListBox](#), [RadioButton](#) e [Slider](#).
- Informazioni utente: [AccessText](#), [Label](#), [Popup](#), [ProgressBar](#), [StatusBar](#), [TextBlock](#) e [ToolTip](#).

Finestre di dialogo

<https://learn.microsoft.com/it-it/dotnet/desktop/wpf/app-development/dialog-boxes-overview?view=netframeworkdesktop-4.8>

Layout

Il requisito principale di qualsiasi layout è quello di adattarsi alle modifiche nella dimensione delle finestre e nelle impostazioni di visualizzazione.

Per layout comuni, ad esempio griglie, sovrapposizioni e ancoraggio, WPF include vari controlli di layout:

- [Canvas](#): i controlli figlio forniscono il proprio layout.
- [DockPanel](#): i controlli figlio sono allineati ai bordi del pannello.
- [Grid](#): i controlli figlio sono posizionati per righe e colonne.
- [StackPanel](#): i controlli figlio sono sovrapposti in verticale o in orizzontale.
- [VirtualizingStackPanel](#): i controlli figlio sono virtualizzati e disposti su una sola riga orientata in senso orizzontale o verticale.
- [WrapPanel](#): i controlli figlio vengono posizionati nell'ordine da sinistra a destra e racchiusi nella riga successiva quando non c'è spazio sufficiente sulla riga corrente.

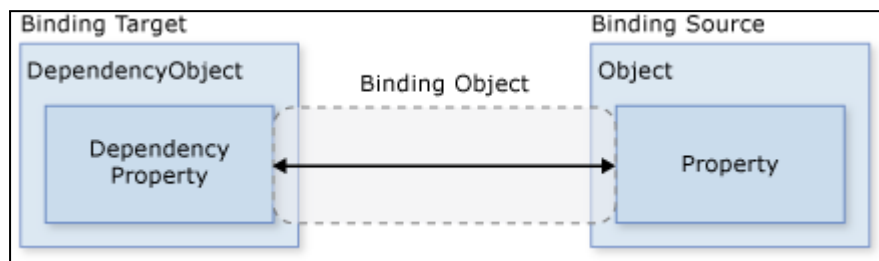
Data binding

La maggior parte delle applicazioni viene creata per consentire agli utenti di visualizzare e modificare i dati. Per le applicazioni WPF, le operazioni di archiviazione e accesso ai dati sono già disponibili per molte librerie di accesso ai dati .NET.

Dopo l'accesso ai dati e il loro caricamento negli oggetti gestiti di un'applicazione, ha inizio la fase più complessa delle applicazioni WPF. Questa fase comporta essenzialmente due punti:

1. Copia dei dati dagli oggetti ai controlli, per consentirne la visualizzazione e la modifica.
2. Verifica che le modifiche apportate ai dati usando i controlli vengano copiate negli oggetti.

Per semplificare lo sviluppo delle applicazioni, in WPF è disponibile un potente motore di associazione dati per gestire automaticamente queste operazioni. L'unità principale di questo motore è la classe [Binding](#), il cui scopo è associare un controllo (destinazione) a un oggetto dati (origine). La figura seguente illustra questa relazione:



WPF supporta la dichiarazione di associazioni direttamente nel markup XAML.

Ad esempio, il codice XAML seguente associa la proprietà [Text](#) dell'oggetto [TextBox](#) alla proprietà Name di un oggetto usando la sintassi XAML "{Binding ... }". Si presuppone che sia presente un oggetto dati impostato sulla proprietà [DataContext](#) di Window con una proprietà Name.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.DataBindingWindow">

  <!-- Bind the TextBox to the data source (TextBox.Text to Person.Name) -->
  <TextBox Name="personNameTextBox" Text="{Binding Path=Name}" />

</Window>
```

Il motore di associazione dati WPF offre più della semplice associazione, in quanto include funzionalità di convalida, ordinamento, filtro e raggruppamento.

Esercitazione:

<https://learn.microsoft.com/it-it/dotnet/desktop/wpf/get-started/create-app-visual-studio?view=net-desktop-9.0>

.NET MAUI

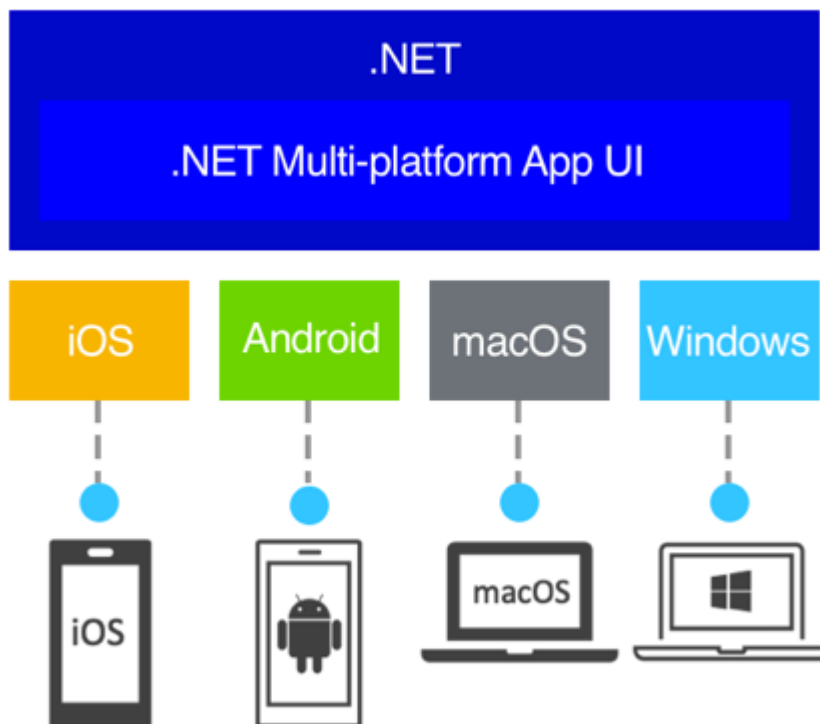
<https://learn.microsoft.com/it-it/dotnet/maui/what-is-maui?view=net-maui-9.0>

<https://dotnet.microsoft.com/it-it/apps/maui>

.NET MAUI è destinato agli sviluppatori che vogliono:

- Scrivere app multiplatforma in XAML e C# da un'unica codebase condivisa in Visual Studio.
- Condividere il layout e la progettazione dell'interfaccia utente tra le piattaforme.
- Condividere codice, test e logica di business tra piattaforme.

.NET MAUI unifica le API Android, iOS, macOS e Windows in un'unica API che consente un'esperienza di sviluppo WORA (Write-Once Run-Anywhere), offrendo inoltre accesso avanzato a ogni aspetto di ogni piattaforma nativa.



Creare la prima APP

<https://learn.microsoft.com/it-it/dotnet/maui/get-started/first-app?pivots=devices-windows&view=net-maui-9.0&tabs=vswin>

Esercitazione guidata

Creare una APP .Net Maui con codice multiplatforma

<https://learn.microsoft.com/it-it/dotnet/maui/tutorials/notes-app/?view=net-maui-9.0>

Nell'esercitazione impareremo a:

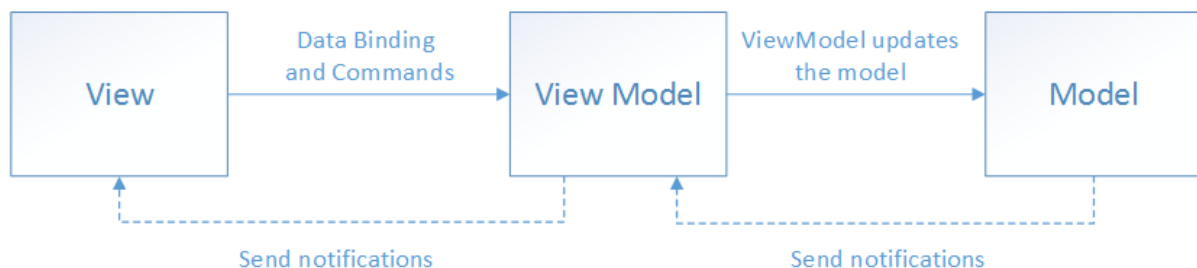
- Creare un'app shell MAUI .NET.
- Eseguire l'app nella piattaforma scelta.
- Definire l'interfaccia utente con eXtensible Application Markup Language (XAML) e interagire con gli elementi XAML tramite codice.
- Creare visualizzazioni e associarle ai dati.
- Usare lo spostamento per spostarsi da e verso le pagine.

Si userà Visual Studio 2022 per creare un'applicazione con cui è possibile immettere una nota e salvarla nell'archiviazione del dispositivo

Aggiungere i concetti MVVM

<https://learn.microsoft.com/it-it/dotnet/maui/tutorials/notes-mvvm/?view=net-maui-9.0>

Il modello model-view-viewmodel (MVVM) consente di separare in modo pulito la logica di business e presentazione di un'app dalla relativa interfaccia utente. La gestione di una separazione netta tra la logica dell'app e l'interfaccia utente consente di risolvere numerosi problemi di sviluppo e semplifica il test, la gestione e l'evoluzione di un'app. Può anche migliorare significativamente le opportunità di riutilizzo del codice e consente agli sviluppatori e ai progettisti dell'interfaccia utente di collaborare più facilmente durante lo sviluppo delle rispettive parti di un'app.



- La **View** è responsabile della definizione della struttura, del layout e dell'aspetto di ciò che l'utente vede sullo schermo. Idealmente, **ogni visualizzazione è definita in XAML**, con un code-behind limitato che **non contiene logica di business**. In alcuni casi, tuttavia, il code-behind potrebbe contenere la logica dell'interfaccia utente che implementa un comportamento visivo difficile da esprimere in XAML, ad esempio nelle animazioni.
- Il **ViewModel implementa proprietà e comandi a cui la visualizzazione può associare i dati e notifica la view di eventuali modifiche dello stato tramite eventi di notifica delle**

modifiche. Le proprietà e i comandi forniti dal ViewModel definiscono la funzionalità da offrire all'interfaccia utente, ma la view determina la modalità di visualizzazione.

- Il **ViewModel** è responsabile del coordinamento delle interazioni della vista con tutte le classi di modello necessarie. In genere esiste una relazione uno-a-molti tra il ViewModel e le classi del modello.

Ogni ViewModel fornisce dati di un modello in un modo che la View può utilizzare facilmente. A tale scopo, il ViewModel a volte esegue la conversione dei dati. L'inserimento di questa conversione dei dati nel ViewModel è una buona idea perché fornisce proprietà a cui è possibile associare la View. Ad esempio, il ViewModel può combinare i valori di due proprietà per semplificare la visualizzazione da parte della View.

- Le classi del **Model** sono classi non visive che incapsulano i dati dell'app. Di conseguenza, si può pensare che il Model rappresenti il modello di dominio dell'app, che di solito include un **modello di dati insieme alla logica di business e di convalida.**

Creare APP per dispositivi mobili e desktop

<https://learn.microsoft.com/it-it/training/paths/build-apps-with-dotnet-maui/>

INotifyPropertyChanged

INotifyPropertyChanged è un'interfaccia che permette di notificare alla UI (o ad altri listener) quando il valore di una proprietà cambia. È molto utilizzata in WPF, e MAUI per supportare il data binding dinamico.

Quando la UI è collegata a un **ViewModel** in un'architettura **MVVM**, vogliamo che si aggiorni automaticamente quando i dati cambiano, senza dover ricaricare manualmente l'interfaccia.

Per far sì che il tutto funzioni è necessario:

- avere un viewModel che implementa INotifyPropertyChanged e che richiama l'evento OnPropertyChanged() alla modifica della proprietà che vogliamo tenere monitorata per l'aggiornamento della UI

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
```

```
public class MainViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    private string _message = "Hello, WPF!";
    public string Message
    {
        get => _message;
        set
        {
            if (_message != value)
            {
                _message = value;
                OnPropertyChanged();
            }
        }
    }

    protected void OnPropertyChanged([CallerMemberName] string? propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Se il valore cambia, chiamiamo OnPropertyChanged(), che avvisa la UI. [CallerMemberName] permette di evitare di scrivere manualmente il nome della proprietà.

- Utilizzare la proprietà nella view

```
<Window.DataContext>
```

```

    <local:MainViewModel/>
</Window.DataContext>

<Grid>
    <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
        <TextBlock Text="{Binding Message}" FontSize="20"/>
        <Button Content="Cambia Messaggio" Command="{Binding ChangeMessageCommand}"/>
    </StackPanel>
</Grid>

```

In questo modo quando **Message** cambia, il **TextBlock** si aggiorna **automaticamente**

Spesso è necessario utilizzare **INotifyPropertyChanged** anche con i comandi

Oggi è possibile attraverso il `CommunityToolkit.Mvvm` evitare di scrivere a mano l'implementazione di `INotifyPropertyChanged` ed utilizzare la classe **ObservableObject** come classe base in modo da avere a disposizione le funzionalità per la gestione delle property e di command. In questo modo il codice sarà più leggibile ed una parte sarà generata automaticamente.

ObservableObject (CommunityToolkit.Mvvm)

Utilizzare `ObservableObject` quando si desidera notificare le modifiche alle proprietà di un singolo oggetto.

`ObservableObject` è una classe di base che implementa l'interfaccia `INotifyPropertyChanged`. **Il suo scopo principale è quello di notificare le modifiche alle proprietà di un oggetto.** Viene comunemente utilizzato nel pattern architetturale MVVM (Model-View-ViewModel) per consentire l'aggiornamento automatico dell'interfaccia utente quando i dati sottostanti cambiano.

Si eredita da `ObservableObject` per creare classi i cui cambiamenti di proprietà devono essere osservabili.

Con il pacchetto `CommunityToolkit.Mvvm` di .NET Community Toolkit c'è la possibilità di utilizzare l'annotazione **[ObservableProperty]** il cui scopo principale è quello di semplificare la creazione di proprietà osservabili, riducendo la quantità di codice che altrimenti sarebbe necessario scrivere.

Se si applica l'annotazione `[ObservableProperty]` a un campo privato all'interno della classe il generatore di codice sorgente analizza il codice e genera una proprietà pubblica con il nome corrispondente (con la prima lettera maiuscola). La proprietà generata include la logica per: Impostare il valore del campo sottostante; generare l'evento `PropertyChanged` quando il valore cambia.

Esempio di utilizzo

```

private string _nome;
public string Nome

```

```

{
    get => _nome;
    set
    {
        if (_nome != value)
        {
            _nome = value;
            OnPropertyChanged(nameof(Nome));
        }
    }
}

```

Si può semplicemente scrivere:

[ObservableProperty]

```
private string _nome;
```

Il generatore di codice si occuperà di creare la proprietà pubblica Nome con la logica necessaria

ObservableCollection

Utilizzare ObservableCollection<T> quando si desidera notificare le modifiche a una raccolta di oggetti.

IObservable<T>

In C#, l'interfaccia **IObservable<T>** fa parte del modello di progettazione "Observer" (Osservatore), ed è utilizzata per implementare la **programmazione reattiva**.

In termini semplici questo tipo di programmazione consente ad un oggetto (l'osservabile) di

Scopo:

- **Gestione di flussi di dati asincroni:**
 - **IObservable<T>** è particolarmente utile per gestire flussi di dati che arrivano in modo asincrono, ovvero quando non si sa esattamente quando i dati saranno disponibili.
 - Questo è comune in scenari come:
 - Eventi dell'interfaccia utente (ad esempio, clic del mouse, input da tastiera).
 - Dati provenienti da sensori o dispositivi hardware.
 - Risposte da servizi web o API.
- **Modello di pubblicazione/sottoscrizione:**
 - **IObservable<T>** implementa un modello di pubblicazione/sottoscrizione, in cui:
 - L'osservabile "pubblica" notifiche quando si verificano cambiamenti.

- Gli osservatori si "iscrivono" all'osservabile per ricevere queste notifiche.

Come funziona:

1. L'osservabile:

- Implementa l'interfaccia **IObservable<T>**.
- Fornisce un metodo `Subscribe(IObserver<T> observer)` che consente agli osservatori di registrarsi.
- Quando si verifica un evento o un cambiamento, l'osservabile chiama i metodi dell'interfaccia `IObserver<T>` sugli osservatori registrati.

2. L'osservatore:

- Implementa l'interfaccia **IObserver<T>**.
- Fornisce tre metodi:
 - `OnNext(T value)`: Riceve i dati dall'osservabile.
 - `OnError(Exception error)`: Riceve notifiche di errore.
 - `OnCompleted()`: Riceve una notifica quando l'osservabile ha terminato di inviare dati.

Vantaggi:

- **Decoupling**: L'osservabile e gli osservatori sono disaccoppiati, il che significa che non hanno bisogno di conoscersi direttamente.
- **Flessibilità**: Più osservatori possono sottoscrivere allo stesso osservabile.
- **Gestione di eventi asincroni**: Fornisce un modo elegante per gestire eventi asincroni.

In sintesi, `IObservable<T>` è uno strumento potente per la programmazione reattiva in C#, che consente di gestire in modo efficiente flussi di dati asincroni e eventi.

Esempio progetto WPF con MVVM:

1. installare **CommunityToolkit.Mvvm**
2. **struttura del progetto:**

```
WpfMvvmApp/  
|-- ViewModels/  
|   |-- MainViewModel.cs  
|-- Models/  
|   |-- Persona.cs  
|-- Views/  
|   |-- MainWindow.xaml  
|-- App.xaml  
|-- MainWindow.xaml.cs  
|-- Program.cs  
|-- WpfMvvmApp.csproj
```

3. **Creare il Model (rappresentazione dei dati)**

```
namespace WpfMvvmApp.Models;
```

```
public class Persona  
{  
    public string Nome { get; set; }  
    public int Età { get; set; }  
}
```

4. **Creare il ViewModel per la gestione dei dati e della logica di business**

```
using CommunityToolkit.Mvvm.ComponentModel;  
using CommunityToolkit.Mvvm.Input;  
using System.Collections.ObjectModel;
```

```
namespace WpfMvvmApp.ViewModels;
```

```
public partial class MainViewModel : ObservableObject  
{  
    // Proprietà osservabile per il nome  
    [ObservableProperty]  
    private string nome = "";  
  
    // Proprietà osservabile per l'età  
    [ObservableProperty]  
    private int età;  
  
    // Lista di persone osservabile  
    public ObservableCollection<Persona> Persone { get; } = new();  
  
    public MainViewModel()
```



```

{
    // Aggiunge qualche dato iniziale
    Persone.Add(new Persona { Nome = "Mario", Età = 30 });
    Persone.Add(new Persona { Nome = "Anna", Età = 25 });
}

// Comando per aggiungere una persona
[RelayCommand]
private void AggiungiPersona()
{
    if (!string.IsNullOrEmpty(Nome) && Età > 0)
    {
        Persone.Add(new Persona { Nome = Nome, Età = Età });
        Nome = "";
        Età = 0;
    }
}
}

```

ObservableProperty genera automaticamente **INotifyPropertyChanged**.

ObservableCollection<Persona> aggiorna automaticamente la UI.

[RelayCommand] crea un **ICommand** automaticamente per il pulsante.

5. Creare la View (MainWindow.xaml)

```

<Window x:Class="WpfMvvmApp.Views.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfMvvmApp.ViewModels"
    Title="Gestione Persone" Height="350" Width="400">

    <Window.DataContext>
        <local:MainViewModel/>
    </Window.DataContext>

    <Grid Margin="10">
        <StackPanel>
            <!-- Campo Nome -->
            <TextBlock Text="Nome:"/>
            <TextBox Text="{Binding Nome, UpdateSourceTrigger=PropertyChanged}" />

            <!-- Campo Età -->
            <TextBlock Text="Età:"/>
            <TextBox Text="{Binding Eta, UpdateSourceTrigger=PropertyChanged}" />

            <!-- Bottone per aggiungere persona -->
            <Button Content="Aggiungi Persona" Command="{Binding AggiungiPersonaCommand}" />

```

```

<!-- Lista delle persone -->
<ListBox ItemsSource="{Binding Persone}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Nome}" FontWeight="Bold"/>
        <TextBlock Text=" - "/>
        <TextBlock Text="{Binding Età}" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
</StackPanel>
</Grid>
</Window>

```

DataContext → Collega la View al MainViewModel.

TextBox Binding → Nome e Età aggiornano automaticamente il ViewModel.

Button Command → Il bottone chiama AggiungiPersonaCommand (generato automaticamente).

ListBox Binding → Mostra dinamicamente la lista di Persone.

Clean Architecture

I principi della **Clean Architecture**, secondo **Uncle Bob (Robert C. Martin)**, mirano a creare un'architettura software che sia **indipendente dai dettagli di implementazione**, altamente **manutenibile e testabile**.

Ci sono quattro principi chiave:

1. **The Dependency Rule (Regola della Dipendenza):**

Le dipendenze devono sempre puntare verso il codice di livello più alto (business logic e use case), mai verso dettagli di implementazione (UI, database, framework, ecc.).

Quindi:

- L'UI non deve conoscere il database.
- La business logic non deve dipendere da framework o librerie.
- Gli adattatori (es. repository, API) dipendono dal core dell'app, non viceversa.

2. **Separation of Concerns (Separazione delle Responsabilità):**

Ogni livello dell'architettura deve avere una sola responsabilità chiara e non interferire con altri livelli.

Quindi:

- Il dominio gestisce solo la logica di business.
- L'Application Layer contiene solo i casi d'uso e le interfacce.
- L'Infrastructure Layer implementa dettagli come database e API.
- L'UI si occupa solo di presentazione.

3. **Independence from Frameworks (Indipendenza dai Framework):**

L'architettura non deve essere legata a un framework specifico.

Questo permette di:

- Evitare il vendor lock-in (es. se il framework diventa obsoleto, il codice rimane riutilizzabile).
- La logica di business funziona indipendentemente da WPF, ASP.NET, Entity Framework, ecc.
- I framework diventano solo strumenti, non il cuore dell'applicazione.

4. **Testability (Testabilità):**

Tutto deve essere facilmente testabile, soprattutto la logica di business.

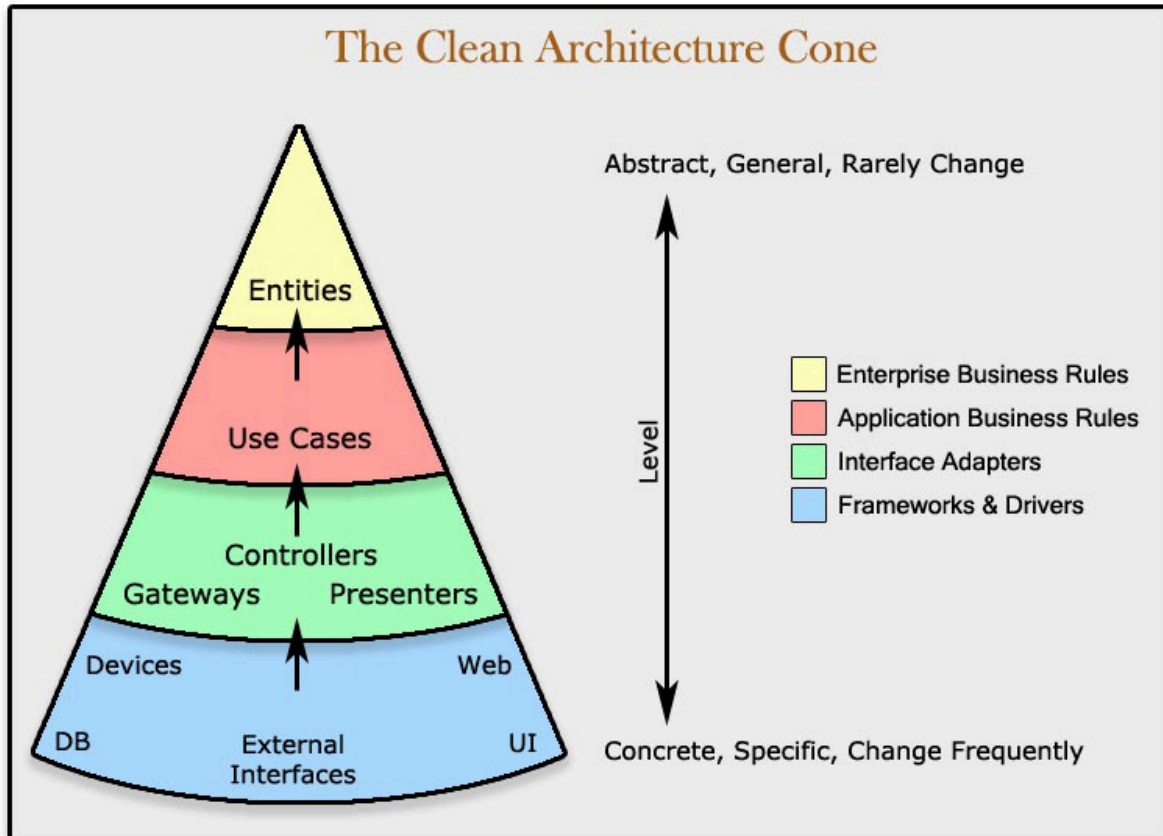
Per fare questo è necessario:

- Implementare l'astrazione grazie all'utilizzo delle interfacce
- Utilizzare il Mocking delle dipendenze nei test (es. simulare il database o l'API).
- Nessuna dipendenza diretta da database, file system o servizi esterni nei casi d'uso.

Un concetto fondamentale derivante da questi 4 principi è che **"Business Logic at the Core"** (La Logica di Business al Centro), ovvero **Il cuore dell'applicazione deve contenere solo logica di business pura, senza dipendenze esterne.**

Quindi:

- **Le entità di dominio sono indipendenti da qualsiasi tecnologia.**
- **I casi d'uso (application layer) sono semplici e gestiscono solo le regole di business.**
- **Il database e l'UI sono dettagli che possono cambiare senza impattare la logica principale.**



Struttura di un progetto Clean

La struttura di base potrebbe essere del tipo:

```

MyApp/
├── MyApp.Presentation/ → **(UI - WPF, Blazor, MAUI, Web API)**
├── MyApp.Application/ → **(Use Cases, Interfacce, Business Logic)**
├── MyApp.Domain/ → **(Entità di dominio, Regole di Business)**
├── MyApp.Infrastructure/ → **(Database, Repository, API, Services)**
└── MyApp.Test/ → **(Unit Test, Integration Test)**

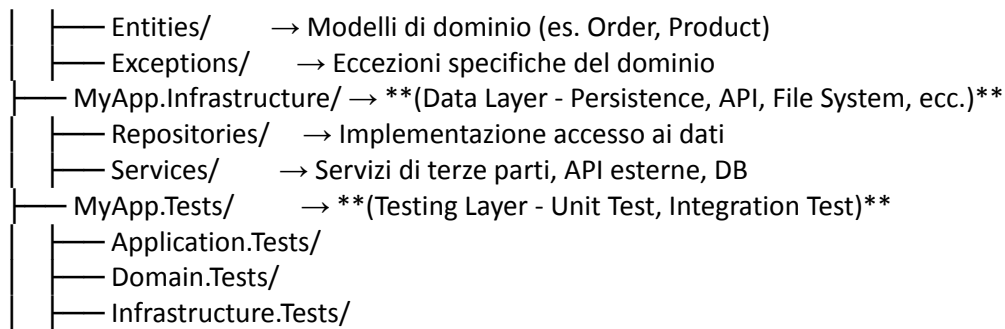
```

Nel dettaglio all'interno delle cartelle di primo livello potremmo avere:

```

MyApp/
├── MyApp.Presentation/ → **(UI Layer - WPF/Maui)** Contiene View, ViewModel, e Binding
│   ├── Views/ → Finestre WPF o MAUI
│   ├── ViewModels/ → Contiene i ViewModel (MVVM)
│   ├── App.xaml → Configurazione UI principale
│   └── DI/ → Configurazione Dependency Injection
├── MyApp.Application/ → **(Application Layer - Business Logic)** Contiene i Use Cases
│   ├── UseCases/ → Contiene la logica applicativa (es. Gestione ordini, calcoli)
│   └── Interfaces/ → Contratti per il dominio
└── MyApp.Domain/ → **(Domain Layer - Core Business Logic)** Contiene le entità

```



Si potrebbe quindi organizzare il progetto in **5 livelli principali** con progetti di vario tipo:

1. Domain
 - Contiene le entità di dominio e la logica di business pura.
 - Non ha dipendenze verso altri livelli.
 - 📁 Progetto: MyApp.Domain (Libreria di classi)
2. Application
 - Contiene i casi d'uso e la logica applicativa.
 - Definisce interfacce per la comunicazione con il livello di infrastruttura.
 - 📁 Progetto: MyApp.Application (Libreria di classi)
3. Infrastructure
 - Implementa le interfacce definite nel livello Application.
 - Contiene repository, accesso ai dati, logging, email, ecc.
 - 📁 Progetto: MyApp.Infrastructure (Libreria di classi)
4. Presentation
 - Contiene l'interfaccia utente (Web API, Console, Blazor, MVC, ecc.).
 - Dipende dall'Application Layer.
 - 📁 Progetto: MyApp.Presentation (Console, WPF, MAUI,...r)
5. Test
 - Contiene i test unitari e di integrazione
 - Dipende dall'Application Layer.
 - 📁 Progetto: MyApp.Test (MSTest, ...)

Esempio Indovina il numero con architettura Clean

