The background of the book cover is a photograph of a wooden pier made of horizontal planks. The pier extends from the bottom right towards the top left. The water is a deep blue with some white foam near the shore. The sky is clear and blue.

Riccardo Crosato

C#

Algoritmi e
Programmi

Riccardo Crosato

C#
Algoritmi e
Programmi

C# – Algoritmi e Programmi

Copyright © 2019 by Riccardo Crosato

Prima edizione: agosto 2019

Edizione n. 2: giugno 2020

Tutti i marchi citati nel testo sono di proprietà dei rispettivi possessori.

Sommario

Introduzione

1. Algoritmi e linguaggi di programmazione

- 1.1. Problemi e algoritmi
- 1.2. Tipi di dato, variabili e costanti
- 1.3. Schemi di composizione delle istruzioni
- 1.4. Espressioni condizionali composte
- 1.5. Linguaggi di programmazione
- 1.6. Processo di traduzione
- 1.7. Il .NET Framework e il linguaggio C#

2. Un primo programma in C#

- 2.1. La classe Console
- 2.2. Compiliamo!
- 2.3. Utilizzo di Visual Studio
- 2.4. La direttiva using static
- 2.5. Compilatori online

3. Variabili, costanti e tipi di dato

- 3.1. Tipi di dato valore e tipi riferimento
- 3.2. Tipi predefiniti C#
- 3.3. Conversione fra tipi
- 3.4. Gestire l'input e output da console
- 3.5. Conversione controllata con TryParse
- 3.6. Tipizzazione implicita con var
- 3.7. Il valore speciale null
- 3.8. Ambito di visibilità delle variabili

4. Operatori

- 4.1. Operatori primari
- 4.2. Operatori unari
- 4.3. Operatori di moltiplicazione

- [4.4. Operatori di addizione](#)
- [4.5. Operatori di traslazione \(shift\)](#)
- [4.6. Operatori relazionali e Operatori di tipo](#)
- [4.7. Operatori di uguaglianza](#)
- [4.8. Operatori logici e condizionali](#)
- [4.9. Operatori di assegnazione](#)
- [4.10. Operatori logici e condizionali](#)
- [4.11. Operatori di assegnazione semplice e assegnazione composta](#)
- [4.12. Operatori di incremento \(++\) e decremento \(--\)](#)

5. Strutture di controllo

- [5.1. Strutture di selezione](#)
- [5.2. Strutture di ripetizione](#)
- [5.3. Ciclo con contatore](#)
- [5.5. Istruzioni break e continue](#)
- [5.6. Controllo delle eccezioni](#)

6. Stringhe

- [6.1. Concatenazione di stringhe](#)
- [6.2. Caratteri di escape e stringhe verbatim](#)
- [6.3. Conversione in stringa – metodi ToString\(\) e Convert.ToString\(...\)](#)
- [6.4. Accesso ai singoli caratteri di una stringa](#)
- [6.5. Metodi per manipolare stringhe](#)
- [6.6. Estrarre una porzione di stringa \(sottostringa\)](#)
- [6.7. Confronto tra stringhe](#)
- [6.8. Ricerca all'interno di una stringa](#)
- [6.9. Modifica della combinazione di maiuscole e minuscole](#)
- [6.10. Inserire, rimuovere e sostituire sottostringhe](#)
- [6.11. “Ripulire” una stringa dai caratteri iniziali e finali](#)
- [6.12. “Imbottire” una stringa](#)
- [6.13. Formattare una stringa](#)

7. Array

- [7.1. Dichiarazione di array](#)
- [7.2. Allocazione e inizializzazione di array](#)

- [7.3. Array jagged](#)
- [7.4. Accesso ai singoli elementi e numero di elementi](#)
- [7.5. Il ciclo foreach](#)
- [7.6. Assegnazione e confronto tra array](#)
- [7.7. Metodi sugli array: la classe Array](#)
- [7.8. Ricerca di elementi](#)
- [7.9. Minimo, massimo e somma degli elementi](#)
- [7.10. Clonazione e copia di elementi](#)
- [7.11. Metodi sulle stringhe che usano array](#)

8. Algoritmi di ricerca e ordinamento

- [8.1. Ricerca sequenziale](#)
- [8.2. Ordinamento per selezione \(*selection sort*\)](#)
- [8.3. Ordinamento per scambio \(*bubble sort*\)](#)
- [8.5. Ordinamento per inserzione \(*insertion sort*\)](#)
- [8.6. Ricerca binaria \(*binary search*\)](#)
- [8.7. Fusione ordinata di due array \(*merge*\)](#)

9. Metodologia top-down e sottoprogrammi

- [9.1. Sottoprogrammi](#)
- [9.2. Metodi in C#](#)
- [9.3. Elementi essenziali di OOP e metodi statici](#)
- [9.4. Campi dato e ambito di visibilità](#)
- [9.5. Metodi funzione e valore di ritorno](#)
- [9.6. Metodi con parametri](#)
- [9.7. Passaggio dei parametri per valore](#)
- [9.8. Passaggio dei parametri per riferimento](#)
- [9.9. Commenti di documentazione](#)
- [9.11. Parametri denominati e parametri opzionali](#)
- [9.12. Modificatore params](#)
- [9.13. Metodi con corpo di espressioni](#)
- [9.14. Metodi di estensione](#)
- [9.15. Metodi annidati](#)

10. Algoritmi ricorsivi

- [10.1. La successione di Fibonacci](#)
- [10.2. La torre di Hanoi](#)
- [10.3. Ricerca binaria ricorsiva](#)
- [10.4. Algoritmo di ordinamento per fusione \(*merge sort*\)](#)
- [10.5. Algoritmo di ordinamento per partizione \(*quick sort*\)](#)
- [10.6. Algoritmi di backtracking](#)

11. Espressioni regolari

- [11.1. Espressioni regolari in C#](#)
- [11.2. Sintassi delle espressioni regolari](#)
- [11.3. Verificare una stringa rispetto ad un pattern](#)
- [11.4. Estrarre le sottostringhe che rispettano un pattern](#)

12. Enumerazioni

- [12.1. Conversioni a stringa e da intero](#)
- [12.2. La classe Enum](#)
- [12.3. Campi di bit](#)

Appendici

- [Stack e Heap](#)
- [Generazione di numeri casuali](#)
- [La classe StringBuilder](#)

Bibliografia

Introduzione

Coltivo la passione per la programmazione dei computer dagli anni '80 del secolo scorso quando all'età di 12 anni fui colpito da un paio di pagine, sul libro di educazione tecnica delle scuole medie, che parlavano di queste macchine fenomenali, i calcolatori elettronici.

Dopo esperienze professionali come analista e programmatore, da 18 anni inseguo questi argomenti negli istituti tecnici industriali per periti informatici e il libro che avete in mano è frutto delle mie lezioni agli studenti che si avvicinano alla programmazione dei computer, competenza impegnativa da acquisire che mi sforzo di far apprendere ai futuri giovani tecnici, in modo rigoroso e professionale.

Il primo capitolo è dedicato ad una panoramica sull'informatica intesa come scienza degli algoritmi (come la si dovrebbe sempre intendere) e non come tecnologia informatica. Il resto dei capitoli è un manuale dedicato al linguaggio C# partendo dalle basi e sviluppando semplici programmi ad interfaccia a carattere. Vengono spiegati nel dettaglio i costrutti basilari, i tipi di dato fondamentali, gli operatori, le stringhe, le strutture di controllo e i metodi, comprendendo anche le novità introdotte dalle ultime versioni del linguaggio (in particolare la versione 7.2); uno degli ultimi capitoli è dedicato alle espressioni regolari.

Vengono affrontati anche alcuni argomenti, come gli algoritmi fondamentali di ricerca e ordinamento e la ricorsione, che ogni programmatore dovrebbe conoscere.

La programmazione ad oggetti non è tra gli obiettivi del libro ma viene comunque introdotta nel capitolo dedicato alla metodologia top-down e ai metodi, con lo scopo di far comprendere meglio i meccanismi del linguaggio.

1. Algoritmi e linguaggi di programmazione

Prima di cominciare lo studio del linguaggio C# vale la pena richiamare alcuni concetti basilari di informatica.

Iniziamo dal termine **informatica**, traduzione del termine francese ***informatique***, tratto da *infor(mation) (autom)atique*, “informazione automatica”.

Il dizionario definisce l’informatica come la scienza che studia l’informazione e più specificatamente l’elaborazione dei dati e il loro trattamento automatico.

Ma una definizione che ci piace di più è quella dell’ACM (*Association for Computing Machinery*, associazione internazionale accademica dedicata a scienziati dell’informatica): l’informatica è la scienza che studia la teoria, l’analisi, la progettazione, l’efficienza, la realizzazione e l’applicazione degli **algoritmi**, cioè dei procedimenti che descrivono e trasformano l’informazione.

1.1. Problemi e algoritmi

Ma che cosa si intende per algoritmo? Un **algoritmo** definisce un insieme di azioni che risolvono un determinato problema, trasformando dati di partenza (**dati in input**) in dati di arrivo (**dati di output**) attraverso le relazioni esistenti tra di essi.

Un algoritmo, per dirsi tale, deve essere composto da un insieme finito e ordinato di passi eseguibili e non ambigui (**istruzioni**) che giunge certamente a terminazione.

Insieme ordinato significa che deve avere un determinato ordine di esecuzione, esiste cioè una prima istruzione ed è sempre possibile stabilire qual è la prossima istruzione; i passi devono essere effettivamente eseguibili cioè devono essere portati a termine e devono essere non ambigui cioè durante l'esecuzione dell'algoritmo lo stato raggiunto permette di stabilire univocamente e completamente le azioni da svolgere subito dopo (l'algoritmo si dice *deterministico*).

La definizione di un algoritmo utilizza una serie di **primitive**.

Le primitive sono componenti fondamentali definite precisamente al fine di eliminare problemi di ambiguità: ogni primitiva ha una rappresentazione simbolica (sintassi) e un significato (semantica).

Un **linguaggio di programmazione** è un insieme di primitive e di regole (che spiegano come possono essere combinate assieme le primitive). Nell'ambito dei linguaggi di programmazione si usa anche il termine istruzione: le **istruzioni** costituiscono i comandi impartiti ad un esecutore (il computer) utilizzando un linguaggio di programmazione.

A seconda di dove si collocano le primitive rispetto alle primitive costituite dalle istruzioni macchina (cioè quelle che la macchina è in grado di eseguire) l'insieme di primitive si dice di “*livello più alto*”; una primitiva di livello più alto è cioè descritta in termini di primitive di livello più basso.

Per descrivere un algoritmo, anziché usare da subito un linguaggio di programmazione formale, si può usare un linguaggio meno formale e più intuitivo chiamato **pseudocodice** che permette di rappresentare l'algoritmo indipendentemente dal linguaggio di programmazione che si utilizzerà.

Lo pseudocodice consiste di notazioni concise e coerenti per rappresentare le strutture sintattiche e semantiche più usate.

Un esempio di pseudocodice che rappresenta l'**algoritmo di Euclide**^[1], per il calcolo del massimo comun divisore tra due numeri naturali, è mostrato di seguito:

INIZIO

Leggi in input A e B

Finchè $A \text{ mod } B = 0$ (*mod* è l'operatore di resto)

calcola $M = A \text{ mod } B$

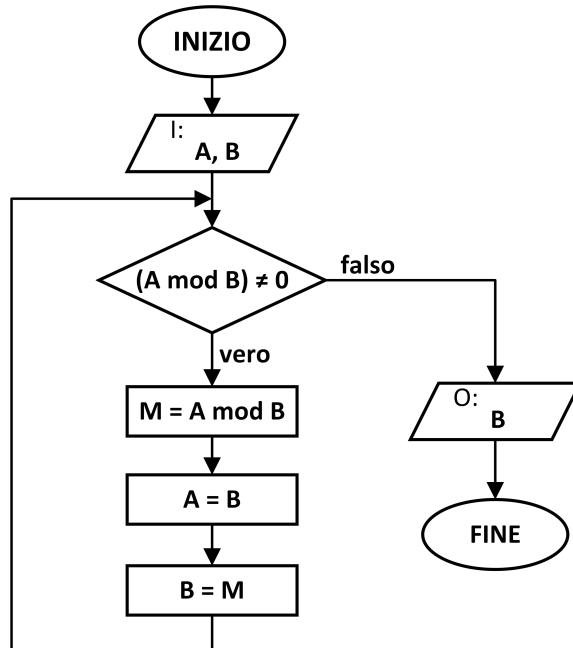
assegna $A = B$

assegna $B = M$

Scrivi in output B

FINE

Un'altra rappresentazione degli algoritmi utilizzata fin dagli albori dell'informatica fa uso di **diagrammi grafici a blocchi** (o **diagrammi di flusso, flow chart**). Ogni blocco ha un ruolo preciso e la rappresentazione si dimostra utile per “visualizzare” il flusso dell'algoritmo. A titolo di esempio, l'algoritmo di Euclide viene rappresentato con il seguente diagramma di flusso:

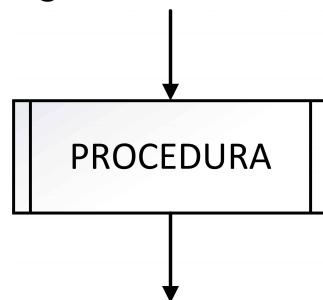


Un diagramma di flusso utilizza i seguenti blocchi di base:

- ellissi per rappresentare l'inizio e la fine dell'algoritmo,

- *parallelogrammi* per rappresentare le istruzioni per la lettura degli input e la scrittura degli output
- *rettangoli* per rappresentare istruzioni che modificano i dati
- *rombi* per indicare condizioni da verificare che possono essere vere o false.

Per rendere più agevole la rappresentazioni di algoritmi complessi, questi vengono suddivisi in **procedure**^[2] (o **sottoprogrammi**) ovvero unità di programma identificate da un nome che risolvono un sottoproblema del problema complessivo da risolvere; rappresentano cioè algoritmi che possono essere riutilizzati e compongono l'algoritmo complessivo che risolve l'intero problema. Il comportamento di una procedura può dipendere da parametri in modo da renderla più generale e utilizzabile per risolvere problemi dello stesso tipo. Nei diagrammi di flusso le procedure vengono rappresentate con la seguente grafica:



1.2. Tipi di dato, variabili e costanti

Gli algoritmi sono azioni che trasformano **informazioni**. Le informazioni rappresentano la conoscenza che abbiamo del mondo e sono rappresentate dai **dati**, ovvero sequenze di simboli. I dati con il significato che gli viene attribuito, costituiscono l'informazione.

I dati vengono classificati in base ad un **tipo** che ne determina l'insieme dei valori rappresentabili e delle operazioni ammesse per quei valori; ad esempio l'operazione di divisione sarà applicabile solo a tipi numerici.

Tutti i linguaggi di programmazione supportano i seguenti tipi di base che è frequente utilizzare nei programmi:

- tipi **numerici**: rappresentano valori numerici di tipo **intero** o **reale**
- tipo **logico** (o **booleano**^[3]): rappresenta i valori vero e falso.
È un tipo utilizzato per rappresentare la verità o falsità di proposizioni logiche ed è indicato nei linguaggi di programmazione con le costanti **true** (vero) e **false** (falso)
- tipo **carattere**: rappresenta singoli caratteri. Solitamente i valori di tipo carattere sono delimitati da singoli apici.
- tipo **stringa**: rappresenta sequenze di caratteri. I valori di tipo stringa sono delimitati da doppi apici.

I linguaggi mettono poi a disposizione tipi più articolati e complessi che vengono definiti sulla base dei suddetti tipi; anche il programmatore può definire propri tipi per rappresentare le situazioni reali gestite dai programmi.

I dati vengono identificati negli algoritmi attraverso l'uso di **variabili**. Possiamo immaginare una variabile come un contenitore a cui è attribuito un nome che la identifica (**identificatore della variabile**); di fatto in un programma per computer le variabili corrispondono ad aree della memoria destinata al programma quando questo viene eseguito.

Il **tipo della variabile** determina quanto grande e come verrà gestita l'area di memoria associata alla variabile.

Molti linguaggi di programmazione obbligano il programmatore a dichiarare quali variabili utilizzerà, altri deducono l'esistenza delle variabili la prima volta che vengono utilizzate; anche il tipo delle variabili può essere

dichiarato esplicitamente oppure può essere dedotto dal contesto e dai valori caricati.

Una variabile, in base al tipo di appartenenza, potrà assumere valori diversi durante l'esecuzione del programma (da qui il nome). Allo scopo esiste un'istruzione fondamentale che consente di "caricare" o variare il valore di una variabile: l'**istruzione di assegnazione** (o *assegnamento*).

Senza l'istruzione di assegnazione non sarebbe possibile gestire dati diversi con lo stesso algoritmo; le variabili consentono cioè di "parametrizzare" l'algoritmo e produrre risultati diversi in uscita a partire da dati diversi in entrata.

L'istruzione di assegnazione è indicata dalla stragrande maggioranza dei linguaggi con il simbolo = ("uguale"); in pseudocodice o nei diagrammi di flusso può essere indicata con una freccia ← .

Ha le seguenti forme:

identificatore_variabile = valore

identificatore_variabile ← valore

a sinistra del segno si indica la variabile da modificare e a destra il valore da assegnare che può essere un valore singolo o il risultato del calcolo di un'espressione composta utilizzando operatori ed altri dati.

Ecco alcuni esempi di istruzioni di assegnazione:

risposta = 42

velocità = 85.7

sesso = 'F'

cognome = "Adams"

risposta = risposta+1

Va sempre ricordato che l'istruzione di assegnazione opera nel seguente ordine:

- viene valutata la parte che sta a destra del simbolo =
- il valore calcolato viene attribuito alla variabile che sta a sinistra.

Tornando agli esempi, le prime quattro istruzioni assegnano un valore fisso alle variabili indicate a sinistra dell’assegnazione: nei primi due casi si tratta di variabili numeriche, nel terzo caso di un variabile carattere e nel quarto caso di una stringa.

L’ultimo esempio è più interessante: viene calcolato il valore dell’espressione a destra utilizzando il valore della variabile **risposta** e il risultato (43) viene poi caricato nella medesima variabile.

Il valore assegnato ad una variabile deve essere compatibile con il suo tipo; quindi non potremo ad esempio assegnare una stringa ad una variabile numerica o, viceversa, un valore numerico ad una variabile di tipo stringa.

I valori che non possono cambiare mai durante l’esecuzione del programma vengono detti **costanti**. Le costanti si suddividono in:

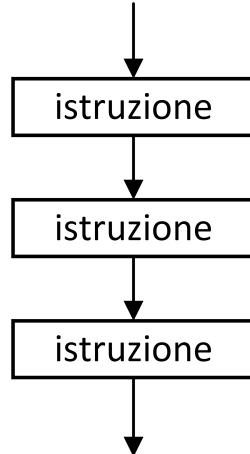
- **costanti letterali (*literal*)** come i valori 42, “Adams”
- **costanti simboliche**: sono costanti letterali a cui è attribuito un identificatore. È convenzione comune usare identificatori tutti maiuscoli per le costanti, ad esempio PI_GRECO, VERSIONE, ...

1.3. Schemi di composizione delle istruzioni

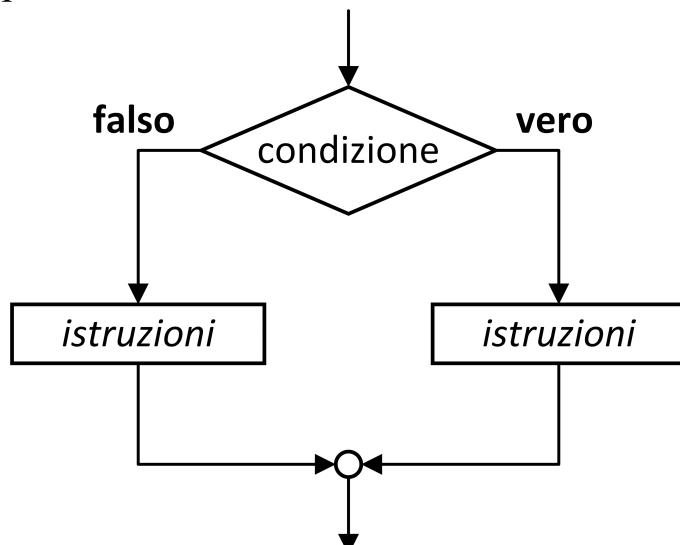
Pochi problemi sono risolvibili da una sequenza lineare di istruzioni.

Piuttosto, gli algoritmi sono descritti utilizzando tre schemi di composizione fondamentali che vengono combinati assieme per comporre gli algoritmi:

- **sequenza**: esecuzione sequenziale di due o più istruzioni

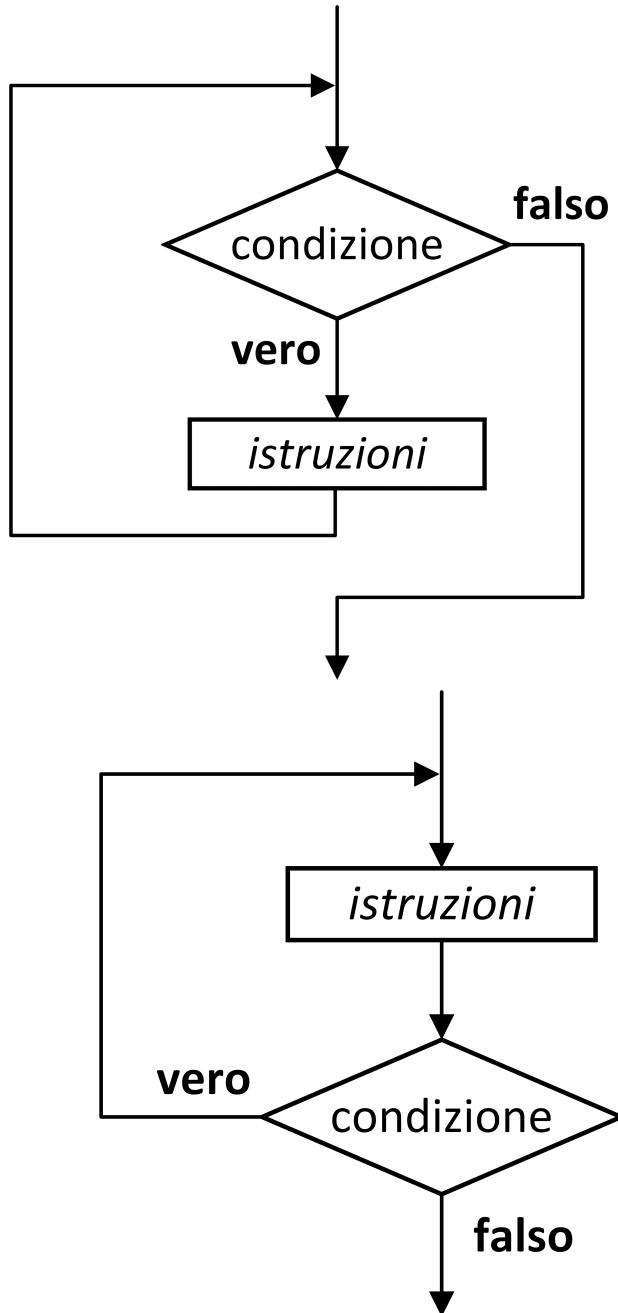


- **selezione** (*scelta, condizione*): esecuzione alternativa di istruzioni in base al verificarsi di una condizione. Il flusso del programma prende cioè un percorso se una condizione risulta vera e un percorso alternativo se la condizione risulta falsa; in entrambi i casi, il flusso prosegue poi dall'istruzione subito successiva al blocco di selezione



- **iterazione** (*ripetizione, ciclo*): esecuzione ripetuta di una o più istruzioni finché è verificata una condizione. È possibile individuare

due schemi per l'iterazione che si differenziano in base a quando viene verificata la condizione di ripetizione del ciclo:



Il ciclo rappresentato a sinistra viene chiamato con **controllo in testa** (o *ciclo pre-condizionale*): la sequenza di istruzioni presente all'interno del ciclo viene eseguita fintantoché la condizione rimane vera.

Il ciclo di destra è invece un ciclo con **controllo in coda** (*ciclo post-condizionale*); in questo caso la sequenza di istruzioni presente

all'interno del ciclo viene eseguita almeno una volta e fintantoché la condizione rimane vera.

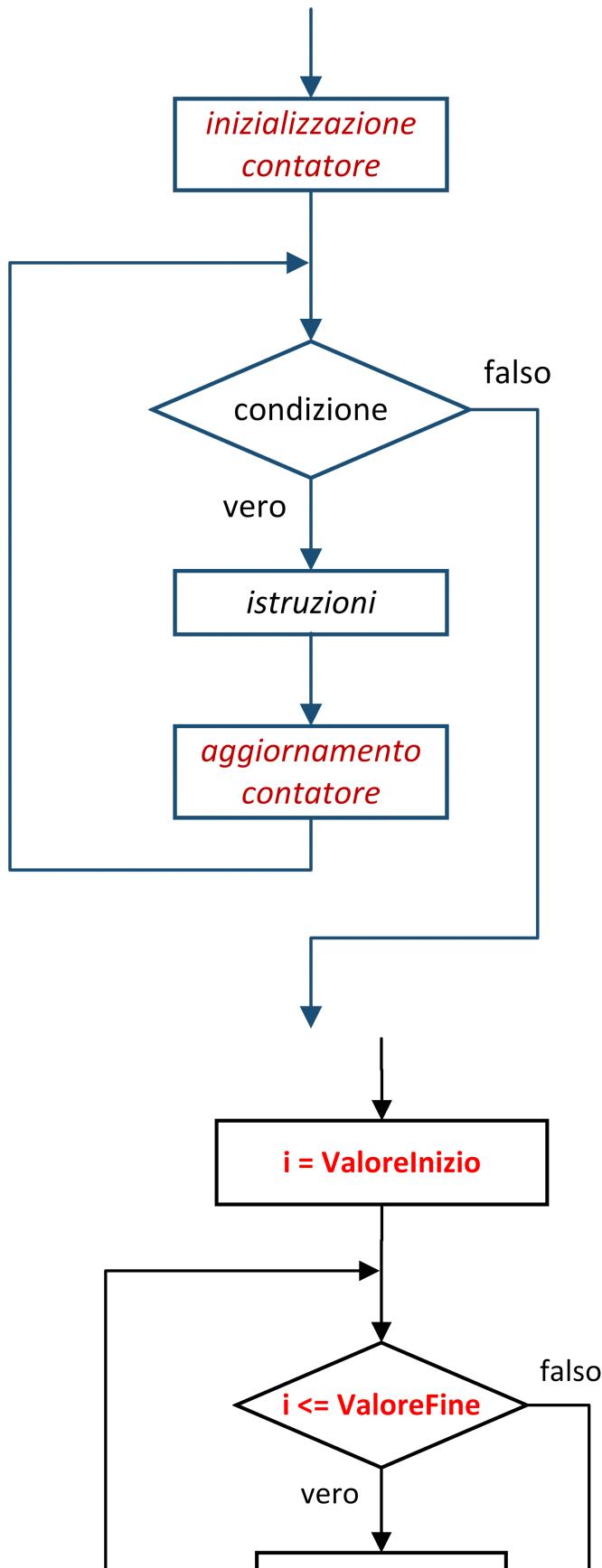
Gli schemi di composizione possono essere considerati come singole istruzioni; ogni schema può cioè usare gli altri schemi per comporre strutture più articolate.

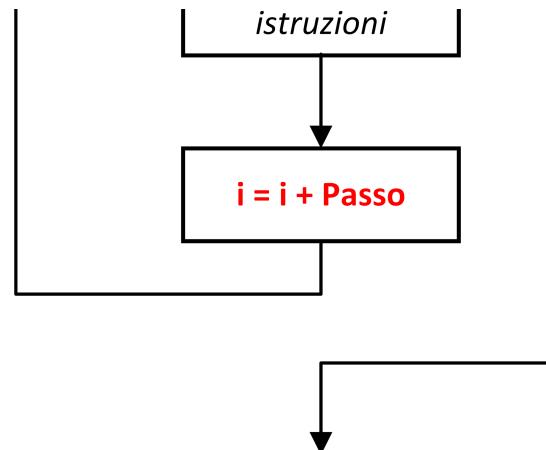
Un importante teorema dimostrato da due informatici italiani (Corrado Böhm e Giuseppe Jacopini) nel 1966 ha stabilito che qualunque algoritmo può essere implementato utilizzando tre sole strutture, la sequenza, la selezione e l'iterazione, da applicare ricorsivamente alla composizione di istruzioni elementari.

Questo risultato ha contribuito alla definizione delle linee guida della **programmazione strutturata** e alla critica sull'uso sconsigliato delle istruzioni di salto (la famigerata istruzione **goto**) che al tempo venivano abbondantemente utilizzate dai programmati.

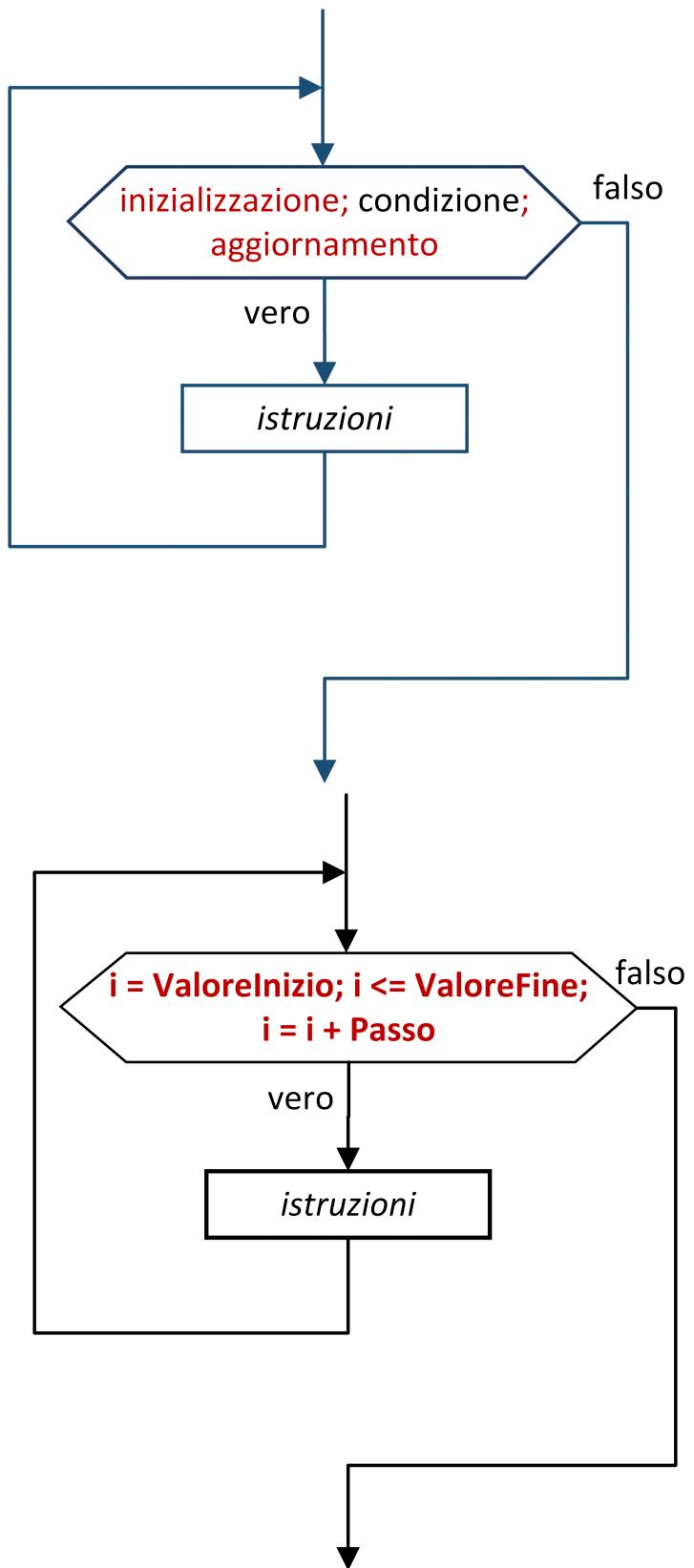
Il termine **programmazione strutturata**^[4] è stato coniato da un grande informatico olandese, Edsger W. Dijkstra, che alla fine degli anni 1960 ha contribuito alla stesura di regole per una programmazione rigorosa e che consentisse una più facile manutenzione e aggiornamento del software. Famoso il suo articolo del 1968, “*Goto statement considered harmful*”, sugli effetti deleteri del goto sulla qualità del software, e in particolare sulla sua leggibilità e manutenzione (il cosiddetto problema dello *spaghetti code*).

Negli algoritmi è comune utilizzare cicli pre-condizionali che effettuano un conteggio; la struttura di questi cicli è mostrata nei seguenti diagrammi ed è stata formalizzata nei linguaggi di programmazione con costrutti sintattici ad hoc.





Nei diagrammi di flusso il ciclo con contatore viene anche rappresentato con la seguente grafica che ingloba nel blocco che rappresenta il ciclo, l'istruzione di inizializzazione, la condizione di iterazione e l'istruzione di aggiornamento del contatore:



1.4. Espressioni condizionali composte

Tutti i linguaggi di programmazione mettono a disposizione gli operatori logici condizionali per scrivere condizioni composte, in modo da semplificare la scrittura degli algoritmi.

I simboli usati in molti linguaggi sono:

- **&&** per il connettivo “e” (**and**)
- **||** per il connettivo “o” (**or**)
- **!** per l’operatore di negazione (**not**)

Richiamiamo le tavole di verità degli operatori logici.

L’operatore **&& (and)** viene utilizzato per legare due o più condizioni e assume valore vero se e solo se tutte le condizioni componenti sono vere:

<i>condizione1</i>	<i>condizione2</i>	<i>condizione1 && condizione 2</i>
falso	falso	falso
falso	vero	falso
vero	falso	falso
vero	vero	vero

L’operatore **|| (or)** assume valore vero se almeno una delle condizioni componenti è vera:

<i>condizione1</i>	<i>condizione2</i>	<i>condizione1 condizione 2</i>
falso	falso	falso
falso	vero	vero
vero	falso	vero
vero	vero	vero

Infine l’operatore **!** (**not**) inverte il valore di verità della condizione a cui viene applicato:

<i>condizione</i>	<i>!condizione</i>
falso	vero
vero	falso

1.5. Linguaggi di programmazione

Abbiamo già visto che un linguaggio di programmazione è costituito da un insieme di primitive e di regole di composizione che spiegano come possono essere combinate assieme le primitive.

In base al livello delle primitive i linguaggi vengono classificati di alto o basso livello: basso perché vicini alla macchina, alto perché più astratti e vicini al linguaggio umano.

I primi linguaggi creati per i computer usavano istruzioni costituite da puri codici numerici; si parla di **linguaggio macchina**.

Questo modo di programmare i computer si è rivelato piuttosto complicato e ostico per i programmatori così sono stati ben presto inventati i **linguaggi assembly** (“di assemblaggio”) che usano istruzioni costituite da codici mnemonici: in sostanza ai codici numerici vengono associati dei comandi testuali più facili da ricordare rispetto ai numeri. Si usano programmi chiamati **assembler** (“assemblatori”) che traducono il programma assembly in linguaggio macchina.

Linguaggi macchina e assembly sono intrinsecamente dipendenti dalla macchina e il programmatore è costretto a pensare al programma in termini di primitive fin troppo elementari.

Un’evoluzione significativa al modo con cui venivano programmati i computer è arrivata con i linguaggi della 3° generazione, chiamati **linguaggi di alto livello**, nati alla fine degli anni 1950. Un linguaggio di alto livello usa primitive di livello più alto che consentono una maggiore astrazione; le primitive vengono poi tradotte in istruzioni macchina da speciali programmi (*compilatori* e *interpreti*).

Vi sono poi linguaggi di 4° generazione che alzano ancora il livello di astrazione rispetto alla macchina; non obbligano il programmatore a sviluppare procedure ma usano sintassi semplici simili al linguaggio naturale. La loro applicazione è specifica per determinate applicazioni.

I linguaggi di 5° generazione sfruttano invece l’intelligenza artificiale e le reti neurali.

I linguaggi di programmazione usati principalmente oggigiorno sono tutti linguaggi di 3° generazione e sono di tipo **general purpose** cioè sono pensati per lo sviluppo di programmi in ambiti applicativi anche molto diversi tra loro.

Nello sviluppo dei linguaggi di programmazione sono stati inventanti diversi **paradigmi di programmazione**; con il termine **paradigma** si intende l'approccio, il modello utilizzato dal linguaggio per descrivere i programmi. I principali paradigmi di programmazione sono:

- imperativo (o procedurale)
- dichiarativo
- funzionale
- orientato agli oggetti

Il **paradigma imperativo**, anche chiamato *paradigma procedurale*, rappresenta l'approccio tradizionale alla programmazione. Viene chiamato imperativo perché la programmazione consiste in una sequenza di comandi che, eseguiti dalla macchina, elaborano i dati per produrre i risultati desiderati. Il programmatore deve progettare un algoritmo per risolvere il problema. L'algoritmo può essere inoltre suddiviso in *procedure* mirate a risolvere determinati sottoproblemi.

Nel **paradigma dichiarativo** il programmatore descrive precisamente il problema anziché progettare un algoritmo per risolverlo; viene realizzato un algoritmo generale per la risoluzione dei problemi e i problemi specifici vengono ricondotti a questo. I linguaggi dichiarativi hanno applicazioni a specifici problemi come ad esempio il linguaggio SQL (*Structure Query Language*), linguaggio dichiarativo per i database.

Linguaggi dichiarativi più generici usano invece le regole della logica matematica (ad esempio PROLOG); il programma è costituito da un insieme di asserzioni iniziali alle quali il programma applica il ragionamento deduttivo.

Il **paradigma funzionale** descrive il programma mediante la concatenazione di funzioni (una funzione accetta un input e produce un output); le primitive di un linguaggio funzionale sono costituite da funzioni elementari a partire dalle quali il programmatore costruisce funzioni più complesse. Uno dei primi e più famosi linguaggi funzionali è stato LISP.

Il **paradigma orientato agli oggetti** rappresenta il paradigma più diffuso nello sviluppo moderno del software (molti linguaggi lo utilizzano mescolato con gli altri paradigmi). Il programma viene descritto come una collezione di unità chiamate oggetti che interagiscono tra di loro per svolgere l'elaborazione richiesta. Il processo di programmazione viene

chiamato OOP (Object Oriented Programming) ed enfatizza la riutilizzabilità del codice e migliora la manutenzione del software.

Le procedure (metodi) all'interno di un oggetto sono piccole unità di codice imperativo. La struttura degli oggetti con le stesse caratteristiche sono definite attraverso il concetto di classe che rappresenta un certo tipo di oggetti.

Concludiamo questo paragrafo con una carrellata dei principali linguaggi di alto livello in ordine cronologico.

1957 → **FORTRAN** (*FORmula TRANslator*) di John Backus (IBM) che fu tra i primi ad avere l'idea di un linguaggio più vicino alla logica umana. Usato ancora oggi per il calcolo scientifico.

1958 → **LISP** (*LISt Processor*) di John McCarthy (MIT), primo linguaggio a usare il paradigma funzionale e pensato per applicazioni di intelligenza artificiale. Dopo il Fortran, è il più vecchio linguaggio di programmazione di alto livello ancora in uso.

1959 → **COBOL** (*COmmon Business-Oriented Language*) di Grace Hopper (Ministero Marina USA), considerata l'autrice del primo compilatore (ha diffuso l'uso del termine *bug*). Usato in moltissime applicazioni bancarie e gestionali.

1960 → **ALGOL** (*ALGOrithmic Language*), sviluppato da un comitato di informatici statunitense ed europeo, ha influenzato una serie di linguaggi successivi, dal Pascal al C.

1964 → **BASIC** (*Beginner's All-purpose Symbolic Instruction Code*) di John George Kemeny e Thomas Eugene Kurtz (Università di Dartmouth), linguaggio imperativo che si è diffuso con l'avvento degli home e personal computer. Criticato per l'utilizzo incondizionato della famigerata istruzione GOTO.

1967 → **Simula**, sviluppato al Norwegian Computing Center di Oslo, principalmente da Ole-Johan Dahl e Kristen Nygaard. Ispirato ad ALGOL, è il primo linguaggio di programmazione orientato agli oggetti (l'elaborazione non è più descritta da una procedura, ma è ottenuta da una "simulazione" del mondo, di cui gli oggetti sono un modello).

1970 → **Pascal**, di Niklaus Wirth (Politecnico di Zurigo), linguaggio imperativo basato su ALGOL che mette in pratica la programmazione

strutturata. Di ampia diffusione creato per l'insegnamento, con sintassi chiara e rigida e dotato di strutture dati avanzate.

1972 → **C**, di Dennis Ritchie (Bell Labs della AT&T), linguaggio imperativo sviluppato con lo scopo di utilizzarlo per la stesura di UNIX (sistema operativo precedentemente realizzato in assembly a partire dal 1969 da Ken Thompson e Dennis Ritchie).

1970 → **Smalltalk**, di Alan Kay (Xerox PARC), derivato da Simula, è uno dei primi linguaggi a dimostrare la potenza della programmazione ad oggetti.

1972 → **Prolog** (dal francese *PROgrammation en LOGique*), di Robert Kowalski e Alain Colmerauer. Primo linguaggio ad usare la programmazione logica, usato nell'intelligenza artificiale e nella linguistica computazionale.

1983 → **C++**, di Bjarne Stroustrup (AT&T Bell Labs) sviluppato come miglioramento del C con l'introduzione della programmazione ad oggetti (si chiamava inizialmente “C con classi”). Ha influenzato molti linguaggi successivi come Java, C#, PHP e altri. Molti software applicativi e librerie dei sistemi operativi sono scritte in C++.

1986 → **Eiffel**, di Bertrand Meyer (Politecnico di Zurigo, Eiffel Software), linguaggio orientato agli oggetti pensato per l'utilizzo industriale. Ha influenzato linguaggi successivi come Java e C#.

1991 → **Python**, di Guido van Rossum, linguaggio orientato agli oggetti che predilige la leggibilità e semplicità. Negli ultimi anni si è molto diffuso nello sviluppo di applicazioni web, nel calcolo scientifico e come linguaggio introduttivo per gli studenti.

1995 → **Java**, di James Gosling (Sun Microsystems), linguaggio di ampia diffusione, orientato agli oggetti e progettato per essere il più possibile indipendente dalla piattaforma di esecuzione (famoso il motto WORA, “*write once, run anywhere*”) perché basato sul concetto di macchina virtuale (JVM, *Java Virtual Machine*).

1995 → **Delphi**, di Anders Hejlsberg (Borland), linguaggio orientato agli oggetti basato sul linguaggio Pascal. È anche un ambiente di sviluppo, primo ad essere completamente visuale e primo ad essere conosciuto come strumento per lo sviluppo rapido di applicazioni (RAD, *Rapid Application Development*).

2000 → C#, di Anders Hejlsberg, linguaggio orientato agli oggetti sviluppato da Microsoft all'interno dell'iniziativa .NET, e successivamente approvato come standard ECMA e ISO.

Ovviamente questi sono solo alcuni della miriade di linguaggi di programmazione esistenti; dovremo citarne anche altri come Objective C (1983), Perl (1987), Visual Basic (1991), JavaScript (1995), PHP (1995), Ruby (1995), Scala (2003), F# (2005), Go (2009), Dart (2011), Kotlin (2011), Swift (2014), ...^[5]

1.6. Processo di traduzione

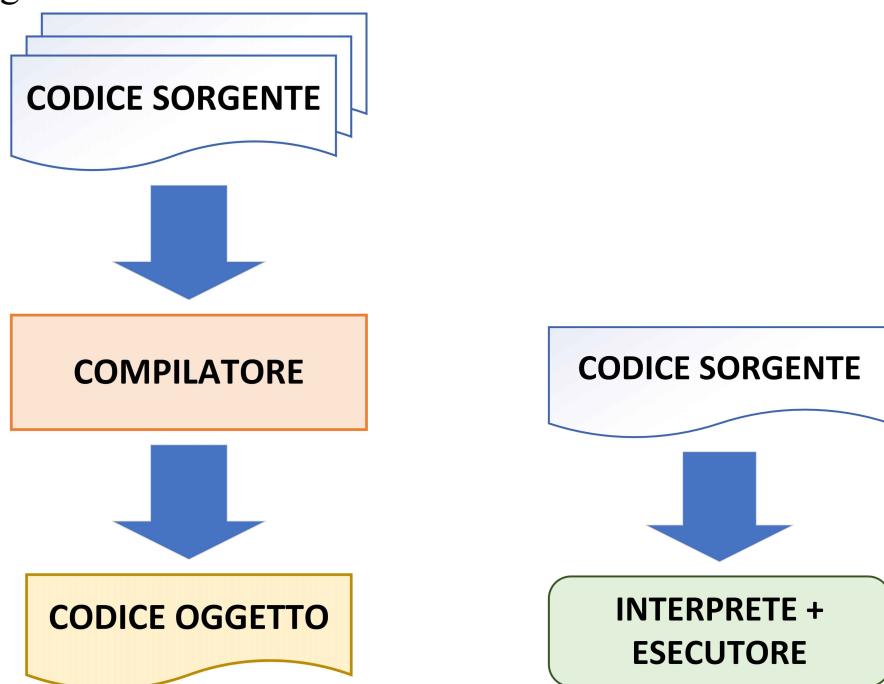
La traduzione di un programma da un linguaggio ad un altro avviene in tre fasi:

1. l'analisi lessicale riconosce le singole entità (*token*) del programma
2. l'analisi sintattica identifica la struttura grammaticale del programma secondo le regole sintattiche del linguaggio usato
3. la generazione del codice cioè la costruzione delle istruzioni in linguaggio macchina che corrispondono alle istruzioni riconosciute ai passi precedenti.

Il programma nella forma originale viene chiamato **programma sorgente** mentre il programma tradotto è il **programma oggetto**.

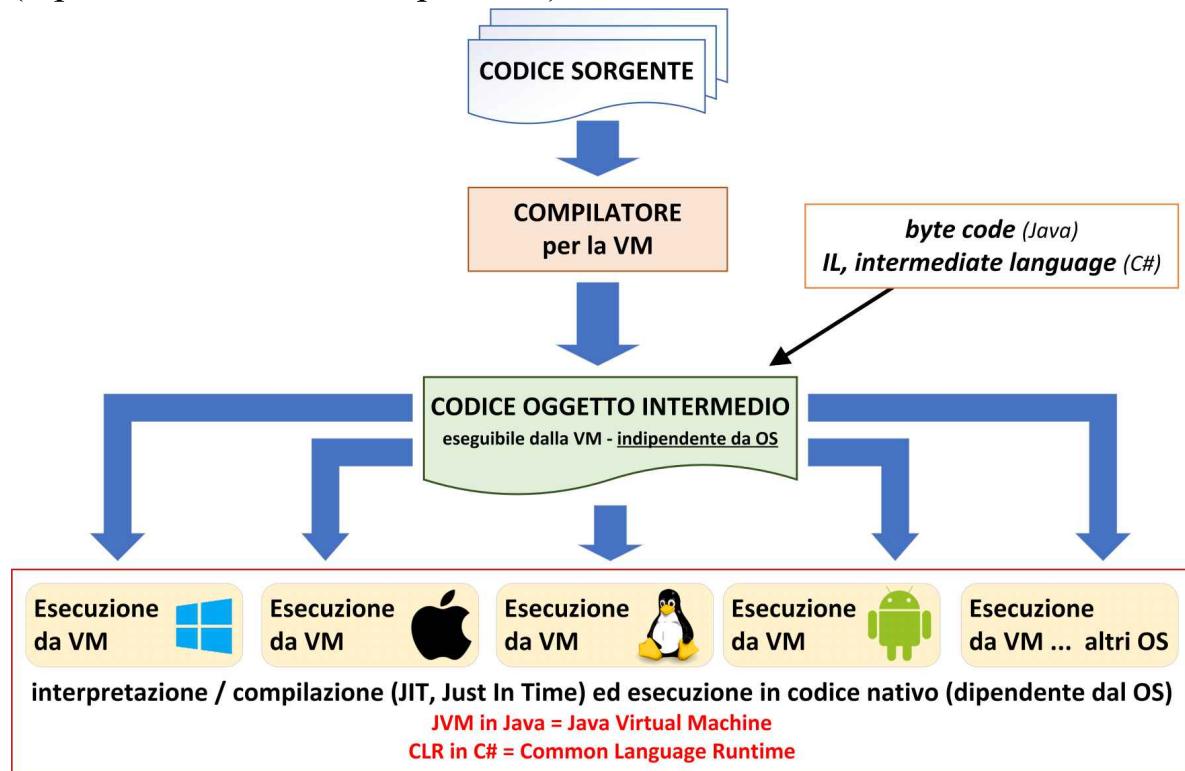
I programmi usati per la traduzione si distinguono in:

- **compilatori**: traducono il programma sorgente una sola volta; solo se intervengono modifiche al programma sorgente si deve rifare la traduzione
- **interpreti**: la traduzione viene fatta ogni volta che il programma è eseguito



Esiste poi una situazione che sta nel mezzo, nella quale il compilatore genera del codice oggetto intermedio per una **macchina virtuale**, indipendente dal sistema operativo.

In fase di esecuzione del programma, la macchina virtuale si occupa di tradurre al volo il codice oggetto intermedio nel codice oggetto nativo (dipendente dal sistema operativo).



Tale modello è utilizzato oggi in diversi linguaggi, a partire dal più vecchio Java fino al più recente C#.

1.7. Il .NET Framework e il linguaggio C#

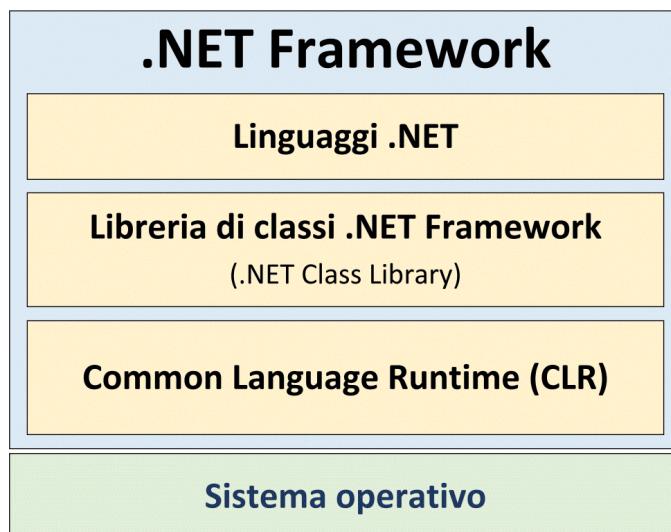
La suite di prodotti .NET (si pronuncia dotnet) è un progetto annunciato da Microsoft nel 2000, all'interno del quale è stata creata una nuova piattaforma di sviluppo e di esecuzione delle applicazioni (il **.NET Framework**) integrando numerose tecnologie degli anni precedenti. Le tecnologie di base di .NET erano originariamente state sviluppate da Microsoft come propria versione di Java, per poi evolvere, nel 1998 nel linguaggio J++ e successivamente in .NET.

Gli obiettivi del .NET Framework sono:

- indipendenza dalla piattaforma hardware (PC, dispositivi mobili, console, ...) e software (sistema operativo)
- integrabilità in ambiente internet garantendo massima sicurezza e integrità dei dati
- facilità di sviluppo rapido delle applicazioni (RAD, Rapid Application Development) utilizzando il concetto di modularità dei componenti software.

Il .NET Framework si pone al di sopra del sistema operativo, che può essere un sistema Windows o, potenzialmente, qualsiasi altro sistema; esiste ad esempio un porting di .NET su Linux chiamato Mono.

.NET è essenzialmente una applicazione che gira sul sistema operativo



Il .NET Framework contiene e/o supporta già i compilatori per C# (csc.exe), Visual Basic (vbc.exe), C++, F# e JScript (jsc.exe).

Oltre a questi linguaggi, forniti da Microsoft, sono utilizzabili altri linguaggi forniti da altri produttori come Pascal, Fortran, RPG, Cobol, Python, Lisp, ecc.^[6]

Il **CLR** (*Common Language Runtime, Ambiente di esecuzione per il linguaggio comune*) è il componente più importante del .NET Framework che sta alla base dell'architettura. Il programma sorgente, scritto in uno qualsiasi dei linguaggi .NET, viene tradotto in un linguaggio macchina intermedio detto **CIL** (*Common Intermediate Language*) o semplicemente **IL**.

Il CLR gestisce ed esegue il codice IL per la macchina hardware e il sistema operativo sottostante. In pratica il ruolo del CLR è quello di una macchina virtuale che si sovrappone alla macchina hardware e al sistema operativo.

In questo assomiglia alla JVM (Java Virtual Machine) anche se diversamente da questa, nasce con la possibilità di usare molti linguaggi^[7].

Possiamo quindi immaginare il CLR come un “simulatore” di un computer con un proprio linguaggio macchina (il linguaggio IL).

Tutti i compilatori che aderiscono alla struttura del CLR devono generare un rappresentazione intermedia del codice, indipendente dalla CPU nel Common Intermediate Language.

Il **runtime**, cioè l'ambiente di esecuzione, utilizza questo linguaggio intermedio per generare codice nativo attraverso il **compilatore JIT** (*Just-In-Time, al volo*) o **JITter**. La modalità di compilazione Just-In-Time permette di ottenere prestazioni maggiori rispetto all'esecuzione di un codice interpretato.

Microsoft fornisce un assembler di linguaggio IL chiamato **ILAsm** e il corrispondente disassemblatore, **ILDasm** che permette di ottenere il codice IL a partire da un programma eseguibile.

Quando un programma viene compilato in IL, il codice ottenuto è conservato all'interno di uno o più file chiamati **assembly**. Le estensioni utilizzate per i file assembly sono ancora **.exe** e **.dll** ma sono differenti dal formato nativo dei file eseguibili e delle librerie Windows.

Ogni assembly contiene: il codice IL, un manifest che descrive l'assembly, dei metadati che descrivono i tipi contenuti nell'assembly ed eventuali

risorse opzionali (immagini, audio, ...).

Il .NET Framework dispone di una vasta **libreria di classi** ovvero di un insieme di tipi riutilizzabili che si integrano con il CLR.

La libreria mette a disposizione tipi per diverse tipologie di applicazioni (desktop, web, mobile, ...) e diversi ambiti applicativi (input/output, database, XML, ...). Le classi sono organizzate in **namespace** cioè raggruppate secondo le medesime funzionalità (il namespace **System** è quello principale). I namespace sono organizzati in modo gerarchico cioè ogni namespace può contenerne altre in modo simile a come avviene con le cartelle del file system.

Con il progetto .NET Microsoft ha presentato anche un nuovo linguaggio di programmazione chiamato **C#** (pronuncia: *SiSharp*), che è il linguaggio che meglio degli altri descrive le linee guida sulle quali ogni programma .NET gira. Questo linguaggio è stato creato da Microsoft (principalmente da Anders Hejlsberg) specificatamente per la programmazione nel Framework .NET ed è stato reso da subito standard ECMA (Standard ECMA-334^[8]) e ISO (ISO/IEC 23270:2018^[9]).

I suoi tipi di dati “primitivi” hanno una corrispondenza univoca con i tipi .NET e molte delle sue caratteristiche sono caratteristiche proprie del .NET framework.

Assomiglia a Java e al C++ ma rispetto a quest’ultimo è stato costruito per evitare errori tipici della programmazione C.

Visual Studio è l’ambiente di sviluppo integrato (IDE, *Integrated Development Environment*) di Microsoft per la piattaforma .NET. È disponibile sia per Windows che Mac OS.

Dal 2015 è disponibile una versione liberamente utilizzabile denominata, **Visual Studio Community**, <https://visualstudio.microsoft.com/it/vs/community/>.

Recentemente, a novembre 2014, Microsoft ha annunciato la distribuzione^[10] della propria piattaforma di sviluppo **.NET Core** con licenza open source.

.NET Core è una versione modulare del .NET framework cross-platform e disponibile per i sistemi operativi Windows, MacOS e Linux. È costituita principalmente da:

- **CoreCLR**: motore di esecuzione (runtime)
- **CoreFX**: libreria di classi fondamentali per lo sviluppo
- **Roslyn**: compilatore C# e Visual Basic.

Elenchiamo infine nella seguente tabella la cronistoria delle versioni di C# con le corrispondenti versioni del .NET Framework e dell'ambiente di sviluppo Visual Studio:

<i>Anno</i>	<i>.NET Framework</i>	<i>C#</i>	<i>Visual Studio</i>
2002	1.0	1.0	Visual Studio.NET 2002
2003	1.1	1.2	Visual Studio.NET 2003
2005	2.0	2.0	Visual Studio 2005
2006	3.0	2.0	Visual Studio 2005
2007	3.5	3.0	Visual Studio 2008
2010	4.0	4.0	Visual Studio 2010
2012	4.5	5.0	Visual Studio 2012
2013	4.5.1 – 4.5.2	6.0	Visual Studio 2013
2015	4.6	6.0	Visual Studio 2015
2017	4.6.2	7.0	Visual Studio 2017
2017-2018	4.7 – 4.7.1 – 4.7.2	7.1 – 7.2 – 7.3	Visual Studio 2017 15.3 – 15.5 – 15.7
2019	4.8	8.0	Visual Studio 2019

2. Un primo programma in C#

C# nasce come linguaggio orientato agli oggetti ma può essere usato in modo semplificato per affrontare la programmazione imperativa.

C# è un linguaggio ***general purpose***, cioè può essere usato per sviluppare praticamente qualsiasi tipo di applicazione, dal software per dispositivi mobili a software per PC e server, dall'app per smartphone al gestionale aziendale.

Per avvicinarci alla programmazione in C# e tradurre algoritmi imperativi svilupperemo ***applicazioni ad interfaccia a carattere***, le cosiddette **app Console**. In questo tipo di applicazioni l'input e l'output con l'utente avvengono attraverso una finestra *shell*.

Questo ci permetterà di concentrarci maggiormente sulla progettazione degli algoritmi risolutivi dei problemi che affronteremo tralasciando la complessità dello sviluppo di applicazioni ad interfaccia grafica, che richiede un'approfondita conoscenza della programmazione ad oggetti.

La **shell** (detta anche ***console*** o ***interprete dei comandi*** o ancora ***prompt dei comandi***) è la parte di un sistema operativo che permette agli utenti di interagire con il sistema stesso, impartendo comandi e richiedendo l'avvio di altri programmi. Insieme al kernel costituisce una delle componenti principali di un sistema operativo.

Il suo nome (dall'inglese shell, *guscio*) deriva dal fatto che questa componente viene considerata l'involucro, la parte visibile del sistema ed è dunque definibile come l'interfaccia utente o il programma software che la rende possibile^[11].

Partiamo quindi con il primo e più semplice programma che possiamo scrivere, il classico “*Hello World!*”^[12], italianizzato in “*Salve Mondo!*”, che visualizza semplicemente un saluto nella finestra shell.

Muniamoci di un qualsiasi editor di testo e scriviamo il seguente codice sorgente che andiamo a salvare con il nome **esempio1.cs** (cs è l'estensione utilizzata per i file sorgente C#):

```
using System; /* spazio di nomi utilizzato  
questa è un'altra riga (esempio di commento su più righe) */  
  
class Saluti // classe principale del programma (commento su una riga)  
{  
    static void Main()  
    {  
        Console.WriteLine("Salve Mondo!");  
    }  
}
```

Alcune cose fondamentali da tenere sempre presenti sono:

- **C# è case-sensitive** cioè fa differenza tra maiuscole e minuscole
- in C# le istruzioni sono separate dal **punto e virgola**
- in C# un blocco di codice viene delimitato dalle **parentesi graffe**
- possiamo inserire dei commenti al codice (utili per chiarire che cosa vogliamo fare e descrivere il nostro algoritmo) in due modi:
 - utilizzando una doppia barra (//) quando il commento è limitato ad una riga (il commento va dalla doppia barra a fine riga)
 - utilizzando i caratteri /* e */ che delimitano un commento che si estende su più righe
- un programma C# ha sempre uno ed un solo metodo **Main** (nota che la M è maiuscola^[13]) che è l'unico punto di entrata del programma cioè il punto dal quale inizierà l'esecuzione del programma.

Abbiamo usato il termine **metodo**: senza tirare in ballo troppi dettagli di programmazione ad oggetti potremo definire un metodo come un blocco di codice autonomo individuato da un nome (una procedura o funzione). Un programma è generalmente composto da più metodi che si richiamano l'uno con l'altro. I metodi sono sempre contenuti in classi o in altri metodi.

Analizziamo ora, istruzione per istruzione, il codice precedente.

using System;

questa istruzione specifica lo **spazio di nomi** (*namespace*) utilizzato dal programma. Un namespace raccoglie e organizza in modo gerarchico le **classi** messe a disposizione dalla libreria del .NET Framework o sviluppate dal programmatore (le classi sono i “mattoni” delle applicazioni che forniscono le funzionalità necessarie). Le classi sono raggruppate in base al lavoro che svolgono. Ogni namespace può inoltre contenere altri namespace.

È possibile evitare di mettere questa istruzione premettendo ad ogni invocazione di metodo il nome del namespace (in questo caso **System**). Ogni programma ha bisogno di usare almeno lo spazio di nomi System.

class Salve

un’applicazione è formata da una o più classi (**class**). Una classe^[14] descrive i dati e le funzionalità del programma. In tutti i programmi che scriveremo avremo una sola classe a cui possiamo dare il nome che preferiamo.

static void Main()

questo indica il “nucleo” del programma ovvero il codice che verrà eseguito quando lanceremo la nostra applicazione. Vi è un solo metodo Main (anche se ci sono più classi).

Console.WriteLine("Salve Mondo!");

con questa istruzione viene visualizzato un messaggio sulla finestra della shell (cioè sul prompt dei comandi). L’input e l’output da shell vengono gestiti attraverso i metodi della classe [Console](#).

2.1. La classe Console

La classe Console contiene una serie di metodi che ci permettono di interagire con la finestra a caratteri; alcuni tra i metodi che potrebbero tornare utili sono i seguenti:

- **Console.WriteLine(*messaggio*)** visualizza il messaggio indicato tra parentesi e ritorna a capo
- **Console.Write(*messaggio*)** visualizza il messaggio indicato tra parentesi senza ritornare a capo
- **Console.ReadLine()** attende l'inserimento di una stringa di caratteri da parte dell'utente (notare che tra parentesi tonde non si mette nulla)
- **Console.ReadKey()** attende la pressione di un tasto sulla tastiera. Il carattere corrispondente al tasto viene visualizzato; per non visualizzarlo chiamare il metodo passando il parametro *true* (che è un valore booleano che appunto indica che il tasto non deve essere visualizzato)
- **Console.Clear()** pulisce la finestra
- **Console.Beep()** riproduce un segnale acustico usando l'altoparlante della console. Possono essere passati due parametri per specificare la *frequenza* del segnale acustico (fra 37 e 32767 Hz) e la *durata* in millisecondi. Provare ad esempio con la chiamata **Beep(400, 2000)**;

Inoltre la classe Console possiede anche alcune proprietà per impostare colori di sfondo e primo piano, posizione del cursore e altre caratteristiche. Ad esempio il codice seguente visualizza lo stesso messaggio “Salve Mondo!” al centro di una finestra con sfondo bianco e colore del testo blu:

```
static void Main()
{
    // imposta colore di sfondo e dei caratteri
    Console.BackgroundColor = ConsoleColor.White;
    Console.ForegroundColor = ConsoleColor.Blue;

    Console.CursorVisible = false; // rende invisibile il cursore

    Console.Clear(); // pulisce lo schermo (con i colori impostati)
```

```
// posiziona il cursore al centro
Console.CursorLeft = Console.WindowWidth / 2;
Console.CursorTop = Console.WindowHeight / 2;

Console.WriteLine("Salve Mondo!"); // visualizza il messaggio

// attende la pressione di un tasto (che non viene mostrato)
Console.ReadKey(true);

// ripristina cursore e colori della finestra al default
Console.CursorVisible = true;
Console.ResetColor();
Console.Clear();
}
```


2.2. Compiliamo!

Procediamo ora alla compilazione del nostro programma invocando il compilatore di C# da una finestra di comando:

```
csc esempio1.cs
```

csc sta per **C Sharp Compiler** (compilatore C#). Se non abbiamo commesso errori, verrà generato il codice eseguibile **esempio1.exe**, altrimenti il compilatore riporterà alcune segnalazioni di errore.

Attenzione che perchè il compilatore csc.exe sia raggiungibile da qualsiasi posizione in cui apriamo la shell è necessario che il suo percorso sia inserito nel percorso di ricerca del sistema operativo (variabile di sistema **Path**).

L'eseguibile csc.exe è normalmente posizionato nella cartella **Microsoft.NET\Framework** all'interno della cartella **Windows**. La posizione potrebbe variare per configurazioni particolari del computer. Se sul computer sono installate più versioni del .NET Framework troveremo più eseguibili csc.exe.

Per scoprire tutte le possibili versioni del compilatore possiamo usare ad esempio il seguente comando (riferito ad una versione a 64 bit di Windows):

```
dir %WINDIR%\Microsoft.NET\Framework64\csc.exe /s
```

In aggiunta o alternativa, se abbiamo installato Visual Studio, possiamo accedere ad una shell già configurata dal **Prompt dei comandi per gli sviluppatori per VS 2019** (oppure **Developer Command Prompt for VS 2019**), voce disponibile nel gruppo di programmi Visual Studio 2019.

2.3. Utilizzo di Visual Studio

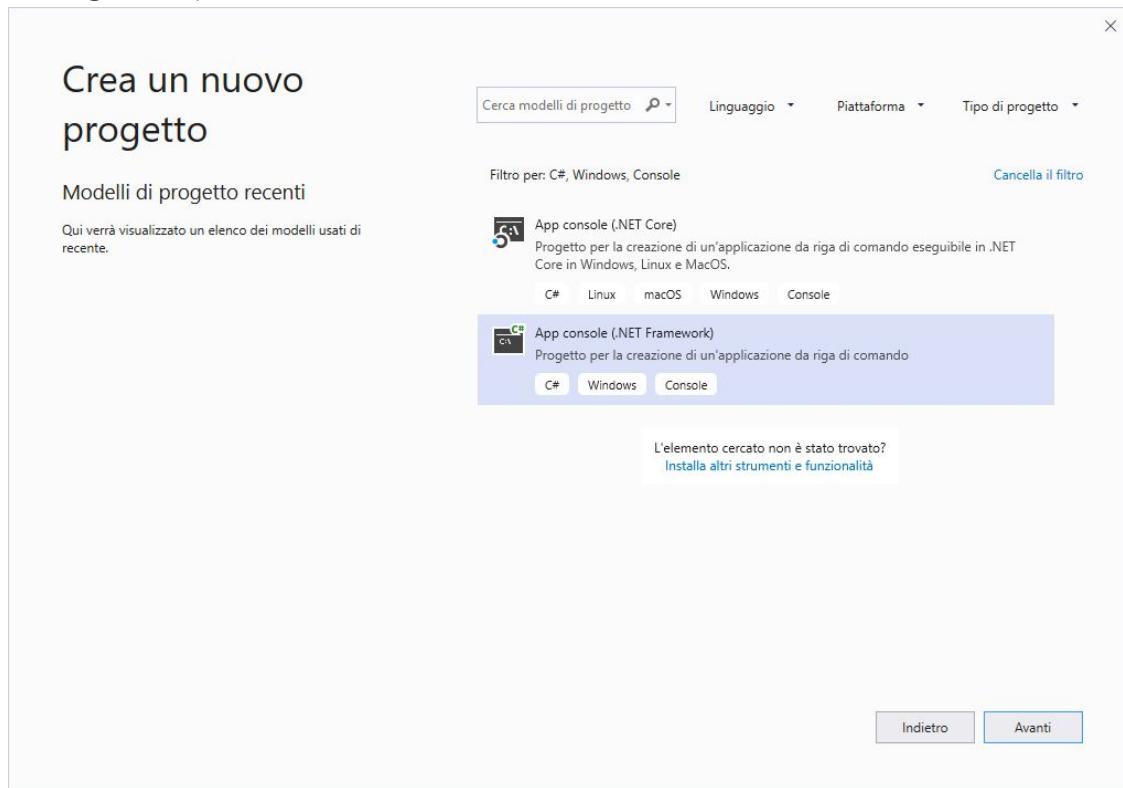
Anziché utilizzare il compilatore a riga di comando, possiamo servirci di un ambiente di sviluppo integrato (**IDE**, *Integrated Development Environment*) come **Visual Studio**.

In questo caso la nostra applicazione verrà sviluppata utilizzando un **progetto**.

Un progetto raccoglie tutto il codice dell'applicazione ma anche molte altre cose, come riferimenti ad altre librerie di classi, icone, immagini, ...

Ad essere più precisi, Visual Studio prevede l'utilizzo delle **soluzioni** che possono raccogliere più progetti assieme. Nel seguito useremo sempre soluzioni contenenti un unico progetto.

Inoltre Visual Studio prevede dei **modelli di applicazione** per mezzo dei quali possiamo costruire automaticamente lo scheletro di una applicazione. Sfruttiamo quindi questa utile funzione utilizzando il modello “**App console (.NET Framework)**” e salviamo il nuovo progetto con il nome proposto dall'IDE cioè “ConsoleApp1” (che sarà anche il nome dell'eseguibile).



L'ambiente creerà il seguente codice:

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

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Come prima, andremo a scrivere il nostro codice all'interno del metodo Main.

Notiamo che rispetto alla struttura scritta prima troviamo alcune novità:

- vi sono più clausole **using** che dichiarano l'utilizzo di altri namespace oltre a System (per il momento non saranno necessari). Notiamo anche che l'editor evoluto di Visual Studio mostra in grigetto i namespace che non sono utilizzati; l'editor ha diverse altre funzionalità che aiutano nella scrittura del codice.
- la classe dell'applicazione viene chiamata da Visual Studio con il nome **Program** ed è contenuta in un namespace con lo stesso nome dato al progetto. Possiamo comunque cambiare sia il nome della classe che quello del namespace.
- il metodo Main ha un parametro **string[] args** che rappresenta gli eventuali parametri passati al programma quando viene richiamato dalla shell (è un *array di stringhe*). Per il momento non ci serviranno.

2.4. La direttiva **using static**

Nell'esempio visto in precedenza abbiamo usato molti metodi e proprietà della classe Console (colori, posizione, ...) e ad ogni chiamata abbiamo dovuto specificare sempre il nome della classe.

Da C# 6.0^[15] c'è la possibilità di omettere il riferimento alla classe utilizzando la direttiva **using static**. Basta inserire nella sezione dedicata agli using la seguente istruzione:

```
using static System.Console;
```

così facendo nel codice possiamo usare direttamente i metodi della classe (ad esempio WriteLine, ReadLine, ...) senza dover anteporre il riferimento Console.

N.B.: questa funzionalità è utilizzabile solo se la classe è statica come lo è classe Console oppure la classe Math (collezione di funzioni matematiche). Capiremo in seguito che cosa si intende con il termine *statico*.

2.5. Compilatori online

Oltre ad utilizzare gli strumenti installati sul proprio PC, il codice C# può essere provato utilizzando un compilatore online. Segnaliamo ad esempio i seguenti due siti utilizzabili liberamente:

- **Ideone** (<https://ideone.com/>) è un compilatore online multi linguaggio che supporta più di 40 linguaggi di programmazione
- **.NET Fiddle** (<https://dotnetfiddle.net/>) è invece dedicato ai linguaggi .NET (C#, F# e VB.NET) con la possibilità di selezionare il compilatore da usare (utile ad esempio per provare funzionalità dei compilatori più recenti).

Citiamo anche Visual Studio Online (VSO), un servizio presentato da Microsoft a maggio 2019 per utilizzare Visual Studio attraverso il browser sulla piattaforma cloud Microsoft Azure (v. <https://visualstudio.microsoft.com/it/vso/>).

3. Variabili, costanti e tipi di dato

Richiameremo in questo capitolo alcuni concetti già incontrati nel capitolo 1 per declinarli nel linguaggio C#.

Come altri linguaggi di programmazione, in C# tutti i dati manipolati dal programma (variabili, costanti) devono avere un tipo.

Si dicono linguaggi a **tipizzazione forte** quei linguaggi, come appunto C#, che obbligano il programmatore a specificare il tipo di ogni valore utilizzato dal programma e il tipo viene assegnato a tempo di compilazione; il linguaggio garantisce che tale valore sia utilizzato in modo coerente.

Variabili e costanti simboliche (cioè i valori costanti a cui è attribuito un identificatore) vanno quindi obbligatoriamente dichiarate (generalmente, per maggiore leggibilità, si mettono all'inizio del programma o della procedura); per le costanti simboliche è obbligatorio specificare un valore contestualmente alla loro dichiarazione.

La variabile o la costante diventa disponibile dall'istruzione subito dopo alla sua dichiarazione.

Anche i valori costanti direttamente utilizzati in un'espressione (si dicono **costanti letterali** o *literal*) hanno un tipo, determinato dal valore stesso; ad esempio i seguenti sono esempi di valori costanti di tipo rispettivamente intero, reale e stringa:

42 23.675433 “Salve Mondo!”

La sintassi per dichiarare una variabile in C# ha la seguente forma:

tipo identificatore;

si indica cioè il tipo della variabile seguito dal suo identificatore; la variabile può essere inizializzata in fase di dichiarazione assegnandogli un valore.

In C# l'assegnazione è indicata con il segno di uguale =.

È possibile dichiarare (e inizializzare) più variabili con una sola dichiarazione separando le variabili con la virgola.

È uso comune in C# utilizzare la *notazione a cammello (camel casing)* per gli identificatori delle variabili: questa notazione prevede che l'identificatore inizi con la lettera minuscola e nel caso sia formato da più parole le successive parole inizino con la lettera maiuscola.

Un'altra notazione per la nomenclatura usata in C# è quella *Pascal (Pascal casing)* nella quale la prima lettera di ciascuna parola dell'identificatore è scritta maiuscola; questa viene normalmente usata nei nomi delle classi e dei metodi.

Ecco alcuni esempi:

```
int a, b;          // dichiarazione di due variabili  
int giornoMese;    // dichiarazione di una variabile  
bool annoBisestile = false; // dichiarazione ed inizializzazione  
int i = 10, j = 20; // dichiarazione ed inizializzazione di 2 variabili
```

Le costanti simboliche sono dichiarate in modo simile alle variabili anteponendo la parola chiave **const**; inoltre *le costanti vanno obbligatoriamente inizializzate*.

const tipo identificatore = valore;

È uso comune usare identificatori tutti maiuscoli per le costanti così da individuarli facilmente nel codice.

Ricordiamo che una costante simbolica non può mai cambiare il suo valore durante l'esecuzione del programma. Una costante è cioè valutata in fase di compilazione e il compilatore sostituisce la costante simbolica, in tutti i posti in cui viene usata, con il rispettivo valore^[16].

Ecco alcuni esempi di dichiarazione di costanti:

```
const int NUMERO_MAX_GIOCATORI = 5;  
const double MIN_TEMP = -50.9, MAX_TEMP = 99.9;
```

Gli identificatori di variabili e costanti (ma anche di altri elementi del linguaggio) non possono iniziare con cifre, contenere spazi o utilizzare parole chiavi del linguaggio^[17]. Possono invece utilizzare caratteri Unicode. C#, infatti, utilizza il set di caratteri **Unicode**^[18] anziché il codice ASCII. Unicode è un formato di carattere a 16 bit progettato per rappresentare

insiemi di caratteri di tutti i linguaggi del mondo.

Le dichiarazioni di variabili e costanti possono essere inserite principalmente in due posizioni del programma:

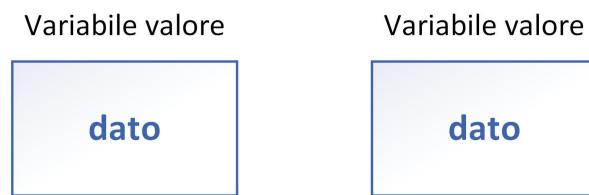
- a livello di classe, cioè al di fuori di ogni metodo
- all'interno di un metodo

nel primo caso è accessibile da tutti i metodi della classe (**visibilità globale**); nel secondo caso è accessibile solo all'interno del metodo (**visibilità locale**).

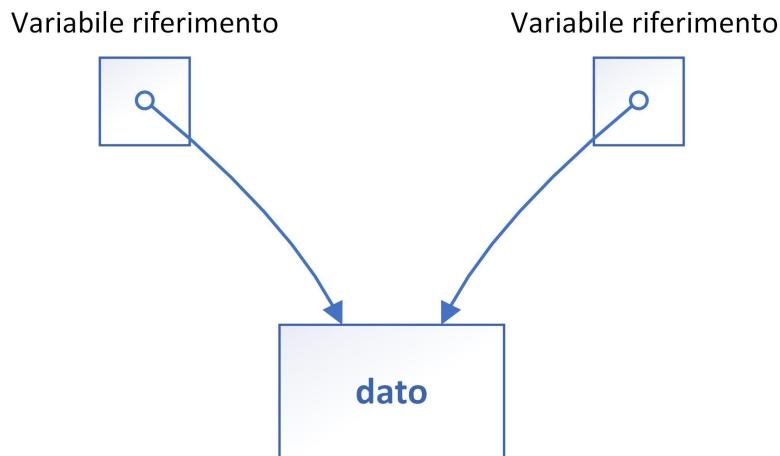
3.1. Tipi di dato valore e tipi riferimento

C# supporta due tipologie di tipi: i **tipi valore** e i **tipi riferimento**.

I **tipi valore** differiscono dai tipi riferimento per il fatto che una variabile dichiarata di un tipo valore contiene direttamente il dato mentre una variabile dichiarata di un tipo riferimento memorizza solo un riferimento al dato (sostanzialmente viene memorizzato, in modo trasparente al programmatore, l'indirizzo della zona di memoria che contiene il dato).



Con i **tipi riferimento** è quindi possibile che due variabili si riferiscano allo stesso oggetto e quindi operando sulla prima variabile si modifica l'oggetto comune a cui si riferisce anche la seconda variabile.



3.2. Tipi predefiniti C#

C# fornisce un insieme completo di *tipi predefiniti* (vengono chiamati anche *tipi incorporati*). Ad ognuno di questi tipi corrisponde una classe definita nel namespace System.

I *tipi predefiniti riferimento* sono i tipi **object** e **string**:

- il tipo **object** è il tipo base dal quale derivano tutti gli altri; questa affermazione sarà più chiara quando affronteremo la programmazione ad oggetti
- il tipo **string** è usato per rappresentare stringhe di caratteri Unicode. Una costante stringa viene specificata delimitandola con doppi apici. Essendo molto importante e versatile verrà trattato in un capitolo a parte.

I *tipi predefiniti valore* comprendono invece tutti i seguenti:

- tipi numerici interi (chiamati anche *integrali*), suddivisi nei tipi interi con segno **sbyte**, **short**, **int** e **long**, e nei tipi interi senza segno **byte**, **ushort**, **uint** e **ulong**
- tipi numerici in virgola mobile che sono **float**, **double** e **decimal**
- tipo **bool**, usato per rappresentare valori booleani; una variabile o costante di tipo bool può assumere solo uno tra i due valori **true** (vero) e **false** (falso)
- tipo **char**, usato per rappresentare singoli caratteri in formato Unicode. Una costante carattere è delimitata da singoli apici. Il tipo char appartiene sempre ai tipi interi perché una costante char memorizza il codice numerico Unicode del carattere rappresentato.

Nelle seguenti tabelle sono riassunti i tipi valore predefiniti.

Tipi interi con segno

Parola chiave C#	Tipo System	Dimensione in bit	Suffisso	Intervallo
sbyte	SByte	8		Da -128 a 127
short	Int16	16		Da -32.768 a 32.767
int	Int32	32		Da -2.147.483.648 a

				2.147.483.647
long	Int64	64	L I	Da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

Tipi interi senza segno

Parola chiave C#	Tipo System	Dimensione in bit	Suffisso	Intervallo
byte	Byte	8		Da 0 a 255
ushort	UInt16	16		Da 0 a 65.536
uint	UInt32	32	U u	Da 0 a 4.294.967.295
ulong	UInt64	64	UL ul	Da 0 a 18.446.744.073.709.551.615

Tipi reali

Parola chiave C#	Tipo System	Dimensione in bit	Suffisso	Intervallo
float	Single	32	F f	Da $\pm 1,5 \times 10^{-45}$ a $\pm 3,4 \times 10^{38}$ (precisione 7 cifre)
double	Double	64	D d	Da $\pm 5,0 \times 10^{-324}$ a $\pm 1,7 \times 10^{308}$ (precisione 15-16 cifre)
decimal	Decimal	128	M m	Da $\pm 1,0 \times 10^{-28}$ a $\pm 7,9 \times 10^{28}$ (precisione 28-29 cifre)

Il tipo **decimal** è utilizzato per rappresentare valori frazionari ed è particolarmente utile per evitare gli errori di arrotondamento causati dalla rappresentazione in virgola mobile.

Come si vede dalla tabella precedente, un valore decimal utilizza infatti ben 128 bit ma, rispetto agli altri tipi a virgola mobile, è caratterizzato da un intervallo minore che lo rende appropriato nei calcoli finanziari e monetari.

Le costanti letterali intere possono essere specificate sia in formato decimale che esadecimale; in quest'ultimo caso il valore va specificato con il prefisso **0x** (o **0X**).

Da C# 7.0^[19] è possibile specificare le costanti intere anche in binario con il prefisso **0b** (o **0B**).

Inoltre, sempre da questa versione di C#, per migliorare la leggibilità, le cifre (decimali, esadecimali o binarie) possono essere separate da caratteri *underscore* _^[20].

Ecco alcuni esempi:

```
int a, b;  
  
a = 2038;          // valore decimale  
a = 0x7F6;        // stesso valore espresso in esadecimale  
a = 0b01111110110; // stesso valore espresso in binario (da C# 7.0)  
  
// uso del separatore _ (da C# 7.0)  
b = 3_897_430;  
a = 0b0111_1111_0110;
```

Le costanti letterali reale possono essere invece specificate anche nella **notazione esponenziale** con il simbolo **E** (o **e**), come nel seguente esempio:

```
double alfa;  
alfa = 1.23E-34;
```

I suffissi indicati nelle precedenti tabelle consentono di forzare l'interpretazione di un valore con il tipo indicato. Infatti, per default, il compilatore deduce il tipo di una costante letterale da come questa viene scritta:

- se il valore contiene un punto decimale o il carattere di esponenziale E, il valore è considerato di tipo **double**
- se il valore è intero, il tipo attribuito è il primo nell'ordine **int**, **uint**, **long** e **ulong** che può contenere il valore specificato.

Alcuni suffissi sono poco utili (U o L) o ridondanti (D) perché, ad esempio, non c'è perdita di informazione nell'assegnare una costante int ad un long.

Invece i suffissi D ed M per i tipi in virgola mobile sono necessari per evitare che le costanti siano interpretate come double quando assegnate a variabili di tipo float o decimal. Ad esempio le istruzioni seguenti non compilerebbero senza usare i prefissi:

```
float f = 3.45f; // non sarebbe compilata senza il suffisso f  
decimal d = 12.34m; // non sarebbe compilata senza il suffisso m
```

I valori minimi e massimi di un tipo si possono ottenere con le costanti **MinValue** e **.MaxValue** disponibili per gran parte dei tipi numerici predefiniti e il tipo char.

Ad esempio **int.MaxValue** restituisce l'estremo superiore dell'intervallo del tipo int (cioè 2.147.483.647).

Tipo booleano (valore logico)

Parola chiave C#	Tipo System	Dimensione in bit	Intervallo
bool	Bool	8	le costanti predefinite false e true

Il tipo **bool** è usato in tutte le condizioni e nelle espressioni che producono un valore boolean.

Potrà sembrare strano che un valore booleano occupi un intero byte quando sarebbe sufficiente un solo bit. La motivazione è da ricercare nell'efficienza, dato che il CLR mappa i tipi di dato di C# nei tipi nativi della macchina, dove un byte è la minima dimensione indirizzabile.

Il linguaggio dispone comunque di operatori e strutture utili per gestire le situazioni in cui occorre gestire singoli bit, come ad esempio la classe **BitArray**^[21].

Tipo carattere

Parola chiave C#	Tipo System	Dimensione in bit	Intervallo
char	Char	16	caratteri Unicode (da 0 a 65.535)

Il tipo **char** è gestito come un tipo intero e può essere usato in espressioni con altre variabili intere, come mostrato nel seguente esempio:

```
int codice = 65;      // codice della lettera 'A'  
char car1 = (char)codice; // conversione da intero a char, car1 contiene 'A'  
  
int salto = 5;  
char car2 = (char)('A' + salto); // car2 contiene 'F', 5 caratteri dopo 'A'
```

Come si è detto una costante letterale carattere deve essere delimitata da singoli apici.

Per i caratteri non visualizzabili (ad esempio un ritorno a capo o una tabulazione) o non direttamente esprimibili (ad esempio caratteri non presenti sulla tastiera) vengono usate le cosiddette **sequenze di escape**. Una **sequenza di escape** è una barra rovesciata \ seguita da un carattere.

Per specificare un carattere (presente o non presente sulla tastiera) si deve usare la sequenza \u (o \x) seguita dal codice del carattere espresso con quattro cifre esadecimale.

Nella seguente tabella sono riassunte le sequenze di escape utilizzabili.

<i>Sequenza di escape</i>	<i>Valore esadecimale</i>	<i>Valore decimale</i>	<i>Significato</i>
\n	0x000A	10	Nuova linea (LF, <i>line feed</i>) ^[22]
\r	0x000D	13	Ritorno carrello (CR, <i>carriage return</i>)
\f	0x000C	12	Avanzamento form (FF, <i>form feed</i>)
\t	0x0009	9	Tabulazione orizzontale
\v	0x000B	11	Tabulazione verticale
\b	0x0008	8	<i>Backspace</i>
\a	0x0007	7	Allerta (l'output di questo carattere provoca generalmente l'emissione di un segnale acustico di allerta)
\0	0x0000	0	Carattere nullo
\'	0x0027	39	Singolo apice
\"	0x0022	34	Doppio apice
\	0x005C	92	Barra rovesciata (<i>backslash</i>)
\u####	0x####		Carattere con il codice esadecimale #### (# è un singola cifra esadecimale). Ad esempio \u00A9 è il carattere ©
\x####			

3.3. Conversione fra tipi

Non è sempre possibile assegnare ad una variabile di un certo tipo una variabile di un altro tipo.

Vi sono tuttavia delle regole di **conversione implicita** che valgono per alcuni tipi di variabili; in generale una conversione implicita fra tipi viene effettuata quando non c'è perdita di informazione: ad esempio è sempre possibile assegnare ad una variabile di tipo *long* una variabile di tipo *int*.

Esiste poi la possibilità di effettuare delle **conversioni esplicite** indicando il tipo di destinazione tra parentesi rotonde (questa operazione si chiama **type casting**, traducibile con *promozione di tipo*); ad esempio:

```
double reale = 12.5;  
int intero;  
intero = (int)reale;
```

nell'esempio precedente il valore assunto dalla variabile *intero* è 12 ovvero il valore di *reale* troncato all'intero. Ovviamente non tutte le conversioni sono ammesse e possono dar luogo ad errori.

Anziché usare il cast è preferibile far ricorso alla classe [Convert](#) (<https://docs.microsoft.com/it-it/dotnet/api/system.convert>) che offre una serie di conversione tra tipi; la utilizzeremo spesso ad esempio per convertire stringhe rappresentanti numeri nei corrispondenti valori numerici.

Alcuni metodi utilizzabili con la classe Convert sono elencati nella seguente tabella.

Metodo	Utilizzo
ToBoolean	Converte un valore specificato in un equivalente valore booleano
ToByte	Converte un valore specificato in un valore integer senza segno a 8 bit
ToChar	Converte un valore specificato in un carattere Unicode
ToDecimal	Converte un valore specificato in un numero Decimal
ToDouble	Converte un valore specificato in un numero in virgola mobile e precisione doppia
ToInt16	Converte un valore specificato in un valore integer con segno a 16 bit

<i>Metodo</i>	<i>Utilizzo</i>
ToInt32	Converte un valore specificato in un valore integer con segno a 32 bit
ToInt64	Converte un valore specificato in un valore integer con segno a 64 bit
ToSByte	Converte un valore specificato in un valore integer con segno a 8 bit
ToSingle	Converte un valore specificato in un numero in virgola mobile e precisione singola
ToString	Converte il valore specificato nell'equivalente rappresentazione in forma String
ToUInt16	Converte un valore specificato in un valore integer senza segno a 16 bit
ToUInt32	Converte un valore specificato in un valore integer senza segno a 32 bit
ToUInt64	Converte un valore specificato in un valore integer senza segno a 64 bit

Vediamo alcuni esempi:

```
double reale = 23.15;
int intero = Convert.ToInt32(reale); // ritorna 23

// ritorna True (ogni valore diverso da 0 viene convertito in true,
// 0 viene convertito in false)
bool logico = Convert.ToBoolean(reale);
string str = Convert.ToString(reale); // ritorna "23.15"
```

Il metodo **Convert.ToString()** ha diverse versioni^[23]; una di queste riceve un intero e restituisce la corrispondente stringa di caratteri. Il metodo può opzionalmente ricevere un secondo parametro che indica la base di destinazione scelta tra 2, 8, 10 o 16.

Ad esempio il seguente codice ottiene le rappresentazioni in binario, ottale ed esadecimale di un numero; inoltre viene usato anche il metodo **Convert.ToChar()** per ottenere il carattere corrispondente al codice 65 inserito nella variabile intera *codiceASCII*:

```
int codiceASCII = 65;
string nBin, nOct, nHex;

// ritorna 'A'
char carattere = Convert.ToChar(codiceASCII);

// converte il valore intero in binario, ottale ed esadecimale
nBin = Convert.ToString(codiceASCII, 2);
nOct = Convert.ToString(codiceASCII, 8);
nHex = Convert.ToString(codiceASCII, 16);
```

3.4. Gestire l'input e output da console

Facciamo una parentesi sui metodi per gestire l'interfaccia a caratteri che abbiamo introdotto nel capitolo 2 con la classe Console. In particolare due metodi sono molto utilizzati, **WriteLine()** e **ReadLine()**.

Come sappiamo il metodo **Console.WriteLine()** consente di visualizzare a video quanto indicato tra parentesi; possiamo specificare una stringa, una variabile (o costante) o concatenare stringhe con altri dati.

Ecco un paio di esempi:

```
Console.WriteLine("Carattere scelto " + carattere + " con codice " + codiceASCII);
Console.WriteLine("Carattere scelto {0} con codice {1}", carattere, codiceASCII);
```

Nel primo esempio l'output viene composto concatenando (con l'operatore `+`) assieme stringhe e variabili, le quali vengono automaticamente convertite in stringhe.

Il secondo esempio ottiene lo stesso effetto usando invece una **stringa di formato**: si tratta di una stringa che contiene al suo interno dei segnaposti (indicati con indici numerici tra parentesi graffe come `{0}`, `{1}`, `{2}`, ... e così via) che verranno sostituiti dai valori delle variabili indicate subito dopo; quindi nell'esempio precedente il segnaposto `{0}` verrà sostituito dalla variabile *carattere* mentre il segnaposto `{1}` dalla seconda variabile specificata, cioè *codiceASCII*.

I segnaposti indicati nella stringa di formato vanno numerati a partire da 0 senza salti.

Ovviamente possiamo ripetere lo stesso segnaposto in più punti della stringa di formato ed è obbligatorio indicare il valore per tutti i segnaposti utilizzati nella stringa di formato.

I segnaposti possono inoltre specificare anche un secondo valore, oltre all'indice, per indicare lo spazio di caratteri utilizzato per la rappresentazione della variabile rappresentata dal segnaposto. Ad esempio:

```
Console.WriteLine("{0,5} {1,2} {2,8} {3}", codiceASCII, nHex, nBin, carattere);
```

produce un output come il seguente:

65 41 1000001 A

in cui *codiceASCII* viene visualizzato su una lunghezza di 5 caratteri, *nHex* su 2 caratteri e *nBin* su 8 caratteri. Notiamo che l'allineamento viene fatto sulla destra; per ottenere l'allineamento a sinistra possiamo utilizzare un valore negativo per cui {0,-5} dedica al primo segnaposto sempre cinque caratteri ma con allineamento a sinistra.

Un ultimo modo per ottenere lo stesso output è quello di utilizzare l'**interpolazione di stringhe**, introdotta da C# 6.0 per semplificare l'utilizzo delle stringhe di formato. Sostanzialmente anziché usare dei segnaposti numerici si indicano i nomi delle variabili direttamente all'interno del segnaposto; per usare questa funzionalità va premesso però un carattere dollaro \$ alla stringa. Così facendo, gli ultimi due esempi possono essere riscritti in questo modo:

```
Console.WriteLine($"Carattere scelto {carattere} con codice {codiceASCII}");  
Console.WriteLine($"{codiceASCII,5} {nHex,2} {nBin,8} {carattere}");
```

Per l'input da console, si usa invece il metodo **Console.ReadLine()** che legge tutto quello che l'utente digita nella finestra di shell fino alla pressione dell'INVIO, restituendolo sotto forma di stringa (nota che il caratteri di INVIO non viene memorizzato nella stringa).

La stringa ricevuta in input andrà quindi convertita, utilizzando uno dei metodi della classe Convert, a seconda di dove inseriremo quanto letto in input.

Ad esempio se ci aspettiamo l'inserimento di un intero, dovremo usare il metodo `Convert.ToInt32()` per convertire la stringa restituita da `Console.ReadLine()`, come mostrato di seguito:

```
int altroIntero = 0;  
Console.WriteLine("Inserisci un intero:");  
altroIntero = Convert.ToInt32(Console.ReadLine());  
Console.WriteLine("Il tuo intero come double vale {0}", Convert.ToDouble(altroIntero));
```

3.5. Conversione controllata con TryParse

Se durante l'esecuzione di un programma la conversione da una stringa al corrispondente valore numerico (fatta utilizzando il cast o la classe Convert) non è possibile il programma genera un errore (si dice che “*solleva un'eccezione*”).

In questi casi l'errore provoca l'interruzione dell'esecuzione del programma. È possibile intercettare queste situazioni attraverso speciali costrutti **try...catch** che vedremo in seguito.

In alternativa per avere maggiore controllo sulla conversione possiamo usare il metodo **TryParse** presente in tutte le classi che rappresentano dati (interi, double, ...); ecco i principali:

- **Int32.TryParse**
- **Double.TryParse**
- **Boolean.TryParse**

Il metodo **TryParse** ha una sintassi più elaborata: riceve la stringa da convertire e il riferimento alla variabile numerica in cui inserire la conversione; restituisce un valore booleano che segnala se la conversione ha avuto esito positivo o meno.

Non viene quindi sollevata nessuna eccezione e possiamo verificare se la conversione è possibile verificando il valore booleano restituito dal metodo. Se la conversione non è ammessa la variabile indicata come destinazione viene impostata ad un valore nullo (0 per i tipi numerici, false per il tipo booleano).

Vediamolo sul seguente esempio:

```
int n;  
bool esito; // conterrà true se la conversione è ammissibile e  
             // false in caso contrario  
string str;  
Console.Write("Inserisci un intero:");  
str = Console.ReadLine();  
esito = Int32.TryParse(str, out n);
```

In **n** ci troveremo il valore convertito da stringa ad intero (e 0 se l'utente ha inserito una stringa che non è convertibile). In **esito** troveremo invece true se la conversione è ammissibile e false altrimenti.

Da notare che è stata usata la parola chiave **out** per indicare che la variabile **n** viene valorizzata dalla chiamata al metodo. Il passaggio di parametri verrà trattato dettagliatamente in seguito.

3.6. Tipizzazione implicita con var

Da C# 3.0^[24] è stata introdotta la parola chiave **var** che consente di dichiarare una variabile senza specificarne il tipo.

Quando si usa **var** è obbligatorio assegnare già in fase di dichiarazione un valore iniziale alla variabile. In questo modo il compilatore è in grado di dedurre il tipo della variabile.

Inoltre **var** si può usare solo nelle dichiarazione di variabili locali ai metodi.

L'utilizzo di **var** facilita il programmatore ma rende meno leggibile il codice; da una dichiarazione “classica” con la specifica del tipo, chi legge il codice ha subito chiaro di quale tipo è la variabile mentre con l’uso di **var** il tipo della variabile potrebbe non essere evidente.

Ecco alcuni esempi:

```
var i = 42;    // c è implicitamente dichiarata di tipo int  
var s = "ciao"; // c è implicitamente dichiarata di tipo string  
var r = 34.5;  // c è implicitamente dichiarata di tipo double
```

Si sottolinea che il tipo della variabile è deciso a tempo di compilazione (ricordiamo che C# è un linguaggio a **tipizzazione forte**) e quindi non è possibile assegnare ad una variabile un valore incompatibile con il tipo che gli è stato attribuito implicitamente.

Nell’esempio di prima, il tentativo di assegnare ad **s** un valore numerico provocherebbe un errore in fase di compilazione.

Altri linguaggi, come ad esempio JavaScript, sono a **tipizzazione debole**, cioè una variabile può cambiare il suo tipo durante l’esecuzione del programma in base ai valori che gli vengono assegnati.

3.7. Il valore speciale **null**

La costante speciale **null** può essere assegnata ad una variabile che appartiene ad un tipo riferimento per indicare che la variabile non si riferisce a nulla.

Viene usata esclusivamente con gli oggetti, cioè con le variabili di un tipo classe, e non può essere usata con i tipi predefiniti valore (ad esempio int, char, ...) che abbiamo visto prima.

Inoltre non può essere usata con la tipizzazione implicita (parola chiave **var**) vista nel paragrafo precedente perché il compilatore non sarebbe in grado di attribuire un tipo definito alla variabile dichiarata con var.

Tra i tipi predefiniti può quindi essere usata ad esempio con il tipo **string** per indicare che una variabile stringa non si riferisce a nessuna stringa, che è diverso da dire che una stringa è vuota (cioè non è costituita da nessun carattere).

3.8. Ambito di visibilità delle variabili

Abbiamo già parlato dell'**ambito di visibilità** di una variabile (in inglese *scope*) come della regione del codice in cui una variabile ha visibilità e può essere utilizzata.

L'ambito di una variabile locale copre l'intero blocco in cui essa è dichiarata e tutti i blocchi eventualmente contenuti in questo.

Ricordiamo che un blocco di codice è delimitato dalla parentesi graffe che possiamo utilizzare anche più volte all'interno del codice.

Ad esempio un codice come il seguente, sebbene poco utile, sarebbe lecito:

```
static void Main(string[] args)
{
    {
        int a = 1; // la variabile intera a è visibile solo in questo blocco
    }

    {
        double a = 2.5; // la variabile double a è visibile solo in questo blocco
    }
}
```

esistono due variabili **a**, una **int** che “vive” nel primo blocco e una, di tipo **double**, nel secondo blocco.

Non è possibile dichiarare una variabile in un blocco annidato con lo stesso nome una variabile dichiarata nel blocco contenitore, anche se la dichiarazione è successiva al blocco annidato^[25]. Ad esempio, modificando il codice precedente nel seguente modo si otterrebbero i due errori in compilazione evidenziati, visto che esiste già una variabile dichiarata nel blocco più esterno:

Si noti bene che, corretto il codice precedente (rimuovendo le dichiarazioni delle variabili locali ai blocchi), la variabile stringa **a** non sarebbe comunque utilizzabile all'interno dei due blocchi perché dichiarata successivamente (anche se il suo ambito di visibilità è l'intero blocco).

4. Operatori

Le espressioni in C# sono costruite utilizzando **operandi** e **operatori**. Gli operatori di un'espressione indicano quale operazione deve essere applicata agli operandi; ad esempio operatori sono quelli abitualmente utilizzati per le quattro operazioni: +, -, * e /.

Ci sono tre tipi di operatori, a seconda del numero di operandi che ricevono:

- operatori **unari**: richiedono un solo operando che può comparire prima (notazione *prefissa*) o dopo (notazione *postfissa*) l'operatore; un esempio è l'operatore aritmetico di negazione $-x$ oppure l'operatore di incremento $x++$
- operatori **binari**: richiedono due operandi che compaiono uno prima e l'altro dopo l'operatore (notazione *infissa*); un esempio sono gli operatori aritmetici come nell'espressione $x+y$
- operatore **ternario**: richiede tre operandi. In C# c'è un solo operatore ternario che è l'*operatore condizionale* (`? :`).

Un'espressione che usa questo operatore ha la forma $c ? y : z$ dove c è un'espressione booleana: se c è verificata l'espressione completa restituisce il valore y altrimenti restituisce il valore z .

L'ordine di valutazione degli operatori in un'espressione è determinato dalla loro precedenza e dalla loro associatività:

- la **precedenza** degli operatori determina l'ordine nel quale gli operatori sono valutati (ad esempio, l'espressione $x+y*z$ è valutata come $x+(y*z)$ poiché l'operatore $*$ ha precedenza rispetto all'operatore $+$)
- l'**associatività** si riferisce invece al modo in cui gli operatori con la stessa precedenza sono valutati: un'espressione può essere valutata da sinistra a destra (*associatività sinistra*) oppure da destra a sinistra (*associatività destra*).

Ad esempio, essendo l'operatore di somma associativo a sinistra, l'espressione $x+y+z$ viene valutata come $(x+y)+z$; invece, essendo l'operatore di assegnazione associativo a destra, l'espressione $x=y=z$ viene valutata come $x=(y=z)$.

L'associatività e la precedenza possono essere modificate (o rese più evidenti) utilizzando le parentesi. Nelle tabelle che seguiranno nei prossimi paragrafi vengono presentati gli operatori disponibili per categoria e *in ordine di precedenza*.

4.1. Operatori primari

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
x.y	Accesso al membro y di x	2 – sinistra
x?.y	Accesso al membro, condizionale	2 – sinistra
f(x)	Chiamata a metodo	2 – sinistra
x[y]	Accesso ad array o indicizzatore	2 – sinistra
x?[y]	Accesso ad array o indicizzatore, condizionale	2 – sinistra
x++	Post-incremento	1 – sinistra
x--	Post-decremento	1 – sinistra
new T(...)	Creazione di oggetto	1 – sinistra
new T(...){ ... }	Creazione di oggetto con inizializzatore	1 – sinistra
new T[...]	Creazione di array	1 – sinistra
typeof(T)	Ottiene l’oggetto System.Type del tipo T	1 – sinistra
checked(x)	Valuta l’espressione x abilitando il controllo dell’overflow per le conversioni e le operazioni aritmetiche sugli interi	1 – sinistra
unchecked(x)	Elimina il controllo sull’overflow a tempo di compilazione (con espressioni con costanti intere)	1 – sinistra
default(T)	Ottiene il valore predefinito del tipo T	1 – sinistra

L’operatore **null condizionale**^[26] ? (chiamato anche “Elvis” per via della somiglianza con il ciuffo di Elvis Presley) è stato introdotto da C# 6.0 ed è utile per accedere a membri di un oggetto (o ad elementi di un array) solo se questo è non nullo.

In caso contrario l’espressione restituisce il valore **null** senza provocare errori (situazione che si verifica se non si usa l’operatore *null condizionale*).

L’operatore **checked** viene invece usato per controllare le espressioni con gli interi. Infatti, se un’espressione con gli interi produce a tempo di esecuzione un overflow questo non viene segnalato e l’applicazione continua a funzionare fornendo risultati errati.

Ad esempio, il seguente codice viene compilato e in esecuzione non produce alcun errore:

```
int a = 2147483647; // massimo valore per int
a = a + 1;          // overflow non segnalato
WriteLine($"a = {a}"); // visualizza a = -2147483648
```

dopo l'incremento di **a**, il suo valore va oltre il limite del tipo **int** ma nessun errore di overflow viene segnalato; invece il valore assunto da **a** diventa uguale al più piccolo valore negativo. Questo per il fatto che i numeri interi con segno vengono rappresentati internamente in *complemento a due*^[27].

Tale comportamento potrebbe non essere desiderabile e portare ad errori difficili da individuare. L'operatore **checked** risolve proprio queste situazioni segnalando l'overflow a tempo di esecuzione; il codice può essere riscritto in questo modo:

```
int a = 2147483647;
a = checked(a + 1); // overflow segnalato a tempo di esecuzione
WriteLine($"n = {a}");
```

L'operatore **unchecked** viene invece usato quando si vuole rimuovere il controllo di overflow sulle espressioni intere quando questo è rilevato a tempo di compilazione.

È il caso ad esempio di un'espressione come la seguente che provocherebbe un errore in compilazione dato che l'espressione è valutabile e provoca un overflow:

```
int b;
b = 2147483647 + 10;
```

scrivendo invece la seguente, l'errore non si verifica e **b** assume il valore -2.147.483.639.

```
int b;
b = unchecked(2147483647 + 10);
```


4.2. Operatori unari

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
+x	+ unario	1 – sinistra
-x	Opposto numerico	1 – sinistra
!x	NOT condizionale (negazione logica applicata a variabili bool)	1 – sinistra
~x	Complemento (negazione bit a bit di x, dove x è di tipo int, uint, long o ulong)	1 – sinistra
++x	Pre-incremento	1 – sinistra
--x	Pre-decremento	1 – sinistra
(T)x	Conversione esplicita di x al tipo T (cast)	1 – sinistra

4.3. Operatori di moltiplicazione

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
$x * y$	Moltiplicazione	2 – sinistra
x / y	Divisione	2 – sinistra
$x \% y$	Resto	2 – sinistra

4.4. Operatori di addizione

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
$x + y$	Addizione, concatenazione di stringhe	2 – sinistra
$x - y$	Sottrazione	2 – sinistra

Notiamo che l'operatore `+` viene usato anche per unire assieme le stringhe e formare un'unica stringa formata dalla *concatenazione* delle due.

C# consente di ridefinire la funzione degli operatori; si parla di **overload degli operatori**.

4.5. Operatori di traslazione (shift)

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
x << y	Traslazione a sinistra	2 – sinistra
x >> y	Traslazione a destra	2 – sinistra

Gli operatori di traslazione spostano il primo operando a sinistra o a destra del numero di bit specificato dal secondo operando (di tipo *int*). Ad esempio il seguente codice effettua la divisione per due e la moltiplicazione per 2 per mezzo dei due operatori:

```
int n = 255;  
n = n >> 1; // ora n vale 127 (shift a destra di 1 bit)  
n = n << 1; // ora n vale 254 (shift a sinistra di 1 bit)
```

4.6. Operatori relazionali e Operatori di tipo

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
x < y	Minore di	2 – sinistra
x > y	Maggiore di	2 – sinistra
x <= y	Minore o uguale a	2 – sinistra
x >= y	Maggiore o uguale a	2 – sinistra
x is T	Controllo tipo: restituisce true se x è di tipo T, false in caso contrario	2 – sinistra
x as T	Cast tra tipi riferimento: restituisce x convertito del tipo T oppure <i>null</i> se x non è un oggetto di tipo T	2 – sinistra

4.7. Operatori di uguaglianza

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
x == y	Uguale	2 – sinistra
x != y	Non uguaglianza	2 – sinistra

4.8. Operatori logici e condizionali

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
x & y	AND logico: effettua l'and bit a bit	2 – sinistra
x ^ y	XOR logico: effettua lo xor bit a bit	2 – sinistra
x y	OR logico: effettua l'or bit a bit	2 – sinistra
x && y	AND condizionale: restituisce y solo se x è true	2 – sinistra
x y	OR condizionale: restituisce y solo se x è false	2 – sinistra
x ?? y	<i>Null-coalescing</i> : restituisce y se x è null, altrimenti restituisce x	2 – sinistra
x ? y : z	Condizionale: restituisce y se x è true e z se x è false	3 – destra

L'operatore **??** è chiamato **operatore di coalesce** o **null-coalescing**^[28] e fornisce un modo semplice e conciso per verificare se un valore è nullo e in tal caso restituire un valore alternativo.

Ad esempio il seguente codice produce in output la stringa “senza nome” dato che la variabile *nome* è nulla:

```
string nome = null;
string s = nome ?? "senza nome";
Console.WriteLine(s); // output: senza nome
```

la seconda istruzione, che usa l'operatore di *coalesce*, è equivalente alla seguente che usa una condizione:

```
string s;
if (nome != null)
    s = nome;
else
    s = "senza nome";
```


4.9. Operatori di assegnazione

<i>Espressione</i>	<i>Descrizione</i>	<i>Numero operandi e Associatività</i>
x = y	Assegnazione semplice	2 – destra
x *= y x /= y x %= y x += y x -= y x <= y x >= y x &= y x = y x ^= y	Assegnazione composta	2 – destra

4.10. Operatori logici e condizionali

Esaminando la tabella precedente, degli operatori logici e condizionali, osserviamo che ci sono due insiemi di operatori che traducono le operazioni logiche come AND e OR.

In particolare abbiamo i seguenti **operatori logici** che possono operare sia su variabili booleane che intere e vengono usati soprattutto per lavorare sui singoli bit di un valore:

- **x & y** effettua l'AND bit a bit
- **x | y** effettua l'OR bit a bit
- **x ^ y** effettua lo XOR bit a bit

Invece gli **operatori condizionali** operano esclusivamente sul tipo bool e sono i seguenti (x e y devono essere variabili, costanti o espressioni che danno come risultato un booleano):

- **x && y** è true se e solo se x e y sono true (and)
- **x || y** è true se e solo se x è true oppure y è true (or)
- **!x** è false se x è true (negazione)

Questi ultimi operatori sono detti anche **operatori di cortocircuito** in quanto l'espressione non viene completamente valutata se è già possibile calcolare il suo valore da una valutazione parziale:

- ad esempio se x è false, sicuramente x && y sarà false e quindi y non sarà valutata
- allo stesso modo se x è true, sicuramente x || y sarà true e quindi y non sarà valutata.

Diversamente le espressioni che usano gli operatori &, | e ^ vengono sempre completamente valutate.

Ad esempio consideriamo la seguente condizione, dove s è una variabile stringa^[29] e s.Length restituisce la lunghezza di s:

```
if (s != null && s.Length > 0)  
.....
```

per la proprietà di cortocircuito, quando s è già in partenza **null** il secondo pezzo **s.Length > 0** non viene valutato perché la condizione è sicuramente falsa (visto che **s != null** è falso).

Invece, nello stesso caso, se si fosse scritto:

```
if (s != null & s.Length > 0)  
.....
```

si sarebbe ottenuto un errore in esecuzione^[30], visto che l'operatore & viene sempre completamente valutato e il secondo pezzo della condizione, **s.Length > 0**, tenta di accedere ad una stringa nulla.

Osserviamo infine che potevamo utilizzare anche l'operatore null condizionale per rendere l'espressione “sicura”:

```
if (s?.Length > 0)  
.....
```

nel caso **s** dovesse essere nulla non viene fatto l'accesso alla proprietà Length e la condizione risulta falsa.

4.11. Operatori di assegnazione semplice e assegnazione composta

Nella tabella precedente abbiamo indicato l'operatore di assegnazione **a = espressione**

Notiamo che l'associatività di questo operatore è a destra il che significa che in un'assegnazione come la precedente, prima viene valutata l'espressione a destra dell'operatore e quindi il valore risultante viene ricopiato nella variabile a sinistra.

Inoltre l'istruzione di assegnazione restituisce l'assegnato; ad esempio potremo scrivere il codice seguente che ha l'effetto di assegnare il valore 42 a tutte e tre le variabili a, b e c.

```
int a, b, c;  
a = b = c = 42;
```

Consideriamo ora la seguente espressione di assegnazione, nella quale la variabile che compare a sinistra, compare anche a destra combinata con un operatore binario (in questo caso la somma):

```
a = a + 5;
```

C# (come tutti i linguaggi con sintassi derivata da quella del C) permette di scrivere espressioni come la precedente nel seguente modo più sintetico:

```
a += 5;
```

l'operatore di somma viene cioè indicato prima dell'operatore di assegnamento: questo tipo di operatore viene detto di **assegnazione composta**. Ripetiamo l'elenco degli operatori (indicato nella tabella) che hanno un corrispondente operatore di assegnazione composta: x *= y, x /= y, x %= y, x += y, x -= y, x <= y, x >= y
x &= y, x |= y, x ^= y.

4.12. Operatori di incremento (++) e decremento (--)

Tra tutti gli operatori unari, due meritano un'attenzione particolare: gli operatori di incremento e decremento. Questi operatori sono derivati dal linguaggio C e risultano molto comodi per rendere più sintetico il codice.

Questi operatori si applicano a variabili di tipo intero e hanno l'effetto di incrementare o decrementare di una unità la variabile a cui sono applicati. L'operatore ++ o -- può inoltre essere scritto subito prima della variabile a cui si applica oppure subito dopo. Questo produce lo stesso effetto se l'istruzione compare da sola ma ha invece un effetto completamente diverso se l'incremento o decremento compare all'interno di una espressione; in particolare:

- se l'operatore è messo prima (si parla di **pre-incremento** e **pre-decremento**), ha luogo l'operazione e quindi il nuovo valore della variabile viene restituito per essere utilizzato nel resto dell'espressione;
- viceversa, se l'operatore compare dopo la variabile (si parla di **post-incremento** e **post-decremento**), il valore di quest'ultima viene restituito per essere utilizzato nell'espressione e subito dopo l'operazione sulla variabile viene eseguita.

Ad esempio le seguenti istruzioni:

```
int conta, indice = 5;  
conta = --indice; // equivale a indice = indice-1; conta = indice;
```

equivalgono a decrementare *indice* e assegnare il nuovo valore a *conta*, portando quindi entrambe le variabili al valore 4.

Invece dopo l'esecuzione del seguente codice:

```
int conta, indice = 5;  
conta = indice++;
```

la variabile ***conta*** vale **5** e la variabile ***indice* 6**. Cioè l'assegnazione della seconda istruzione comporta le seguenti azioni:

- viene valutata l'espressione a destra che restituisce il valore di ***indice*** (cioè **5**)
- viene incrementata la variabile ***indice***

- viene assegnata alla variabile ***conta***, a destra dell'assegnazione, il valore calcolato cioè **5**

Si osservi il seguente esempio, un po' strano e perfettamente inutile, di utilizzo dell'operatore di incremento che potrebbe trarre in inganno:

```
int i = 1;
i = i++;
```

per l'ordine in cui vengono eseguite le azioni, la variabile ***i*** non subisce nessuna variazione e rimane al valore che aveva inizialmente cioè **1**: la parte destra dell'assegnazione restituisce il vecchio valore di ***i*** e poi incrementa ***i***, quindi alla variabile di sinistra (***i*** stessa) viene assegnato il vecchio valore e quindi ***i*** rimane al valore **1**.

Ancora più strano^[31] potrebbe sembrare il seguente esempio (le parentesi sono state inserite per chiarezza):

```
int i = 1;
i = (i++) + i;
```

i assumerà il valore **3** in quanto la valutazione dell'espressione avviene da sinistra a destra e quindi **(i++) + i** dà come risultato **1+2** (il vecchio valore di ***i*** più il nuovo valore assunto da ***i***).

Invertendo gli addendi si ottiene invece un risultato diverso:

```
int i = 1;  
i = i + (i++);
```

i assume il valore **2**; in questo la valutazione dell'espressione considera il vecchio valore di *i* al quale somma lo stesso valore, prima di incrementare *i*.

5. Strutture di controllo

Gli schemi di composizione fondamentale di selezione e iterazione si traducono in C# in modo pressoché identico a come viene fatto in C e in altri linguaggi derivati da esso (come ad esempio C++ e Java).

5.1. Strutture di selezione

Lo schema di selezione si traduce in C# con l'istruzione **if ... else**.

La sintassi, nella forma più completa, di questa istruzione è la seguente:

```
if ( <condizione> )
{
    // istruzioni da eseguire se la condizione è vera
}
else
{
    // istruzioni da eseguire se la condizione è falsa
}
```

L'istruzione è composta da un primo blocco **if (<condizione>)** ... nel quale viene specificata la condizione da verificare.

<**condizione**> indica una qualsiasi espressione che ha come risultato un valore booleano, **true** o **false**; può quindi essere anche semplicemente il nome di una variabile di tipo **bool**.

Se la *condizione* è verificata, vengono eseguite le istruzioni contenute nel primo blocco; nel caso contrario, vengono invece eseguite le istruzioni contenute nel secondo blocco, subito dopo la parola chiave **else**. Quindi l'esecuzione procede in entrambi i casi dall'istruzione che segue.

Il blocco else è opzionale, cioè non è necessario specificarlo se non serve eseguire qualche azione nel caso la condizione non sia verificata.

Se l'intenzione è quella di eseguire le istruzioni solo nel caso la condizione non sia verificata, è molto poco elegante lasciare vuoto il blocco **if** e inserire le istruzioni nel blocco **else**; conviene piuttosto ripensare la condizione (negandola) e inserire le istruzioni nel blocco **if** evitando l'**else**.

Le parentesi graffe che delimitano i blocchi di istruzioni sono necessari solo quando c'è più di una istruzione, altrimenti la loro presenza è opzionale. Ad esempio potremo scrivere il codice:

```
if (n % 2 == 0)
    Console.WriteLine("Il numero è pari");
else
    Console.WriteLine("Il numero è dispari");
```

ma anche:

```
if (n % 2 == 0)
{
    Console.WriteLine("Il numero è pari");
}
else
{
    Console.WriteLine("Il numero è dispari");
}
```

Si noti invece che nel seguente esempio le parentesi del blocco else sono necessarie perché vi sono due istruzioni al suo interno:

```
if (n % 2 == 0)
    Console.WriteLine("Il numero è pari");
else
{
    Console.WriteLine("Il numero è dispari");
    conta++;
}
```

Naturalmente l'istruzione all'interno di un'istruzione condizionale può essere essa stessa un'istruzione condizionale; possiamo quindi costruire catene di condizioni annidate, come nel seguente esempio:

```
if (numeroLati == 3)
    Console.WriteLine("Triangolo");
else
    if (numeroLati == 4)
        Console.WriteLine("Rettangolo");
    else
        if (numeroLati == 5)
            Console.WriteLine("Pentagono");
        else
```

```
Console.WriteLine("Figura non riconosciuta");
```

Quando ci sono catene di condizioni, come la precedente, che cioè coinvolgono la stessa variabile confrontata per uguaglianza con una serie di valori, si può usare, in alternativa alla catena di condizione, un *blocco multidecisionale* attraverso l'istruzione **switch**.

L'esempio precedente può essere riscritto più elegantemente nel seguente modo:

```
switch (numeroLati)
{
    case 3:
        Console.WriteLine("Triangolo");
        break;
    case 4:
        Console.WriteLine("Rettangolo");
        break;
    case 5:
        Console.WriteLine("Pentagono");
        break;
    default:
        Console.WriteLine("Figura non riconosciuta");
        break;
}
```

La sintassi completa dell'istruzione **switch** è indicata di seguito:

```
switch (<espressione>)
{
    case <costante 1>:
        // azioni eseguite quando <espressione> uguaglia <costante 1>
        break;
    case <costante 2>:
        // azioni eseguite quando <espressione> uguaglia <costante 2>
        break;

    // ... eventuali altri blocchi case ...
    default:
        // azioni di default
        break;
}
```

il blocco è intestato con la parola chiave **switch**. All'interno delle parentesi deve essere indicata una variabile o un'espressione di un tipo intero, bool, char o string^[32].

Tra le parentesi graffe che delimitano il blocco vi possono essere più etichette **case**^[33] che specificano i valori costanti con i quali dovrà essere confrontata la variabile o l'espressione. Può inoltre essere presente un'etichetta **default**.

Quando lo switch viene eseguito, vengono effettuate nell'ordine le seguenti azioni:

- la variabile o l'espressione specificata nello **switch** viene valutata
- se una delle costanti specificate in una delle etichette **case** è uguale al valore dell'espressione switch, il controllo è trasferito alle azioni che seguono il case corrispondente
- se nessuna delle costanti specificate nei case corrispondono e se l'etichetta **default** è presente, il controllo passa a questa. Se invece l'etichetta **default** non è presente, lo switch termina e l'esecuzione passa all'istruzione presente subito dopo la parentesi di chiusura del blocco switch.

Si noti che nei blocchi case non è necessario utilizzare le parentesi graffe nel caso di più azioni.

Inoltre è sempre necessario chiudere i blocchi case ed il blocco default (se è presente) con l'istruzione **break**.

Possiamo anche usare la stessa azione per più case. È sufficiente elencare i **case** lasciandoli vuoti tutti tranne l'ultimo; all'interno dell'ultimo specifichiamo le istruzioni da eseguire e inseriamo il break.

Ecco un esempio:

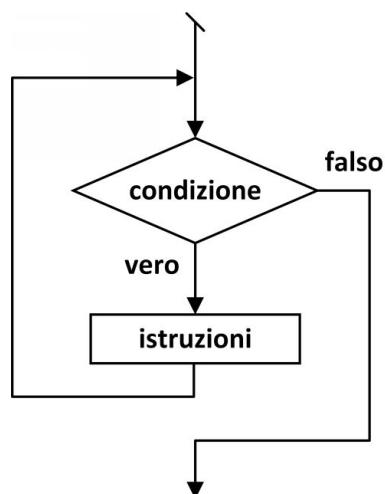
```
switch (n)
{
    case 1:
    case 2:
    case 3:
        Console.WriteLine("n vale uno, due o tre");
        break;
    case 4:
        Console.WriteLine("n vale quattro");
        break;
}
```


5.2. Strutture di ripetizione

Abbiamo visto che esistono tre tipologie di strutture ripetitive: il **ciclo pre-condizionale**, il **ciclo post-condizionale** ed il **ciclo con contatore**.

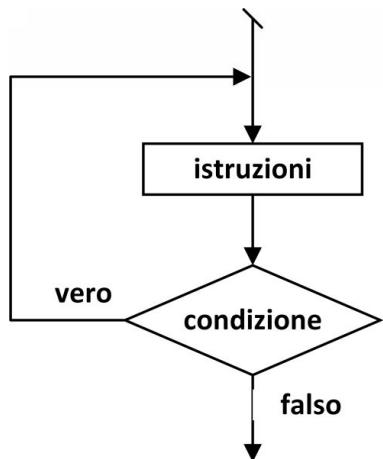
In C# queste strutture si traducono rispettivamente con le istruzioni **while**, **do...while** e **for**.

Nel **ciclo pre-condizionale** (o con controllo in testa) **while** le istruzioni all'interno del ciclo sono ripetute finché la condizione specificata all'inizio rimane vera:



```
while ( <condizione> )
{
    // istruzioni eseguite
    // finchè <condizione> rimane vera
}
```

Nel **ciclo post-condizionale** (o con controllo in coda) **do...while**, come prima, le istruzioni all'interno del ciclo vengono ripetute finché la condizione specificata rimane vera ma, visto che la verifica della condizione è fatta alla fine, le istruzioni sono eseguite senz'altro almeno una volta.



```

do  

{  

    // istruzioni eseguite almeno una volta  

    // e finchè <condizione> rimane vera  

}  

while ( <condizione> );

```

si noti la presenza (necessaria) del punto e virgola in chiusura.

In C# (come del resto in tutti i linguaggi che hanno preso spunto dal C) anche il ciclo post-condizionale termina non appena la condizione diventa falsa. In altri linguaggi come il Pascal il ciclo termina non appena la condizione diventa vera^[34].

5.3. Ciclo con contatore

Il **ciclo con contatore** non è nient'altro che una versione più comoda del ciclo while quando c'è la necessità di ripetere una o più azioni un certo numero di volte.

Il ciclo con contatore si traduce in C# con il ciclo **for** ed ha in generale la seguente struttura:

```
for ( <inizializzatore>; <condizione>; <iteratore> )
{
    // CORPO DEL CICLO: istruzioni eseguite finché <condizione> rimane vera
}
```

Le tre parti dell'intestazione del ciclo **for** vanno separate con un punto e virgola e hanno le seguenti funzioni:

<**inizializzatore**> indica un'istruzione che viene eseguita una sola volta prima di iniziare il ciclo. Generalmente si mette qui un'istruzione che imposta il valore iniziale del contatore; possiamo anche inserire la dichiarazione del contatore in questo punto: il contatore esisterà e avrà visibilità solo all'interno del ciclo, terminato il ciclo il contatore non sarà più accessibile.

Ad ogni iterazione viene verificata <**condizione**> e se è vera vengono eseguite le istruzioni presenti nel corpo del ciclo.

Quindi, alla fine di queste, viene eseguita l'istruzione specificata in <**iteratore**>; normalmente viene inserita un'istruzione di incremento o decremento.

Per quanto detto, il ciclo for corrisponde quindi al seguente codice:

```
<inizializzatore>;
while ( <condizione> )
{
    // istruzioni eseguite finché condizione rimane vera
    <iteratore>;
}
```

Osserviamo, che come per il ciclo while, se la condizione specificata nel ciclo è già inizialmente falsa, le istruzioni all'interno del ciclo non vengono mai eseguite.

Inoltre le tre parti del ciclo for non sono obbligatorie e possono essere lasciate vuote (ma vanno sempre inseriti i punti e virgola).

In particolare non inserire *<condizione>* equivale a una condizione sempre vera quindi il ciclo non avrà termine (ciclo infinito)^[35].

Nel caso più estremo potremo lasciare tutte e tre le parti vuote e scrivere un ciclo infinito come il seguente:

```
for (;;)
    Console.WriteLine("non mi fermo più");
```

Vediamo un esempio completo che calcola la somma dei primi **n** numeri naturali, con **n** richiesto all'utente:

```
int n; // numero inserito dall'utente
int somma = 0; // conterrà la somma dei primi n numeri naturali

Console.Write("Inserisci un numero naturale: ");
n = Convert.ToInt32(Console.ReadLine());

for (int i = 1; i <= n; i++)
    somma = somma + i;
```

notiamo la dichiarazione della variabile **i** nella parte di inizializzazione del ciclo; **i** esiste solo all'interno del ciclo, si comporta cioè come una variabile locale.

Un'ultima annotazione: la prima e ultima parte dell'intestazione di un ciclo for (<inizializzatore> e <iteratore>) possono contenere più istruzioni di assegnazione separate da una virgola. Ad esempio potremo scrivere il seguente ciclo:

```
int i = 0, j = 0; // l'inizializzazione delle variabili è necessaria
for (i = 1, j = 10; i <= j; i++, j--)
    Console.WriteLine(i + " " + j);
```

che produce il seguente output:

```
1 10
2 9
3 8
4 7
5 6
```

il ciclo precedente poteva essere scritto anche dichiarando e inizializzando i e j locali al ciclo:

```
for (int i = 1, j = 10; i <= j; i++, j--)  
    Console.WriteLine(i + " " + j);
```


5.4. Istruzioni break e continue

L'istruzione **break** consente di interrompere un ciclo e si può usare in tutti i cicli.

Ad esempio il seguente ciclo si ferma quando *i* raggiunge il valore 5 e stampa quindi i valori di *i* da 1 a 5:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine(i);
    if (i == 5)
        break;
}
```

Non è molto elegante interrompere un ciclo prima del suo naturale termine (cioè quando la condizione di iterazione diventa falsa). Per scrivere codice strutturato si dovrebbe fare cioè affidamento alla sola condizione del ciclo, che dovrebbe essere l'unica espressione da valutare per decidere se concludere o meno il ciclo. Inserire dei **break** in un ciclo obbliga chi legge il nostro codice a controllare l'interno del ciclo per capire quando il ciclo si può fermare, anziché controllare la sola condizione nell'intestazione del ciclo.

L'istruzione **continue** ha invece un ruolo diverso: essa permette di saltare all'iterazione successiva del ciclo. In corrispondenza di una istruzione **continue** è cioè come se ci fosse un salto all'inizio del ciclo.

Ad esempio il seguente codice salta tutti i numeri pari e produce in output i soli numeri dispari da 1 a 10:

```
for (int i = 1; i <= 10; i++)
{
    if (i % 2 == 0)
        continue;
    Console.WriteLine(i);
}
```


5.5. Controllo delle eccezioni

Come abbiamo accennato nel capitolo 3, C#, come molti altri linguaggi di programmazione, mette a disposizione il costrutto **try...catch** utilizzabile per intercettare tutte quelle situazioni che possono portare il programma in errore provocando il sollevamento di un'eccezione con la conseguente interruzione del programma stesso.

Il costrutto **try...catch** è composto da due parti: una prima parte (**try**) che consente di *provare* ad eseguire un codice che potrebbe provocare qualche eccezione, ed una seconda parte (**catch**) che viene eseguita soltanto se un'eccezione viene *catturata*; questa seconda parte dovrà contenere il codice per gestire l'eccezione:

```
try
{
    // codice sono controllo di eccezione
}
catch
{
    // codice eseguito in caso di eccezione
}
```

Ecco un semplice esempio di conversione controllata dell'input dell'utente in intero:

```
int n;
try
{
    n = Convert.ToInt32(Console.ReadLine());
}
catch
{
    n = 0;
    Console.WriteLine("Hai inserito un intero non valido. Assumo 0");
}
```

solo se l'input dell'utente non è convertibile in intero, ovvero se l'istruzione `Convert.ToInt32(...)` solleva un'eccezione, il blocco catch viene eseguito. In tutti i casi l'esecuzione prosegue dall'istruzione subito successiva alla parentesi di chiusura del catch.

In alcuni casi è necessario eseguire del codice in seguito all'eccezione ma lasciare comunque al compilatore la naturale gestione della stessa. Per

ottenere tale comportamento è sufficiente inserire l'istruzione **throw** all'interno del blocco catch:

```
try
{
    // codice sotto controllo di eccezione
}
catch
{
    // codice eseguito in caso di eccezione
    throw; // risolve l'eccezione
}
```

L'istruzione **throw** è utilizzabile anche al di fuori del blocco catch per sollevare esplicitamente un'eccezione. In tal caso va seguita dal nome del tipo di eccezione che si vuole provocare come mostrato nel seguente esempio:

```
if (n < 0 || n > 10)
    throw new Exception("valore fuori dai limiti");
```

La classe **Exception**^[36] rappresenta una qualsiasi eccezione; vi sono classi più specifiche^[37] per rappresentare eccezioni di una particolare categoria: ad esempio la classe **IndexOutOfRangeException** rappresenta errori di fuori uscita da un range oppure **DivideByZeroException** rappresenta errori di divisione per zero. Ad esempio il codice precedente poteva essere riscritto nel seguente modo, sollevando un'eccezione più mirata:

```
if (n < 0 || n > 10)
    throw new IndexOutOfRangeException("valore fuori dai limiti");
```

Le classi che rappresentano eccezioni possono essere usate anche nel blocco catch per intercettare solo particolari tipi di eccezioni; in tal caso è possibile inserire anche più blocchi catch per intercettare più tipi di eccezioni, come mostrato nel seguente esempio:

```
try
{
    // codice sotto controllo di eccezione
}
catch (IndexOutOfRangeException errore)
{
    // il codice qui dentro viene eseguito solo se si verifica un'eccezione di tipo
    IndexOutOfRangeException
}
catch (DivideByZeroException errore)
```

```
{
    // il codice qui dentro viene eseguito solo se si verifica un'eccezione di tipo
    DivideByZeroException
}
```

oppure più in generale possiamo definire una variabile per accedere alle informazioni dell'eccezione, ad esempio al messaggio di errore ritornato dal compilatore:

```
try
{
    // codice sotto controllo di eccezione
}
catch (Exception errore)
{
    Console.WriteLine("si è verificata la seguente eccezione: " +
                      errore.Message);
}
```

Un altro costrutto che è opportuno utilizzare in certi casi è il blocco **try...finally**.

È anch'esso formato da due blocchi: il blocco **try** in cui va inserito il codice critico e il blocco **finally** che viene eseguito in tutti i casi, sia che il blocco try non abbia generato errori, sia nel caso li abbia provocati. Il blocco finally assicura quindi l'esecuzione del codice anche in presenza di errori; in quest'ultimo caso l'eccezione provoca l'interruzione del programma ma il codice inserito nel finally verrà eseguito certamente. Viene utilizzato ad esempio per operazioni di “pulizia” come la cancellazione di file temporanei o la scrittura di eventi in file di log.

```
try
{
    // codice sotto controllo di eccezione
}
finally
{
    // codice eseguito in ogni caso
}
```

Il blocco finally può essere usato anche assieme al blocco catch. In definitiva potremo usare la seguente struttura **try...catch...finally** per gestire eventuali eccezioni e assicurare l'esecuzione di codice in qualsiasi caso:

```
try
```

```
{  
    // codice sono controllo di eccezione  
}  
catch  
{  
    // codice eseguito in caso di eccezione  
}  
finally  
{  
    // codice eseguito in ogni caso  
}
```


6. Stringhe

Abbiamo già incontrato le stringhe nei paragrafi precedenti. Vediamo più rigorosamente di che cosa si tratta. Le stringhe rappresentano sequenze di caratteri. Mentre una variabile di tipo carattere (**char**) può contenere un solo carattere, una variabile di tipo stringa può contenere una qualsiasi sequenza di caratteri.

Il tipo stringa non appartiene ai tipi fondamentali visti in precedenza in quanto è definito come una classe in grado di contenere più variabili di tipo char. Comunque, per l'utilizzo che ne faremo nei programmi, possiamo tranquillamente trascurarne l'implementazione interna.

In C# le variabili di tipo stringa vengono dichiarate utilizzando come tipo, la parola chiave **string**^[38]. Una costante di tipo **string** va sempre delimitata da due caratteri di doppi apici ("") come mostrato nei seguenti esempi:

```
string nome;           // dichiarazione di una variabile stringa
nome = "Paolo";       // assegnazione con una stringa costante
string cognome = "Rossi"; // dichiarazione ed inizializzazione
```

Come tutte le altre variabili, anche le variabili di tipo stringa vanno sempre inizializzate, o attraverso un'assegnazione con una costante stringa oppure attraverso la lettura in input.

Come abbiamo più volte visto, il metodo **Console.ReadLine()** richiede in input l'immissione di una stringa che è restituita al programma per essere normalmente assegnata ad una variabile; vediamo un esempio:

```
string materia;
Console.Write("Quale materia scolastica preferisci?");
materia = Console.ReadLine(); // inizializzazione mediante input
```

Quando c'è la necessità di inizializzare una stringa possiamo assegnargli la **costante di stringa vuota**, contrassegnata da una coppia di doppi apici:

```
string corso = ""; // questa stringa è inizializzata come stringa vuota
```

Trattandosi di un tipo riferimento, una variabile stringa può assumere anche il valore speciale **null** che indica un riferimento nullo; è come dire che la variabile non contiene nessuna stringa:

```
string scuola = null; // questa stringa non è allocata
```

Una stringa in C# è infine un *oggetto immutabile* cioè il suo contenuto non può essere modificato in nessun modo. Questo non significa che non possiamo assegnare ad una variabile di tipo stringa una nuova stringa ma non possiamo sostituire, eliminare o modificare una parte di una stringa. Come vedremo, tutti i metodi di manipolazione delle stringhe creano un nuovo oggetto assegnando a questo la stringa modificata.

6.1. Concatenazione di stringhe

Possiamo unire assieme due o più stringhe per ottenerne una unica utilizzando l'operatore +, che viene chiamato **operatore di concatenazione**.

Ad esempio possiamo scrivere:

```
string nominativo;  
nominativo = nome + cognome; // concatenazione di due variabili stringa  
nominativo = "mi chiamo " + nominativo;
```

L'operatore di concatenazione è molto potente permettendo di concatenare stringhe con valori numerici senza la necessità di convertire esplicitamente questi ultimi: i valori numerici vengono cioè automaticamente convertiti in stringhe e concatenati.

È importante ricordare che perché la conversione possa avvenire è necessario che sia utilizzata almeno una variabile o una costante stringa:

```
string test = ""; // è importante inizializzare la stringa  
int n = 1;  
test = test + n; // n viene convertita in stringa e concatenata con test
```

in questo modo, la variabile test conterrà la stringa "1".

Questa conversione implicita è molto comoda, ad esempio nell'utilizzo con i metodi di output **Console.WriteLine**:

```
double x1, x2; // variabili per le due soluzioni  
// ... algoritmo per la soluzione di equazioni di 2o grado ...  
Console.WriteLine("Le soluzioni dell'equazione sono " + x1 + " e " + x2);
```

si faccia attenzione che se si fosse scritto:

```
Console.WriteLine(x1 + x2);
```

l'output (errato!) sarebbe stato la somma delle due soluzioni e non le due soluzioni distinte.

In conclusione, l'operatore + ha quindi un duplice utilizzo: quando coinvolge solo variabili numeriche, ne esegue la somma come valore numerico, mentre se coinvolge stringhe e valori numerici, ne esegue la concatenazione restituendo una stringa.

Altro esempio: cosa produce il seguente codice in output? Se avete dei dubbi, provate ad eseguirlo!

```
string sequenza = "";
for (int i = 0; i < 10; i++)
    sequenza = sequenza + i;
Console.WriteLine(sequenza);
```

Cosa otterremmo se sostituiamo l'istruzione all'interno del ciclo con la seguente?

```
sequenza = i + sequenza;
```

6.2. Caratteri di escape e stringhe verbatim

È possibile includere nelle stringhe le sequenze di escape incontrate nel capitolo 3 come ad esempio i caratteri '\n' (nuova riga) e '\t' (tabulazione) o caratteri specificati attraverso il loro codice Unicode.

Il codice:

```
string colonne = "Colonna 1\tColonna 2\tColonna 3";
Console.WriteLine(colonne);
```

produce come output:

Colonna 1 Colonna 2 Colonna 3

invece:

```
string righe = "Riga 1\nRiga 2\nRiga 3";
Console.WriteLine(righe);
```

produce come output:

Riga 1

Riga 2

Riga 3

o ancora:

```
Console.WriteLine("Copyright \u00A9 by \"ACME Corporation\"");
```

produce in output:

Copyright © by "ACME Corporation"

Ricordiamo che per includere una barra rovesciata in una stringa oppure i caratteri di singolo o doppio apice, li dobbiamo far precedere da una barra rovesciata.

Il simbolo @ davanti ad una costante stringa permette di ottenere una stringa **verbatim**^[39] ovvero una stringa nella quale non viene effettuata nessuna interpretazione dei *caratteri di escape*.

Questo è utile ad esempio nelle stringhe che devono contenere percorsi di file: in questo caso una barra rovesciata verrebbe interpretata come carattere

di escape e, per evitarlo, sarebbe necessario utilizzare la doppia barra al posto di ogni singola barra presente nel percorso.

Utilizzando le stringhe *verbatim* questo non è più necessario, per cui le due stringhe seguenti sono equivalenti:

```
string p1 = "C:\\Users\\ric\\";  
string p2 = @"C:\Users\ric\";
```

6.3. Conversione in stringa – metodi **ToString()** e **Convert.ToString(...)**

Tutti i tipi di dato disponibili in C# dispongono del metodo **ToString()** che consente di convertirne il valore in stringa. Ecco un esempio:

```
int anno = DateTime.Today.Year; // restituisce l'anno corrente  
string msg = "Siamo nell'anno " + anno.ToString();  
Console.WriteLine(msg); // output "Siamo nell'anno 2019"
```

in effetti, quando non è utilizzato **ToString()**, la concatenazione tra stringhe e valori effettua implicitamente la chiamata a **ToString()** per tutte le variabili presenti nell'espressione di concatenazione.

Il metodo **ToString()** può essere chiamato su una costante, come nel seguente esempio:

```
string valore;  
valore = 2019.ToString(); // converte la costante numerica 2019 in stringa
```

notiamo che nel precedente esempio la conversione della costante numerica in stringa è necessaria, non potendo assegnare direttamente alla variabile stringa di sinistra, la costante numerica di destra.

In alternativa al metodo **ToString()** applicato ad una variabile, per convertire un valore in stringa possiamo usare la classe **Convert** vista nel capitolo 3 ed in particolare il suo metodo **ToString()**.

Questo metodo riceve una variabile di tipo qualsiasi e ne restituisce la conversione in stringa.

Con variabili di tipo intero può essere utilizzato anche con un secondo parametro che specifica la base numerica utilizzata per la conversione in stringa: la base può essere 2, 8, 10 o 16. Senza scrivere un algoritmo per la conversione dalla base 10 alle basi 2, 8 o 16, abbiamo quindi a disposizione un metodo che fornisce la conversione.

Vediamo un programma completo (in modalità console) in cui sfruttiamo questa versione del metodo per generare la tabella dei codici ASCII presentando per ogni carattere con codici tra 13 e 255, i rispettivi codici in decimale, esadecimale e binario.

```

static void Main(string[] args)
{
    for (int codice = 13; codice <= 255; codice++)
    {
        string nHex = Convert.ToString(codice, 16);
        string nBin = Convert.ToString(codice, 2);
        Console.WriteLine("{0,5} {1,2} {2,8} {3}",
                          codice, nHex, nBin, Convert.ToChar(codice));
    }
}

```

Si osservi che nell'istruzione `Console.WriteLine` è stato utilizzato il metodo **Convert.ToChar(codice)** per ottenere il carattere corrispondente al codice ASCII.

Inoltre la stringa da visualizzare è stata prodotta con una *stringa di formato*, di cui abbiamo già parlato nel capitolo 3: abbiamo cioè specificato un primo parametro di tipo stringa che contiene testo fisso e alcuni segnaposto indicati con `{0}`, `{1}`, ...; ogni segnaposto viene sostituito da una delle variabili indicate come parametri successivi nella `WriteLine`.

Nell'esempio, il segnaposto indicato con `{0}` verrà sostituito dal valore di `codice`, `{1}` dal valore di `nHex`, `{2}` da `nBin` ed infine `{3}` dal valore di **Convert.ToChar(codice)**.

Ovviamente possiamo ripetere lo stesso segnaposto in più punti della stringa di formato.

Come ulteriore esempio, l'output dato dalla seguente istruzione:

```
Console.WriteLine("Mi chiamo " + nome + " " + cognome);
```

potrebbe essere sostituito da quest'altra, in cui è stata utilizzata una stringa di formato:

```
Console.WriteLine("Mi chiamo {0} {1}", nome, cognome);
```

Notiamo poi che nel primo esempio, oltre all'indice del segnaposto è stato indicato un secondo valore, ad esempio per il primo `{0,5}`: il 5 messo come secondo valore indica lo spazio in caratteri utilizzato per la rappresentazione della variabile rappresentata dal segnaposto; nel nostro esempio n verrà visualizzato su cinque caratteri allineato a destra. Per ottenere l'allineamento a sinistra possiamo utilizzare un valore negativo per

cui `{0,-5}` dedica al primo segnaposto sempre cinque caratteri ma con allineamento a sinistra.

Nel capitolo 3 avevamo anche introdotto l'**interpolazione di stringhe** (disponibile da C# 6.0), un ulteriore modo di formattare le stringhe al posto della stringa di formato e della concatenazione: basta anteporre un simbolo \$ alla stringa e inserire direttamente variabili ed espressioni nella stringa tra parentesi graffe.

Con l'interpolazione di stringhe, l'istruzione inserita all'interno del ciclo può essere riscritta in questo modo:

```
Console.WriteLine($"'{codice,5} {nHex,2} {nBin,8}  
{Convert.ToString(codice)}'");
```

6.4. Accesso ai singoli caratteri di una stringa

Come si è detto una stringa è una sequenza di char.

È possibile recuperare uno specifico carattere di una stringa utilizzando il nome della variabile stringa seguito da una coppia di parentesi quadre all'interno delle quali specifichiamo l'indice del carattere che vogliamo ottenere^[40], dove l'indice può variare da 0 al numero di caratteri della stringa meno uno.

Ad esempio, se la variabile **nome** contiene la stringa “Paolo”

- **nome[0]** restituirà il carattere 'P' (di tipo char)
- **nome[1]** restituirà il carattere 'a'
- **nome[2]** restituirà il carattere 'o'
- **nome[3]** restituirà il carattere 'l'
- **nome[4]** restituirà il carattere 'o'

Attenzione che non è possibile modificare il carattere con la notazione a parentesi quadre. Cioè non possiamo scrivere un'istruzione come la seguente:

nome[2] = 'A'; // VIETATO

Naturalmente, il contenuto di una variabile stringa non è predeterminato nel programma ma può variare durante l'esecuzione del programma stesso o essere richiesto in input all'utente.

Per ottenere la lunghezza (intera) di una variabile stringa possiamo utilizzare la proprietà **Length** ovvero scrivere l'identificatore della variabile stringa seguito da un punto e dalla parola chiave **Length**, come mostrato nel seguente esempio:

```
string nome;  
int n;  
Console.WriteLine("Inserisci il tuo nome:");  
nome = Console.ReadLine();  
n = nome.Length; // n contiene la lunghezza della stringa nome  
Console.WriteLine("La lunghezza del tuo nome è {0}", n);
```

come ulteriore esempio, vediamo un'applicazione in un ciclo (seguito del programma precedente):

```
string nomer = "";
```

```
// ciclo dal primo all'ultimo carattere di nome
for (int i = 0; i <= nome.Length - 1; i++)
{
    // aggiungo il carattere in testa alla nuova stringa nomer
    nomer = nome[i] + nomer;
}
Console.WriteLine("Nome rovesciato: {0}", nomer);
```

notiamo che l'indice del ciclo varia da 0 alla lunghezza meno uno: ciò ricordando quanto detto all'inizio, ossia che i caratteri di una stringa sono indicizzati a partire da 0 e quindi ci dobbiamo fermare quando l'indice arriva a **nome.Length – 1**.

Era ovviamente equivalente scrivere come condizione del ciclo, **i < nome.Length** anziché **i <= nome.Length-1**.

6.5. Metodi per manipolare stringhe

Il namespace **System** contiene una serie di metodi per manipolare stringhe. Questi metodi sono in gran parte delle funzioni che lavorano su una stringa e restituiscono una nuova stringa prodotta dall'elaborazione della stringa di partenza.

Come si diceva all'inizio del capitolo, le stringhe sono oggetti immutabili e quindi non è possibile modificare direttamente una variabile stringa ma, sfruttando i metodi che vedremo tra poco, possiamo ottenere una "elaborazione" della stringa e riassegnarla nuovamente alla nostra variabile^[41].

Elenchiamo quindi alcuni dei metodi più utili^[42]: per ogni metodo indicheremo quanti e di quale tipo sono i parametri ricevuti e all'inizio specificheremo il tipo del valore restituito dal metodo.

6.6. Estrarre una porzione di stringa (substring)

`string s.Substring(int indicePartenza)`

`string s.Substring(int indicePartenza, int numeroCaratteri)`

Il metodo **SubString** estraе da una stringa *s* una sua parte. Può essere usato in due modi:

- con un solo parametro, *indicePartenza*, che è l'indice del carattere da cui partire
- con due parametri: il secondo, *numeroCaratteri*, è il numero di caratteri da estrarre; se non è specificato estraе fino alla fine della stringa.

Il metodo restituisce una nuova stringa con la porzione estratta (la stringa originale non viene modificata):

- se *indicePartenza* è uguale a **Length** o *indiceCarattere* è **0**, **SubString** estraе una stringa nulla,
- se *indicePartenza+numeroCaratteri* è maggiore di **Length** si verifica un errore in esecuzione.

Esempi:

```
string nome = "Paolo Rossi";
string s1, s2;
s1 = nome.Substring(0, 5); // s1 conterrà "Paolo"
s2 = nome.Substring(6);   // s2 conterrà "Rossi"
```

6.7. Confronto tra stringhe

```
bool sA.Equals(string sB)
int sA.CompareTo(string sB)
int string.Compare(string sA, string sB)
int string.Compare(string sA, string sB, bool ignoraCase)
```

Il modo più semplice per confrontare due stringhe consiste nell'utilizzare gli operatori `==` e `!=`, che eseguono un confronto binario tra le stringhe distinguendo quindi tra maiuscole e minuscole^[43].

Un altro modo è utilizzare il metodo **Equals** che restituisce *true* se le due stringhe sono uguali, sempre distinguendo tra maiuscole e minuscole.

Non è invece possibile utilizzare gli operatori `>`, `<`, `>=`, `<=` per capire se una stringa precede o segue un'altra. Per confrontare due stringhe abbiamo a disposizione due metodi, **CompareTo** e **Compare**. Per il confronto viene utilizzato il valore Unicode dei caratteri che compongono le stringhe considerando che le lettere minuscole hanno un valore inferiore alle lettere maiuscole.

Il metodo **CompareTo** confronta una stringa *sA* con una stringa *sB* e restituisce un valore intero con il seguente significato:

- minore di 0 se *sA* è minore di *sB* (*sA* precede *sB*)
- uguale a 0 se *sA* è uguale a *sB*
- maggiore di 0 se *sA* è maggiore di *sB* (*sA* segue *sB*)

Il metodo distingue tra maiuscole e minuscole ed applica le impostazioni internazionali della macchina^[44].

Esempi:

```
string s3 = "ABC", s4 = "abc";
if (s3.CompareTo(s4) > 0) // "ABC" segue "abc"
    Console.WriteLine(s3 + " segue " + s4);
```

Il metodo **Compare**, a differenza del precedente, viene richiamato specificando la parola chiave **string** e riceve le due stringhe *sA* e *sB* da confrontare ed eventualmente un terzo parametro, *ignoraCase*, che indica

se il confronto deve ignorare o meno il case delle stringhe (il valore di default è *false* cioè nel confronto non viene ignorato il case delle stringhe). Il valore restituito segue le stesse regole del metodo **CompareTo**.

Esempi:

```
string s5 = "rossi", s6 = "Rossi";
int confronto;
confronto = string.Compare(s5, s6);    // confronto vale -1
confronto = string.Compare(s5, s6, true); // confronto vale 0
```

6.8. Ricerca all'interno di una stringa

```
bool s.Contains(string sottostringa)
int s.IndexOf(string sottostringa)
int s.IndexOf(string sottostringa, int indicePartenza)
int s.IndexOf(string sottostringa, int indicePartenza,
               int numCaratteri)
int s.IndexOf(char carattere)
int s.IndexOf(char carattere, int indicePartenza)
int s.IndexOf(char carattere, int indicePartenza,
               int numCaratteri)
int s.LastIndexOf(string sottostringa)
int s.LastIndexOf(string sottostringa, int indicePartenza)
int s.LastIndexOf(string sottostringa, int indicePartenza,
                  int numCaratteri)
int s.LastIndexOf(char carattere)
int s.LastIndexOf(char carattere, int indicePartenza)
int s.LastIndexOf(char carattere, int indicePartenza,
                  int numCaratteri)
```

Il metodo **Contains** restituisce *true* se la stringa *s* contiene il parametro specificato.

I metodi **IndexOf** ricercano una *sottostringa* (o un *carattere*) all'interno di una stringa *s* e restituiscono -1 se la stringa (o il carattere) ricercata non viene trovata oppure l'indice della prima posizione in cui è presente (ricordare che gli indici partono sempre da 0).

In modo analogo **LastIndexOf** inizia la ricerca di una sottostringa o di un carattere a partire dalla fine di *s* restituendo l'indice della posizione in cui viene trovata la sottostringa (o il carattere) o -1 se non viene trovata.

Opzionalmente **IndexOf** e **LastIndexOf** possono ricevere l'indice del carattere dal quale deve iniziare la ricerca e il numero di caratteri da esaminare: IndexOf procede sempre in avanti da sinistra a destra mentre LastIndexOf a ritroso da destra a sinistra.

Tutti questi metodi distinguono tra maiuscole e minuscole ma è possibile impostare un ulteriore parametro per specificare il tipo di confronto (ad esempio specificando il valore 1 non viene fatta distinzione tra maiuscole e minuscole; si veda **StringComparison**, <https://docs.microsoft.com/it-it/dotnet/api/system.stringcomparison>).

Esempi:

```
nome = "Massimo Rossi";
bool esito = nome.Contains("Ro"); // true
int p;
p = nome.IndexOf("ss"); // 2 (prima occorrenza)
p = nome.LastIndexOf("ss"); // 10 (ultima occorrenza)
```

6.9. Modifica della combinazione di maiuscole e minuscole

string s.ToUpper()

string s.ToLower()

Per modificare le lettere in una stringa in maiuscole o minuscole possiamo utilizzare i metodi **ToUpper()** o **ToLower()** che non ricevono parametri e restituiscono la stringa a cui vengono applicati, rispettivamente, con le lettere tutte maiuscole e con le lettere tutte minuscole.

Esempi:

```
nome = "Paolo Rossi";
nome = nome.ToUpper(); // ora nome contiene "PAOLO ROSSI"
nome = "Paolo Rossi";
nome = nome.ToLower(); // ora nome contiene "paolo rossi"
```

6.10. Inserire, rimuovere e sostituire sottostringhe

```
string s.Insert(int posizione, string sottostringa)
string s.Remove(int indicePartenza)
string s.Remove(int indicePartenza, int numeroCaratteri)
string s.Replace(string vecchia, string nuova)
```

Il metodo **Insert** restituisce la stringa *s* a cui viene applicato, con inserita *sottostringa* a partire dall'indice *posizione*. Se *posizione* è maggiore di *Length* si verifica un errore in esecuzione.

Il metodo **Remove** restituisce *s* rimuovendo il numero di caratteri *numeroCaratteri* a partire dall'indice *indicePartenza*. Se *indicePartenza+numeroCaratteri* è maggiore di *Length*, si verifica un errore in esecuzione; se il parametro opzionale *numeroCaratteri* non è specificato, rimuove da *indiceCarattere* fino alla fine.

Il metodo **Replace** restituisce *s* nella quale tutte le occorrenze della sottostringa *vecchia* vengono sostituite con la sottostringa *nuova*.

Esempi:

```
nome = "Rossi";
string s7;
s7 = nome.Insert(0, "Paolo ");    // s7 contiene "Paolo Rossi"
nome = "Paolo Rossi";
string s8;
s8 = nome.Remove(0, 6);          // s8 contiene "Rossi"
s8 = nome.Remove(nome.IndexOf(' ')); // s8 contiene "Paolo"
nome = "Paolo Rossi";
string s9;
s9 = nome.Replace("Rossi", "Reds"); // s9 contiene "Paolo Reds"
```

6.11. “Ripulire” una stringa dai caratteri iniziali e finali

string s.Trim()
string s.Trim(**char** c1, **char** c2, **char** c3, ...)

Il metodo **Trim** restituisce la stringa **s** togliendo i caratteri iniziali e finali: è disponibile in due forme, la prima, senza parametri, toglie solo i caratteri spazio, la seconda toglie tutti i caratteri elencati come parametri.

Esempi:

```
nome = " Paolo Rossi ";
string s10;
s10 = nome.Trim(); // s10 contiene "Paolo Rossi"

nome = "*+*Paolo*Rossi**-+";
s10 = nome.Trim('+', '*', '-'); // s10 contiene "Paolo*Rossi"
```

Esistono anche i metodi analoghi **TrimStart** e **TrimEnd**, rispettivamente, per togliere solo i caratteri all'inizio e i caratteri alla fine della stringa.

6.12. “Imbottire” una stringa

```
string s.PadLeft(int lunghezzaTotale)
string s.PadLeft(int lunghezzaTotale, char carattere)
string s.PadRight(int lunghezzaTotale)
string s.PadRight(int lunghezzaTotale, char carattere)
```

Il metodo **PadLeft** restituisce, a partire dalla stringa *s*, una nuova stringa della lunghezza specificata in *lunghezzaTotale* in cui l'inizio della stringa viene riempito con spazi o con il carattere *carattere*, se specificato. Se il numero di caratteri specificato in *lunghezzaTotale* è inferiore alla lunghezza della stringa, il metodo restituisce la stringa originale.

Il metodo **PadRight** opera in modo analogo aggiungendo spazi o il carattere specificato alla fine della stringa.

Esempi:

```
nome = "Paolo Rossi";
string s11;
s11 = nome.PadRight(15); // s11 contiene "Paolo Rossi   "

string binario = "10";
binario = binario.PadLeft(8, '0'); // binario contiene "00000010"
```

6.13. Formattare una stringa

`string string.Format(string formato, argomento1, argomento2, ...)`

Il metodo **String.Format** lavora in modo simile alla stringa di formato utilizzabile con il metodo `Console.WriteLine` vista in precedenza.

Cioè restituisce la stringa passata come primo parametro sostituendo al posto dei segnaposti i valori delle variabili elencate nei parametri successivi.

Ricordiamo che i segnaposti sono indicati tra parentesi graffe e devono avere numeri progressivi che partono da 0, ad esempio `{0}`, `{1}`, `{2}`, ...

Inoltre possono specificare:

- separato da una virgola, il numero di caratteri da usare per la rappresentazione della variabile come in `{0, 10}` dove la variabile viene allineata a destra in uno spazio di complessivamente dieci caratteri; se il numero è negativo l'allineamento è a sinistra. Se il numero di caratteri non è sufficiente la variabile non viene troncata.
- separato da due punti, il formato da usare per presentare la stringa come in `{0: 0.00}` dove per rappresentare la variabile (supposto di tipo numerico) vengono usati due decimali.

Si può consultare il seguente link per altri formati:

<https://docs.microsoft.com/it-it/dotnet/standard/base-types/custom-numeric-format-strings>

Esempi:

```
string sf;
sf = string.Format("L'area vale {0} e il perimetro vale {1}", a, p);
```

in `sf` troveremo la stringa “**L'area vale...**” dove al posto dei segnaposti `{0}` e `{1}` vengono inseriti i valori delle variabili `a` e `p`, rispettivamente.

```
string sp;
sp = string.Format("Pi greco vale {0,10:0.0000}", Math.PI);
```

`sp` conterrà la stringa “**Pi greco vale 3,1416**”

7. Array

Le variabili appartenenti ai tipi primitivi presenti in C# sono in grado di mantenere un unico valore in un certo istante. Anche le variabili di tipo string mantengono un'unica stringa di caratteri.

Ci sono altri tipi di dati che invece hanno la possibilità di mantenere più valori referenziati con la stessa variabile.

Tra questi il tipo più comune in tutti i linguaggi di programmazione è l'**array**.

Ricordiamo che un array è una variabile strutturata in grado di memorizzare un insieme di valori **tutti dello stesso tipo**, individuati da un unico identificatore.

Un array può avere una o più dimensioni:

- l'array ad una dimensione viene chiamato **vettore**; possiamo immaginarlo come una lista di celle, ognuna contenente un valore;
- l'array a due dimensioni viene invece chiamato **matrice**; possiamo pensare ad esempio ad una scacchiera o ad una tabella;
- possiamo utilizzare anche array a più di due dimensioni anche se il loro utilizzo non è frequente.

Un array è memorizzato in memoria in posizioni consecutive senza salti tra i vari elementi e i singoli elementi occupano lo stesso spazio. Questo sistema ha il grosso vantaggio che l'accesso ai singoli elementi può avvenire direttamente: se ad esempio vogliamo accedere al terzo elemento, ci basterà sommare alla locazione di inizio del vettore il doppio dello spazio occupato da un singolo elemento.

L'identificatore di un array lo individua come un tutt'uno. Abbiamo bisogno di un meccanismo che ci permetta di accedere ai singoli valori contenuti dell'array. Questo è fornito da un **indice** numerico (o dagli indici, nel caso di array a più di una dimensione) che ci permette di individuare univocamente ogni singola “cella” dell'array.

L'indice è un intero che inizia da zero e prosegue fino alla lunghezza del vettore meno uno, un po' come per accedere ai singoli caratteri di una stringa. Per le matrici avremo bisogno di due indici per individuare un elemento e analogamente per un array a **n** dimensioni avremo bisogno di **n** indici interi per specificare ogni dimensione.

Nella seguente figura sono rappresentati due array: un array ad una dimensione (*vettore*) di 6 elementi e individuato dalla variabile v , ed un array a due dimensioni (*matrice*) di 16 elementi (4×4) individuato dalla variabile m :

		<i>indice colonna</i>				
		0	1	2	3	
<i>indice riga</i>	0	$m[0,0]$	$m[0,1]$	$m[0,2]$	$m[0,3]$	
	1	$m[1,0]$	$m[1,1]$	$m[1,2]$	$m[1,3]$	
	2	$m[2,0]$	$m[2,1]$	$m[2,2]$	$m[2,3]$	
	3	$m[3,0]$	$m[3,1]$	$m[3,2]$	$m[3,3]$	
		0	1	2	3	
		4	5	<i>indice</i>		
v	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]

7.1. Dichiaraone di array

In C# una variabile di tipo vettore si dichiara specificando il ***tipo base*** (cioè il tipo che avranno gli elementi) seguita da una coppia di parentesi quadre aperta e chiusa e dall'identificatore; la dichiarazione ha la seguente sintassi:

***tipo_base*[] identificatore_array;**

se si tratta di una matrice, invece, la dichiarazione assume la seguente forma:

***tipo_base*[,] identificatore_array;**

la virgola denota cioè che vi sono due dimensioni e quindi avremo bisogno di specificare due indici per accedere ad ogni singolo valore.

In generale, per dichiarare un array a ***n*** dimensioni dovremo indicare ***n-1*** virgole.

Ogni elemento di un array è trattata come una normale variabile di ***tipo base*** alla stessa stregua di una variabile semplice. Possiamo pensare ad un array anche come ad una collezione di variabili semplici tutte dello stesso tipo.

In C# le variabili di tipo array sono ***variabili riferimento*** a differenza delle variabili di tipo semplice (che sono create direttamente durante la dichiarazione) come int, bool, char, double che sono ***tipi valore***. La differenza tra i tipi riferimento e i tipi valore è stata spiegata nel capitolo 3 e va sempre tenuta presente, specialmente quando si lavora con gli array.

In particolare vedremo che l'assegnazione tra due variabili array assume un significato completamente diverso rispetto all'assegnazione tra due variabili di tipo valore.

Vediamo alcuni esempi di dichiarazione di variabili array:

```
double[] voti;      // dichiarazione di un vettore di double
string[] studenti; // dichiarazione di un vettore di stringhe
int[] voti1q, voti2q; // dichiarazione di due vettori di interi
```

7.2. Allocazione e inizializzazione di array

La dichiarazione di un array non occupa (*alloca*) lo spazio in memoria per gli elementi dell'array; la dichiarazione definisce solo una variabile che conterrà il riferimento all'array. Dobbiamo utilizzare l'operatore **new** per creare effettivamente l'array specificandone le dimensioni, con la seguenti sintassi:

identificatore_array = new int[dimensioni_array];

Se si tratta di un array a più di una dimensione, dovremo specificare la lunghezza delle singole dimensioni.

Esempi:

```
voti = new double[30]; // creazione del vettore voti con 30 elementi  
studenti = new string[30]; // creazione del vettore studenti
```

```
Console.WriteLine("Quanti voti vuoi inserire ?");  
// la dimensione è letta in input  
int n = Convert.ToInt32(Console.ReadLine());  
voti1q = new int[n]; // creazione del vettore voti1q di n elementi
```

Quindi con l'operatore **new** possiamo allocare l'array dichiarato precedentemente ripetendo il tipo base e specificando la dimensione tra parentesi quadre.

Quando un array viene allocato, i suoi elementi assumono tutti il valore di default, che dipende dal tipo base del vettore: **0** per i tipi numerici, '\0' (carattere nullo) per il tipo **char**, false per il tipo **bool** e **null** per i tipi riferimento (come ad esempio il tipo **string**).

Una volta creato l'array non è più possibile modificarne la dimensione. Se ad esempio riutilizzassimo l'operatore **new** su un array già allocato, verrebbe creato un nuovo array e il vecchio contenuto dell'array non sarebbe più raggiungibile e andrebbe perso.

È anche possibile dichiarare e allocare un array nella stessa istruzione, come nel seguente esempio:

```
string[] nazioni = new string[100];
```

inoltre nella medesima istruzione che crea un array è possibile *inizializzarlo* con determinati valori che verranno specificati tra parentesi graffe; questo è utile ad esempio anche per definire sequenze di costanti.

Nei seguenti esempi sono mostrate varie modalità di dichiarazione/allocazione/inizializzazione:

```
string[] numeri = new string[] { "uno", "due", "tre", "quattro" };
```

```
// sintassi estesa (con specificazione del numero di elementi)
string[] settimana = new string[7] { "lunedì", "martedì", "mercoledì",
    "giovedì", "venerdì", "sabato", "domenica" };
```

```
// sintassi estesa (la inizializzazione può essere fatta anche dopo)
int[] a;
a = new int[] { 10, 20, 30 };
```

```
// sintassi compatta, utilizzabile solo durante la dichiarazione
int[] b = { 10, 20, 30 };
```

Si noti che nell'ultima dichiarazione è stato omesso l'operatore **new** che è implicito. Questa sintassi compatta è utilizzabile solo in fase di dichiarazione dell'array.

Allo stesso modo, anche gli array a più dimensioni possono essere creati ed eventualmente inizializzati; ecco un paio di esempi con matrici:

```
int[,] m;      // dichiarazione di una matrice
m = new int[5, 3]; // creazione della matrice con 5 righe e 3 colonne
// creazione di una matrice 3 x 3 con contestuale inizializzazione
char[,] tris = new char[3, 3]
{
    { 'X', 'O', 'O' },
    { 'O', 'X', 'O' },
    { 'O', 'O', 'X' }
};
```

nel secondo esempio notiamo le parentesi graffe più interne per specificare i valori per gli elementi delle tre righe della matrice che chiaramente devono avere lo stesso numero di elementi. Avendo inizializzato la matrice, possiamo anche omettere le dimensioni 3, 3.

Inoltre, come per gli array ad una dimensione, se l'array viene inizializzato in fase di dichiarazione, possiamo omettere il **new**; il precedente esempio può essere riscritto più sinteticamente così:

```
char[,] tris =
```

```
{  
    {'X', 'O', 'O'},  
    {'O', 'X', 'O'},  
    {'O', 'O', 'X'}  
};
```

7.3. Array jagged

Con le dichiarazione degli ultimi due esempi abbiamo ottenuto due **matrici rettangolari** ovvero nelle quali il numero di elementi per ogni riga è sempre lo stesso.

È possibile dichiarare anche **matrici irregolari** che vengono chiamate **array jagged** (*jagged* in inglese significa “stracciato”, “seghettato”).

Sostanzialmente viene dichiarato un vettore di vettori cioè un array ad una dimensione che ha come tipo base un array ad una dimensione: in questo modo ogni elemento dell'array è a sua volta un array con un numero variabile di elementi.

Per capire bene come usare gli **array jagged**, immaginiamo di voler scrivere un programma per memorizzare le temperature registrate in città di alcune regioni; per ogni regione l'utente decide quante città considerare.

Ci serve quindi un array come il seguente:

```
int[][] temperature;
```

a questo punto gestiamo l'input del numero di regioni e andiamo ad allocare l'*array jagged* come mostrato di seguito:

```
int numeroRegioni;
```

```
Console.WriteLine("Inserire il numero di regioni: ");
if (Int32.TryParse(Console.ReadLine(), out numeroRegioni))
{
    temperature = new int[numeroRegioni][];
    temperature[0] = new int[numeroCittà];
```

il programma si conclude con il ciclo che itera per il numero di regioni scelto e per ogni iterazione alloca il singolo elemento dell'*array jagged* al numero di città scelte per quella regione e quindi richiede all'utente le singole temperature:

```
for (int i = 0; i < numeroRegioni; i++)
{
    int numeroCittà;
    Console.WriteLine($"Numero di città della regione n. {i+1}: ");
    if (Int32.TryParse(Console.ReadLine(), out numeroCittà))
    {
        temperature[i] = new int[numeroCittà];
        for (int j = 0; j < numeroCittà; j++)
        {
            Console.WriteLine($"Inserire la temperatura della città n. {j+1}: ");
            if (Int32.TryParse(Console.ReadLine(), out numeroTemperatura))
                temperature[i][j] = numeroTemperatura;
        }
    }
}
```

```
        Console.WriteLine($"Temperatura della {j+1}^ città: ");
        Int32.TryParse(Console.ReadLine(), out temperature[i][j]);
    }
}
}
```

7.4. Accesso ai singoli elementi e numero di elementi

Come già ricordato, per accedere ai singoli elementi di un array si utilizza la notazione indicizzata con le parentesi quadre, in modo simile all'accesso dei singoli caratteri in una stringa.

Come per le stringhe, per sapere qual è il numero di elementi di un array possiamo usare la proprietà **Length**, come mostrato nel seguente esempio:

```
for (int i = 0; i < voti.Length; i++)
{
    Console.WriteLine($"Inserisci il {i+1}º voto = ");
    Double.TryParse(Console.ReadLine(), out voti[i]);
}
```

Se l'array ha più di una dimensione, la proprietà **Length** restituisce il numero complessivo di elementi.

Possiamo inoltre utilizzare la proprietà **Rank** (*rango*) che fornisce il numero di dimensioni dell'array e il metodo **GetLength** che restituisce la lunghezza in ogni dimensione specificata tra parentesi con un intero da 0 a Rank-1:

```
int[,] m = new int[5, 3];
Console.WriteLine("Numero di elementi = {0}", m.Length); // stampa 15
Console.WriteLine("Numero di dimensioni = {0}", m.Rank); // stampa 2
Console.WriteLine("Numero di righe = {0}", m.GetLength(0)); // stampa 5
Console.WriteLine("Numero di colonne = {0}", m.GetLength(1)); // stampa 3
```

Per accedere ai singoli elementi della matrice utilizziamo sempre la notazione a parentesi quadre specificando stavolta due indici, rispettivamente per la riga e la colonna. Per poter “visitare” l'intera matrice abbiamo bisogno evidentemente di due cicli annidati, il primo che scorre sulle righe ed il secondo che, per ogni riga, scorre sulle colonne:

```
for (int r = 0; r < m.GetLength(0); r++) // ciclo sulle righe
    // per ogni riga, ciclo sulle colonne
    for (int c = 0; c < m.GetLength(1); c++)
        m[r, c] = r * m.GetLength(1) + c + 1;
```

il precedente esempio riempie la matrice **m** con i numeri da 1 a 15.

Il codice usa il metodo **GetLength** per ottenere le dimensioni della matrice; in questo modo lo abbiamo reso più generico e utilizzabile per riempire

qualsiasi matrice con interi progressivi da 1 in poi.

7.5. Il ciclo foreach

In diverse situazioni c'è la necessità di leggere il contenuto di un intero array ad esempio per visualizzare tutti gli elementi o calcolare qualcosa.

In questi casi anziché utilizzare un ciclo con contatore, possiamo utilizzare l'istruzione **foreach**.

Il ciclo **foreach** consente di scorrere dall'inizio alla fine senza far uso di un contatore. Il ciclo foreach non si può applicare in qualsiasi situazione ma è molto comodo con array, stringhe e altre strutture dati, quando si voglia scorrere interamente la struttura.

La sintassi dell'istruzione è la seguente:

```
foreach (tipo_base elemento in collezione)
    corpo_del_ciclo
```

il ciclo funziona in questo modo:

- la variabile **elemento** è dichiarata all'interno del *foreach* e si comporta come una variabile locale al ciclo; deve essere dello stesso tipo del tipo base della **collezione** specificata (che può essere un array, una stringa o altre strutture dati^[45])
- la struttura collezione viene percorsa totalmente dal primo all'ultimo elemento e per ogni iterazione **elemento** viene valorizzato con l'elemento corrente

Quando si usa l'istruzione *foreach* si deve tener presente che non è possibile modificare gli elementi della collezione e la variabile locale **elemento** non è modificabile ma solo leggibile (il tentativo di modificarla genera un errore di compilazione). Sebbene non generi errori in compilazione è da evitare anche la modifica della composizione della collezione (ad esempio togliendo elementi); tali modifiche possono provocare errori in esecuzione.

Va da sé che il ciclo *foreach* si presta in tutte quelle situazioni in cui si vogliono leggere tutti gli elementi della struttura e non ci sia la necessità di modificarne il contenuto.

È comunque possibile usare l'istruzione **break** all'interno del ciclo per interromperne l'esecuzione o l'istruzione **continue** per passare all'iterazione successiva.

Ecco un esempio, riferito all'array voti usato in precedenza, che calcola la media dei voti:

```
double somma = 0;  
foreach (int voto in voti)  
    somma += voto;  
Console.WriteLine($"La media dei voti vale {somma / voti.Length}");
```

notiamo che, come per le altre strutture di controllo, le parentesi graffe non sono necessarie se è presente una sola istruzione.

Come si diceva, possiamo usare un ciclo *foreach* anche per scorrere tutti i caratteri di una stringa; nel seguente esempio vengono contate le vocali della stringa inserita dall'utente:

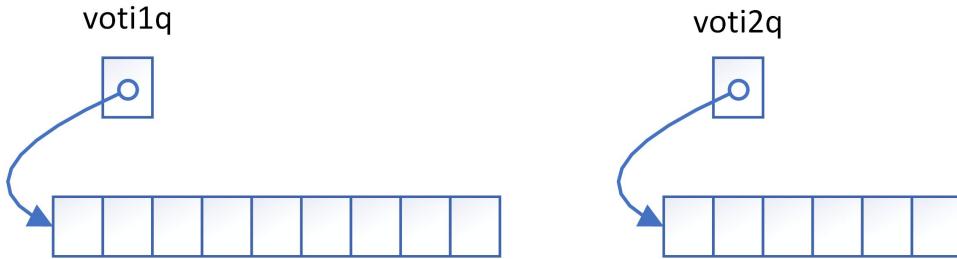
```
Console.Write("Digita una stringa: ");  
string nome = Console.ReadLine();  
int conta = 0;  
foreach (char carattere in nome.ToLower())  
    if ("aioue".Contains(carattere.ToString()))  
        conta++;  
Console.WriteLine($"'{nome}' contiene {conta} vocali");
```

7.6. Assegnazione e confronto tra array

L'unica operazione ammessa per gli array è l'assegnazione.

Come si è detto gli array sono un tipo riferimento e quindi quello che viene copiato con un'assegnazione è solo il riferimento non tutto l'array.

Ad esempio supponiamo di avere dichiarato e allocato i due vettori *voti1q* e *voti2q*: la situazione in memoria potrebbe essere la seguente:

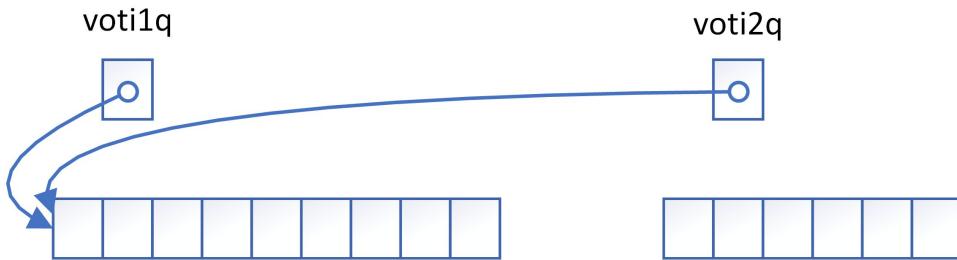


l'assegnazione seguente:

```
voti2q = voti1q;
```

non ricopia tutti gli elementi del vettore *voti1q* nel vettore *voti2q* ma ricopia solo il valore del riferimento contenuto in *voti1q* in *voti2q*, per cui *voti2q* non farà più riferimento al vettore di destra ma a quello di sinistra. L'accesso ad un elemento *voti2q[i]* farà perciò riferimento ad un elemento di *voti1q*.

La seguente figura esemplifica la situazione:



Una domanda sorge spontanea: cosa succede al vettore di destra, inizialmente individuato da *voti2q*? In effetti, dopo l'assegnazione precedente, il contenuto di questo vettore non è più raggiungibile, non esistendo più una variabile che permetta di capire dove si trova in memoria.

La memoria da esso occupata viene automaticamente liberata da un processo automatico chiamato *garbage collector* (letteralmente “spazzino” o “raccoglitore della spazzatura”).

Questa è una caratteristica di linguaggi come appunto C# ma anche Java, Python e altri.

In altri linguaggi, invece, come Pascal e C/C++ la responsabilità di liberare la memoria non più referenziabile è del programmatore che dovrebbe quindi liberare (con opportune istruzioni) esplicitamente la memoria occupata da *voti2q* prima di procedere all’assegnazione che abbiamo esaminato.

È possibile confrontare anche due variabili di tipo array ma vengono confrontati sempre i riferimenti non i valori contenuti negli array. Per confrontare i valori necessariamente dobbiamo considerare i singoli elementi.

7.7. Metodi sugli array: la classe **Array**

Le diverse dichiarazioni e inizializzazioni di array che abbiamo visto all'inizio del capitolo sottendono l'utilizzo, da parte del CLR, della classe **Array** del namespace **System**.

Una variabile di tipo array è un oggetto di un classe derivata dalla classe **Array**.

Di conseguenza ad una variabile di tipo array possiamo applicare i metodi previsti dalla classe **Array**.

Vediamone alcuni che possono tornare utili nei programmi.

7.8. Ricerca di elementi

```
int Array.IndexOf(vettore, valore);
int Array.IndexOf(vettore, valore, int indicePartenza);
int Array.IndexOf(vettore, valore,
                  int indicePartenza, int numElementi);
int Array.LastIndexOf(vettore, valore);
int Array.LastIndexOf(vettore, valore, int indicePartenza);
int Array.LastIndexOf(vettore, valore,
                      int indicePartenza, int numElementi);
int Array.BinarySearch(vettore, valore);
int Array.BinarySearch(vettore, valore, int indicePartenza);
int Array.BinarySearch(vettore, valore,
                      int indicePartenza, int numElementi);
```

I tre metodi **IndexOf**, **LastIndexOf** e **BinarySearch** cercano all'interno dell'array ad una dimensione *vettore*, indicato come primo parametro, il *valore* indicato come secondo parametro e ne restituiscono la posizione^[46].

In particolare il metodo **IndexOf** restituisce la posizione della prima occorrenza del valore cercato mentre il metodo **LastIndexOf** l'ultima occorrenza; il metodo **IndexOf** effettua la ricerca da sinistra a destra mentre il metodo **LastIndexOf** da destra a sinistra.

Il metodo **BinarySearch** restituisce sempre la prima posizione del valore cercato ma si può applicare solo se il vettore è ordinato (in caso contrario il risultato restituito potrebbe non essere corretto); se sappiamo che il vettore è ordinato o lo abbiamo in precedenza ordinato, questo metodo risulta nettamente più efficiente rispetto ai metodi **IndexOf** e **LastIndexOf** che usano una ricerca sequenziale all'interno del vettore.

I tre metodi possono inoltre ricevere altri due parametri per indicare l'indice dal quale deve partire la ricerca e il numero di elementi da esaminare; se quest'ultimo non viene specificato la ricerca prosegue fino alla fine del vettore (o all'inizio per il metodo **LastIndexOf**).

Se l'elemento cercato non viene trovato i metodi restituiscono un valore minore di zero.

Vediamo un semplice esempio che ricerca la prima e ultima posizione di un nome all'interno di un elenco:

```
string[] nomi = { "Paolo", "Alice", "Mario", "Carlo", "Mario" };
string valore = "Mario";
int posizione;
posizione = Array.IndexOf(nomi, valore);
Console.WriteLine($"posizione = {posizione}"); // visualizza 2
posizione = Array.LastIndexOf(nomi, valore);
Console.WriteLine($"posizione = {posizione}"); // visualizza 4
```

7.9. Minimo, massimo e somma degli elementi

tipo_base vettore.Min()
tipo_base vettore.Max()
tipo_base vettore.Sum()
double vettore.Average()

I metodi precedenti non appartengono alla classe Array ma si possono applicare ad array ad una dimensione perché rappresentano delle sequenze di valori^[47].

I metodi **Min** e **Max** restituiscono rispettivamente il minimo e massimo del vettore e si possono applicare sia a vettori numerici che a vettori di stringhe: in quest'ultimo caso restituiscono la stringa minore e maggiore, nell'ordine alfabetico.

I metodi **Sum** e **Average** possono essere applicati solo a vettori numerici (interi o reali) e restituiscono la somma e la media degli elementi.

Vediamo un esempio su un vettore che contiene delle temperature:

```
int[] temperature = { 12, 28, 10, -4, 8, 5, 15, 7 };
Console.WriteLine("Temperatura minima: " + temperature.Min()); // -4
Console.WriteLine("Temperatura massima: " + temperature.Max()); // 28
Console.WriteLine("Somma: " + temperature.Sum()); // 81
Console.WriteLine("Media: " + temperature.Average()); // 10,125
```


7.10. Clonazione e copia di elementi

```
Object variableArray.Clone()  
vettore.CopyTo(vettoreDestinazione, int indicePartenza)  
Array.Copy(vettoreSorgente, vettoreDestinazione,  
          int elementiDaCopiare )  
Array.Copy(vettoreSorgente, int indiceSorgente,vettoreDestinazione,  
          int indiceDestinazione, int elementiDestinazione)
```

Il metodo **Clone** consente di clonare un qualsiasi array ovvero di creare un nuovo array che è una copia dell'array a cui viene applicato; non è quindi necessario allocare il nuovo array, ci pensa il metodo a crearlo e a riempirlo copiando tutti gli elementi dall'array sorgente.

Il metodo restituisce un oggetto generico (variabile di tipo **Object**) ed è quindi necessario fare un cast al tipo dell'array.

Vediamo un esempio clonando il vettore delle temperature:

```
int[] copiaTemperature;  
copiaTemperature = (int[])temperature.Clone(); // N.B.: è necessario fare il cast perchè Clone  
restituisce un oggetto generico
```

```
Console.WriteLine("Copia di temperature:");  
foreach (int temp in copiaTemperature)  
    Console.WriteLine(temp);
```

Il metodo **Clone** si può ovviamente usare su array di qualsiasi dimensione; ad esempio potremo clonare la matrice *tris* utilizzata in precedenza:

```
char[,] copiaTris;  
copiaTris = (char[,])tris.Clone(); // la clonazione si può fare anche per array a più dimensioni
```

I metodi **CopyTo** e **Array.Copy** servono invece per copiare elementi da un array ad un altro già esistente; si applicano solo ad array ad una dimensione.

Il metodo **CopyTo** in particolare viene chiamato su una variabile array e ne copia tutti i valori in un array di destinazione (precedentemente allocato) a partire da una determinata posizione.

Ad esempio il seguente codice copia tutte le *temperature* nell'array *copia1* a partire dalla posizione di indice 2; in questo esempio l'array di destinazione dovrà quindi essere creato con due elementi in più rispetto all'array sorgente:

```
int[] copia1 = new int[temperature.Length+2];
temperature.CopyTo(copia1, 2);

foreach (int temp in copia1)
    Console.WriteLine(temp + " ");
// output prodotto dal ciclo: 0, 0, 12, 28, 10, -4, 8, 5, 15, 7,
```

Il metodo **Array.Copy** fa una cosa simile ma offre maggiore versatilità.

Nella prima forma, riceve due array (si parla sempre di array ad una dimensione) e copia il numero di elementi specificato dal primo array, a partire dal primo elemento, e lo incolla nel secondo array, sempre a partire dal primo elemento.

Nell'esempio seguente vengono copiati gli elementi del vettore *temperature* a partire dal primo fino al terzultimo nel vettore *copia2*:

```
int[] copia2 = new int[temperature.Length - 3];

// Array.Copy( arraySorgente, arrayDestinazione , elementiDaCopiare )
Array.Copy(temperature, copia2, temperature.Length - 3);
foreach (int temp in copia2)
    Console.WriteLine(temp + ", ");
// output prodotto dal ciclo: 12, 28, 10, -4, 8,
```

Nella seconda forma, più completa, permette di specificare da quali indici del primo e secondo array deve partire la copia e quanti elementi devono essere copiati.

Ad esempio per copiare 3 elementi del vettore *temperature*, a partire dal terzo, nel vettore *copia3* a partire dalla sua seconda posizione, potremo scrivere il seguente codice:

```
int[] copia3 = new int[temperature.Length];

// Array.Copy( arraySorgente, indiceSorgente, arrayDestinazione, indiceDestinazione,
// elementiDestinazione)
Array.Copy(temperature, 2, copia3, 1, 3);

foreach (int temp in copia3)
```

```
Console.WriteLine(temp + ", ");
// output prodotto dal ciclo: 0, 10, -4, 8, 0, 0, 0, 0,
```


7.11. Metodi sulle stringhe che usano array

La classe **String** dispone di alcuni metodi che hanno come valore restituito o tra i loro parametri, oggetti di tipo array. Vediamone qualcuno.

char[] s.ToCharArray()

char[] s.ToCharArray(int indicePartenza, int numCaratteri)

il metodo **ToCharArray** crea un array di char contenente tutti i caratteri della stringa *s*; se vengono specificati anche i due parametri interi l’array prodotto sarà creato con *numCaratteri* caratteri della stringa *s* a partire dall’indice *indicePartenza*.

Esempi:

```
string nome = "Alan Turing";
char[] caratteri;
caratteri = nome.toCharArray();
// contenuto array: {'A', 'l', 'a', 'n', ' ', 'T', 'u', 'r', 'i', 'n', 'g'}
```

```
caratteri = nome.toCharArray(nome.IndexOf(" ") + 1, 3);
// contenuto array: {'T', 'u', 'r'}
```

Esistono poi i due metodi **IndexOfAny** e **LastIndexOfAny**, che funzionano come i metodi *IndexOf* e *LastIndexOf* (incontrati nel capitolo sulle stringhe) ma cercando nella stringa uno tra i caratteri presenti in un array di char passato come parametro:

int s.IndexOfAny(char[] caratteri)

int s.IndexOfAny(char[] caratteri, int indicePartenza)

**int s.IndexOfAny(char[] caratteri, int indicePartenza,
 int numCaratteri)**

int s.LastIndexOfAny(char[] caratteri)

int s.LastIndexOfAny(char[] caratteri, int indicePartenza)

**int s.LastIndexOfAny(char[] caratteri, int indicePartenza,
 int numCaratteri)**

Infine, vediamo altri due metodi che possono tornare utili:

- il metodo **Split** consente di spezzare una stringa in parti considerando come separatori un elenco di caratteri o stringhe contenuto in un vettore
- il metodo **Join** effettua l'operazione opposta unendo più stringhe contenute in un vettore restituendo una stringa con tutti gli elementi separati da un separatore specificato

Il metodo **Split** può essere usato nei seguenti modi:

```
string[] s.Split()  
string[] s.Split(char separatore)  
string[] s.Split(char[] separatori)  
string[] s.Split(char[] separatori,  
                  StringSplitOptions.RemoveEmptyEntries)  
string[] s.Split(char[] separatori, int numeroMaxStringhe)  
string[] s.Split(char[] separatori, int numeroMaxStringhe,  
                  StringSplitOptions.RemoveEmptyEntries)  
string[] s.Split(string[] separatori,  
                  StringSplitOptions.RemoveEmptyEntries)  
string[] s.Split(string[] separatori, int numeroMaxStringhe,  
                  StringSplitOptions.RemoveEmptyEntries)
```

il metodo crea e ritorna un array di stringhe con tutte le parti in cui la stringa **s** può essere suddivisa utilizzando i separatori specificati come primo parametro; l'array dei separatori che può essere sia di caratteri che di stringhe.

Se l'array dei separatori è vuoto o *null*, viene considerato come separatore lo spazio.

Vediamo un esempio con un array di char come separatori:

```
string frase = "Uno, due e tre. Quattro e cinque.";  
char[] separatori = { ',', ' ', '.' };  
string[] pezzi;  
pezzi = frase.Split(separatori);  
// contenuto array:  
// { "Uno", "", "", "due", "e", "tre", "", "Quattro", "e", "cinque", "" }
```

dall'esempio notiamo che quando vengono trovati più separatori adiacenti vengono inserite delle stringhe vuote; lo stesso dicasì quando i separatori sono all'inizio o alla fine della stringa.

Si può specificare il parametro **StringSplitOptions.RemoveEmptyEntries** per escludere le stringhe vuote.

Nell'array prodotto non figureranno comunque mai i separatori.

Il metodo può essere chiamato anche senza parametri e in tal caso verrà usato lo spazio come separatore; oppure può essere chiamato con un solo parametro di tipo char; le seguenti chiamate sono quindi valide (e comode):

```
pezzi = frase.Split(); // utilizza come separatore lo spazio  
pezzi = frase.Split('!'); // utilizza come separatore '!'
```

Inoltre possiamo specificare il numero massimo di stringhe **numeroMaxStringhe** che devono essere estratte; consideriamo il seguente esempio:

```
frase = "ab-cd-efg";
pezzi = frase.Split(new char[] {'-'}, 2);
// contenuto array: { "ab", "cd-efg" }
```

Il metodo **Join** viene chiamato sul tipo **string** e può essere usato nei seguenti modi:

```
string string.Join(string separatore, object[] valori)
string string.Join(string separatore, string[] valori)
string string.Join(string separatore, string[] valori,
int indicePartenza, int numeroValori )
```

nei primi due modi vengono concatenati i **valori** presenti nell'array (che può essere di un tipo generico oppure di tipo **string**) separandoli con separatore.

Con array di stringhe possono inoltre essere passati altri due parametri interi per specificare da quale elemento considerare e quanti elementi prendere.

Esempi:

```
int[] a = { 2, 4, 12, 7 };
string[] valori = { "uno", "due", "tre", "quattro", "cinque", "sei" };

string unione;
unione = string.Join(",", a);    // "2,4,12,7"

unione = string.Join("#", valori); // "uno#due#tre#quattro#cinque#sei"

unione = string.Join("#", valori, 1, 3); // "due#tre#quattro"
```


8. Algoritmi di ricerca e ordinamento

Gli informatici, fin dalla nascita dei primi calcolatori si sono interessati ai problemi di **ricerca** e **ordinamento** dei dati.

In diverse attività umane c'è la necessità di cercare dati all'interno di un archivio; ad esempio all'interno di un'anagrafe dei residenti di un comune, di un elenco di ordini di un'azienda.

Anche nelle attività personali sono frequenti operazioni come cercare un contatto all'interno della rubrica del telefono, un messaggio nel sistema di messaggistica che usiamo tutti i giorni o un prodotto in un sito di e-commerce.

Vedremo che se gli archivi vengono ordinati la ricerca diventa molto rapida. L'ordinamento è un altro classico problema che gli informatici hanno cercato di risolvere nel modo più efficiente possibile, a partire dall'ordinamento delle schede perforate usate nei primi calcolatori elettronici.

Ci concentreremo quindi sui due problemi della ricerca e dell'ordinamento illustrando alcuni algoritmici classici per risolverli. Come insieme di dati su cui ricercare o da ordinare ci limiteremo a vettori di numeri interi ma gli stessi algoritmi sono adattabili a qualsiasi struttura di dati. Inoltre ordineremo i numeri del vettore in ordine crescente; per ottenere un ordinamento decrescente basterà semplicemente cambiare la condizione di confronto.

Nel seguito considereremo il seguente, come vettore di esempio al quale applicare gli algoritmi; salviamo le dimensioni dell'array nella variabile **n**:

```
int[] v = new int[] { 19, 44, 38, 15, 40 };
int n = v.Length;
```

	0	1	2	3	4	
v	19	44	38	15	40	

Inoltre ci concentreremo su algoritmi di ordinamento che lavorano “*sul posto*” ovvero scambiano gli elementi del vettore e non usano array di supporto.

8.1. Ricerca sequenziale

Per ricercare un dato (chiamiamolo *valore*) all'interno di un insieme il modo più naturale è quello di procedere sequenzialmente esaminando ogni elemento e confrontandolo con quello che si sta cercando.

Ci potremo fermare appena abbiamo trovato il valore.

Con il nostro vettore *v* possiamo quindi procedere sequenzialmente dal primo elemento (di indice **0**) fino all'ultimo elemento (di indice **n-1**) e confrontare ogni elemento dell'array con il valore cercato. Se nessun elemento risulta uguale al valore cercato, il ciclo proseguirà fino alla fine.

Possiamo utilizzare un indicatore booleano (un *flag*) per memorizzare l'esito della ricerca:

il flag, che chiameremo *trovato*, sarà inizialmente impostato a *false* per indicare che non sappiamo ancora se l'elemento è presente e diventerà *true* nel caso trovassimo un elemento uguale al valore cercato. In tal caso, possiamo interrompere il ciclo senza esaminare gli elementi restanti. Ecco l'implementazione dell'algoritmo:

```
int valore;
Write("Valore da cercare: ");
Int32.TryParse(ReadLine(), out valore);

bool trovato = false;
for (int i = 0; i < n && !trovato; i++)
{
    if (v[i] == valore)
        trovato = true;
}

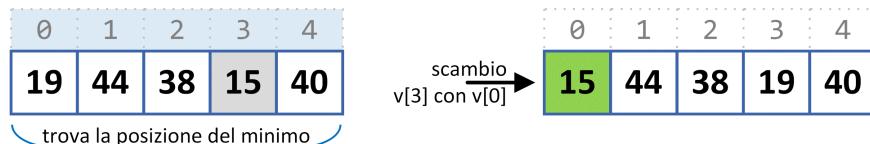
WriteLine($"Esito della ricerca di {valore} = {trovato}");
```

notiamo la condizione inserita come “guardia” nel ciclo for: il ciclo termina se l'indice va oltre il limite oppure se il valore viene trovato. Se avessimo tolto la seconda parte della condizione **&& !trovato** il ciclo sarebbe proseguito sempre fino alla fine del vettore anche se l'elemento viene trovato.

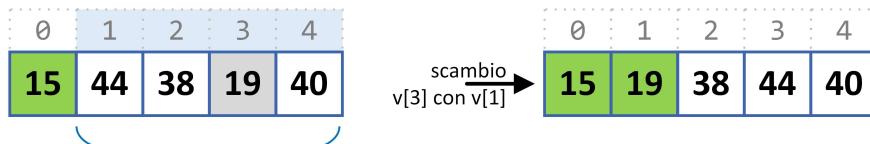
8.2. Ordinamento per selezione (*selection sort*)

L'ordinamento per selezione è uno tra i più semplici algoritmi di ordinamento.

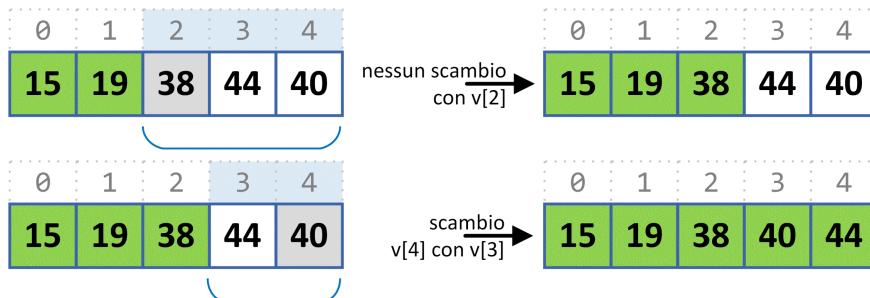
L'idea è quella di *selezionare* il minimo elemento del vettore e scambiarlo con il primo elemento. Visualizziamo la situazione con il vettore che abbiamo preso ad esempio:



Quindi, consideriamo gli elementi dal secondo in poi, *selezioniamo* il minimo tra questi e scambiamolo con l'elemento in seconda posizione:



continuiamo questo processo fino a quando abbiamo esaurito gli elementi:



L'implementazione dell'algoritmo è la seguente:

```
int iMin = 0; // variabile per salvare la posizione del minimo
for (int i = 0; i < n - 1; i++)
{
    // trova la posizione (iMin) del minimo tra le posizioni i e n-1
    iMin = i;
    for (int j = i + 1; j < n; j++)
        if (v[j] < v[iMin])
            iMin = j;

    // scambia il minimo con l'elemento di posto i (lo scambio viene effettuato solo se il minimo
    // non era già in posizione i)
    if (iMin != i)
```

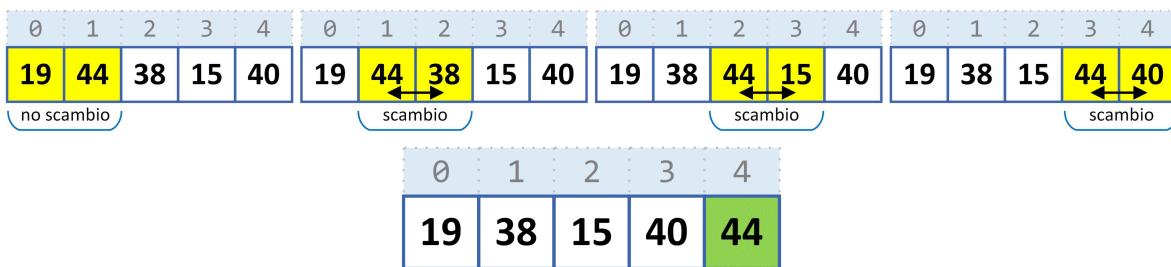
```
{  
    int temp = v[i];  
    v[i] = v[iMin];  
    v[iMin] = temp;  
}
```

8.3. Ordinamento per scambio (*bubble sort*)

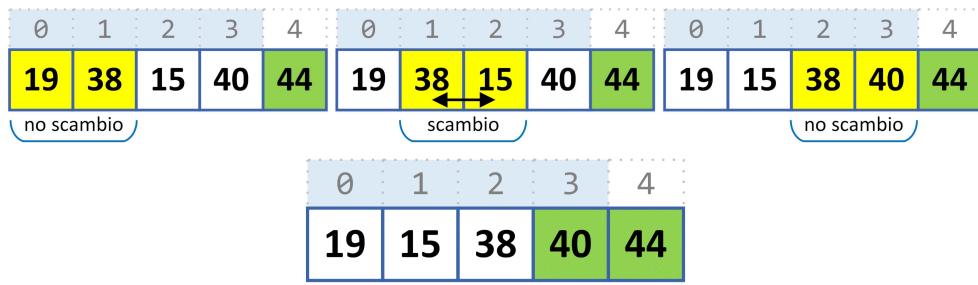
L'idea dell'ordinamento per scambio è quella di confrontare a due a due gli elementi del vettore e scambiarli in modo tale da far risalire verso la fine del vettore gli elementi più grandi. Per questo motivo viene anche chiamato ***bubble sort*** (**ordinamento a bolle**): come bollicine in un calice di spumante gli elementi più grandi “risalgono” verso l'alto, cioè verso la fine del vettore.

Partendo quindi dal vettore iniziale andiamo a confrontare il 1^o con il 2^o elemento e li scambiamo se il 1^o risulta più grande del secondo, procediamo analogamente confrontando 2^o con 3^o elemento, 3^o con 4^o elemento e così via fino ad arrivare alla fine del vettore.

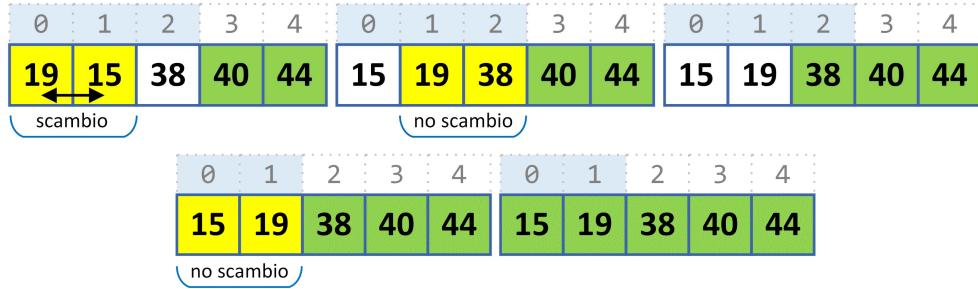
Come mostrato nella seguente immagine, avremo ottenuto nell'ultima posizione del vettore l'elemento più grande:



A questo punto ripetiamo il processo per tutti gli elementi dal primo al penultimo (l'ultimo è già a posto). Avremo così ottenuto l'elemento più grande (dei primi n-1 elementi) in penultima posizione:



Ripetendo il processo fino a considerare due soli elementi, avremo ordinato completamente il vettore:



Possiamo osservare che se in una passata del vettore non operiamo scambi significa che gli elementi erano già in ordine. **Bubble sort** è in grado quindi di “accorgersi” se gli elementi sono in ordine e fermarsi quando non vengono effettuati scambi. Questo lo rende in generale più efficiente dell’ordinamento per selezione quando il vettore è già ordinato o parzialmente ordinato.

Ecco l’implementazione in C#:

```
bool scambio = true; // indica se sono avvenuti scambi

for (int i = 0; scambio && i < n - 1; i++)
{
    // confronta gli elementi adiacenti e porta
    // l'elemento massimo in fondo (in posizione n-1-i)
    scambio = false;

    for (int j = 0; j < n - 1 - i; j++)
    {
        if (v[j] > v[j + 1])
        {
            int temp = v[j];
            v[j] = v[j + 1];
            v[j + 1] = temp;
            scambio = true;
        }
    }
}
```


8.4. Ordinamento per inserzione (*insertion sort*)

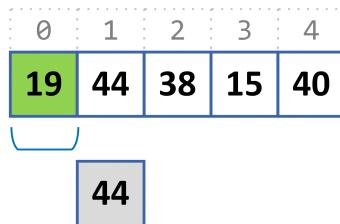
L'ordinamento per inserzione (o inserimento) è il modo utilizzato dai giocatori di carte per ordinare una mano.

Immaginiamo che le prime 3 carte di una mano siano già in ordine progressivo: dobbiamo inserire la 4[^] carta tra le prime 3 nella posizione corretta rispettando l'ordinamento progressivo.

Possiamo quindi confrontare la carta da inserire con la 3[^] carta e spostare quest'ultima verso destra se risulta maggiore; quindi ripetiamo l'operazione confrontando la carta da inserire con la 2[^] carta, e così via fino a trovare una carta che sia inferiore (o uguale) alla carta da inserire. Avremo così trovato la posizione dove inserire la carta.

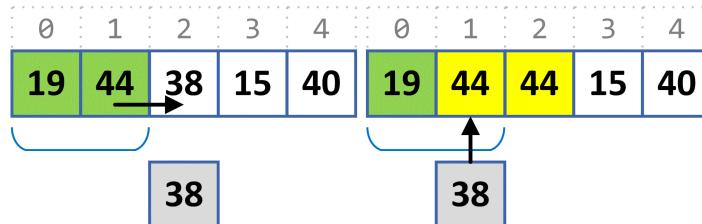
Con questa prima passata le prime 4 carte saranno in ordine. Ripetiamo il processo inserendo la 5[^] carta nella posizione corretta tra le prime quattro, inserendo la 6[^] carta tra le prime cinque e così via fino a quando non avremo ordinato l'intera mano.

Visualizziamo cosa succede con il nostro vettore di prova. Al primo passo dobbiamo confrontare il secondo elemento (44) con l'unico elemento che lo precede (19): non essendo minore non dobbiamo effettuare nessun slittamento a destra:



a questo punto i primi due elementi sono in ordine.

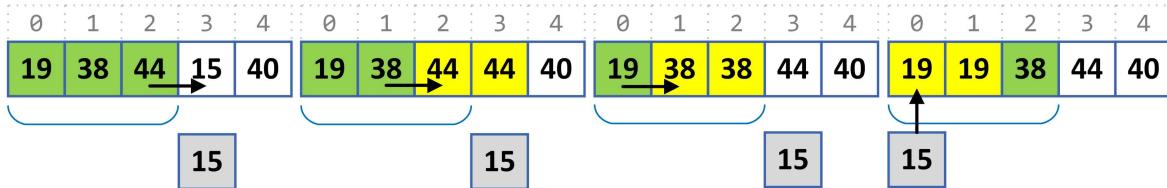
Dobbiamo quindi inserire nel posto giusto il terzo elemento (38):



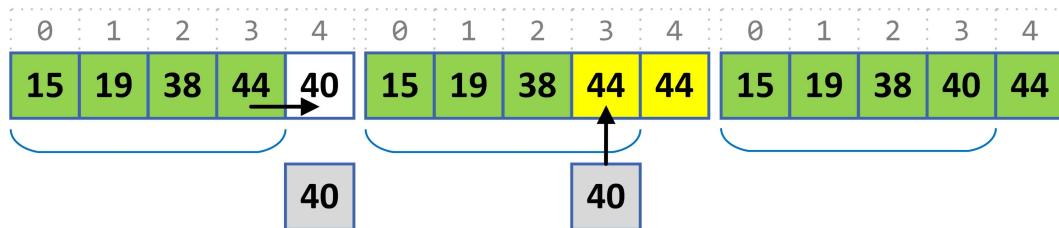
38 è minore di 44 quindi slittiamo 44 verso destra di una posizione; ci fermiamo quando non ci sono più elementi da far slittare e a questo punto

inseriamo 38 nella posizione a cui siamo arrivati.

Partiamo ora con i primi tre elementi in ordine e troviamo il posto dove inserire il quarto elemento (15): in questo caso (visto che 15 è il valore minimo) slitteremo tutti gli elementi verso destra fino ad arrivare in prima posizione dove potremo inserirlo:



L'ultimo passaggio viene effettuato considerando i primi quattro elementi in ordine e cercando il posto dove inserire l'ultimo elemento (40): ciò comporta un solo spostamento verso destra per inserire 40 in posizione 3:



Ecco l'implementazione in C#:

```

for (int i = 1; i < n; i++)
{
    // assunzione: gli elementi da 0 a i-1 sono in ordine,
    // inseriamo v[i] tra i primi i-1 elementi, nella giusta posizione
    int daInserire = v[i];

    // troviamo il "buco" dove inserire daInserire
    int j = i - 1;
    while (j >= 0 && v[j] > daInserire)
    {
        v[j + 1] = v[j];
        j--;
    }

    // daInserire va inserito in posizione j+1
    // (non serve se la sua posizione è rimasta la stessa)
    if (j+1 != i)
        v[j + 1] = daInserire;
}

```

Osserviamo che il ciclo più interno si ferma appena viene trovato un “buco” dove inserire l’elemento. Se il vettore è già inizialmente in ordine crescente, il ciclo più interno non verrà mai eseguito e l’algoritmo verrà eseguito in sole $n-1$ iterazioni.

8.5. Ricerca binaria (*binary search*)

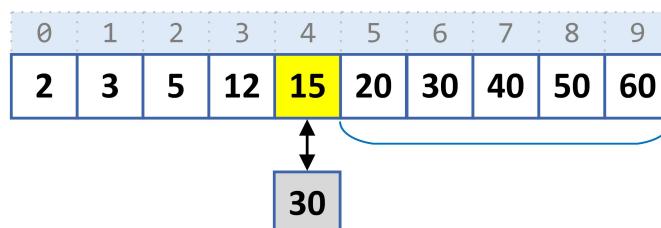
Ordinare un insieme di dati è conveniente perché consente una ricerca molto rapida di un elemento al suo interno, sfruttando il fatto che gli elementi sono in ordine.

Immaginiamo di dover ricercare un termine in un dizionario (dove, come è risaputo, i termini sono inseriti in ordine alfabetico). Possiamo procedere in questo modo:

- apriamo il dizionario a metà e controlliamo se la parola da ricercare si trova nelle due facciate aperte. Se lo è abbiamo concluso.
- altrimenti se la parola ricercata precede le parole indicate nelle facciate, possiamo ignorare la seconda metà del dizionario perché la parola (se esiste) sarà sicuramente presente nella prima parte; analogamente se fosse stata seguente avremo considerato solo la seconda metà del dizionario. Comunque vada abbiamo dimezzato in un sol colpo le parole da esaminare.
- procediamo allo stesso modo con la metà del dizionario rimasta: la dividiamo a metà e confrontiamo cosa troviamo con la parola ricercata.

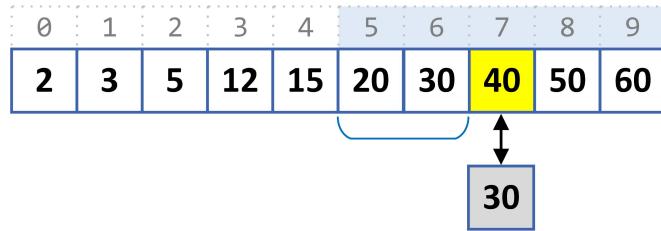
Seguendo questo metodo arriveremo o a trovare il termine da ricercare oppure, nel caso peggiore, a non trovarci in mano nessuna pagina rimasta e concludere quindi che il termine non esiste. Questo metodo va sotto il nome di **ricerca binaria** (o **ricerca dicotomica**^[48]) ed è il metodo più efficiente che possiamo applicare a insiemi ordinati di dati.

Applichiamo il metodo al seguente vettore di interi; vogliamo capire se il valore 30 esiste nel vettore. Andiamo quindi ad individuare l'elemento che sta al centro (15) e confrontiamolo con il valore da cercare:

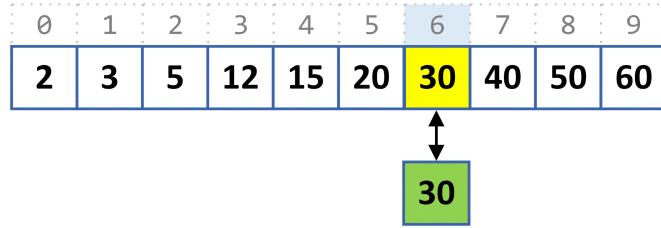


30 è maggiore di 15 quindi se è presente si troverà nella seconda parte del vettore (gli elementi da posizione 5 alla fine). Su questa procediamo allo

stesso modo individuando l'elemento centrale:



questa volta 30 è minore di 40, l'elemento centrale, e quindi possiamo concentrarci sulla porzione dall'elemento di posto 5 all'elemento di posto 6:



siamo rimasti con un solo elemento, quello di posto 6 che è quello cercato.

Vediamo l'implementazione dell'algoritmo:

```

int sx, dx, centro;
trovato = false;
sx = 0;
dx = n - 1;
while (sx <= dx && !trovato)
{
    centro = (sx + dx) / 2;
    if (valore > v[centro])
        // la ricerca prosegue nella porzione di indici [centro+1 ... dx]
        sx = centro + 1;
    else
        if (valore < v[centro])
            // la ricerca prosegue nella porzione di indici [sx ... centro-1]
            dx = centro - 1;
        else
            // L'elemento è stato trovato
            trovato = true;
}

```

WriteLine(\$"Esito della ricerca di {valore} = {trovato}");

abbiamo bisogno di due variabili, che abbiamo chiamato **sx** e **dx**, che mantengono gli indici della porzione di vettore da considerare. Inizialmente saranno valorizzati, il primo a **0** e il secondo all'indice dell'ultimo elemento, **n-1**.

Calcoliamo l'indice dell'elemento che sta al centro come $(sx+dx)/2$. Confrontiamo l'elemento centrale con il valore da cercare e in base a dove casca andremo a modificare **sx** e **dx**: in particolare se il valore da cercare è maggiore dell'elemento centrale dovremo continuare la ricerca sulla porzione di destra, modificando quindi **sx**, altrimenti continueremo la ricerca sulla porzione di sinistra modificando **dx**.

Il processo continua finché abbiamo trovato l'elemento oppure fino a quando **sx** supererà **dx**: in quest'ultimo caso possiamo concludere che l'elemento cercato non è presente nel vettore.

Come ultima considerazione notiamo quanto è molto più efficiente la ricerca binaria rispetto alla ricerca sequenziale: immaginando di dover cercare un valore in un vettore di 1000 elementi, la ricerca sequenziale implicherebbe, nel caso peggiore (che l'elemento sia l'ultimo oppure non esista) 1000 iterazioni. Con la ricerca binaria dovremo invece fare al massimo 10 iterazioni, visto che 2^{10} supera 1000.

L'algoritmo di ricerca binaria comporta un piccolo aumento di iterazioni all'aumentare del numero di elementi. Pensando ad esempio di raddoppiare il numero di elementi, con la ricerca sequenziale avremo 2000 iterazioni, il doppio di prima; con la ricerca binaria dovremo invece fare una sola iterazione in più ($2^{11} = 2048$). Un bel risultato!

8.6. Fusione ordinata di due array (*merge*)

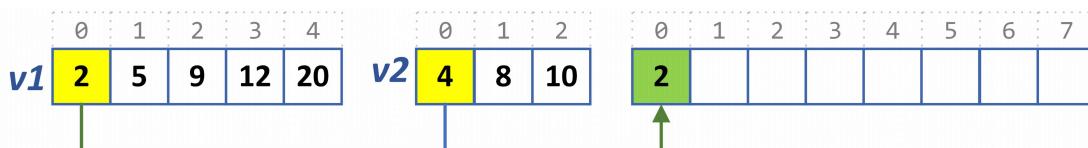
Supponiamo di avere due vettori **v1** e **v2**, di interi entrambi ordinati in ordine crescente.

Vogliamo ricopiare tutti gli elementi di **v1** e **v2** in un terzo vettore (che chiameremo **vm**) mantenendo l'ordine crescente: questo tipo di elaborazione viene chiamata **fusione ordinata** (o *merge*).

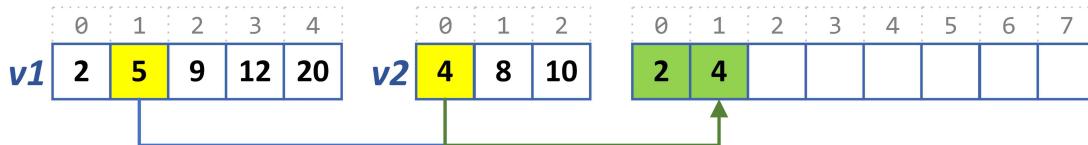
La prima idea potrebbe essere quella di ricopiare così come sono gli elementi di **v1** in **vm**, quindi ricopiare tutti gli elementi di **v2** in **vm** ed infine ordinare l'array **vm**.

Il numero di iterazioni per attuare questo metodo può però essere ridotto di molto (esattamente alla somma delle lunghezze dei due vettori) risparmiando il riordinamento finale di **vm** e sfruttando invece il fatto che **v1** e **v2** sono già in ordine: potremo cioè confrontare il primo elemento di **v1** con il primo elemento di **v2** e ricopiare l'elemento più piccolo in **vm**.

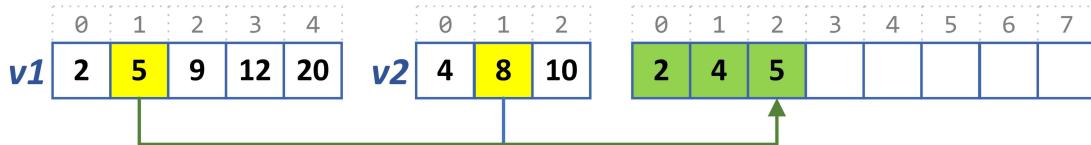
Visualizziamo quanto appena scritto con un'immagine considerando due vettori di esempio di, rispettivamente, 5 elementi e 3 elementi; è chiaro che il vettore **vm** dovrà avere lunghezza almeno 8 elementi per poter contenere tutti gli elementi di **v1** e **v2**:



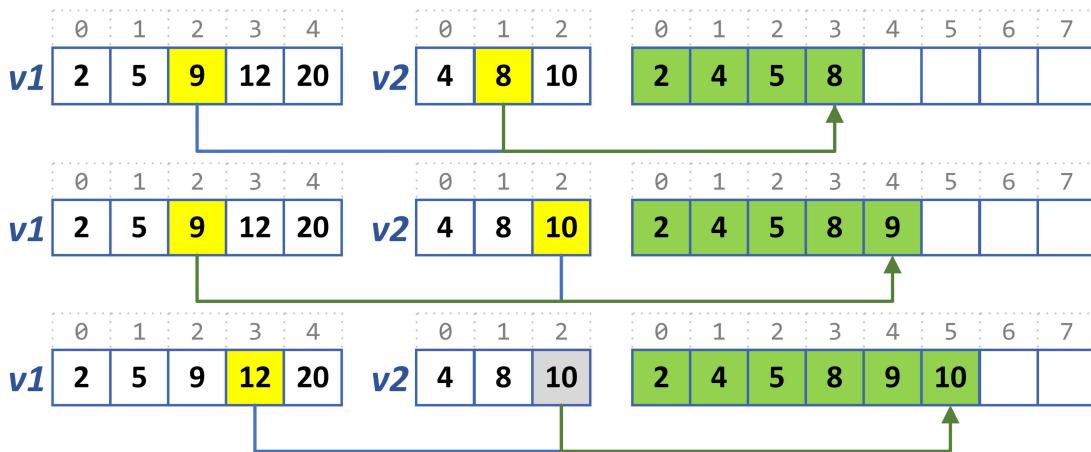
a questo punto, procediamo confrontando il secondo elemento di **v1** con il primo di **v2**; copieremo stavolta il valore 4 (il minore dei due) nel vettore **vm**:



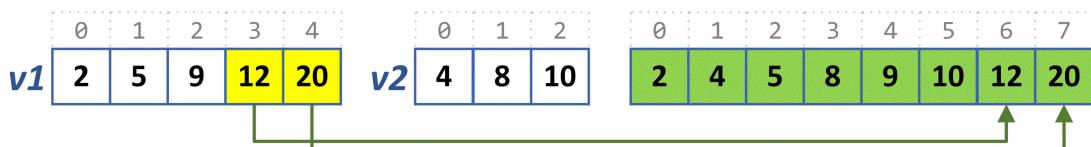
al prossimo passo il confronto è tra 5 e 8 con la scelta di 5 da ricopiare in **vm**:



continuiamo in questo modo fino a quando uno dei due vettori viene esaurito:



A questo punto il contenuto del vettore **v2** è stato completamente ricopiato nel vettore **vm**. Possiamo concludere ricopiando i due elementi rimanenti di **v1** in **vm**.



Riepilogando, l'algoritmo di fusione confronta, partendo dalla prima posizione, un elemento del primo vettore con un elemento del secondo vettore e ricopia l'elemento minore nel vettore risultante; poi passa all'elemento successivo nel vettore da cui è stata appena effettuata la copia e continua in questo modo fino ad esaurire uno dei due vettori. Alla fine, ricopia la parte restante di uno dei due vettori (quello non esaurito) nel vettore fusione.

Avremo bisogno quindi di tre contatori, **i** per iterare su **v1**, **j** per iterare su **v2** e **k** per iterare su **vm**. Ecco l'algoritmo in C#:

```
int[] vm = new int[v1.Length + v2.Length];
int i = 0, j = 0, k = 0;
```

```
while (i < v1.Length && j < v2.Length)
```

```

{
    // inserisce in vm l'elemento minore tra v1[i] e v2[j]
    if (v1[i] < v2[j])
    {
        vm[k] = v1[i];
        i++;
    }
    else
    {
        vm[k] = v2[j];
        j++;
    }
    k++;
}

// copia l'eventuale parte restante di v1
while (i < v1.Length)
{
    vm[k] = v1[i];
    i++; k++;
}

// copia l'eventuale parte restante di v2
while (j < v2.Length)
{
    vm[k] = v2[j];
    j++; k++;
}

```


9. Metodologia top-down e sottoprogrammi

Finora abbiamo affrontato problemi semplici per i quali abbiamo sviluppato programmi che racchiudevano tutto il codice all'interno del metodo *Main*. Nei casi reali questo non è più sufficiente.

Immaginiamo di voler sviluppare un programma che consenta di gestire gli arrivi ad una gara sportiva permettendo l'inserimento degli arrivi (nomi degli atleti con il relativo tempo impiegato) e producendo quindi la classifica ordinata dal primo arrivato all'ultimo nonché il tempo medio impiegato dagli atleti.

Anziché affrontare il problema come un tutt'uno risulta più semplice scomporre il problema in *sottoproblemi*, concentrarsi su questi ultimi e assemblare infine la soluzione al problema generale.

Ad esempio, il programma precedente potrebbe essere suddiviso nei seguenti tre compiti o *macro azioni*:

1. input dei dati degli arrivi,
2. output della classifica ordinata,
3. calcolo e output del tempo medio.

A loro volta, le tre macro azioni potrebbero essere affrontate singolarmente ed eventualmente suddivise in altri compiti ancora più *granulari*.

Questa tecnica di analisi dei problemi viene chiamata di *raffinamento a più passi* o *metodologia top-down*, dall'alto al basso, dal generale al particolare.

La metodologia top down oltre a semplificare l'analisi dei problemi consente di trovare soluzioni più gestibili e mantenibili nel tempo; possiamo anche immaginare di assegnare la risoluzione dei sottoproblemi a più progettisti, magari specializzati in quella determinata categoria di problemi.

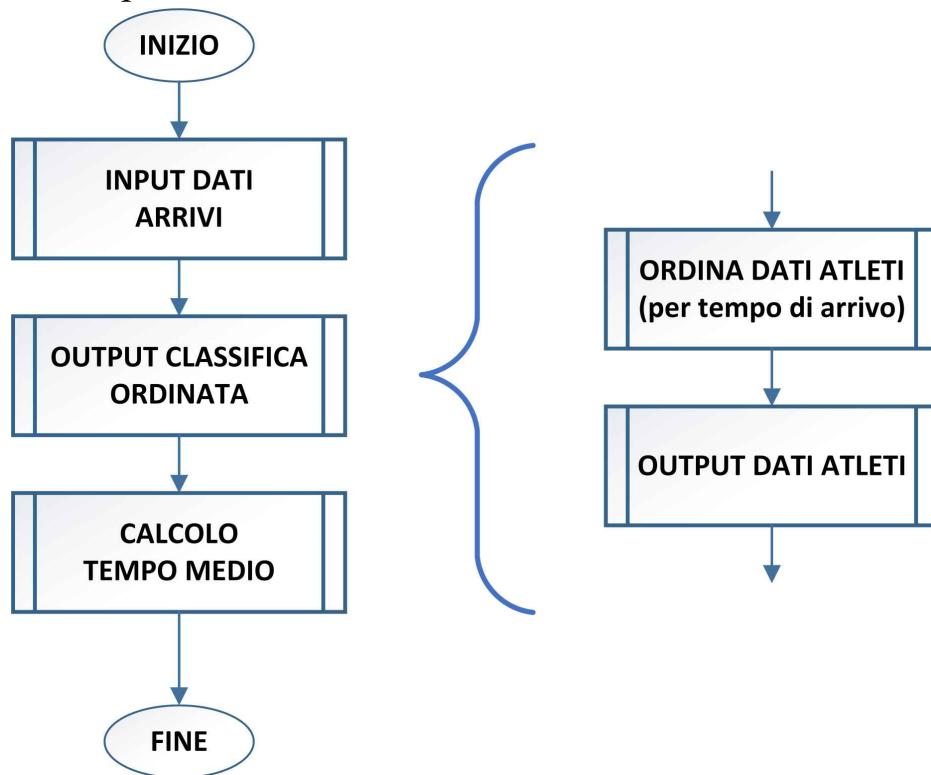
Inoltre i sottoproblemi possono essere ricorrenti e quindi la loro soluzione può essere sfruttata più volte all'interno del problema più generale.

Anche lo sviluppo di un sito web, ad esempio di e-commerce, viene affrontato in questo modo suddividendo la progettazione in molti sotto

compiti come la realizzazione dell'interfaccia con l'utente, la gestione delle iscrizioni, del carrello acquisti, del pagamento, ecc.

Se ci pensiamo molti problemi pratici, non solo informatici, vengono affrontati seguendo questo approccio: anche il progetto di una macchina industriale viene scomposto nei progetti dei vari sottosistemi che dovranno comporre la macchina e a loro volta questi sottosistemi saranno composti da sistemi via via più semplici.

Ritornando al problema proposto, possiamo rappresentare la situazione con il seguente flow chart in cui abbiamo usato il blocco grafico tipico che si usa per rappresentare le *procedure* (o *sottoprogrammi*) che risolvono i sottoproblemi; la procedura intermedia, della produzione della classifica, è stata ulteriormente scomposta nell'ordinamento dei risultati per tempo di arrivo e nell'output della classifica così ottenuta:



9.1. Sottoprogrammi

Il termine **sottoprogramma** è usato genericamente nei linguaggi di programmazione ad alto livello per indicare unità di codice specializzate a risolvere un determinato sottoproblema.

Il concetto di sottoprogramma si ritrova anche nei primi linguaggi a basso livello spesso con il nome di **subroutine**, per indicare unità di codice *assembly* o in un linguaggio macchina.

Nei linguaggi ad alto livello si può distinguere anche tra funzione e procedura:

- per **funzione** si intende un sottoprogramma che a partire da dati in ingresso produce un risultato (unico) che restituisce al programma principale; il concetto è analogo a quello di *funzione matematica*
- per **procedura** si intende invece un sottoprogramma che non restituisce nessun risultato ma effettua solo una serie di operazioni.

Alcuni linguaggi di programmazione come il Pascal distinguono con parole chiavi apposite (*function* e *procedure*) le due tipologie di sottoprogrammi.

Altri linguaggi come il C e tutti quelli che hanno derivato la loro sintassi da questo linguaggio^[49] (come C# e Java), usano invece soltanto funzioni senza fare uso di una specifica parola chiave.

Le procedure in C e in questi linguaggi sono semplicemente funzioni che non ritornano alcun valore; come vedremo per indicarlo si contrassegna la funzione con la parola chiave **void** (*vuoto*).

Nell'ambito della programmazione orientata agli oggetti (paradigma adottato dalla maggior parte dei linguaggi moderni) si usa più propriamente il termine **metodo**. Un metodo può essere definito come una funzione associata ad un oggetto ed è sempre obbligatoriamente definito all'interno di una classe.

9.2. Metodi in C#

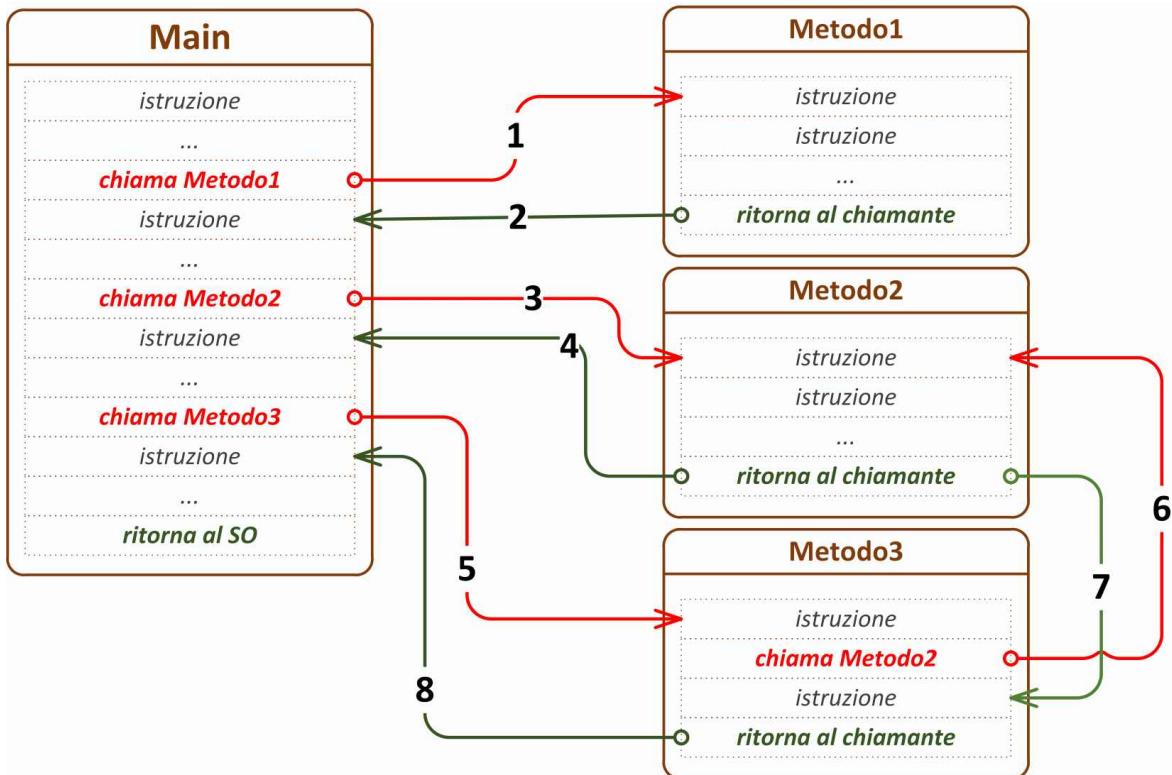
I programmi che abbiamo visto finora contengono già un metodo, il metodo **Main** che è metodo principale del programma, il punto di ingresso dal quale l'esecuzione inizia. Il metodo **Main** viene chiamato automaticamente dal CLR all'avvio dell'applicazione^[50].

Una classe può contenere altri metodi secondari che vengono dichiarati e definiti all'interno della classe in cui è inserito il metodo **Main** o di altre classi.

Il metodo **Main** potrà quindi chiamare un altro metodo che eseguirà le istruzioni in esso definite per ritornare, al termine, il controllo nuovamente al *Main* che riprenderà l'esecuzione dall'istruzione immediatamente successiva a quella di chiamata del metodo secondario; lo stesso metodo potrà essere chiamato anche più volte, ad esempio per svolgere compiti ricorrenti.

A loro volta i metodi secondari potranno chiamare altri metodi o, come vedremo in seguito, richiamare se stessi (si parla in questo caso di **metodi ricorsivi**).

La seguente immagine mostra un esempio di sequenza di chiamate a metodi secondari:



Vediamo qual è la sintassi più usuale di un metodo in C#, prendendo ad esempio una semplice funzione che restituisce il massimo tra due interi:

nome metodo	tipo di ritorno
lista parametri	modificatore di accesso

**intestazione
del metodo**

public int Massimo (int a, int b)

{

```
if (a > b)
corpo
del metodo           return a;
else
    return b;
}
```

sintatticamente la definizione di un metodo è costituita da due parti:

- l'**intestazione del metodo** che ne descrive le caratteristiche come il valore di ritorno, il nome e gli eventuali parametri
- il **corpo del metodo** (racchiuso tra parentesi graffe) che è l'insieme delle istruzioni che lo costituiscono.

L'intestazione del metodo è composta dalle seguenti parti:

- **modificatore di accesso:** è una parola chiave che determina la visibilità del metodo dalle altre classi. I modificatori di accesso più frequenti sono:
 - ***private***: il metodo è richiamabile solo da altri metodi della stessa classe;
 - ***public***: il metodo è richiamabile anche da altre classi;
 - ***protected***: il metodo è richiamabile anche da altre classi ma solo se sono derivate^[51] dalla classe in cui è definito.
- Possiamo omettere il modificatore di accesso: in questo caso verrà assunto il modificatore *private*.
- **tipo di ritorno:** questo è il tipo del valore ritornato dal metodo:
 - se un metodo è una funzione, cioè ritorna un valore a chi lo ha chiamato, deve esserne indicato il tipo del valore. Una funzione può ritornare un unico valore che però può essere anche di tipo complesso, ad esempio una stringa o un array.
Inoltre, come mostrato nell'esempio, una funzione dovrà usare la parola chiave **return** *seguita dal valore* che si vuole ritornare al chiamante.
 - se un metodo è una procedura cioè non ritorna nulla al chiamante, il tipo di ritorno deve essere la parola chiave **void**. In questo caso la parola chiave *return* non viene utilizzata nel corpo del metodo. È eventualmente possibile usarla per interrompere anzitempo l'esecuzione del metodo senza però indicare nessun valore di ritorno (si scrive **return;**)
- **nome del metodo:** questo è il nome del metodo che verrà usato per richiamarlo da altri metodi. In C# si usa la convenzione **Pascal Case** (*notazione Pascal*) per i nomi dei metodi, cioè si usano iniziali maiuscole per tutte le parole che compongono il nome^[52].
- **lista parametri:** questi sono i parametri che verranno passati al metodo e sulla base dei quali il metodo effettuerà le sue operazioni o calcolerà un risultato:
 - i parametri, se più di uno, vanno separati con la virgola

- ogni parametro viene definito da un tipo e un nome
- la lista dei parametri può essere vuota se il metodo non ha bisogno di parametri; in questo caso le parentesi tonde vanno indicate lo stesso. Anche nella chiamata del metodo sarà obbligatorio indicare le parentesi tonde.

Per **firma del metodo** si intende la parte dell'intestazione costituita dal nome e dalla lista dei parametri. La firma lo identifica rispetto agli altri metodi e deve essere unica all'interno della classe.

È quindi possibile dichiarare metodi con lo stesso nome ma parametri diversi; in questo modo il compilatore sarà in grado di discriminare tra più metodi con lo stesso nome. Questa caratteristica si chiama **overload dei metodi** (in italiano, *sovraffunzione*).

La firma viene anche chiamata **segnatura** (dall'inglese *signature*) o **prototipo**, quest'ultimo è un termine tipico dei linguaggi C e C++.

9.3. Elementi essenziali di OOP e metodi statici

Per capire meglio come scrivere nuovi metodi all'interno delle classi, è utile introdurre qualche nozione di **programmazione orientata agli oggetti** (**OOP, Object Oriented Programming**).

In questo paradigma di programmazione gli elementi gestiti dal programma sono chiamati **oggetti** e modellano gli aspetti della realtà che il programma sta gestendo.

Ad esempio in un'applicazione per la gestione di una rubrica, gli oggetti potrebbero essere i singoli contatti inseriti nella rubrica: ogni contatto ha le sue caratteristiche ovvero i dati che lo descrivono (nome, cognome, telefono, ...) e le azioni (i metodi) che possono essere applicati su di esso.

Tutti gli oggetti della rubrica condividono quindi lo stesso tipo, che in OOP viene chiamato **classe**, concetto principale della OOP. La classe funge da *stampo* per gli oggetti di un determinato tipo, definisce cioè le caratteristiche e le funzionalità che tutti gli oggetti di quel tipo dovranno avere. Gli oggetti vengono anche chiamati **istanze della classe**.

Ad esempio i contatti delle rubrica potranno essere oggetti della classe **Contatto** definita dal seguente codice (molto semplificato):

```
class Contatto
{
    private string nome;
    private string cognome;
    private string telefono;

    public void SetInfo(string n, string c, string t)
    {
        nome = n.Trim().ToUpper();
        cognome = c.Trim().ToUpper();
        telefono = t.Trim();
    }

    public string GetNominativo()
    {
        return $" {nome} {cognome}";
    }

    public string GetTelefono()
    {
        return telefono;
    }
}
```

Il codice che userà la classe, ad esempio nel metodo Main, dovrà creare oggetti del tipo Contatto utilizzando l'operatore **new**, che abbiamo incontrato con gli array o altre classi della libreria .NET (ad esempio la classe **Random**). Vediamo un esempio di codice che crea due contatti e ne visualizza i dati:

```
class Program
{
    static void Main(string[] args)
    {
        Contatto c1, c2;

        c1 = new Contatto();
        c1.SetInfo("Toni", "Di Grigio", "0445 234567");

        c2 = new Contatto();
        c2.SetInfo("Lara", "Gazza", "0445 229988");

        Console.WriteLine("il numero di " + c1.GetNominativo() +
            " è " + c1.GetTelefono());
        Console.WriteLine("il numero di " + c2.GetNominativo() +
            " è " + c2.GetTelefono());
        Console.ReadKey();
    }
}
```

quindi ogni oggetto di tipo *Contatto* mantiene le informazioni di *nome*, *cognome* e *telefono*. Sugli oggetti *c1* e *c2* abbiamo richiamato i metodi previsti dalla classe, per impostare le informazioni del contatto e per ottenere le informazioni come il nominativo (nome e cognome) e il telefono.

In generale un metodo viene richiamato a partire da un'istanza di una classe, cioè da un oggetto precedentemente creato.

Nei programmi visti finora non abbiamo mai avuto la necessità di aggiungere nuove classi al nostro progetto. Abbiamo fatto tutto dal metodo *Main* che, osserviamo, è anch'esso inserito in una classe, la classe **Program**.

Una caratteristica del metodo *Main* è che si tratta di un **metodo statico**. Un metodo statico può essere utilizzato senza che esista un oggetto di quel tipo, è condiviso e unico per tutti gli oggetti di quel tipo; per richiamarlo dobbiamo fare riferimento esclusivamente al nome della classe.

Esempi di metodi statici sono tutti quelli della classe **Console**: per richiamarli usiamo sempre la sintassi **Console.NomeMetodo(...)**. Non sarebbe nemmeno possibile creare oggetti di tipo Console dato che la classe Console è una **classe statica**, cioè contiene esclusivamente metodi statici e non è istanziabile, cioè non è possibile creare oggetti di tipo Console.

Anche la classe **Math** è una classe statica, tutti i suoi metodi vanno richiamati come **Math.NomeMetodo()**.

Altre classi come la classe String invece contengono sia metodi statici che metodi non statici (i metodi “normali”, non statici, vengono chiamati anche **metodi di istanza**): ad esempio il metodo **ToUpper()** si applica ad un oggetto di tipo **String** mentre il metodo **Join** si richiama con il riferimento alla classe, cioè come **String.Join(...)**.

In conclusione per poter utilizzare un metodo senza bisogno di creare un oggetto su cui utilizzarlo possiamo dichiararlo statico con la parola chiave **static**, che va inserita prima del tipo di ritorno del metodo (indifferentemente prima o dopo l’eventuale modificatore di accesso).

Riprendendo l’esempio iniziale sulla funzione *Massimo*, per poterla usare dal metodo *Main*, avremo dovuto creare un oggetto (in questo caso di classe *Program1*). Per evitarlo possiamo semplicemente dichiarare la funzione **static** come mostrato nel seguente esempio di programma completo (nel quale abbiamo tolto il modificatore di accesso alla funzione *Massimo*, rendendola di fatto utilizzabile solo dall’interno della classe *Program1*):

```
class Program1
{
    static int Massimo(int a, int b)
    {
        if (a > b)
            return a;
        else
            return b;
    }

    static void Main(string[] args)
    {
        int n1, n2, m;
        Console.WriteLine("Inserisci due numeri interi");
        int.TryParse(Console.ReadLine(), out n1);
        int.TryParse(Console.ReadLine(), out n2);

        m = Massimo(n1, n2);
        Console.WriteLine($"Il massimo tra {n1} e {n2} è {m}");
    }
}
```

```
        Console.ReadKey();
    }
}
```

D'ora in avanti dichiareremo quindi tutti i metodi statici in modo da poterli usare “liberamente” senza dover interessarci alla definizione di nuove classi e alla creazione di oggetti.

9.4. Campi dato e ambito di visibilità

Come abbiamo visto con la classe *Contatto*, all'interno di una classe possiamo avere metodi ma anche variabili, che nel gergo OOP, vengono chiamati **campi dato** (o **membri dato** o semplicemente **campi**).

Nella classe *Contatto* abbiamo come campi le tre stringhe *nome*, *cognome* e *telefono*.

Un oggetto di tipo *Contatto* avrà quindi tre stringhe ad esso associate distinte da quelle associate ad altri oggetti dello stesso tipo.

I campi dato hanno visibilità globale all'interno di una classe, sono cioè accessibili da tutti i metodi della classe.

Anche i campi dato di una classe possono essere dichiarati statici (in tal caso vengono chiamati **campi di classe**): in questo modo diventano unici per tutti gli oggetti di quel tipo e non sarà necessario creare oggetti per accedervi. Come esempio completo, riprendiamo il programma di gestione dei risultati della gara sportiva presentato all'inizio del capitolo:

```
class Program2
{
    static int n; // numero arrivi
    static string[] nomiAtleti;
    static int[] tempiAtleti; // tempi di arrivo in secondi

    static void AcquisisciArrivi()
    {
        for (int i = 0; i < n; i++)
        {
            Console.WriteLine("Inserire il nome ciclista: ");
            nomiAtleti[i] = Console.ReadLine();
            Console.WriteLine("Inserire il tempo di arrivo: ");
            int.TryParse(Console.ReadLine(), out tempiAtleti[i]);
        }
    }

    static void VisualizzaArrivi()
    {
        for (int i = 0; i < n; i++)
            Console.WriteLine($"{nomiAtleti[i]}, {tempiAtleti[i]}");
    }

    static void OrdinaArrivi()
    {
        // insertion sort
        for (int i = 1; i < n; i++)
```

```

{
    int tempo = tempiAtleti[i];
    string nome = nomiAtleti[i];
    int j = i - 1;
    while (j >= 0 && tempiAtleti[j] > tempo)
    {
        tempiAtleti[j + 1] = tempiAtleti[j];
        nomiAtleti[j + 1] = nomiAtleti[j];
        j--;
    }
    if (j + 1 != i)
    {
        tempiAtleti[j + 1] = tempo; nomiAtleti[j + 1] = nome;
    }
}
}

static void Main(string[] args)
{
    Console.WriteLine("Numero ciclisti:");
    int.TryParse(Console.ReadLine(), out n);

    nomiAtleti = new string[n];
    tempiAtleti = new int[n];

    Console.WriteLine("----- ACQUISIZIONE DATI:");
    AcquisisciArrivi();
    VisualizzaArrivi();

    Console.WriteLine("----- CLASSIFICA ORDINATA:");
    OrdinaArrivi();
    VisualizzaArrivi();

    Console.ReadKey();
}
}

```

La classe è composta da *tre campi dato statici*, il numero di atleti e i due vettori che serviranno per contenere i nomi degli atleti e i rispettivi tempi. Abbiamo poi i tre metodi statici per acquisire i dati, visualizzare l'elenco degli atleti con i rispettivi tempi e ordinare l'elenco. Come detto tutti i metodi possono accedere ai campi dato.

Il metodo **Main** diventa quindi molto sintetico e si limita a richiamare i metodi precedenti; come vediamo il metodo per visualizzare i dati viene sfruttato due volte.

Notiamo che il metodo *Main* viene definito alla fine della classe: in realtà può essere posizionato in qualsiasi punto all'interno della classe anche prima dei metodi richiamati^[53].

Sempre riguardo ai campi dato, diversamente dalle variabili locali e analogamente agli elementi di un array, vengono automaticamente inizializzati ai valori di default dei rispettivi tipi^[54], quando il programma viene caricato.

Altra cosa importante da ricordare riguardo all'ambito di visibilità è che se un metodo dichiara una variabile locale con lo stesso nome di un campo di classe, quest'ultima nasconde la variabile globale^[55].

Per accedere alla variabile globale dobbiamo esplicitamente specificare la classe di appartenenza con la sintassi **Classe.Campo**.

Vediamo di seguito un esempio completo:

```
class Program3
{
    static int a; // è inizializzata automaticamente a 0
    static int b = 1;

    static void Main(string[] args)
    {
        int b = 2;

        // possiamo accedere alla variabile globale a anche
        // se non ha ricevuto esplicitamente un valore
        Console.WriteLine(a); // stampa 0

        // la variabile b locale nasconde la variabile b globale
        Console.WriteLine(b); // stampa 2

        // per accedere alla variabile b globale dobbiamo
        // specificare la classe di appartenenza
        Console.WriteLine(Program3.b); // stampa 1

        Console.ReadKey();
    }
}
```


9.5. Metodi funzione e valore di ritorno

Come abbiamo visto, un metodo può essere una **funzione** cioè ritornare un valore al metodo che lo ha invocato:

- nell'intestazione del metodo va specificato, al posto di **void**, il tipo del valore ritornato.
- nel corpo del metodo si usa l'istruzione **return** seguita dal valore da ritornare; l'istruzione **return** causa inoltre l'interruzione del metodo ed il ritorno al chiamante.

È possibile usare **return** (senza alcun valore) anche nei metodi **void** per interrompere anticipatamente l'esecuzione del metodo. Questa è una pratica diffusa tra i programmatori anche se toglie leggibilità al codice, soprattutto se il metodo è formato da molte righe.

In effetti le regole della **programmazione strutturata** prevedono che ogni funzione abbia un solo punto di uscita. Ciò vale per ogni altro blocco di codice, che dovrebbe avere un solo ingresso e una sola uscita; quindi sarebbero da evitare istruzioni come *break* o *continue* e tanto meno istruzioni *goto*.

Nel seguente programma leggiamo in input la lingua scelta dall'utente che memorizziamo in un campo di classe per usarlo quindi da un metodo che saluta in base alla lingua scelta:

```
class Program4
{
    static string lingua; // campo di classe (visibilità globale)
    tipo di ritorno ←
    static string Saluta()
    {
```

```
        string s;
        if (lingua == "I") s = "Salve ";
        corpo della
        funzione      else
```

```
if (lingua == "E") s = "Hello ";
else
    s = ":) ";
return s;
ritorna il controllo al chiamante
restituendo il valore di s
}
```

```
static void Main(string[] args)
{
    string nome;
    Console.Write("Come ti chiami?");
    nome = Console.ReadLine();
    Console.Write("Scegli una lingua (I = italiano - E = inglese):");
    lingua = Console.ReadLine().ToUpper();
    Console.WriteLine(Saluta() + nome);
}
```

9.6. Metodi con parametri

Come abbiamo visto nel primissimo esempio di questo capitolo, un metodo può ricevere uno o più parametri. In questo modo l'algoritmo rappresentato dal metodo può essere generalizzato e usato in modo più versatile con valori diversi.

Immaginiamo ad esempio di avere la necessità frequente di visualizzare il contenuto di un array di stringhe. Se realizziamo un metodo senza parametri (come *VisualizzaArrivi()* visto in precedenza) potremo usarlo solo per visualizzare sempre lo stesso array globale definito a livello di classe; se invece passiamo l'array da visualizzare come parametro, potremo sfruttare lo stesso metodo per visualizzare array diversi.

Un metodo con parametri deve definire tipo e nome di ogni parametro ricevuto: i parametri indicati nell'intestazione vengono detti **parametri formali** di contro ai **parametri effettivi** (o **attuali**) che sono i valori usati nella singola chiamata al metodo: al momento dell'invocazione del metodo da un altro metodo, i valori dei parametri effettivi vengono copiati nei parametri formali e il metodo viene eseguito con quei valori.

I parametri formali si comportano come variabili locali al metodo ed è perciò vietato definire variabili locali al metodo che usino gli stessi identificatori usati per i parametri.

Vediamo un esempio di una semplice funzione che calcola la somma degli interi tra due estremi:

```
parametri formali           class Program5
{
    static int Sommatoria(int a, int b)
    {
        int s = 0;
        for (int i = a; i <= b; i++) s = s + i;
        return s;
    }
    parametri
    effettivi

    static void Main(string[] args)
    {
        int inf, sup;
        WriteLine($"Somma da 1 a 100 = { Sommatoria(1, 100) }");
        Write("Estremo inferiore? ");
    }
}
```

```
parametri  
effettivi  
    inf = Convert.ToInt32(Console.ReadLine());  
  
    Write("Estremo superiore? ");  
    sup = Convert.ToInt32(Console.ReadLine());  
  
    WriteLine($"Somma da {inf} a {sup} = {Sommatoria(inf, sup)}");  
}  
}
```

Nella prima chiamata al metodo vengono passati due costanti letterali (1 e 100) che verranno ricopiate nei parametri formali *a* e *b*.

Nella seconda chiamata, invece, i valori delle variabili *inf* e *sup* richieste in input, vengono ricopiatati nei parametri formali *a* e *b*.

Il nome assegnato ai parametri formali non è quindi assolutamente legato ai parametri passati durante la chiamata al metodo. I valori dei parametri effettivi viene copiato nell'ordine sui parametri formali: il primo parametro effettivo viene copiato sul primo parametro formale, il secondo sul secondo, e così via.

L'invocazione del metodo deve quindi rispettare rigorosamente tipo e numero di parametri previsti dalla definizione del metodo.

9.7. Passaggio dei parametri per valore

Come abbiamo visto nel precedente paragrafo, i valori dei parametri effettivi vengono ricopiatati sui parametri formali del metodo. Questo comportamento viene detto *passaggio per valore*.

Anche se il metodo modifica il valore dei parametri formali (ed è possibile farlo), ai parametri effettivi non succederà nulla.

Consideriamo il seguente esempio che dovrebbe scambiare il valore delle variabili passate come parametri:

```
class Program6
{
    static void ScambiaFake(int a, int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }

    static void Main(string[] args)
    {
        int x = 5, y = 12;

        ScambiaFake(x, y);
        // N.B: x vale ancora 5 e y vale ancora 12 (passaggio per valore)
    }
}
```

Il programma non modifica i valori originari di *x* e *y* perché semplicemente questi vengono ricopiatati su *a* e *b* che hanno “vita” propria.

Per default il passaggio dei parametri ad un metodo è per valore. Nel prossimo paragrafo vedremo come sia possibile ottenere lo scopo desiderato utilizzando un altro tipo di passaggio.

Consideriamo ora il seguente programma che implementa un metodo per l’ordinamento di un array con l’algoritmo *bubble sort*:

```

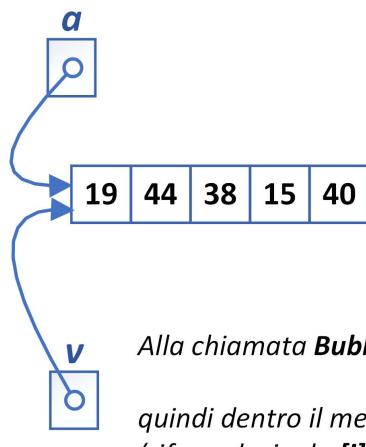
class Program7
{
    static void BubbleSort(int[] v)
    {
        int n = v.Length;
        bool scambio = true;
        for (int i = 0; scambio && i < n - 1; i++)
        {
            scambio = false;
            for (int j = 0; j < n - 1 - i; j++)
            {
                if (v[j] > v[j + 1])
                {
                    int temp = v[j];
                    v[j] = v[j + 1];
                    v[j + 1] = temp;
                    scambio = true;
                }
            }
        }
    }

    static void Main(string[] args)
    {
        int[] a = { 19, 44, 38, 15, 40 };
        BubbleSort(a);
    }
}

```

Come è possibile che il vettore venga ordinato se è passato per valore?

È in effetti così ed è perfettamente coerente: quello che viene copiato è il valore della variabile *a* che, essendo un array (che è un tipo riferimento), contiene l'indirizzo in memoria di dove stanno gli elementi del vettore. Il parametro formale *v* otterrà quindi lo stesso valore e si riferirà perciò allo stesso array:



Alla chiamata **BubbleSort(a)** il valore di **a** viene ricopiato su **v**,
quindi dentro il metodo, **a** si riferisce allo stesso array
(riferendosi ad **a[i]** è come ci si riferisse a **v[i]**)

9.8. Passaggio dei parametri per riferimento

Come si diceva, per far sì che il metodo *Scambia* si comporti come vogliamo, scambiando i valori delle variabili con cui viene chiamato, è necessario ricorrere al *passaggio per riferimento*.

In C# ciò si attua utilizzando la parola chiave **ref** che va inserita sia nell'intestazione del metodo (prima del tipo del parametro), sia nella chiamata al metodo: in questo modo i parametri formali si riferiranno ai parametri effettivi (sono sostanzialmente altri nomi per identificare i parametri effettivi).

La versione corretta del metodo *Scambia* è quindi la seguente:

```
static void Scambia(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
static void Main(string[] args)
{
    int x = 5, y = 12;

    Scambia(ref x, ref y);
    // ora x vale 12 e y vale 5
}
```

Come sappiamo prima di poter utilizzare una variabile è obbligatorio che abbia ricevuto almeno una volta un valore. Questo vale anche per i parametri: nel precedente esempio se *x* e/o *y* non avessero un valore iniziale, il compilatore segnalerebbe un errore di variabile non assegnata.

In alcuni casi si ha l'esigenza di scrivere un metodo che restituisca più valori ed è quindi il metodo ad avere il compito di valorizzare i parametri per la prima volta: in questo caso si parla di *parametri di uscita* e C# prevede a questo scopo la parola chiave **out**.

Si badi bene che si tratta sempre di un passaggio per riferimento; semplicemente non è necessario che il parametro abbia un valore prima della chiamata al metodo.

Il passaggio con **out** è stato utilizzato parecchie volte ad esempio con il metodo **int.TryParse**.

Vediamo un programma completo con un metodo per risolvere equazioni di secondo grado che usa due parametri di uscita per memorizzare le due radici.

Notiamo che il metodo dovrà obbligatoriamente valorizzare i due parametri **out** (per questo, abbiamo impostato inizialmente a 0 le due soluzioni):

```
class Program8
{
    static bool EquazioniSecondoGrado(double a, double b, double c,
                                       out double x1, out double x2)
    {
        double delta = b * b - 4 * a * c;
        x1 = 0; x2 = 0;
        if (delta >= 0)
        {
            x1 = (-b + Math.Sqrt(delta)) / (2 * a);
            x2 = (-b - Math.Sqrt(delta)) / (2 * a);
            return true;
        }
        else
            return false;
    }

    static void Main(string[] args)
    {
        double a, b, c, radice1, radice2;

        Console.WriteLine("Soluzioni dell'equazione a*x^2 + b*x + c = 0");
        Console.WriteLine("Inserire i coefficienti a, b e c");
        Console.Write("a = ");
        a = Convert.ToDouble(Console.ReadLine());
        Console.Write("b = ");
        b = Convert.ToDouble(Console.ReadLine());
        Console.Write("c = ");
        c = Convert.ToDouble(Console.ReadLine());

        if (EquazioniSecondoGrado(a, b, c, out radice1, out radice2))
            Console.WriteLine($"Soluzioni = {radice1}, {radice2}");
        else
            Console.WriteLine($"Equazione impossibile");
    }
}
```

Visto che i parametri **out** vengono calcolati e valorizzati dal metodo che li riceve, non dovrebbero nemmeno essere valorizzati dal chiamante. Per questo motivo, da C# 7.0^[56] è possibile dichiarare direttamente le variabili **out** nell'elenco degli argomenti della chiamata al metodo (*dichiarazione in linea*) anziché dichiararli separatamente.

Il codice prodotto risulta più compatto e leggibile e viene evitata l'assegnazione accidentale di un valore alla variabile prima della chiamata al metodo. Nel codice precedente, avremo quindi potuto omettere la dichiarazione delle variabili *radice1* e *radice2* e inglobarla direttamente nella chiamata al metodo che diventa:

```
EquazioniSecondoGrado(a, b, c, out double radice1, out double radice2);
```

Inoltre, sempre dalla versione 7.0, è possibile “*dimenticare*” alcuni parametri **out** di un metodo specificando un carattere *underscore* “_” dopo la parola chiave **out**.

Può essere utile se non abbiamo interesse ad utilizzare tutti i parametri **out** prodotti dal metodo. Ad esempio la seguente chiamata al metodo *EquazioniSecondoGrado* ignora la seconda radice calcolata dal metodo:

```
EquazioniSecondoGrado(a, b, c, out radice1, out _); // ignora il secondo parametro
```

Un terzo modo per passare i parametri per riferimento è usare la parola chiave **in**, introdotta da C# 7.2^[57]. Questa specifica che il parametro è un *parametro di ingresso* e il comportamento è come per la parola chiave **ref** con la differenza che il parametro formale non può essere modificato dal metodo, in altre parole è un *parametro ref in sola lettura*. Diversamente da **ref** e **out** non è obbligatorio specificare la parola chiave **in** nella chiamata al metodo.

Può essere utile se vogliamo assicurarci che il metodo non possa manipolare i parametri ricevuti, cosa che sarebbe possibile sia con il passaggio per valore che con i passaggi per riferimento con **ref** e **out**.

Inoltre l'uso dei parametri **in**, al posto di parametri passati per valore, ha prestazioni migliori in programmi che richiamano ciclicamente uno stesso metodo.

Riassumiamo, nella seguente tabella, le caratteristiche delle parole chiavi **in**, **ref** e **out**:

in	out	ref
-----------	------------	------------

<i>in</i>	<i>out</i>	<i>ref</i>
passaggio per riferimento in sola lettura	passaggio per riferimento per parametri di uscita	passaggio per riferimento per parametri sia di ingresso che uscita
i parametri con i quali si chiama il metodo devono essere inizializzati	non è richiesta l'inizializzazione dei parametri con i quali si chiama il metodo	i parametri con i quali si chiama il metodo devono essere inizializzati
la parola chiave in è opzionale quando si chiama il metodo	la parola chiave out è obbligatoria nella chiamata al metodo	la parola chiave ref è obbligatoria nella chiamata al metodo
non è necessario usare il parametro all'interno del metodo	è necessario assegnare un valore al parametro all'interno del metodo (anche se il parametro effettivo lo aveva già)	non è necessario usare il parametro all'interno del metodo
i parametri in non possono essere dichiarati in linea	i parametri out possono essere dichiarati in linea	i parametri ref non possono essere dichiarati in linea

Osservazione importante: diversamente dal passaggio per valore, non possiamo passare per riferimento (con **in**, **out** o **ref**) costanti letterali o espressioni, perché queste non sono identificate e quindi ad esse non è associato un indirizzo in memoria.

Si tenga infine presente che, nonostante le parole chiave **in**, **ref** e **out** determinino un comportamento diverso in fase di esecuzione, non sono considerate parte della firma del metodo in fase di compilazione. Non è quindi possibile definire più metodi con lo stesso nome differenziandoli solo con uno di questi tre modificatori.

9.9. Commenti di documentazione

Apriamo una piccola parentesi sulla documentazione del codice.

Sappiamo quanto sono importanti i commenti per il programmatore e come possiamo inserirli nel codice: come in tanti altri linguaggi possiamo usare la doppia barra // per il commento in riga oppure la coppia /* e */ per il commento di un blocco di righe.

Visual Studio consente di aggiungere in testa ad un metodo anche un **commento di documentazione**: sono commenti che vengono utilizzati dall'editor di codice per essere mostrati, durante la digitazione, quando viene usato il metodo [\[58\]](#).

Inoltre possono essere estratti dal codice sorgente e usati per generare la documentazione di un progetto; a tal scopo esistono strumenti per la generazione automatica della documentazione come l'applicativo open source **Doxxygen** [\[59\]](#).

Per inserire un commento di documentazione possiamo sfruttare gli automatismi dell'IDE: una volta completata la scrittura del metodo (o almeno dell'intestazione del metodo) possiamo inserire tre barre, ///, subito sopra l'intestazione e premendo INVIO l'editor genererà la struttura del commento.

Proviamo la funzione con l'ultimo metodo visto per la risoluzione di un'equazione di secondo grado; l'IDE genera automaticamente la sequenza di commenti seguenti:

```
/// <summary>
///
/// </summary>
/// <param name="a"></param>
/// <param name="b"></param>
/// <param name="c"></param>
/// <param name="x1"></param>
/// <param name="x2"></param>
static void EquazioniSecondoGrado(double a, double b, double c,
                                    out double x1, out double x2)
```

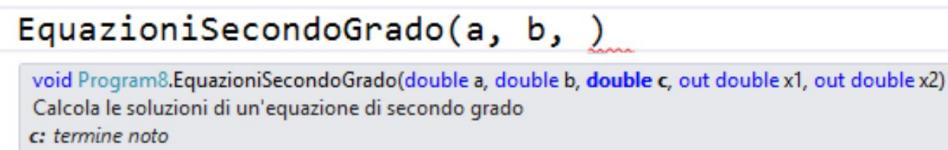
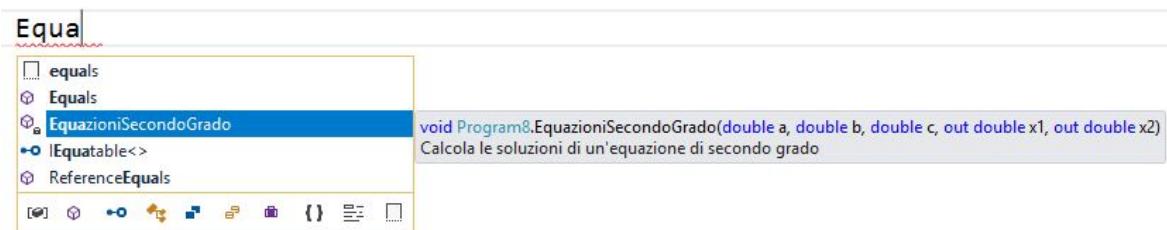
iniziando con una doppia barra sono considerati dal compilatore normali commenti ma possono essere sfruttati con **IntelliSense** oppure per generare, con sistemi automatici, la documentazione del codice.

Notiamo che vengono usati speciali marcatori per specificare le varie caratteristiche del metodo (il linguaggio usato è basato su **XML**): ad esempio <**summary**> è un tag in cui inseriremo la descrizione del metodo, i tag <**param**> indicano i vari parametri, e così via^[60].

Completiamo quindi il commento:

```
///<summary>
/// Calcola le soluzioni di un'equazione di secondo grado
///</summary>
///<param name="a">coefficiente di x al quadrato</param>
///<param name="b">coefficiente di x</param>
///<param name="c">termine noto</param>
///<param name="x1">prima soluzione</param>
///<param name="x2">seconda soluzione</param>
```

Quando useremo il metodo, come mostrato nelle seguenti immagini, il sistema di completamento dell'editor ci proporrà il metodo con i commenti descrittivi inseriti:



9.10. Parametri denominati e parametri opzionali

Da C# 4.0^[61] è stata introdotta la possibilità di specificare il nome dei parametri formali quando un metodo viene invocato. Questo rende il codice più leggibile e consente di specificare i parametri in un ordine diverso rispetto a come sono elencati nella firma del metodo.

Consideriamo ad esempio il seguente metodo per il calcolo dell'area di trapezi:

```
static double AreaTrapezio(double baseMaggiore, double baseMinore,  
                           double altezza)  
{  
    return (baseMaggiore + baseMinore) * altezza / 2;  
}
```

il metodo potrebbe essere chiamato con un ordine completamente diverso dei parametri come mostrato di seguito:

```
double area = AreaTrapezio(altezza: 3, baseMaggiore: 5, baseMinore: 4);
```

Si possono anche mescolare parametri posizionali con parametri denominati ma in tal caso è necessario specificare per ultimi i parametri denominati. Ad esempio, la seguente chiamata è perfettamente lecita:

```
area = AreaTrapezio(4, 5, altezza: 3);
```

mentre non lo è la seguente perché non sarebbe chiaro a quali parametri si riferiscono il secondo e terzo valore:

```
area = AreaTrapezio(altezza: 3, 5, 4);
```

Sempre da C# 4.0, nella definizione di un metodo possiamo specificare dei valori di default per alcuni o tutti i parametri; questi parametri diventano **opzionali** e il metodo potrà essere chiamato senza specificare i valori di questi parametri che assumeranno i valori di default.

Il seguente metodo potrebbe essere utilizzato per ordinare un vettore in modo crescente o decrescente in base al valore del secondo parametro:

```
static void Ordina(int[] a, bool crescente = true)  
{  
    // codice per l'ordinamento ...  
}
```

una chiamata al metodo, per ordinare in modo crescente un vettore, potrebbe essere quindi la seguente:

Ordina(vettore);

Analogamente ai parametri denominati, nel caso siano presenti sia parametri obbligatori che parametri opzionali, i parametri opzionali vanno indicati per ultimi nella firma del metodo.

9.11. Modificatore params

È possibile scrivere metodi che ricevono un numero variabile di parametri aggiungendo il modificatore **params** all'ultimo parametro di un metodo.

Il modificatore può presentarsi una sola volta e il parametro a cui è applicato deve essere un array.

Ad esempio potremo scrivere un metodo per sommare un numero variabile di valori interi:

```
static int Sommatoria(params int[] a)
{
    int somma = 0;
    foreach (int valore in a)
        somma += valore;
    return somma;
}
```

Il metodo *Sommatoria* può essere richiamato con qualsiasi numero di valori interi (anche 0) oppure con un array precedentemente creato o allocato in linea; ecco alcuni esempi di chiamate:

```
int s;
s = Sommatoria(1, 2, 5, 6); // s vale 14
s = Sommatoria();          // s vale 0
s = Sommatoria(new int[] { 1, 2, 3 }); // s vale 6
```

Proponiamo un ulteriore esempio di un metodo che determina il massimo tra un numero qualsiasi di valori interi:

```
static int Massimo(params int[] a)
{
    if (a.Length == 0)
        throw new ArgumentException("deve essere specificato almeno un valore");

    int max = a[0];
    foreach (int valore in a)
        if (valore > max)
            max = valore;
    return max;
}
```

9.12. Metodi con corpo di espressioni

La versione 6.0 ha introdotto i *corpi di espressione* o *espressioni corpo (expression-bodied)* ovvero la possibilità di definire sinteticamente metodi composti da un'unica espressione, ad esempio una funzione con una sola istruzione *return* oppure una procedura contenente una sola istruzione.

La sintassi fa uso della “freccia” => e prevede la scrittura della firma del metodo seguita dalla freccia e quindi dall'espressione.

Ecco un paio di esempi applicati a metodi statici (ma lo stesso si può fare per metodi non statici^[62]):

```
// esempio di procedura definita con corpo di espressione
public static void Saluta() => Console.WriteLine("Salve!");
```

```
// esempio di funzione definita con un corpo di espressione
public static bool Pari(int n) => n % 2 == 0;
```

si noti che per le funzioni non va indicata la parola chiave **return** ma solo l'espressione da ritornare. Inoltre non possono essere specificate parentesi graffe per includere più istruzioni; si possono inserire esclusivamente espressioni compatibili con la firma del metodo.

Questa come altre caratteristiche costituiscono il cosiddetto “**zucchero sintattico**” (*syntactic sugar*) ovvero caratteristiche sintattiche che non introducono nuove funzionalità ma aumentano solo la capacità espressiva di un linguaggio di programmazione per facilitare e/o rendere più sintetica la scrittura del codice.

Questi nuovi modi di scrittura del codice vengono ricondotti dal compilatore a strutture fondamentali già disponibili.

Per rendersi conto di come vengano tradotte si può far uso di tools per esaminare il linguaggio intermedio (IL) generato dal compilato. Allo scopo si possono ad esempio usare decompilatori per C# come **dotPeek**^[63] e **ILSpy**^[64]: con strumenti di questo tipo si può decostruire lo *zucchero sintattico* e capire come le nuove strutture sintattiche vengano trattate dal compilatore.

9.13. Metodi di estensione

I metodi di estensione, introdotti da C# 3.0, consentono di estendere con nuovi metodi una classe statica, senza aver accesso e dover quindi modificare l'implementazione della classe stessa.

Un metodo di estensione è quindi un metodo statico che aggiunge nuove funzionalità alla classe statica.

La sintassi per definire un metodo di estensione prevede di usare un primo parametro marcato con la parola chiave **this** e del tipo della classe, seguito da eventuali altri parametri.

Inoltre i metodi di estensione vanno definiti all'interno di una classe che deve essere dichiarata statica.

Ad esempio potremo definire la seguente classe che implementa due funzioni non previste per le stringhe: la prima per calcolare il numero di vocali di una stringa e la seconda che determina il numero di parole in una stringa considerando come separatori singoli o sequenze di spazi.

```
public static class EstensioniString
{
    public static int ContaVocali(this string s)
    {
        int conta = 0;
        foreach (char c in s.ToUpper())
            if ("AEIOU".Contains(c.ToString()))
                conta++;
        return conta;
    }

    public static int ContaParole(this string s)
    {
        int conta = 0;
        if (s != "")
        {
            conta = 1;
            for (int i = 0; i < s.Length; i++)
                if (i > 0 && s[i] == ' ' && s[i - 1] != ' ')
                    conta++;
        }
        // controllo che la stringa termini con degli spazi
        if (s[s.Length - 1] == ' ')
            conta--;
    }
    return conta;
}
```

{}
{}

Così facendo avremo a disposizione di ogni variabile stringa i nuovi metodi per il conteggio.

Di seguito un esempio di codice che le sfrutta:

```
string frase = " La donzelletta vien dalla campagna ";
```

```
WriteLine("Numero di vocali = " + frase.ContaVocali());  
WriteLine("Numero di parole = " + frase.ContaParole());
```

Ciò non toglie che i metodi di estensione possano essere utilizzati come dei normali metodi statici passando come parametro la stringa da esaminare:

```
WriteLine("Numero di vocali = " + EstensioniString.ContaVocali(frase));  
WriteLine("Numero di parole = " + EstensioniString.ContaParole(frase));
```

9.14. Metodi annidati

Dalla versione 7.0 di C# è possibile definire **metodi annidati** all'interno di altri metodi; vengono anche chiamati **metodi locali**. Un metodo locale così definito è utilizzabile solo all'interno del metodo in cui è definito ed ha accesso alle variabili locali definite nel metodo che lo contiene.

Potrebbe essere utile per rendere più chiaro lo sviluppo del codice definendo metodi “di servizio” che hanno ragione di essere usati solo all'interno di un certo metodo.

Riportiamo di seguito un esempio di metodo che calcola la distanza minima di una serie di punti dall'origine; le coordinate **x** e **y** dei punti vengono passate, rispettivamente, attraverso due array. All'interno definiamo un metodo locale che restituisce la distanza tra un determinato punto e l'origine.

```
public static double DistanzaMinima(double[] puntiX, double[] puntiY)
{
    if (puntiX.Length != puntiY.Length)
        throw new ArgumentException("gli array con le coordinate X e Y devono avere la stessa
dimensione");

    double Distanza(int indicePunto) => Math.Sqrt(puntiX[indicePunto] *
        puntiX[indicePunto] + puntiY[indicePunto]*puntiY[indicePunto]);

    double dMin = 0;
    for (int i = 0; i < puntiX.Length; i++)
    {
        double d = Distanza(i);
        if (d < dMin)
            dMin = d;
    }
    return dMin;
}
```

Il metodo **Distanza**, essendo formato da un'unica istruzione, è definito attraverso un'espressione corpo. Può accedere alle variabili locali e ai parametri del metodo *DistanzaMinima*. Attenzione che non potrebbe accedere all'indice **i** del ciclo in quanto questo ha un ambito di visibilità limitato al ciclo e non all'intero metodo.

Un metodo locale segue le *regole di scope* già viste: non può cioè usare come nomi dei parametri o delle variabili locali, identificatori già utilizzati nel metodo in cui è contenuto.

I metodi locali annidati ad un metodo possono essere definiti in qualunque punto del metodo anche dopo il loro utilizzo, sempre che non accedano a variabili dichiarate dopo.

Dalla versione 8.0 di C# (disponibile in preview) è stata aggiunta la possibilità di utilizzare il modificatore **static** in testa al metodo locale. In questo modo si limita il metodo impedendo l'accesso alle variabili locali.

10. Algoritmi ricorsivi

Abbiamo visto che un metodo può chiamare altri metodi per eseguire delle sotto attività.

Ma un metodo può chiamare anche se stesso: in tal caso si parla di **metodo ricorsivo**.

Usare la tecnica ricorsiva si presta bene alla risoluzione di certi problemi difficilmente risolvibili in modo iterativo. Alcuni problemi possono cioè essere espressi in termini del problema medesimo applicato a dati diversi.

Partiamo da un semplice esempio di un problema che abbiamo già risolto iterativamente (cioè con un ciclo): la somma dei primi **n** numeri naturali.

Supponiamo di avere una funzione **Somma(n)** che calcola la somma dei primi **n** numeri. Tale funzione può essere espressa in termini di sé stessa nel seguente modo:

$$\text{Somma}(n) = \text{Somma}(n-1) + n$$

cioè la somma dei primi **n** numeri è data dalla somma fino a **n-1** a cui va addizionato **n**. Per come l'abbiamo definita, anche **Somma(n-1)** può essere espressa seguendo la stessa regola come:

$$\text{Somma}(n-1) = \text{Somma}(n-2) + n-1$$

possiamo procedere in questo modo fino ad arrivare ad una chiamata **Somma(0)** che restituirà ovviamente 0. Sostanzialmente quindi abbiamo definito la funzione **Somma** nel seguente modo:

$$\begin{aligned} & \text{Somma}(n-1) + n && \text{se } n > 0 \\ \text{Somma (n)} = & \\ & 0 && \text{altrimenti} \end{aligned}$$

Possiamo tradurre pari pari la funzione precedente in un metodo C#:

```
static int Somma(int n)
{
```

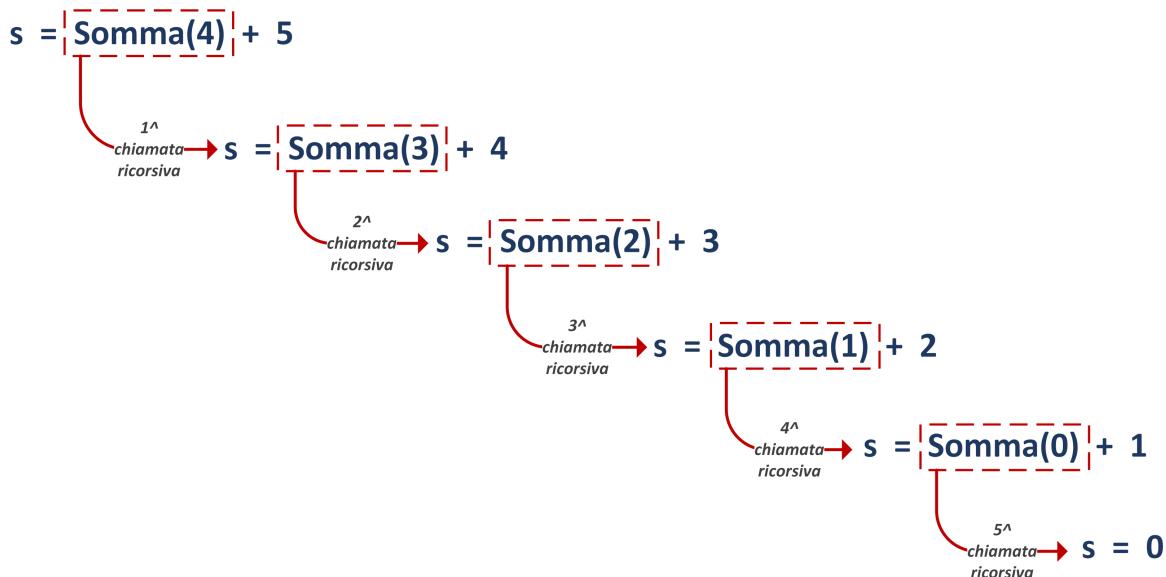
```

int s;
if (n > 0)
    s = Somma(n - 1) + n;
else
    s = 0;
return s;
}

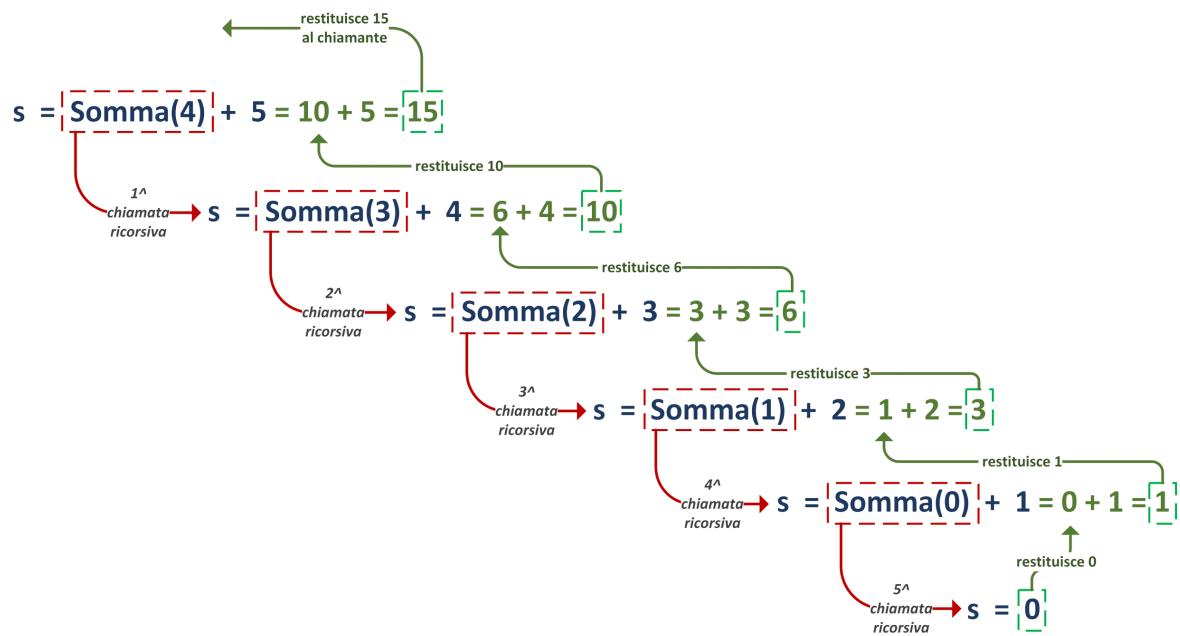
```

quindi con la definizione ricorsiva della funzione **Somma** non abbiamo più bisogno di un ciclo per calcolare la somma ma le chiamate che vengono innescate vanno ad accumulare la somma fino a calcolarla per tutti gli **n** numeri.

Possiamo simulare l'esecuzione delle chiamate con **n=5**: la funzione viene chiamata ricorsivamente fintantoché **n** si mantiene positivo. Ecco raffigurata di seguito la sequenza delle chiamate:



quando si arriva all'ultima chiamata, **Somma(0)**, viene eseguito il ramo *else* della condizione che assegna **0** alla variabile **s** e la restituisce. A questo punto il valore restituito va a completare il calcolo della chiamata **Somma(1)** che produce un valore per **s** pari a **1** e lo restituisce; il processo continua a ritroso fino a giungere alla chiamata iniziale. Nella seguente figura è mostrata la situazione:



Ogni chiamata ricorsiva della funzione utilizza una variabile locale `s` a sé stante; si creano cioè in memoria tante variabili `s` quante sono le chiamate ricorsive.

È chiaro quindi che la soluzione ricorsiva al problema è più dispendiosa, in termini di memoria, rispetto alla soluzione iterativa con un ciclo che aggiorni un “accumulatore” per calcolare la somma.

10.1. La successione di Fibonacci

Affrontiamo un secondo problema, un classico calcolo che può essere effettuato in modo ricorsivo: la determinazione dell'n-esimo numero della **successione di Fibonacci**.^[65]

Si tratta di una successione numerica scoperta nel tredicesimo secolo del matematico pisano Leonardo Fibonacci. La successione è costituita dai seguenti elementi:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \dots$$

ogni numero della successione, a partire dal terzo, è dato dalla somma dei due numeri che lo precedono.

La successione è stata fornita dal matematico come soluzione al problema proposto dall'imperatore Federico II di Svevia nel 1223 in un torneo di matematica. Il problema era quello di determinare come cresce una popolazione di conigli assumendo che una coppia di conigli possa diventare fertile dopo un mese dalla nascita e dia alla luce una nuova coppia dopo un altro mese.

Ecco quindi che se inizialmente partiamo da una coppia di conigli appena nati (1, primo elemento della successione), dopo un mese avremo ancora una coppia (1, secondo elemento) ma dopo due mesi avremo 2 coppie di conigli, al terzo mese avremo 3 coppie (le 2 coppie del mese precedente e 1 nuova coppia concepita dalla coppia più anziana), al quarto mese avremo 3+2 coppie e così via.

La successione può essere espressa naturalmente in modo ricorsivo nel seguente modo:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \text{ se } n > 1$$

La traduzione in C# è immediata:

```
static int Fibonacci(int n)
{
    int fib;
    if (n <= 0)
        fib = 0;
    else
        if (n == 1)
            fib = 1;
        else
            fib = Fibonacci(n-1) + Fibonacci(n - 2);

    return fib;
}
```

In una procedura ricorsiva individuiamo quindi sempre due azioni:

- il *passo base* che determina la soluzione direttamente senza far ricorso alle soluzioni dello stesso problema per altri input
- il *passo ricorsivo* che definisce la soluzione in termini della soluzione del problema.

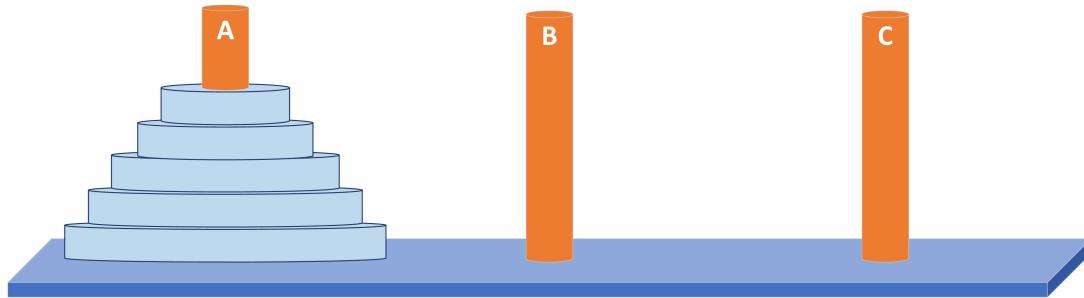
10.2. La torre di Hanoi

Il rompicapo della *torre di Hanoi* è un classico problema la cui soluzione ricorsiva risulta di gran lunga più semplice è intuitiva rispetto ad una versione iterativa.

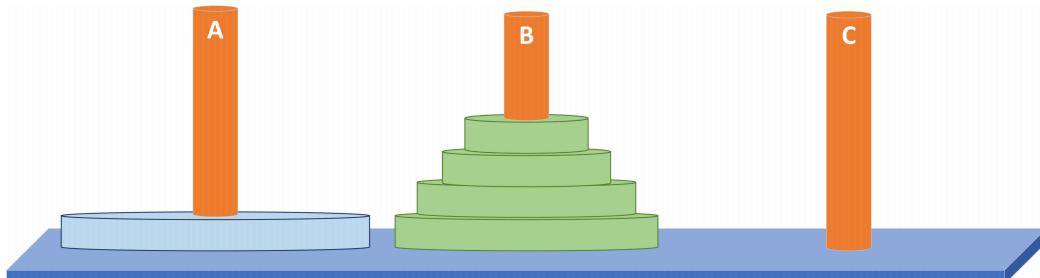
Il rompicapo, inventato alla fine del 1800, è costituito da tre pioli nei quali sono impilati dei dischi con diametri via via maggiori: inizialmente i dischi sono inseriti nel primo piolo e l'obiettivo del gioco è spostarli nel terzo piolo utilizzando il piolo intermedio come supporto. Lo spostamento deve seguire due semplici regole:

- si deve spostare un disco alla volta
- un disco può essere posizionato sopra un altro disco solo se ha un diametro inferiore.

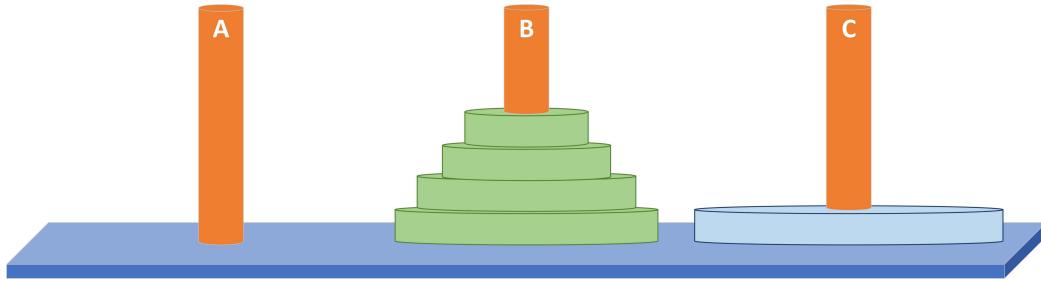
La situazione con cinque dischi è rappresentata dalla seguente figura^[66]:



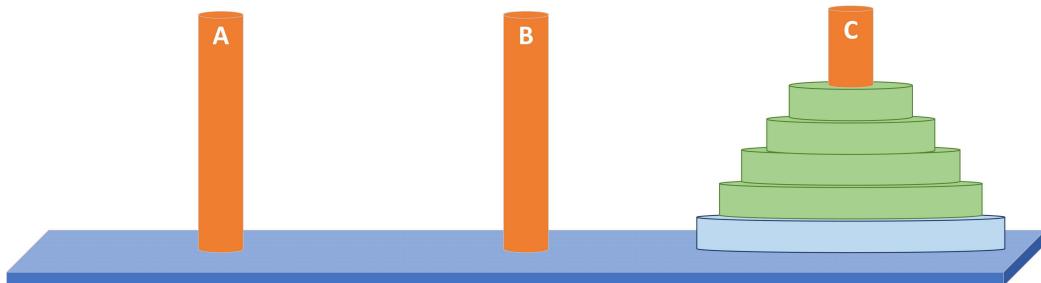
Potremo affrontare il problema riducendolo allo spostamento dei primi quattro dischi dal piolo A al piolo B. Immaginiamo cioè di aver già risolto il rompicapo per 4 dischi:



così facendo avremo liberato il disco più grande del piolo A, che potremo così spostare con una sola mossa sul piolo C:



a questo punto possiamo spostare la torre di quattro dischi che abbiamo “parcheggiato” nel piolo B sul piolo di destinazione C:



Notiamo che lo spostamento della pila di quattro dischi può avvenire sfruttando tutti i tre pioli compreso il piolo che ospita il disco più grande: uno spostamento di uno qualsiasi degli altri dischi sopra di questo rispetta sempre la seconda regola del gioco.

Cosa abbiamo ottenuto? Abbiamo scomposto il problema al problema di spostare quattro dischi, nella prima fase dal piolo A al piolo B usando come piolo di supporto il piolo C: quest’ultimo può essere affrontato allo stesso modo scomponendolo nei tre passi seguenti:

1. sposta 3 dischi dal piolo A al piolo C
2. sposta un disco dal piolo A al piolo B
3. sposta 3 dischi dal piolo C al piolo B

Anche in questo caso il problema di spostare 3 dischi può essere espresso in termini del problema di spostare 2 dischi.

In definitiva possiamo esprimere in modo ricorsivo il problema più generale di spostare **n** dischi dal piolo A al piolo B, nel seguente modo:

1. sposta $n-1$ dischi dal piolo A al piolo B usando come piolo di supporto il piolo C
2. sposta un disco (l’unico disco rimasto) dal piolo A al piolo C
3. sposta $n-1$ dischi dal piolo B al piolo C usando come piolo di supporto il piolo A.

Questo schema può essere tradotto pari pari nella seguente procedura ricorsiva:

```
// sposta n dischi da pioloA a pioloC usando come piolo di supporto pioloB
static void Hanoi(int n, int pioloA, int pioloC, int pioloB)
{
    if (n > 0)
    {
        Hanoi(n - 1, pioloA, pioloB, pioloC);
        WriteLine($"spostamento {pioloA} -> {pioloC}");
        Hanoi(n - 1, pioloB, pioloC, pioloA);
    }
}
```

Il numero minimo di mosse m_n necessarie per spostare n dischi è pari a $2^n - 1$, infatti:

- per $n = 1$, $m_1 = 2^1 - 1 = 1$
- per $n = 2$, $m_2 = 2^2 - 1 = 3$
- per $n = 3$, dello schema di soluzione possiamo osservare che $m_3 = m_2 + 1 + m_2 = 2^2 - 1 + 1 + 2^2 - 1 = 2 * 2^2 - 1 = 2^3 - 1 = 7$
- in generale quindi è sempre $m_n = m_{n-1} + 1 + m_{n-1} = 2^{n-1} - 1 + 1 + 2^{n-1} - 1 = 2 * 2^{n-1} - 1 = 2^n - 1$

10.3. Ricerca binaria ricorsiva

Anche l'algoritmo di ricerca binaria studiato nel capitolo 8 può essere rivisto in modo ricorsivo. In effetti confrontando l'elemento da cercare con quello centrale nell'array (che ricordiamo deve essere ordinato), possiamo rimandare la ricerca alla parte sinistra dell'array se l'elemento da cercare è minore dell'elemento centrale e alla parte destra se l'elemento è maggiore dell'elemento centrale.

```
static bool RicercaBinaria(int[] v, int n, int sx, int dx)
{
    bool trovato = false;
    if (sx <= dx)
    {
        int centro = (sx + dx) / 2;
        if (v[centro] == n)
            trovato = true;
        else
            if (n < v[centro])
                trovato = RicercaBinaria(v, n, sx, centro - 1);
            else
                trovato = RicercaBinaria(v, n, centro + 1, dx);
    }
    return trovato;
}
```

In questa versione, il metodo riceve gli indici **sx** e **dx**, che indicano da quale a quale elemento considerare. Per ricercare nell'intero array il metodo dovrà essere chiamato con 0 come valore per sx e a.Length-1 come valore per dx.

Il ciclo della versione iterativa viene quindi sostituito dalle due chiamate ricorsive che vengono scelte in base a dove si posiziona il numero da cercare rispetto all'elemento centrale: se l'elemento da cercare è minore la ricerca proseguirà tra gli elementi di indici sx e centro-1 mentre se è maggiore, proseguirà tra gli elementi di indice centro+1 e dx.

10.4. Algoritmo di ordinamento per fusione (*merge sort*)

Esaminiamo ora l'**algoritmo di ordinamento per fusione (*merge sort*)**, un metodo di ordinamento particolarmente efficiente che sfrutta la ricorsione con una tecnica algoritmica chiamata *divide et impera*, dividi e conquista^[67]:

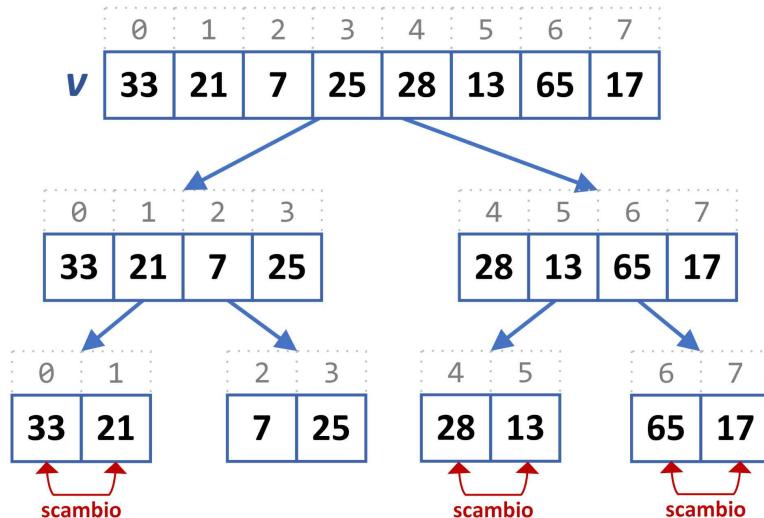
- **dividi**: si divide il problema in un certo numero di sottoproblemi che sono istanze dello stesso problema ma di dimensione inferiore, e
- **conquista**: si risolvono ricorsivamente i sottoproblemi e quando il sottoproblema è abbastanza piccolo lo si risolve direttamente. Si usano le soluzioni dei sottoproblemi per determinare la soluzione del problema dato.

L'idea è quella di dividere a metà l'array da ordinare: se immaginiamo di ordinare le due parti in modo indipendente, possiamo poi applicare l'algoritmo di fusione visto nel capitolo 8 per ottenere l'ordinamento dell'intero array.

A questo punto l'ordinamento di ognuna delle due parti può essere risolta a sua volta allo stesso modo: dividiamo in due l'array e ordiniamo le due parti per poi ricomporre l'array fondendo le due sotto parti ordinate.

Quando arriveremo ad un sotto array di soli due elementi l'algoritmo di fusione non fa altro che posizionare gli elementi nel giusto ordine.

Di seguito un'immagine esemplificativa con un array di 8 elementi:



Quindi riepilogando:

- dividiamo l'array in due parti
- richiamiamo ricorsivamente l'algoritmo su ognuna delle due parti
- ricomponiamo l'array originario fondendo ordinatamente le due parti ordinate.

Per realizzare l'algoritmo abbiamo bisogno quindi di un metodo per fondere assieme due parti ordinate dello stesso array; è una variante dello stesso metodo visto nel capitolo 8 che lavora però sul medesimo array anziché su due array distinti.

Di seguito l'implementazione del metodo:

```
// Fonde ordinatamente le due partizioni dell'array da sx a m e da m+1 a dx
// (0 <= sx <= m <= dx < n, dove n sono le dimensioni dell'array)
static void Merge(int[] v, int sx, int m, int dx)
{
    int i = sx, j = m + 1, k = 0;
    if (sx >= 0 && sx <= m && m <= dx && dx < v.Length)
    {
        int[] supporto = new int[dx - sx + 1];
        while (i <= m && j <= dx)
        {
            if (v[i] < v[j]) supporto[k++] = v[i++];
            else supporto[k++] = v[j++];
        }
        // ricopia la parte restante
        if (i <= m)
            while (i <= m)
                supporto[k++] = v[i++];
        else
            while (j <= dx)
                supporto[k++] = v[j++];
        // ricopia tutto l'array di supporto nell'array originario a partire dalla posizione sx
        supporto.CopyTo(v, sx);
    }
}
```

il metodo usa un array di supporto all'interno del quale ricopiare gli elementi del vettore originale **v**.

Con questo metodo a disposizione l'implementazione del merge sort è immediata:

```
static void MergeSort(int[] v, int sx, int dx)
{
    if (sx < dx)
    {
        int m = (sx + dx) / 2; // calcola l'indice di mezzo

        MergeSort(v, sx, m); // chiamata ricorsiva con la prima metà
        MergeSort(v, m + 1, dx); // chiamata ricorsiva con la seconda metà

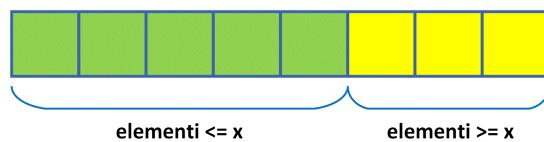
        Merge(v, sx, m, dx); // fusione ordinata delle due metà
    }
}
```


10.5. Algoritmo di ordinamento per partizione (quick sort)

Un algoritmo di ordinamento sul posto efficiente quanto il merge sort è stato sviluppato nel 1962 dall'informatico britannico Tony Hoare ed è comunemente noto con il nome **quick sort** in ragione delle sue prestazioni.

È l'algoritmo più efficiente per l'ordinamento sul posto di un array generico^[68] ed ha valso al suo inventore il premio Turing^[69] nel 1980.

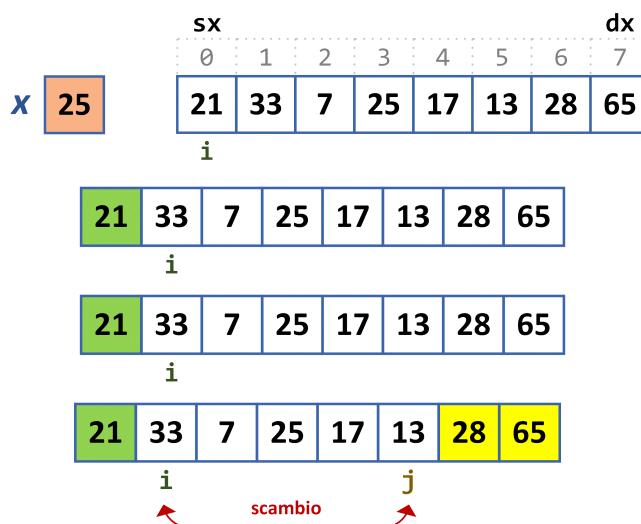
Il metodo si basa sulla creazione di un partizionamento dell'array in due parti aventi la caratteristica di contenere, la prima elementi non superiori ad un elemento x , scelto come perno (o *pivot*) e la seconda, elementi non inferiori ad x :



così facendo avremo ottenuto che le due partizioni sono tra loro ordinate. Applichiamo quindi ricorsivamente l'algoritmo alle due partizioni fino a quando è possibile cioè fino a quando le partizioni non sono formate da un unico elemento.

Scelto un elemento x qualsiasi come perno (il valore che funge da perno può essere l'elemento centrale dell'array o il primo elemento), la tecnica per partizionare l'array è quella di scorrere l'array da sinistra fino a trovare elementi minori di x e da destra fino a trovare elementi maggiori di x .

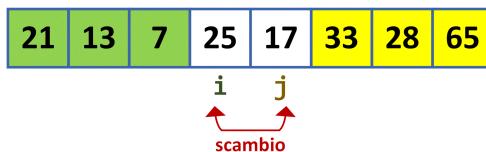
Ecco un'immagine che rappresenta questa prima fase su un array di esempio in cui viene scelto come valore del perno l'elemento centrale:



i e j sono i contatori usati per scorrere, rispettivamente da sinistra e da destra l'array. Lo scorrimento da sinistra terminerà appena troviamo un elemento $\geq x$ e da destra su un elemento $\leq x$; possiamo quindi scambiare i due elementi indicati da i e j per allargare le due partizioni, incrementando al contempo i e decrementando j :



proseguiamo in modo analogo con i e j :



e quindi, come prima, scambiamo gli elementi che hanno interrotto la scansione:



il processo continua fino a quando i e j si scavalcano. A questo punto avremo ottenuto la partizione voluta: dagli indici sx a j avremo elementi minori o uguali a x mentre da i a dx tutti elementi maggiori o uguali ad x .

Possiamo applicare lo stesso metodo alle due partizioni ovvero chiamare ricorsivamente quick sort da sx a j e da i a dx .

Riportiamo di seguito il metodo completo:

```
static void QuickSort(int[] v, int sx, int dx)
{
    // partiziona v
    int i = sx, j = dx;
    int x = v[(sx+dx) / 2];
    while (i <= j)
    {
        // percorro da sx finchè trovo elementi < x
        while (v[i] < x) i++;

        // percorro da dx finchè trovo elementi > x
        while (v[j] > x) j--;

        // a questo punto, v[i] >= x e v[j] <= x, li scambio
        if (i <= j)
        {
            int temp = v[i]; v[i] = v[j]; v[j] = temp;
            i++; j--;
        }
    }
}
```

```
}

if (sx < j) QuickSort(v, sx, j);
if (i < dx) QuickSort(v, i, dx);

}
```

10.6. Algoritmi di backtracking

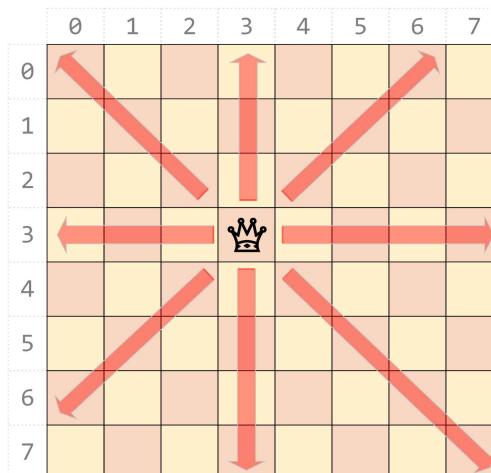
In alcuni problemi nasce l'esigenza di scrivere algoritmi che non seguono una regola fissa di calcolo ma procedano piuttosto per tentativi. Algoritmi di questo tipo sono applicati ad esempio ai programmi per giocare a scacchi o simili.

L'idea è quella di scomporre il problema in obiettivi parziali che possono essere espressi in termini ricorsivi e consistono nell'esplorazione di un numero finito di sotto obiettivi.

Questa tecnica di ricerca esaustiva (o di *forza bruta*, come viene anche chiamata) è detta **backtracking** (letteralmente “retrocedere”) e consente di provare sistematicamente tutte le strade che partono da un certo punto dopo che alcune di esse non portano da nessuna parte. Usando il backtracking possiamo sempre ritornare ad una posizione che offre altre possibilità per risolvere il problema con successo.

Un problema che si presta bene all'uso di questa tecnica di soluzione è il **“problema della otto regine”**: *“Otto regine devono essere poste su di una scacchiera in modo che nessuna di loro possa catturare le altre. Trovare tutte le soluzioni possibili.”*.

Nel gioco degli scacchi la regina può catturare tutti i pezzi che si trovano nella stessa riga, nella stessa colonna o nelle due diagonali.



Tornando al problema, è evidente quindi che potrà esserci un'unica regina su una stessa riga, colonna o diagonale.

Questo problema fu studiato dal grande matematico tedesco Carl Friedrich Gauss nel 1850, ma egli non riuscì a risolverlo completamente, appunto per il fatto che il problema non ammette una soluzione analitica, ma richiede piuttosto una paziente ricerca esaustiva.

Per determinare tutte le soluzioni procederemo quindi per tentativi. Il metodo generale sarà:

- generare le possibili soluzioni in modo sistematico;
- confrontarle con un criterio che caratterizzi la soluzione.

Utilizziamo una tecnica di backtracking, ovvero proviamo sistematicamente tutte le strade:

- posizioniamo una regina per riga, partendo dalla prima ed aggiungendo un'altra “sicura” nella riga successiva
- la prima regina è posizionata nella prima casella della prima riga e vi resta finché sono generate tutte le configurazioni “sicure” a partire da questa iniziale. Quindi essa viene spostata di una casella più a destra, e così via ...

Numeriamo le righe e colonne della scacchiera da 0 a 7 e scriviamo il seguente pseudocodice:

```
inizializza scacchiera  
per colonna0 da 0 a 7 esegui  
    posiziona una regina in (0, colonna0)  
    genera tutte le configurazioni con la regina 0 fissa  
    togli la regina da (0, colonna0)
```

la procedura “*genera tutte le configurazioni con la regina 0 fissa*” è a sua volta:

```
per colonna1 da 0 a 7 esegui  
    se (1, colonna1) è sicura allora  
        posiziona una regina in (1, colonna1)  
        genera tutte le configurazioni con le regine 0 e 1 fisse  
        togli la regina da (1, colonna1)
```

e così via

Questo suggerisce di costruire una procedura ricorsiva:

```
procedura GeneraSoluzioni(riga)  
per colonna da 0 a 7 esegui  
    se sicura(riga, colonna) allora
```

```
posiziona una regina in (riga, colonna)
se riga = 7 allora stampa la soluzione
altrimenti GeneraSoluzioni(riga+1)
togli regina da (riga, colonna)
```

Per implementare l'algoritmo precedente in C# dobbiamo prima decidere come rappresentare la scacchiera: la scelta più naturale potrebbe essere quella di utilizzare una matrice 8x8 di interi. Lo stato di ogni casella è rappresentato dal numero inserito nella cella; ad esempio potremo decidere di inizializzare a 0 tutte le celle e porre ad 1 una cella quando è posizionata una regina. È chiaro che ogni volta che viene posizionata una regina dovranno essere marcate come non disponibili anche le celle della riga, colonna e diagonali che interessano la regina. Per ogni regina posizionata o tolta da una cella, vanno quindi modificate molte altre celle.

Questo approccio non si rivela quindi molto efficiente: la scacchiera è vista infatti dalla prospettiva del giocatore che vede l'intera scacchiera e tutti i pezzi allo stesso tempo, ma, se focalizziamo la nostra attenzione soltanto sulle regine, possiamo considerare la scacchiera dalla loro prospettiva.

Per le regine, la scacchiera non è divisa in quadrati, ma in colonne, righe e diagonali. Se una regina è posta in una singola posizione, essa si trova non solo in tale posizione, ma anche sull'intera riga, colonna e diagonale.

Possiamo allora utilizzare tre array di booleani per indicare che una certa colonna e le due diagonali sono “bloccate” da una regina. Le diagonali principali, in numero di 15, hanno la proprietà che mantengono costante la differenza tra l'indice della riga e l'indice della colonna **r-c** che varia tra -7 e +7; per le diagonali secondarie rimane costante invece la somma tra gli indici di riga e colonna **r+c**, che varia tra 0 e 14. Avremo 15 diagonali principali e altrettante diagonali secondarie.

La singola soluzione può essere rappresentata con un array di 8 elementi nel quale l'elemento di indice *i* indica la colonna della *regina i-esima*.

Di seguito impostiamo il codice per dichiarare tutti gli array di cui abbiamo bisogno e il metodo per inizializzarli.

```
static int[] soluzione;
static bool[] colonne; // stato delle colonne
static bool[] diagonale1; // stato delle diagonali principali
static bool[] diagonale2; // stato delle diagonali secondarie
```

```

static void Inizializza()
{
    colonne = new bool[8];
    diagonale1 = new bool[15];
    diagonale2 = new bool[15];
    soluzione = new int[8];
}

```

Scriviamo ora i metodi di supporto per posizionare e togliere una regina da una certa casella, per capire se una casella è sicura perché possa essere occupata da una regina e per stampare la soluzione trovata.

```

// Posiziona una regina in (r,c) mettendo sotto scacco colonna e diagonali
static void Posiziona(int r, int c)
{
    soluzione[r] = c;
    colonne[c] = true;
    diagonale1[r - c + 7] = true; // +7 perché r-c inizia da -7
    diagonale2[r + c] = true;
}

// Toglie la regina da (r,c) cioè libera dallo scacco la colonna e le diagonali. Nota che non è
// necessario azzerare anche la posizione della soluzione in quanto questa verrà sovrascritta dal
// prossimo tentativo
static void Togli(int r, int c)
{
    colonne[c] = false;
    diagonale1[r - c + 7] = false;
    diagonale2[r + c] = false;
}

// Verifica se la posizione (r,c) è sicura
static bool Sicura(int r, int c)
{
    if (!colonne[c] && !diagonale1[r - c + 7] && !diagonale2[r + c])
        return true;
    else
        return false;
}

// Stampa la soluzione
static void StampaSoluzione()
{
    Write("\nSoluzione: ");
    for (int r = 0; r < 8; r++)
        Write(soluzione[r] + " ");
}

```

Ecco infine la procedura ricorsiva per generare tutte le soluzioni:

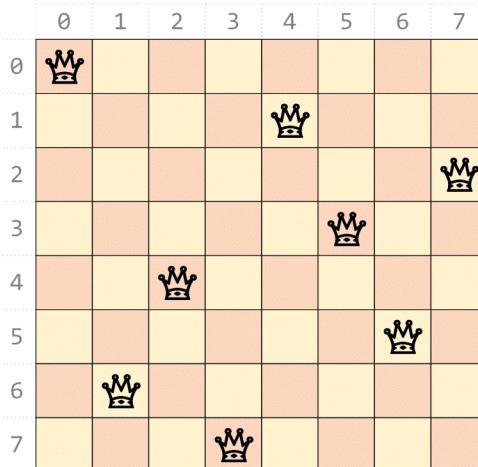
```
// Genera tutte le configurazioni cercando di mettere una regina nella riga data dal parametro passato
public static void GeneraSoluzioni(int riga = 0)
{
    if (riga == 0)
        Inizializza();

    for (int colonna = 0; colonna < 8; colonna++)
    {
        if (Sicura(riga, colonna))
        {
            Posiziona(riga, colonna);
            if (riga == 7)
                StampaSoluzione();
            else
                GeneraSoluzioni(riga + 1);
            Togli(riga, colonna);
        }
    }
}
```

Per completezza riportiamo di seguito le 92 soluzioni trovate dall'algoritmo:

1: 04752613	20: 25164073	39: 35720641	57: 46027531	75: 52630714
2: 05726314	21: 25307461	40: 36074152	58: 46031752	76: 53047162
3: 06357142	22: 25317460	41: 36271405	59: 46137025	77: 53174602
4: 06471352	23: 25703641	42: 36415027	60: 46152037	78: 53602417
5: 13572064	24: 25704613	43: 36420571	61: 46152073	79: 53607142
6: 14602753	25: 25713064	44: 37025164	62: 46302751	80: 57130642
7: 14630752	26: 26174035	45: 37046152	63: 47302516	81: 60275314
8: 15063724	27: 26175304	46: 37420615	64: 47306152	82: 61307425
9: 15720364	28: 27360514	47: 40357162	65: 50417263	83: 61520374
10: 16257403	29: 30471625	48: 40731625	66: 51602473	84: 62057413
11: 16470352	30: 30475261	49: 40752613	67: 51603742	85: 62714053
12: 17502463	31: 31475026	50: 41357206	68: 52064713	86: 63147025
13: 20647135	32: 31625704	51: 41362750	69: 52073164	87: 63175024
14: 24170635	33: 31625740	52: 41506372	70: 52074136	88: 64205713
15: 24175360	34: 31640752	53: 41703625	71: 52460317	89: 71306425
16: 24603175	35: 31746025	54: 42057136	72: 52470316	90: 71420635
17: 24730615	36: 31750246	55: 42061753	73: 52613704	91: 72051463
18: 25147063	37: 35041726	56: 42736051	74: 52617403	92: 73025164
19: 25160374	38: 35716024			

Sono indicati solo gli indici di colonna per le righe da 0 a 7; ad esempio la prima soluzione corrisponde alla seguente disposizione sulla scacchiera:



11. Espressioni regolari

Immaginiamo di voler scrivere un programma per verificare se una stringa inserita dall'utente può rappresentare un numero di telefono composto da sole cifre e con una lunghezza minima di 6 e massima di 14 cifre.

Dovremo verificare se la stringa ha una lunghezza compresa tra 6 e 14 ed eseguire un ciclo per esaminare i singoli caratteri verificando che siano cifre.

Oppure, pensiamo a problemi analoghi di verificare se una stringa contiene una data composta con i numeri per giorno mese e anno separati da barre, come in 02/08/2019. Oppure, ancora, verificare se una stringa rappresenta o meno un indirizzo e-mail.

In molte situazioni reali c'è l'esigenza di verificare la validità di una stringa rispetto ai dati che deve rappresentare; pensiamo alla compilazione di moduli on-line in cui vengono richiesti dati anagrafici, fiscali o altro.

In tutte queste situazioni dovremo scrivere degli algoritmi ad hoc per la validazione dei dati, aiutandoci ad esempio con le funzioni sulle stringhe viste nel capitolo 6.

Le **espressioni regolari** (*regular expression*) sono nate proprio per rispondere a queste esigenze evitando al programmatore la scrittura di algoritmi per ogni tipo di dato da validare. Concretamente un'espressione regolare è definita attraverso una sequenza di caratteri e simboli per definire lo schema (*pattern*) che le stringhe di input dovranno soddisfare.

Tutti i linguaggi di programmazione integrano, o supportano attraverso librerie, l'utilizzo delle espressioni regolari: dispongono cioè di funzioni per capire se una stringa segue o meno lo schema definito da una certa espressione regolare.

Le espressioni regolari sono state inventate ancora negli anni 1970 con il sistema operativo UNIX^[70] ma il loro utilizzo si è diffuso a partire da metà degli anni '80 con il linguaggio Perl^[71] che supporta nativamente l'utilizzo delle espressioni per validare i dati. Tanto è vero che oggigiorno la sintassi per definire un'espressione regolare segue molto spesso le regole sintattiche definite dal linguaggio Perl.

11.1. Espressioni regolari in C#

Anche C# supporta le espressioni regolari attraverso la classe **Regex** del namespace **System.Text.RegularExpressions**. Il namespace contiene poi altre classi di supporto utilizzate dai metodi della classe Regex.

La classe **Regex** contiene una serie di metodi per verificare una stringa rispetto ad un criterio descritto da un'espressione regolare (*pattern*) ma anche per estrarre le parti che la soddisfano. I principali metodi sono i seguenti:

- **.IsMatch**^[72] verifica se una stringa di input soddisfa una espressione regolare data
- **Match**^[73] estrae la porzione di una stringa che soddisfa (se esiste) l'espressione regolare. Il metodo restituisce la prima occorrenza della sottostringa come oggetto di tipo **Match**^[74]
- **Matches**^[75] è simile al precedente ma restituisce tutte le occorrenze dell'espressione regolare. Viene restituito un oggetto di tipo **MatchCollection**^[76], una raccolta di oggetti Match
- **Replace**^[77] a partire da una stringa di input, la restituisce sostituendo tutte le sottostringhe corrispondenti ad una espressione regolare, con una stringa sostitutiva
- **Split**^[78] suddivide una stringa di input in base ad una espressione regolare e restituisce un array di stringhe con tutte le parti nelle quali è stata suddivisa la stringa di input.

I precedenti metodi sono previsti sia nella versione statica (utilizzabile cioè direttamente sulla classe Regex) sia nella versione di istanza da utilizzare quindi su un oggetto di tipo Regex precedentemente creato.

Regex prevede l'impostazione del pattern per l'espressione regolare, seguendo una specifica sintassi che vedremo nel prossimo paragrafo. C# segue le regole sintattiche del linguaggio Perl 5.

11.2. Sintassi delle espressioni regolari

La stringa che ha la funzione di *pattern* può contenere singoli caratteri e le sequenze di escape viste nel capitolo 3. Ad esempio una stringa come la seguente è perfettamente lecita come pattern:

```
string pattern = @"abc";
```

le stringhe soddisfano il precedente pattern solo se contengono la costante stringa “abc”.

Oltre a normali caratteri, il pattern può utilizzare speciali segnaposti per definire schemi più elaborati; è proprio questo aspetto che rende potenti le espressioni regolari.

Notiamo che nel precedente esempio abbiamo usato una stringa *verbatim*: nello specifico non sarebbe stato necessario ma abituiamoci a definire i pattern con stringhe verbatim dato che diversi segnaposti utilizzano la barra \. Non definendo una stringa verbatim, potremo incorrere in errori di compilazione per sequenze di escape non riconosciute.

Nella seguente tabella sono descritti i principali segnaposti^[79]:

Segnaposto	Descrizione
\	Quando è seguito da un carattere non riconosciuto come carattere di escape o da altri segnaposti, trova la corrispondenza di tale carattere. È importante usare la barra ad esempio per individuare caratteri come ., *, + o ? che altrimenti verrebbero interpretati come segnaposti (v. descrizioni successive).
.	È un carattere jolly per indicare un qualsiasi singolo carattere ad eccezione di \n Ad esempio il pattern a.c viene validato da tutte le stringhe che contengono la lettera “a” seguita da un qualsiasi altro carattere e quindi dalla lettera “c”
\w	Un singolo carattere alfabetico ovvero lettere e cifre
\W	Un singolo carattere non alfabetico ovvero segni di punteggiatura, spazi e altri caratteri non alfabetici
\s	Un singolo carattere di spazio. Oltre al carattere di spaziatura ‘ ‘ vengono individuati come caratteri di spazi anche la tabulazione, il ritorno a capo, in definitiva i seguenti caratteri (descritti attraverso sequenze di escape): \t \r \n \f
\S	Un singolo carattere diverso da uno spazio

<i>Segnaposto</i>	<i>Descrizione</i>
\d	Una qualsiasi cifra decimale
\D	Un singolo carattere diverso da una cifra decimale
[carattere1-carattere2]	Un singolo carattere compreso nell'intervallo tra carattere1 e carattere2 . Ad esempio il pattern [a-z] viene validato da una qualsiasi stringa contenente almeno una lettera tra la “a” e la “z”
[gruppo_caratteri]	Un singolo carattere contenuto in gruppo_caratteri . Ad esempio il pattern qu[ioa] viene validato da stringhe contenenti le parole “qui”, “quo” o “qua”. Possiamo inserire in gruppo_caratteri anche intervalli come ad esempio con il pattern [0-9a-f] che viene validato da stringhe contenenti cifre o le lettere tra “a” ed “f”
[^gruppo_caratteri]	Un singolo carattere non contenuto in gruppo_caratteri
(sottoespressione)	Le parentesi tonde sono utilizzate per raggruppare delle sotto espressioni. Ciò può essere utile ad esempio per applicare un alternativa o un quantificatore (v. prossime righe) o estrarre le singole sottostringhe che validano le sotto espressioni.
	Consente di definire delle alternative. Ad esempio il pattern libr(eria o i) viene validato da stringhe contenenti le parole “libreria”, “libro” o “libri”
*	L'elemento che precede * si presenta zero o più volte
+	L'elemento che precede + si presenta una o più volte
?	L'elemento che precede ? si presenta zero volte o una volta
{n}	L'elemento precedente si presenta esattamente n volte
{n,}	L'elemento precedente si presenta almeno n volte
{n,m}	L'elemento precedente si presenta almeno n volte, ma non più di m volte
^	Questo carattere viene usato per “ancorare” l'espressione all'inizio della stringa. Ad esempio il pattern ^[a-z][A-Z] viene validato da stringhe che iniziano con una lettera maiuscola o minuscola
\$	Analogamente al precedente, questo carattere viene usato per ancorare l'espressione alla fine della stringa

11.3. Verificare una stringa rispetto ad un pattern

Per verificare una stringa rispetto ad uno schema possiamo usare il metodo **.IsMatch** della classe **Regex**.

La versione statica è immediata da usare e nella forma più semplice ha il seguente prototipo:

```
static bool IsMatch(string input, string pattern);
```

dove **input** indica la stringa da controllare e **pattern** lo schema dell'espressione regolare da utilizzare. Il metodo ritorna **true** se **input** viene validata dal criterio definito da **pattern** e **false** in caso contrario.

Una seconda versione del metodo può ricevere un terzo parametro per impostare delle opzioni per la corrispondenza; ecco alcune opzioni significative^[80]:

- **RegexOptions.IgnoreCase** non fa distinzione tra maiuscole e minuscole; se non è specificata questa opzione, il metodo **.IsMatch** distingue tra maiuscole e minuscole
- **RegexOptions.CultureInvariant** non considera le impostazioni legate alla cultura
- **RegexOptions.Compiled** precompila l'espressione regolare così da renderne più rapida l'esecuzione. Questo può rivelarsi utile non tanto con la versione statica ma con la versione di istanza del metodo **IsMatch** per validare una serie di stringhe rispetto allo stesso pattern.

Più opzioni possono essere combinate assieme con l'operatore or, ad esempio potremo scrivere **RegexOptions.IgnoreCase | RegexOptions.CultureInvariant** per ignorare il case e la cultura.

Esiste anche un ulteriore overload del metodo che riceve un quarto parametro per impostare un tempo limite superato il quale la validazione viene interrotta. Ciò può essere utile nei casi di espressioni regolari complesse che potrebbero portare a tempi lunghi di verifica.

Come dicevamo, nel caso la stessa espressione regolare deva essere verificata per una serie di stringhe, conviene creare un oggetto di tipo **Regex** associato al pattern, e usare su di questo il metodo **.IsMatch**:

```
Regex re = new Regex(pattern);
bool valida = re.IsMatch(input);
```

la versione di istanza del metodo **IsMatch**, rispetto a quella statica, può ricevere un secondo parametro di tipo intero che specifica la posizione del carattere dalla quale iniziare la ricerca:

```
bool valida = re.IsMatch(input, 5); // inizia la ricerca dal carattere di posizione 5
```

Vediamo ora un programma completo che consente di validare gli input dell’utente solo se rappresentano numeri espressi in formato esadecimale che devono iniziare con **0x** o **0X**:

```
string pattern = @"^0x(\d|[a-f])+$";
Regex re = new Regex(pattern, RegexOptions.IgnoreCase);

string input;
do
{
    Write("Inserire una stringa da controllare: ");
    input = ReadLine();

    if (re.IsMatch(input))
        WriteLine("La stringa inserita è un numero esadecimale");
    else
        WriteLine("La stringa inserita non è un numero esadecimale");

}
while (input != "");
```

Esaminiamo il pattern utilizzato:

- con i carattere ^ e \$ ancoriamo l’espressione in modo tale che la stringa in input sia validata solo se corrisponde all’espressione e non contiene niente altro
- (\d|[a-f]) viene validata da una cifra (segnaposto \d) o da una lettera tra “a” e “f”
- il segno + indica che l’elemento precedente (lettera o cifra) può presentarsi una o più volte. Quindi ad esempio non sarà validata una stringa con il solo prefisso come “0x”.

Riprendendo l’esempio fatto all’inizio del capitolo, se volessimo verificare se l’input è un numero di telefono composto da sole cifre e con una

lunghezza minima di 6 e massima di 14 cifre potremo scrivere il seguente codice; questa volta usiamo la versione statica di **IsMatch**:

```
if (Regex.IsMatch(input, @"^\d{6,14}$"))
    WriteLine("Numero telefonico valido");
else
    WriteLine("Numero telefonico NON valido");
```

Si faccia attenzione a non inserire spazi nel quantificatore tra le parentesi graffe.

Infine, il codice per validare una data nella forma **gg/mm/aaaa** è il seguente:

```
if (Regex.IsMatch(input, @"^\d{2}/\d{2}/\d{4}$"))
    WriteLine("Data valida");
else
    WriteLine("Data NON valida");
```

11.4. Estrarre le sottostringhe che rispettano un pattern

I metodi **Match** e **Matches** consentono di estrarre da una stringa in input tutte le sottostringhe che validano un pattern.

Il metodo Match individua la prima occorrenza della sottostringa che valida il pattern, come oggetto di classe Match; esaminiamo il seguente esempio:

```
input = "Mi chiamo James Bond e sono nato il 11/11/1924 e la mia prima missione è iniziata il  
23/03/1953";
```

```
Match m = Regex.Match(input, @"\d{2}\/\d{2}\/\d{4}");  
if (m.Success)  
    WriteLine("Identificata data " + m.Value +  
              " in posizione " + m.Index + " di lunghezza " + m.Length);
```

viene cercata la prima data presente nella stringa di input: l'oggetto **m** restituito da Match ha diverse proprietà, alcune delle quali vengono utilizzate nel codice precedente:

- **m.Success** restituisce *true* se una data è stata trovata
- **m.Value** è la sottostringa individuata, nell'esempio “11/11/1924”
- **m.Index** è la posizione dalla quale inizia la sottostringa, nell'esempio 36
- **m.Length** la lunghezza della sottostringa, nell'esempio 10.

Sfruttando il metodo **NextMatch** della classe Match possiamo individuare tutte le occorrenze. Nell'esempio precedente sostituendo la condizione con il seguente ciclo possiamo ottenere tutte le date contenute nella stringa:

```
while (m.Success)  
{  
    WriteLine("Identificata data " + m.Value +  
              " in posizione " + m.Index + " di lunghezza " + m.Length);  
    m = m.NextMatch();  
}
```

In alternativa possiamo utilizzare il metodo **Matches** che restituisce una collezione di oggetti Match (classe **MatchCollection**).

Il seguente esempio individua tutte le date presenti nella stringa di input:

```
MatchCollection mc = Regex.Matches(input, @"\d{2}/\d{2}/\d{4}");
WriteLine("Trovate " + mc.Count + " date all'interno della stringa di input");
foreach (Match m1 in mc)
    WriteLine("Identificata la data " + m1.Value +
        " in posizione " + m1.Index);
```

La proprietà **Count** di MatchCollection restituisce il numero di occorrenze delle sottostringhe che validano il pattern. Le singole occorrenze possono essere ottenute utilizzando un indice, ad esempio mc[0], mc[1], ...

Quando nel pattern si utilizzano le parentesi tonde per racchiudere delle sottosstringhe, possiamo estrarre i vari pezzi usando la proprietà **Groups** dell'oggetto Match. Il seguente esempio è una variante del precedente per ottenere, per ogni data, giorno mese ed anno:

```
mc = Regex.Matches(input, @"(\d{2})/(\d{2})/(\d{4})");
foreach (Match m2 in mc)
{
    WriteLine("Identificata la data " + m2.Value);
    // Nota: l'intera data è ottenibile anche con m2.Groups[0].Value
    WriteLine("giorno = " + m2.Groups[1].Value);
    WriteLine("mese = " + m2.Groups[2].Value);
    WriteLine("anno = " + m2.Groups[3].Value);
}
```

Il metodo **Split** permette invece di ottenere un array con tutte le sottostringhe che validano un'espressione regolare. Con il seguente esempio si ottengono le parole che compongono una frase considerando come separatori i segni di punteggiatura e gli spazi (singoli o consecutivi):

```
input = "Questa: è una frase, di esempio.";
string[] parole = Regex.Split(input, @"\s,;:+");

WriteLine("Elenco parole:");
foreach (string s in parole)
    if (s != "")
        WriteLine(s);
/* Output:
Elenco parole:
Questa
è
```

```
una  
frase  
di  
esempio  
*/
```

La condizione **parola != ""** usata all'interno del ciclo è necessaria per evitare di visualizzare le stringhe vuote. Infatti il segno di punteggiatura che conclude la stringa produce un'ultima stringa, di lunghezza 0.

Se si delimita il pattern con le parentesi tonde anche le occorrenze del pattern vengono restituite in output. Si osservi ad esempio cosa produce in output il seguente codice:

```
input = "111;222;333";  
string[] token = Regex.Split(input, @"(;)");  
foreach (string s in token)  
    WriteLine(s);  
/* Output:  
111  
;  
222  
;  
333  
*/
```

usando invece il pattern ";" si otterrà come output il seguente:

```
111  
222  
333
```

In rete si trovano diversi siti con collezioni di espressioni regolari già pronte, ad esempio **RegExLib.com** (www.regexlib.com) oppure siti in cui è possibile verificare il funzionamento delle espressioni regolari digitate, ad esempio **Regex Tester** (<http://regexstorm.net/tester>), **RegExr** (<https://regexr.com/>) oppure **RegEx101** (<https://regex101.com/>).

12. Enumerazioni

Per raggiungere un maggiore livello di astrazione e rendere il codice più leggibile, molti linguaggi di programmazione consentono di definire insiemi di costanti simboliche chiamate **enumerazioni**.

Il nome è dovuto al fatto che vengono elencate le costanti che fanno parte dell'enumerazione.

Si immagini ad esempio un'applicazione che abbia bisogno di memorizzare in una variabile una direzione che possa assumere uno dei quattro punti cardinali, “nord”, “sud”, “est”, “ovest”.

Il programmatore potrebbe adottare una soluzione come la seguente, nella quale usare come valori per la variabile direzione, una delle quattro costanti definite:

```
const int NORD = 0, SUD = 1, EST = 2, OVEST = 3;  
int direzione = NORD;
```

E' una soluzione sicuramente percorribile che comunque richiede una certa attenzione da parte del programmatore nell'assegnare alle variabili che rappresentano direzioni, una delle quattro costanti previste; assegnare un valore intero sarebbe ovviamente possibile.

Un'alternativa è definire un tipo di enumerazione che elenchi i quattro punti cardinali, e usarlo come tipo delle variabili che vogliono usare per memorizzare direzioni; si può fare in C# con la parola chiave **enum** nel seguente modo:

```
public enum Direzioni  
{  
    Nord,  
    Sud,  
    Est,  
    Ovest,  
}
```

come mostrato nell'esempio l'elenco può eventualmente terminare con una virgola così che sia più immediato allungarlo.

La dichiarazione di un tipo di enumerazione può essere inserita indifferentemente a livello di namespace o annidata all'interno di una

classe.

Se definiamo un’enumerazione all’interno di una classe, con lo stesso nome di un’altra enumerazione definita a livello di namespace, quest’ultima è “nascosta” dalla dichiarazione più interna: possiamo comunque riferirci all’enumerazione più esterna anteponendo il nome del namespace.

Possiamo usare il nuovo tipo ad esempio scrivendo dichiarazioni come la seguente:

```
Direzioni d1 = Direzioni.Nord, d2 = Direzioni.Sud;
```

Si noti che i valori dell'enumerazione devono essere indicati specificando il nome del tipo seguito da un punto e dal valore desiderato.

Il compilatore tratta i tipi di enumerazione come elenchi di costanti di tipo **int** attribuendo, per default, ai valori elencati valori da 0 in poi.

E' possibile però attribuire, in fase di dichiarazione dell'enumerazione, valori differenti. Ecco un esempio commentato:

```
enum Colori
{
    Rosso = 1, // possono essere assegnati valori esplicativi agli elementi
    Verde = 2,
    Blu, // se un elemento non ha un valore, assume quello del precedente aumentato di 1 (Blu prende quindi il valore 3)
    Pomodoro = 1, // anche ripetendo il valore (non ha molto senso)
    Lattuga = Verde, // o usando un altro elemento come riferimento
    Peperoncino = Pomodoro + 1 // o anche usando un altro elemento con un'operazione
}
```

Inoltre un tipo di enumerazione può essere basato su un tipo intero diverso da **int**, quindi scelto tra **byte**, **sbyte**, **short**, **ushort**, **uint**, **long** o **ulong**. Il tipo va specificato subito dopo il nome dell'enumerazione precedendolo con un carattere due punti, come mostrato nel seguente esempio:

```
public enum TipoAuto : byte
{
    Cabriolet,
    Coupè,
    SUV
}
```

Come per le altre costanti, tutti i riferimenti ai singoli valori di un tipo di enumerazione vengono convertiti in costanti letterali numeriche in fase di compilazione.

Le enumerazioni sono usate nella libreria di classi .NET per definire elenchi di opzioni per i metodi: ad esempio l'enumerazione **StringSplitOptions** nel

metodo **Split** (v. capitolo 7 sulle stringhe) oppure **RegexOptions** nelle espressioni regolari (v. capitolo 11).

12.1. Conversioni a stringa e da intero

Come per tutte le variabili di tipo semplice anche le variabili di un tipo di enumerazione sono “inscatolate” in oggetti; è quindi possibile usare il metodo **ToString()** per ottenere la stringa corrispondente al valore della variabile.

Può ricevere un parametro, (anche maiuscolo, non è case-sensitive) che può essere “**g**” (default), “**d**”, “**x**” o “**f**”, per restituire rispettivamente il nome dell’elemento, il valore numerico, il valore numerico esadecimale e il valore come flag (v. oltre).

Ecco un esempio riferito all’enumerazione definita all’inizio:

```
Direzioni d = Direzioni.Ovest;  
WriteLine(d.ToString()); // Ovest  
WriteLine(d.ToString("d")); // 3  
WriteLine(d.ToString("x")); // 00000003  
WriteLine(d.ToString("f")); // Ovest
```

Visto che in sostanza le variabili di un tipo di enumerazione sono interi, possiamo assegnare costanti intere ad una variabile; ad eccezione del valore 0 è sempre necessario utilizzare il cast esplicito. L’assegnazione di un valore fuori dal range previsto dall’enumerazione non provoca comunque alcuna eccezione:

```
d = 0; // il valore 0 (solo quello) può essere attribuito ad una variabile di un tipo enumerazione  
senza bisogno di cast  
d = (Direzioni)3; // d assumerà valore Ovest  
d = (Direzioni)4; // il cast di un valore non contenuto non provoca eccezioni  
WriteLine(d.ToString()); // 4
```

12.2. La classe Enum

La classe **Enum**^[81] contiene alcuni metodi statici utili per operare con i tipi di enumerazione. Ad esempio il metodo **Parse** permette di convertire una stringa nel corrispondente valore dell'enumerazione; se la conversione non è possibile viene sollevata un'eccezione:

```
d = (Direzioni)Enum.Parse(typeof(Direzioni), "est", true);
WriteLine(d.ToString()); // Est
```

l'ultimo parametro a true ha la funzione di ignorare il case della stringa da convertire.

I metodi **GetNames** e **GetValues** consentono di ottenere degli array rispettivamente, con i nomi degli elementi e dei valori dell'enumerazione:

```
string[] nomi = Enum.GetNames(typeof(Direzioni));
WriteLine(string.Join(", ", nomi)); // Nord,Sud,Est,Ovest
```

```
int[] valori = (int[])Enum.GetValues(typeof(Direzioni));
WriteLine(string.Join(", ", valori)); // 0,1,2,3
```

Il ruolo della classe **Enum** è quello di fungere da unificazione per tutti i tipi di enumerazione. Possiamo ad esempio utilizzarlo come tipo per un metodo che riceva un qualsiasi tipo di enumerazione:

```
void Visualizza(Enum valore)
{
    WriteLine(valore.ToString());
}
```

```
Direzioni dir = Direzioni.Ovest;
Visualizza(dir);
```

```
Colori col = Colori.Rosso;
Visualizza(col);
```

12.3. Campi di bit

Un tipo di enumerazione può essere usato per elencare un insieme di valori che si escludono a vicenda. Una variabile del tipo di enumerazione può assumere i valori previsti dall'enumerazione ma anche combinare assieme più valori; ciò può essere utile per maschere di bit o flag.

Ad esempio la seguente è una definizione per un'enumerazione che rappresenta colori descritti dalle tre componenti R (red), G (green), B (blue):

```
enum RGB
{
    Red = 0xFF0000,
    Green = 0x00FF00,
    Blue = 0x0000FF
}
```

Agli elementi Red, Green e Blue sono attribuiti valori che corrispondono a potenze di 2 differenti in modo tale cioè che i valori si escludano a vicenda. Così facendo potremo dichiarare una variabile come la seguente, per combinare entrambi i valori Red e Green:

```
RGB giallo = (RGB)0xFFFF00;
WriteLine(giallo.ToString("f")); // Green, Red (il valore di giallo combina entrambi i valori)
```

Il metodo **ToString**, con il parametro “f”, permette di ottenere, in questo caso, le componenti combinate.

La combinazione può essere effettuata anche usando gli operatori logici:

```
RGB bianco = RGB.Red | RGB.Green | RGB.Blue;
WriteLine(bianco.ToString("f")); // Blue, Green, Red
```

Su un valore di un tipo di enumerazione può essere usato il metodo **HasFlag** per capire se il valore combinato contiene uno dei valori dell'enumerazione.

Ad esempio con il seguente codice viene richiesto in input un intero esadecimale, interpretato come colore, e rilevato se contiene la componente di colore verde:

```
Write("Digita un colore espresso in esadecimale: ");
RGB colore = (RGB)Convert.ToInt32(ReadLine(), 16); // converte la stringa come valore esadecimale

if (colore.HasFlag(RGB.Green))
    WriteLine("Il colore contiene la componente verde");

WriteLine("Componenti del colore inserito: " + colore.ToString("f"));
```

Nell'ultima riga otteniamo le componenti del colore inserito utilizzando il parametro “f” (*flags*) del metodo `ToString`. Se non si usava si avrebbe ottenuto solo il valore numerico.

In alternativa, il tipo di enumerazione può essere dichiarato con l'attributo **[Flags]** come mostrato di seguito; questo dichiara esplicitamente che l'enumerazione è un campo di bit a valori esclusivi. Così facendo il comportamento di default del metodo `ToString`, senza parametri, avrà lo stesso effetto del parametro “f”.

```
[Flags]
enum RGB
{
    Red = 0xFF0000,
    Green = 0x00FF00,
    Blue = 0x0000FF
}
```

Vediamo un altro esempio definendo un'enumerazione che elenca i permessi di un file (le costanti intere sono specificate usando letterali binari, con il prefisso 0b, introdotti da C# 7):

```
[Flags]
enum PermessiFile: byte
{
    Nessuno = 0b_0000_0000, // 0
    Esecuzione = 0b_0000_0001, // 1
```

```
Scrittura = 0b_0000_0010, // 2
Lettura  = 0b_0000_0100, // 4
}
```

proviamo il nuovo tipo:

```
PermessiFile pf = PermessiFile.Esecuzione | PermessiFile.Lettura;
WriteLine(pf); // ToString() viene chiamato implicitamente e stampa "Esecuzione, Lettura"
```

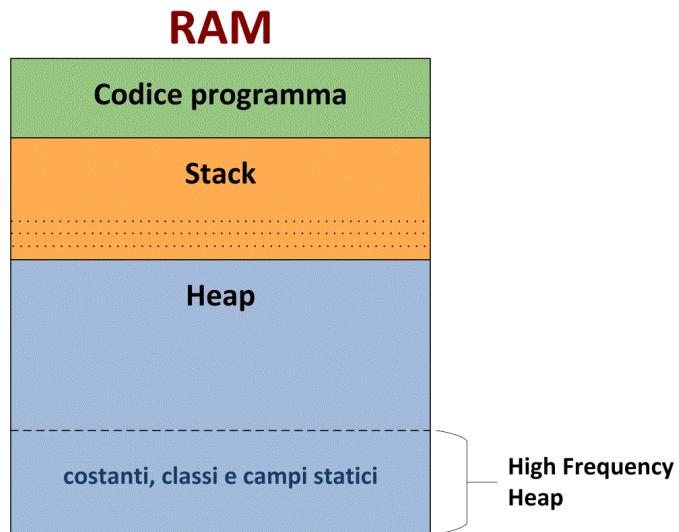

Appendici

Stack e Heap

La memoria assegnata ad un programma quando viene eseguito è suddivisa in due blocchi con ruoli diversi:

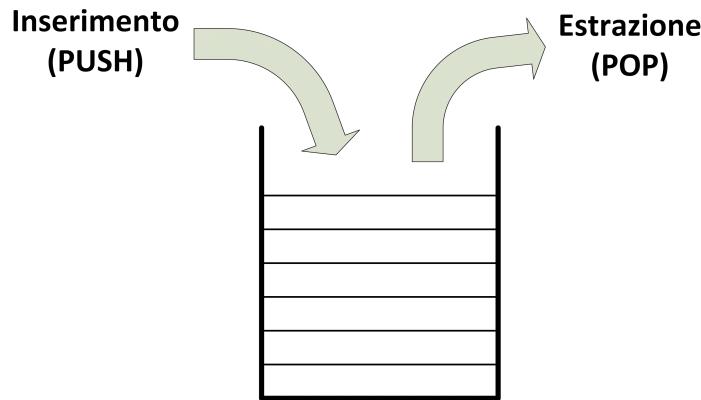
- lo **stack** (“pila”) è un blocco di memoria la cui dimensione è prefissata quando il programma viene mandato in esecuzione. In questa area di memoria vengono allocate tutte le variabili locali^[82], i parametri dei metodi e i valori di ritorno
- lo **heap** (“mucchio”) è un blocco di memoria che può allargarsi durante l’esecuzione del programma e contiene tutte le variabili di tipo riferimento come oggetti, stringhe, array, ecc.

Inoltre l’heap mantiene anche tutte le variabili statiche e le costanti definite nel programma. Questi dati, essendo mantenuti in memoria per l’intero ciclo di vita del programma, non sono influenzati dal lavoro del garbage collector e sono memorizzati in un’apposita sezione dello heap chiamata “**High Frequency Heap**”^[83].



Lo **stack** organizza la memoria al suo interno impilando i dati uno di seguito all’altro (o uno sopra l’altro): l’ultimo dato che verrà inserito nello stack (operazione di **push**) sarà sempre il primo ad essere estratto (operazione di **pop**).

Si dice che lo stack segue una logica di funzionamento **LIFO**, *Last-In First-Out* (l'ultimo ad entrare è il primo ad uscire).



Questa è una delle strutture dati più usate in informatica, anche ad esempio nella memoria interna delle CPU^[84].

Proprio per la sua semplice logica di funzionamento, lo **stack** è utilizzato nella gestione dei sottoprogrammi. Nello stack vengono memorizzati i cosiddetti “**record di attivazione**” dei metodi.

Il record di attivazione di un metodo comprende l’indirizzo dell’istruzione successiva a quella che ha invocato il metodo (“*indirizzo di ritorno*”), che consente al codice del metodo di sapere a quale riga di codice saltare una volta conclusa la sua attività, gli eventuali parametri del metodo e le variabili locali al metodo.

Questo meccanismo consente quindi di “passare” dati tra il metodo chiamante e il metodo chiamato. Permette inoltre una semplice gestione dei metodi ricorsivi. Attenzione comunque che la catena di chiamate ricorsive può causare il superamento dei limiti dello stack provocando un errore di “*stack overflow*”^[85].

Riguardo allo **heap**, se una variabile locale è di un tipo riferimento (ad esempio un array) il riferimento viene memorizzato nello stack mentre il valore referenziato (il contenuto dell’array) è memorizzato nello heap.

Il *garbage collector* (di cui si è parlato al capitolo 7) provvede a liberare la memoria dello heap non più raggiungibile. Questo accade, ad esempio, alla terminazione di un metodo che utilizza variabili locali di tipo riferimento; più in generale succede tutte le volte che una variabile esce dal proprio scope.

L'area di memoria dedicata allo **stack** di un programma viene allocata all'avvio del programma stesso^[86]; non è quindi possibile cambiarne la dimensione durante l'esecuzione.

La memoria dello **heap** è gestita invece dinamicamente in base alle esigenze che si presentano durante l'esecuzione del programma.

I progetti realizzati con Visual Studio sono di default impostati per una compilazione per qualsiasi CPU; di conseguenza la memoria allocabile per lo heap è al massimo di 2 GiB, limite imposto dai sistemi a 32 bit. Possiamo superare questo limite impostando il progetto per un sistema a 64 bit: dal menu *Progetto -> Proprietà di ... ->* scheda *Compilazione*, impostare **“Piattaforma di destinazione”** a **x64**. Così facendo la memoria sarà limitata soltanto dalla RAM che abbiamo a disposizione^[87].

In C# possiamo determinare quanta memoria è utilizzata da un programma utilizzando la classe statica **GC**^[88] che è una classe che permette di controllare il garbage collector. Dispone del metodo **GetTotalMemory** che restituisce il numero totale di byte allocati; il metodo riceve un parametro booleano che se true (il default è false), forza l'esecuzione del garbage collector (per liberare eventuali oggetti che devono essere rimossi dalla memoria) prima di restituire il risultato.

Il seguente esempio ne dimostra l'utilizzo:

```
long memPrima, memDopo;
memPrima = GC.GetTotalMemory(true);
byte[] a = new byte[10000]; // allocazione di un array di 10000 byte
memDopo = GC.GetTotalMemory(true);
Console.WriteLine("Differenza memoria = " + (memDopo-memPrima) + " B"); // stampa
"10024 B"
```

Come ci si aspetta, l'allocazione dell'array determina un consumo di circa 10000 byte di memoria (i pochi byte aggiuntivi vengono utilizzati per la gestione dell'oggetto array).

Dal punto di vista delle prestazioni la gestione della memoria dello stack è più veloce rispetto allo heap: questo a causa della semplicità delle operazioni di gestione, che vengono generalmente tradotte in istruzioni in linguaggio macchina molto veloci. Inoltre i dati dello stack tendono ad

essere riutilizzati frequentemente e quindi tendono ad essere mappati nella cache del processore con un notevole guadagno nei tempi di accesso.

Generazione di numeri casuali

In alcune situazioni risulta comodo riempire un array con dati casuali. La generazione di valori casuali è in generale utile in tutte quelle situazioni in cui dobbiamo simulare qualche evento reale come l'estrazione di una pallina da un'urna, il lancio di un dado, la definizione casuale di uno stato iniziale in un videogioco, ecc.

Nei linguaggi di programmazione è sempre disponibile qualche funzione per generare numeri casuali.

Anche C# non fa eccezione fornendo la classe **Random**, che vedremo fra poco.

In realtà si parla più propriamente di **numeri pseudo-casuali** in quanto i numeri generati dalla macchina non sono completamente casuali ma vengono costruiti sulla base di successioni numeriche il cui valore di partenza (chiamato **seme**) è normalmente inizializzato sfruttando un valore letto dalla macchina come ad esempio il tempo dall'orologio di sistema.

La caratteristica di queste successioni numeriche è quella di assicurare che tutti i numeri nel range scelto abbiano la stessa probabilità di presentarsi proprio come succede in una sequenza di numeri casuali generata ad esempio lanciando un dado^[89].

Per generare numeri casuali in C# si deve dichiarare una variabile di tipo **Random**. Questa variabile sarà il nostro generatore.

Per poterla di fatto usare dovremo creare il generatore attraverso l'operatore **new**, come mostrato nel seguente esempio:

```
Random gen; // dichiara un generatore di numeri casuali  
gen = new Random(); // crea il generatore
```

nella creazione del generatore può essere indicato anche un parametro intero che verrà utilizzato come **seme** per calcolare un valore iniziale per la sequenza di numeri pseudo-casuali.

Se non si passa nulla, viene usato un seme predefinito dipendente dal tempo; più precisamente viene sfruttato il numero di millisecondi trascorsi dall'ultimo avvio del computer, ottenibili dalla proprietà **Environment.TickCount**^[90].

Una volta creato il generatore, possiamo utilizzarlo usando uno dei seguenti metodi:

- **int Next()** restituisce un numero casuale maggiore o uguale a zero e minore del massimo valore per il tipo int. Ricordiamo che il massimo valore intero a 32 bit è 2.147.483.647 e si può ottenere con la costante **int.MaxValue**
- **int Next(int max)** restituisce un numero casuale maggiore o uguale a zero e minore di **max**
- **int Next(int min, int max)** restituisce un numero casuale maggiore o uguale a **min** e minore di **max**
- **double NextDouble()** restituisce un numero casuale in virgola mobile maggiore o uguale a 0,0 e minore di 1,0
- **NextBytes(byte[] buffer)** riempie il vettore **buffer** (vettore di *byte*) con valori casuali maggiori o uguali a zero e minori della costante **byte.MaxValue** che vale 255

Ad esempio possiamo usare il generatore creato per simulare il lancio di un dado o di una moneta:

```
int dado;  
dado = gen.Next(1, 7); // genera un numero nell'intervallo [1, 6]  
  
bool moneta;  
moneta = (gen.Next(2) == 0); // genera un numero tra due possibili e lo utilizza per simulare il  
lancio di una moneta
```

Potrebbe succedere che, provando ad utilizzare più generatori casuali uno di seguito all'altro (senza interazioni con l'utente), venga generata la stessa sequenza. La velocità di esecuzione del codice comporta che i due generatori usino come seme lo stesso valore di millisecondi.

Possiamo verificarlo eseguendo il seguente codice che genera due sequenze di 100 interi:

```
gen = new Random(); // crea un generatore  
  
// genera una prima sequenza di 100 interi  
int[] casuali1 = new int[100];  
for(int i = 0; i < casuali1.Length; i++)  
    casuali1[i] = gen.Next();
```

```
gen = new Random(); // ricrea il generatore  
// genera una seconda sequenza di 100 interi  
int[] casuali2 = new int[100];  
for (int i = 0; i < casuali2.Length; i++)  
    casuali2[i] = gen.Next();
```

scrivendo quindi il codice per confrontare le due sequenze:

```
bool uguali = true;
for (int i = 0; uguali && i < 100; i++)
    if (casuali1[i] != casuali2[i])
        uguali = false;
if (uguali)
    WriteLine("le due sequenze sono identiche");
```

scopriremo che le due sequenze hanno lo stesso contenuto.

Per ovviare a questo problema dovremo utilizzare un seme sicuramente diverso nelle creazioni successive degli oggetti Random.

Una soluzione è quella di sfruttare un **GUID** (*Globally Unique Identifier*, identificatore unico globale). Si tratta di numeri pseudo-casuali generati dai sistemi in modo tale da essere sempre diversi; vengono ad esempio usati per identificare univocamente due componenti software al fine di distinguerle^[91].

C# consente di ottenere un tale numero attraverso la struttura **Guid**^[92] del namespace System. Ad esempio potremo creare un generatore con un seme affidabile in questo modo:

```
gen = new Random( Guid.NewGuid().GetHashCode() );
```

La struttura **Guid**, senza bisogno di usare Random, potrebbe tornare utile anche per generare sequenze di interi non ripetuti.

Per completezza segnaliamo che esiste anche un'altra classe utile per la generazione di numeri casuali che viene usata nella crittografia. Si tratta della classe **RNGCryptoServiceProvider**^[93] (RNG sta per “*Random Number Generator*”) del namespace **System.Security.Cryptography**. Nella pagina della documentazione ufficiale si può trovare un esempio di codice che la utilizza.

La classe **StringBuilder**

Visto che il tipo **string** rappresenta oggetti immutabili, è poco efficiente utilizzarlo quando dobbiamo effettuare diverse modifiche al contenuto di una stringa; come abbiamo visto nel capitolo 6 l'utilizzo dei metodi di manipolazione comporta sempre la creazione di nuove stringhe.

C# mette a disposizione una classe apposita di nome **StringBuilder**^[94] (namespace **System.Text**) per esigenze di modifiche frequenti ad una stringa.

Un oggetto **StringBuilder** può essere creato con l'operatore `new` con uno dei possibili costruttori utilizzabili che vedremo tra poco.

La stringa memorizzata in un oggetto **StringBuilder** viene gestita attraverso un array di `char` con una lunghezza di default di 16 caratteri. Si può impostare una capacità iniziale dell'array interno, nel costruttore.

La proprietà **Capacity** permette di conoscere o modificare (successivamente alla creazione) la capacità dell'oggetto fino al valore massimo per gli interi, vale a dire `2.147.483.647`(= `int.MaxValue`).

La lunghezza della stringa gestita da un oggetto **StringBuilder** si ottiene invece con la proprietà **Length** che è anche modificabile per accorciare o allungare la stringa: se si aumenta la lunghezza, i caratteri aggiuntivi assumeranno valore 0 (carattere '\0') e, se necessario, verrà aumentata la capacità dell'oggetto; attenzione invece che se si diminuisce `Length`, la stringa verrà troncata ma la capacità dell'oggetto rimarrà la stessa.

Non è infine possibile attribuire a **Capacity** un valore inferiore all'attuale lunghezza dell'oggetto **StringBuilder**.

Elenchiamo i possibili costruttori che possiamo usare per creare un oggetto **StringBuilder**:

- **StringBuilder()**
crea l'oggetto inizializzandolo alla stringa vuota
- **StringBuilder(int capacità)**
crea l'oggetto con una stringa vuota e la capacità iniziale specificata
- **StringBuilder(string valore)**
crea l'oggetto inizializzandolo al valore della stringa specificata

- **public StringBuilder(int capacità, int capacitàMassima)**
crea l'oggetto con la capacità iniziale specificata che può raggiungere un valore massimo specificato. Se durante l'elaborazione dell'oggetto la capacità supera il valore massimo viene sollevata un'eccezione.

- **StringBuilder(string valore, int capacità)**
crea l'oggetto usando la stringa e la capacità specificate
- **StringBuilder(string valore, int indiceInizio, int lunghezza, int capacità)**
crea l'oggetto inizializzandolo alla sottostringa estratta dal parametro, dall'indice e per il numero di caratteri specificati. Imposta anche una capacità iniziale.

Vediamo un paio di esempi di creazione di un oggetto:

```
StringBuilder sb;
```

```
sb = new StringBuilder();
WriteLine("lunghezza = " + sb.Length + ", capacità = " + sb.Capacity); // lunghezza = 0,
capacità = 16
```

```
sb = new StringBuilder("rossi");
WriteLine("lunghezza = " + sb.Length + ", capacità = " + sb.Capacity); // lunghezza = 5,
capacità = 16
```

Un oggetto di tipo `StringBuilder` consente di elaborare in modo efficiente una stringa mettendo a disposizione una serie di metodi per aggiungere, togliere e inserire elementi; non è invece disponibile l'operatore di concatenazione `+` come per il tipo `string`.

Di seguito un elenco dei principali metodi:

- **Clear()** pulisce l'oggetto `StringBuilder`
- **Append()** accoda una stringa ad un oggetto `StringBuilder`
- **Insert()** inserisce una stringa a partire da una certa posizione di un oggetto `StringBuilder`
- **Remove()** rimuove i caratteri da un oggetto `StringBuilder` a partire da una certa posizione e per un determinato numero di caratteri
- **Replace()** sostituisce parti di un oggetto `StringBuilder`.

I metodi `Append` e `Insert` possono ricevere anche variabili di tipo diverso dal tipo `string`; in tal caso l'elemento da inserire viene prima convertito in stringa per essere aggiunto o inserito nell'oggetto `StringBuilder`.

Anziché agire sulla proprietà `Length`, è preferibile usare questi metodi per allungare o accorciare un oggetto `StringBuilder` in quanto la capacità verrà

adattata di conseguenza.

Su un oggetto di tipo `StringBuilder` possiamo usare il metodo **ToString()** per ottenere una stringa con lo stesso contenuto dell'oggetto.

Inoltre, come per il tipo `string`, è possibile usare la notazione indicizzata con le parentesi quadre, per accedere ad un singolo carattere della stringa. Diversamente dal tipo `string`, possiamo anche modificare i singoli caratteri.

Riportiamo di seguito alcuni esempi di utilizzo dei metodi:

```
string s = "rossi";
sb = new StringBuilder(s);

sb[0] = 'R'; // modifica del primo carattere
sb.Append("Paolo"); // aggiunge una stringa alla fine

WriteLine(sb); // "RossiPaolo" (chiama implicitamente sb.ToString())

sb.Insert(5, " ");
WriteLine(sb); // "Rossi Paolo"

sb.Clear(); // svuota la stringa
for (int i = 1; i <= 10; i++)
{
    sb.Append(i);
    if (i < 10)
        sb.Append(" ");
}
// nuovo contenuto di sb: "1 2 3 4 5 6 7 8 9 10"

sb.Replace(" ", "-");
WriteLine(sb); // "1-2-3-4-5-6-7-8-9-10"
```

Come si diceva, l'uso del metodo **Remove** può comportare la modifica della capacità dell'oggetto; verifichiamolo con il seguente esempio:

```
WriteLine("lunghezza = " + sb.Length + ", capacità = " + sb.Capacity); // lunghezza = 20,
capacità = 32

sb.Remove(9, sb.Length - 9); // rimuove dal carattere di posto 9 fino alla fine, "contenuto di sb:
"1-2-3-4-5"

WriteLine("lunghezza = " + sb.Length + ", capacità = " + sb.Capacity); // lunghezza = 9,
capacità = 25
```

modificando invece la proprietà **Length**, la capacità non cambia. Possiamo ad esempio sostituire nel precedente esempio le ultime due righe di codice per verificare che la capacità rimane la medesima:

```
sb.Length = 9;
WriteLine("lunghezza = " + sb.Length + ", capacità = " + sb.Capacity); // lunghezza = 9,
capacità = 32
```

L'assegnazione tra oggetti `StringBuilder` copia il riferimento dell'oggetto di destra sulla variabile di sinistra. Potremo quindi avere più variabili che si riferiscono allo stesso oggetto, come mostrato di seguito:

```
StringBuilder sb1;
```

```
sb1 = sb; // sb1 si riferisce alla stessa stringa contenuta in sb  
sb1[0] = 'A'; // cambia la prima lettera di sb (lo stesso di sb1)  
WriteLine(sb); // "A-2-3-4-5"
```

Per creare invece un oggetto con lo stesso contenuto di un altro dovremo ad esempio usare un codice come il seguente:

```
sb1 = new StringBuilder(sb.ToString());
```

`sb1` è ora un nuovo oggetto indipendente da `sb`

Diversamente dal tipo `string`, l'operatore `==` non è stato ridefinito e, come per ogni altra classe, confronta i riferimenti di due oggetti `StringBuilder` che saranno quindi in generale sempre diversi.

Per confrontare il contenuto si usa invece il metodo **Equals** che restituisce **true** solo se l'oggetto al quale viene applicato ha la stessa stringa e gli stessi valori `Capacity` e `MaxCapacity` del parametro specificato.

```
StringBuilder sb2, sb3;
```

```
sb2 = new StringBuilder("ciao");  
sb3 = new StringBuilder("ciao");
```

```
bool test1 = (sb2 == sb3);  
bool test2 = sb2.Equals(sb3);
```

```
WriteLine(test1 + ", " + test2); // false, true
```


Bibliografia

- J. G. Brookshear – *Informatica. Una panoramica generale* – Pearson
- J. Albahari, B. Albahari – *C# 7.0 in a Nutshell* – O'Reilly
- A. Pelleriti – *Programmare con C# 7. Guida completa* – LSWR
- C. Nagel – *C# 7 and .NET Core 2.0* – WROX
- S. Del Furia, P. Meozzi – *Programmare con il .Net Framework* – Mondadori
- N. Wirth – *Algoritmi+Strutture dati=Programmi* – Tecniche Nuove
- <https://it.wikipedia.org/>
- https://en.wikipedia.org

[1] L'algoritmo di Euclide è uno tra i più antichi algoritmi conosciuti, risalente al 300 a.C.

[2] Parleremo diffusamente dell'argomento nel capitolo 9

[3] “booleano” deriva dal nome del grande matematico inglese George Boole che nel 1847 definì un’algebra per operare su dati che possono assumere solo i valori vero o falso.

[4] cfr. https://en.wikipedia.org/wiki/Structured_programming

[5] Per un indice dei linguaggi di programmazione più popolari può essere consultato il sito <https://www.tiobe.com/tiobe-index/>

[6] Per una lista dei linguaggi vedere https://en.wikipedia.org/wiki/List_of_CLI_languages

[7] La JVM nasce inizialmente solo con Java; dal 2003 altri linguaggi si sono basati sulla JVM, primi Groovy e Scala – cfr. https://en.wikipedia.org/wiki/List_of_JVM_languages

[8] <http://www.ecma-international.org/publications/standards/Ecma-334.htm>

[9] [https://standards.iso.org/ittf/PubliclyAvailableStandards/c042926_ISO_IEC_23270_2006\(E\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/c042926_ISO_IEC_23270_2006(E).zip)

[10] <https://github.com/dotnet>

[11] cfr. [https://it.wikipedia.org/wiki/Shell_\(informatica\)](https://it.wikipedia.org/wiki/Shell_(informatica))

[12] “Hello World” è il primo programma che abitualmente si scrive quando si impara un nuovo linguaggio di programmazione. E’ stato menzionato per la prima volta da Brian Kernighan nel suo tutorial al linguaggio B ed è diventato ampiamente conosciuto grazie al libro di Kernighan & Ritchie, “*The C Programming Language*”, del 1978 (libro famosissimo conosciuto anche semplicemente come K&R, tradotto in più di 20 lingue).

[13] Molti altri linguaggi (che hanno preso ispirazione dal linguaggio C) usano questo nome ma con l’iniziale minuscola, *main*.

[14] Si veda il capitolo 9 per una trattazione più precisa del concetto di classe e in generale della programmazione ad oggetti.

[15] La versione 6.0 è stata introdotta nel 2015 con il .NET Framework versione 4.6.

[16] Per questo motivo non possono essere usati tutti i tipi per una costante simbolica; i tipi ammissibili sono tutti quelli predefiniti in C# cioè i tipi numerici, string, char e bool oltre che i tipi enum.

[17] In realtà è possibile usare come identificatori parole riservate del linguaggio, nominando l’identificatore con il prefisso @. Generalmente si evita (renderebbe il codice poco chiaro) ma può

essere indispensabile se si utilizzano librerie esterne scritte in altri linguaggi che hanno parole chiavi diverse da quelle di C#. Per un elenco delle parole chiavi del linguaggio vedere

<https://docs.microsoft.com/it-it/dotnet/csharp/language-reference/keywords/>

[18] v. <https://it.wikipedia.org/wiki/Unicode>

[19] La versione 7.0 è stata introdotta nel 2017.

[20] La versione 7.2 (dicembre 2017) ha aggiunto anche la possibilità di inserire l'underscore per separare il prefisso 0b o 0x dal numero, scrivendo quindi ad esempio 0b_0111_1111_0110

[21] v. <https://docs.microsoft.com/en-us/dotnet/api/system.collections.bitarray>

[22] I diversi sistemi operativi usano modi diversi per indicare la fine di una riga in un testo; i sistemi Windows e altri usano la coppia di caratteri CR+LF, i sistemi Unix e derivati il solo carattere LF mentre i sistemi Commodore, Apple MacOS (fino alla versione 9) il solo carattere CR.

cfr. https://it.wikipedia.org/wiki/Ritorno_a_capo

[23] Il termine corretto sarebbe **overload** con il quale si intende che possono coesistere metodi con lo stesso nome ma parametri diversi. L'editor di Visual Studio suggerisce durante la digitazione quali sono i possibili overload di un metodo.

[24] La versione 3.0 è stata introdotta nel 2007.

[25] Altri linguaggi come C, C++ e Java si comportano in modo diverso: l'ambito di visibilità di una variabile inizia dal punto in cui la variabile è dichiarata fino alla fine del blocco. In C e C++ è anche possibile dichiarare una variabile locale ad un blocco con lo stesso nome di una variabile del blocco contenitore: in questo caso la variabile locale nasconde la variabile più esterna (*hiding delle variabili*). Questo comportamento c'è anche in C# e Java ma solo per i campi di classe che possono essere nascosti dalle variabili locali ai metodi.

[26] <https://docs.microsoft.com/it-it/dotnet/csharp/language-reference/operators/null-conditional-operators>

[27] Si veda ad esempio https://it.wikipedia.org/wiki/Complemento_a_due per una spiegazione del metodo utilizzato per rappresentare numeri interi con segno.

[28] Il verbo inglese **to coalesce** ha il significato di connettere, combinare

[29] Le stringhe verranno trattate nel dettaglio nel capitolo 6

[30] L'errore che si ottiene è del tipo "**System.NullReferenceException: 'Riferimento a un oggetto non impostato su un'istanza di oggetto'**"

[31] Provando questi esempi in C o C++ si potrebbero avere risultati diversi. In particolare gli ultimi due esempi assegnano alla variabile *i* il valore 3 in entrambi i casi. Si tratta di casi che appartengono ai casi di comportamento indefinito (*Undefined behavior*, cfr.

https://en.wikipedia.org/wiki/Undefined_behavior) ovvero a codice il cui risultato non è definito dalle specifiche del linguaggio di programmazione ma piuttosto da assunzioni del compilatore.

[32] Da C# 7.0 l'espressione dello switch può essere qualsiasi espressione non null.

[33] In realtà il compilatore ammette (seppur segnalandolo come warning) anche un blocco switch vuoto. Tale uso non ha molto senso!

[34] Il ciclo post-condizionale si traduce in Pascal con **repeat ... until**

[35] Nei cicli **while** e **do...while** non è invece possibile lasciare vuota la condizione

[36] rif. <https://docs.microsoft.com/it-it/dotnet/api/system.exception>

[37] Queste sono classi ereditate dalla classe Exception. Si veda il capitolo 9 per una trattazione generale della programmazione orientata agli oggetti.

[38] La parola chiave **string** corrisponde alla versione abbreviata della classe **System.String**. Per dichiarare una stringa possiamo usare equivalentemente anche la parola chiave **String**.

[39] *Verbatim* è un avverbio latino che significa “testualmente”, “parola per parola”. E’ anche il nome di una famosa azienda produttrice di supporti di memorizzazione.

[40] Ciò è possibile perché la classe String definisce un **indicizzatore**.

[41] Un modo alternativo e più efficiente per elaborare stringhe, utile nei casi in cui si devono fare modifiche frequenti, è usare la classe **StringBuilder** (v. appendice).

[42] Per una descrizione completa di tutti i metodi disponibili si veda <https://docs.microsoft.com/it-it/dotnet/api/system.string?#metodi>

[43] Gli operatori == e != sono stati ridefiniti nella classe String per effettuare un confronto carattere per carattere anziché un confronto tra riferimenti. Ricordiamo che il tipo String, come qualsiasi altro tipo classe, è un tipo riferimento.

[44] In particolare, per la lingua italiana, le lettere maiuscole vengono considerate successive alle lettere minuscole. Si noti che nell’ordine Unicode i codici delle lettere minuscole sono dopo le lettere maiuscole per cui un confronto tra char 'a' > 'A' dà esito positivo.

[45] Si veda <https://docs.microsoft.com/it-it/dotnet/csharp/language-reference/keywords/foreach-in> per dettagli su quali strutture possono essere usate con **foreach**.

[46] Per utilizzare questi metodi è necessario che l’array sia di un tipo per il quale sia definito un ordinamento.

[47] Questi metodi sono applicabili ad altri tipi di strutture dati, ad esempio alle liste.

[48] **Dicotomia** deriva dal greco con il significato di “divido in due parti”

[49] vengono talvolta chiamati linguaggi C-like

[50] Questo è comune ad altri linguaggi. Anche in C, C++ o Java il metodo eseguito all’avvio del programma si chiama **main** (con la “m” minuscola rispetto al maiuscolo usato in C#).

[51] Il concetto di derivazione o **ereditarietà** è un concetto importante nella programmazione ad oggetti. Una classe può ereditare da un’altra classe tutte le sue caratteristiche senza doverle reimplementare nuovamente.

[52] Un’altra convenzione utilizzata è la **Camel Case (notazione a cammello)** che viene adottata in Java sia per i metodi che per le variabili. In C# questa notazione viene invece utilizzata solo per le variabili.

[53] Invece in C e C++ è necessario dichiarare le funzioni prima di utilizzarle.

[54] Campi numerici a zero, campi booleani a false, campi stringa a **null**

[55] Ricordiamo invece (v. capitolo 3 su variabili, costanti e tipi di dato) che, riguardo alle variabili locali in un metodo, non è possibile dichiarare una variabile in un blocco annidato con lo stesso nome di una variabile dichiarata nel blocco contenitore, anche se la dichiarazione è successiva al blocco annidato.

[56] La versione 7.0 è stata introdotta a marzo 2017 con .NET Framework versione 4.6.2. Per una cronologia delle versioni di C# si veda <https://docs.microsoft.com/it-it/dotnet/csharp/what's-new/csharp-version-history> oppure la tabella alla fine del capitolo 1.

[57] Versione 7.2 introdotta a novembre 2017 con .NET Framework versione 4.7.1

[58] Questo sistema di completamento del codice va sotto il nome di **IntelliSense**

[59] <http://www.doxygen.nl/>

[60] Per una trattazione completa dei tag che si possono usare si veda <https://docs.microsoft.com/it-it/dotnet/csharp/codedoc>

[61] Versione 4.0 introdotta ad aprile 2010 con .NET Framework versione 4

- [62] La versione 7.0 ne ha esteso l'utilizzo anche ad altri elementi di una classe (costruttori, distruttore, proprietà e indicizzatori).
- [63] <https://www.jetbrains.com/decompiler/>
- [64] <https://github.com/icsharpcode/ILSpy>
- [65] cfr. https://it.wikipedia.org/wiki/Successione_di_Fibonacci
- [66] La leggenda (inventata dalla ditta che ha messo in commercio il rompicapo) narra che in un non ben precisato monastero Indù i monaci sono impegnati nello spostare 64 dischi d'ora su tre colonne di diamante e quando finiranno il lavoro il mondo finirà (cfr. https://it.wikipedia.org/wiki/Torre_di_Hanoi).
- [67] Il motto latino è riferito alla strategia, utilizzata fin dal tempo dei romani, secondo la quale creando divisioni all'interno di un popolo lo si poteva conquistare con più facilità.
- [68] Quick sort viene normalmente usato come algoritmo di ordinamento da molte funzioni di libreria presenti nei linguaggi di programmazione. Esistono altri algoritmi che hanno efficienze migliori ma si applicano a situazioni particolari, ad esempio l'ordinamento di numeri interi.
- [69] https://amturing.acm.org/award_winners/hoare_4622167.cfm. Il Premio A. M. Turing, intitolato al grande matematico inglese Alan Mathison Turing, è considerato il premio Nobel per gli informatici. È stato istituito nel 1966 dall'ACM (Association for Computing Machinery) e viene annualmente attribuito ad uno scienziato che si è distinto nel campo della computer science.
- [70] <https://it.wikipedia.org/wiki/Unix>
- [71] <https://it.wikipedia.org/wiki/Perl>
- [72] <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.regex.IsMatch>
- [73] <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.regex.Match>
- [74] <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.match>
- [75] <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.regex.matches>
- [76] <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.matchcollection>
- [77] <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.regex.replace>
- [78] <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.regex.split>
- [79] Per un elenco completo si può consultare <https://docs.microsoft.com/it-it/dotnet/standard/base-types/regular-expression-language-quick-reference>
- [80] v. <https://docs.microsoft.com/it-it/dotnet/api/system.text.regularexpressions.regexoptions> per altre opzioni.
- [81] <https://docs.microsoft.com/it-it/dotnet/api/system.enum>
- [82] Ciò è completamente vero per la prima versione di C#; per alcune tipologie di metodi (metodi anonimi, ...) introdotte nelle versioni successive, le variabili locali possono essere allocate anche nello heap (cfr. <https://devblogs.microsoft.com/premier-developer/dissecting-the-local-functions-in-c-7/>).
- [83] cfr. <https://csharp.2000things.com/2011/01/03/200-static-data-and-constants-are-stored-on-the-heap/>
- [84] Il linguaggio Assembly dei processori Intel prevede le due istruzioni PUSH e POP per manipolare i dati nello stack.
- [85] Visto che si tratta di un problema molto noto ai programmati, il nome “*Stack overflow*” è stato usato anche per un famoso sito web (<https://stackoverflow.com/>) nel quale si possono fare domande su vari linguaggi e argomenti di programmazione.

[86] Lo stack viene allocata in un blocco contiguo di memoria (senza salti); la sua dimensione di default è di 1 MiB su sistemi a 32 bit e di 4 MiB su sistemi a 64 bit.

[87] Rimane comunque il limite di 2GiB per i singoli oggetti (ad esempio array) allocati dal programma. Per superare anche tale limite (su sistemi a 64 bit) occorre impostare il progetto come indicato al seguente link <https://docs.microsoft.com/it-it/dotnet/framework/configure-apps/file-schema/runtime/gcallowverylargeobjects-element>

[88] <https://docs.microsoft.com/it-it/dotnet/api/system.gc>

[89] Si può verificare che la distribuzione dei numeri estratti è casuale, sviluppando un piccolo programma che generi una sequenza di numeri in un certo range e calcola la frequenza di uscita di ogni numero. Si noteranno valori delle frequenze molto vicini (tanti più sono i numeri estratti).

[90] La classe statica Environment consente di ottenere informazioni sul sistema e sull'ambiente di esecuzione, ad esempio la versione del sistema operativo, il numero di processori, ecc. (v. <https://docs.microsoft.com/it-it/dotnet/api/system.environment>)

[91] v. <https://it.wikipedia.org/wiki/GUID>

[92] <https://docs.microsoft.com/it-it/dotnet/api/system.guid>

[93] <https://docs.microsoft.com/it-it/dotnet/api/system.security.cryptography.rngcryptoserviceprovider>

[94] <https://docs.microsoft.com/it-it/dotnet/api/system.text.stringbuilder>