# Appunti di programmazione

## Table of Contents

# 1. Program design

Progettare un programma richiede prima di tutto di pensare a cosa si vuole che il programma faccia, e pianificare come lo deve fare. un buon punto di partenza a questo fine è considerare i 5 step di un programma, che sono :

- Inizializzazione
    - Il programma ed il sistema operativo compiono le operazioni di allocazione della memoria necessaria, e tutto ciò che è necessario per preparare l'ambiente per ricevere input dall'utente (crea le finestre, i form, pulsanti ecc.)
- Input
    - Il programma attende l'input dell'utente, e quando una data azione è compiuta, come la pressione del tasto invio, o di un pulsante ad esempio, passa alla fase successiva
- Process
    - Il programma prende i dati di input inseriti dall'utente, li interpreta, li elabora applicandoci i vari metodi, ed una volta finito, passa alla fase successiva
- Display
    - Il programma prende i dati ottenuti dalla fase di process e li mostra all'utente e poi torna in attesa di ulteriori input
- Termination
    - Il programma controlla cosa ha inizializzato e lo chiude, assicurandosi un'uscita dal programma senza errori e senza processi che continuano, o con memoria ancora occupata.

È buona norma scrivere il codice usando una tecnica chiamata *defensive coding*, ovvero scrivere il codice in modo che altri programmatori riescano a comprenderlo facilmente, e, utilizzare degli accorgimenti per ridurre il numero di errori:

- Impiegare i commenti in modo utile ed esplicativo
    - I commenti sono righe di codice che non vengono interpretate e compilate. Vengono utilizzate per creare documentazione per un programma e per fornire delle spiegazioni sui vari elementi del codice.

- Usare nomi sensati per le variabili
- Evitare numeri "magici"
    - I numeri magici sono numeri usati in espressioni, apparentemente senza motivo per un lettore esterno.
- Utilizzare costanti simboliche
    - le costanti sono valori che, appunto, non cambiano e dovrebbero avere un nome che rifletta il loro scopo.
- Utilizzare uno stile di sviluppo coerente e costante
    - Lo stile in cui viene scritto il codice dovrebbe rimanere invariato per tutto il programma, così che se un programmatore esterno dovesse leggere il codice sorgente, una volta interpretato lo stile riesca a leggerlo con relativa semplicità.
- Fare pause
    - Nel caso di debugging, lunghe sessioni tendono ad essere controproducenti, in quanto si finisce per vedere ciò che si vuole o pensa di vedere e non quanto sta realmente accadendo.
- Farsi aiutare
    - Sempre nel caso di debugging, può essere utile chiedere ad altri programmatori di dare un'occhiata al codice, un osservatore esterno può individuare un errore più in fretta.

Un programma è composto da *statement* che sono formati da diverse *espressioni*, le quali a loro volta sono composte da *operandi* ed *operatori*.

| Codice | Elemento |
|---|---|
| i = 10 * 6; | Statement |
| 10 * 6 | Espressione |
| * | Operatore |
| 10, 6 | Operandi |

L'operando è in genere un valore oppure una variabile, l'operatore è un operatore matematico, di assegnazione o logico che compie un'azione sugli operandi. L'espressione è uno o più operandi ed il loro operatore associato trattati come un'unica entità. Lo statement è una o più espressioni chiuse da un punto e virgola.

## 1.1. Operatori

In tutti i linguaggi di programmazione sono disponibili degli operatori che compiono delle azioni sugli operandi.

Gli operatori matematici consentono di eseguire operazioni matematiche sugli operandi, e restituiscono il risultato dell'operazione. Gli operatori relazionali e gli operatori logici invece restituiscono valori di tipo booleano (true/false) e vengono usati per validare gli input, come controller per i loop o per prendere eseguire un comando invece che un altro nel codice. L'operatore di assegnazione invece serve per attribuire ad una variabile un valore, che sia hard coded, il risultato di un'espressione, o un input dell'utente.

> Hard coding: inserire nel programma dei valori costanti non consentendone la modifica senza andare a modificare il codice sorgente del programma. L'hard coding rende il programma inflessibile.

In genere, gli operatori sono *left-associative*, ovvero si risolvono da sinistra a destra, tranne per l'operatore di assegnazione, che è *right-associative*. Inoltre generalemte gli operatori sono binari, cioè eseguono le operazioni con due operatori o espressioni. L'unica eccezione è l'operatore not che semplicemente inverte il risultato di altri operatori relazionali e logici.

| **Operatori Matematici** | | |
|---|---|---|
| **Operatore** | **Nome** | **Effetto** |
| + | Somma | Somma l'operando a sinistra per quello a destra |
| - | Sottrazione | Sottrae l'operando a sinistra a quello a destra |
| * | Moltiplicazione | Moltiplica l'operando a sinistra per quello a destra |
| / | Divisione | Divide l'operando a sinistra per quello a destra |
| % | Modulo | Restituisce il resto della divisione tra l'operando a sinistra per quello a destra |
| **Operatori di Assegnazione** | | |

| Operatori Matematici | | |
|---|---|---|
| = | Assegnazione | Assegna il valore dell'operando a destra all'operando a sinistra |
| **Operatori relazionali** | | |
| == | Uguaglianza | Restituisce un valore true se l'operando o l'espressione a sinistra è uguale a quello di destra |
| != | Disuguaglianza | Restituisce un valore true se l'operando o l'espressione a sinistra non è uguale a quello di destra |
| < | Minore | Restituisce un valore true se l'operando o l'espressione a sinistra è minore di quello di destra |
| <= | Minore o uguale | Restituisce un valore true se l'operando o l'espressione a sinistra è minore o uguale a quello di destra |
| > | Maggiore | Restituisce un valore true se l'operando o l'espressione a sinistra è maggiore a quello di destra |
| >= | Maggiore o uguale | Restituisce un valore true se l'operando o l'espressione a sinistra è maggiore o uguale a quello di destra |
| **Operatori Logici** | | |
| ! | Not | Inverte il risultato di una condizione |

| Operatori Matematici | | |
|---|---|---|
| && | And | Restituisce un valore true se entrambe le condizioni si verificano |
| \|\| | Or | Restituisce un valore true se almeno una condizione si verifica |

Sono disponibili delle versioni abbreviate di alcuni operatori.

| Operatori Matematici | | |
|---|---|---|
| **Operatore** | **Espressione abbreviata** | **Espressione estesa** |
| += | var += a | var = var + a |
| -= | var -= a | var = var - a |
| *= | var *= a | var = var * a |
| /= | var /= a | var = var / a |
| %= | var %= a | var = var % a |
| ++ | var++ | var = var + 1 |
| - - | var-- | var = var - 1 |

## Ordine degli operatori

Gli operatori hanno un ordine di risoluzione, ovvero l'ordine con cui vengono eseguite le operazioni. Questo ordine degli operatori può essre controllato tramite l'uso di parentesi. L'ordine degli operatori è il seguente:

1. ., ++, --, new
2. !
3. *, /, %
4. +, -,
5. <, >, <=, >=
6. ==, !=
7. &&, ||
8. ?:

9. =, *=, /=, %=, +=, -=

10,

# 2. Programmazione ad oggetti

Alla base della programmazione ad oggetti c'è la necessità di avere un sistema che permetta il riutilizzo del codice, e renda il codice stesso relativamente semplice da comprendere per un lettore esterno. Tutto ciò è reso possibile tramite l'utilizzo di **classi**

# 3. Classe

La classe è una descrizione semplificata dell'oggetto che si vuole implementare, un template per ottenere oggetti di quel tipo. Essendo un template, una classe non è utilizzata finché non viene creato un oggetto di quella classe. La descrizione dell'oggetto che si vuole utilizzare si ottiene attraverso **proprietà** e **metodi**.

- **PROPRIETÀ**
  - Le proprietà di una classe sono i valori e gli attributi che la descrivono.
- **METODO**
  - I metodi di una classe sono le azioni associate all'oggetto, che in genere, vanno a modificarne le proprietà.

```
ESEMPIO DI UNA CLASSE.
public clsFattura
float prezzoUnitario;
int quantitaOrdinata;
double impostaIva;
decimal prezzoTotale;
CalcoloIva();
CalcoloTotale();
```

DICHIARAZIONE != DEFINIZIONE
La dichiarazione di una qualsiasi variabile è ciò di cui il programma ha bisogno per creare quella variabile. una dichiarazione, come ad esempio `int prezzoUnitario;` dice al sistema operativo di assegnare a `prezzoUnitario` un indirizzo di memoria adeguato al tipo di dati che deve contenere, e che valori possono essere assegnati a quella variabile. La dimensione dell'indirizzo di memoria e

la tipologia di valori accettabili sono entrambi indicati dal data type, in questo caso `int`. La definizione di una variabile invece, oltre a fare ciò che fa una dichiarazione, assegna anche un valore alla variabile. La definizione `int prezzoUnitario = 20;`, oltre a creare la variabile `prezzoUnitario`, stabilisce che ha un valore di `20`. Definire una variabile può essere utile per evitare che vengano assegnati valori indesiderati o che possano rompere il codice. Inoltre è necessaria nell'utilizzo dei contatori dei cicli. Una variabile può essere dichiarata ovunque nel codice, può anche non essere mai definita, anche se ciò comporta ad uno spreco di memoria.

# 4. Proprietà

Come detto in precedenza le proprietà di un oggetto sono delle variabili che servono a descrivere l'oggetto stesso. Per variabili si intendono tutti i dati che possono cambiare in un programma. I dati delle proprietà possono essere di 3 tipi principali, che a loro volta si dividono in altri sottotipi.

- Numeric
- Textual
- Boolean

## 4.1. Numeric Data

I dati numerici si dividono in due categorie: integer e float.

## 4.2. Integer Data

L'integer data type si usa per rappresentare numeri interi, sia positivi che negativi. In base allo spazio occupato in memoria, alla grandezza del numero, ed alla presenza o meno di numeri negativi, ci sono diversi tipi di integer data.

| Nome | Dimensione | Range |
|---|---|---|
| byte | 8 | da 0 a 255 |
| sbyte | 8 | da -128 a 127 |
| short | 16 | da -32768 a 32767 |
| ushort | 16 | da 0 a 65535 |
| int | 32 | da -2,147,483,648 a 2,147,483,647 |

| Nome | Dimensione | Range |
|---|---|---|
| uint | 32 | da 0 a 4,294,967,295 |
| long | 64 | da -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807 |
| ulong | 64 | da 0 a 18,446,744,073,709,551,615 |

## 4.3. Float Data

Il data type float viene utilizzato per rappresentare numeri decimali, sia positivi che negativi. In base allo spazio occupato in memoria, alla grandezza del numero, alla precisione, ed alla presenza o meno di numeri negativi, ci sono diversi tipi di float data. La precisione di un float indica il numero di cifre rappresentate. Il decimal viene utilizzato principalmente per valori monetari, vista la sua grande precisione, ma è anche molto più lento da utilizzare rispetto agli altri tipi.

| Nome | Dimensione | Range | Precisione |
|---|---|---|---|
| foat | 32 | da ±1.5 * 10^-45 a ±3.4 * 10^38 | 7 |
| double | 64 | da ±5.0 * 10^-324 a ±1.7 * 10^308 | 15 |
| decimal | 128 | da ±1.8 * 10^-28 a ±7.9 * 10^28 | 28 |

CONSIDERAZIONI SUI DATI NUMERICI:

Con l'uso di dati numerici, ci sono delle considerazioni da fare per quanto riguarda le prestazioni e le necessità del programma. Cosa considerare

- Range:
  - bisogna tenere conto del range di numeri di cui si può aver bisogno, ed anche dell'utilizzo del programma che si sta sviluppando. Ad esempio, per controllare la temperatura di un forno per la fusione del vetro, il byte non è sufficiente.
- Memoria:

    - può capitare che la memoria disponibile sia limitata, come nel caso di piccoli elettrodomestici, quindi i tipi di numero dovranno essere il più piccolo possibile.

- CPU:

    - le cpu lavorano in modo molto più efficente con numeri del loro stesso tipo. Ad esempio, una cpu a 64b ci mette molto meno a processare un double o un long, rispetto ad un int o un float, nonostante siano più piccoli.

- Librerie:

    - i vari framework e linguaggi di programmazione hanno dei metodi che possono richiedere un tipo preciso di numero, come ad esempio i metodi math.

È possibile convertire un tipo di dato numerico in un altro tipo. Se la conversione ne aumenta le dimensioni si parla di *data widening* (ad esempio passare da float a double). Il processo inverso, ovvero la conversione ad un data type più piccolo, si chiama *data narrowing*.

## 4.4. Boolean Data

I dati booleani non sono propriamente un tipo a se stante di dati, e non hanno un valore, piuttosto controllano lo stato di una variabile. Una variabile di tipo booleano, dichiarata con `bool`, ha 2 possibili stati: `true` e `false`. In alcuni linguaggi, come sqlServer, il boolean viene sostituito dal `bit`, che come stati possibili ha 1 e 0.

## 4.5. Textual Data

Textual data indica tutti i dati che non vengono trattati come valori numerici o come valori booleani. Per tenere in memoria i singoli caratteri che compongono questi valori, si utilizzano le **stringhe**. Una stringa quindi non è altro che un insieme di caratteri interpretati dal programma come tali, quindi non processati, se non con metodi propri delle stringhe. In genere tutti i dati inseriti dall'utente sono inseriti come stringhe ed in seguito convertiti in altre tipologie di dati. Le stringhe hanno proprietà e metodi, il che le rende degli oggetti a tutti gli effetti.

**Concatenazione delle stringhe** * Uno dei modi per manipolare le stringhe è la concatenazione, ovvero la possibilità di unire una o più stringhe in successione.

```
string nome = "Mario";
```

```
string cognome = "Rossi";

string nomeCompleto = nome + " " + cognome;

Il risultato sarà

nomeCompleto = "Mario Rossi";
```

**Proprietà delle stringhe:**

- length:
  - l'unica proprietà delle stringhe in c#, inserisce la lunghezza di una stringa in una variabile int. La sintassi per richiamare la lunghezza di una stringa `string` è la seguente:

```
int length = string.Length;
```

**Metodi delle stringhe:**

Qui sono riportati i metodi principali delle stringhe in C#.

- ToUpper()
  - trasforma il testo in maiuscolo.

```
string.ToUpper();
```

- ToLower()
  - trasforma il testo in minuscolo.

```
string.ToLower();
```

- IndexOf(char, start)
  - trova la posizione di un carattere e restituisce un int; richiede come argomenti il carattere da cercare e la posizione nella stringa da cui parte la ricerca. Le posizioni in una stringa di 10 caratteri vanno da 0 a 9, se la ricerca fallisce, la ricerca restituisce un valore di -1.

```
int index = string.IndexOf(a, 5);

cerca la lettera a nella stringa string, ed assegna un valore ad index in
 base al successo o meno della ricerca, ed alla posizione del carattere.
```

- LastIndexOf(char)
    - trova l'ultima posizione di un carattere e restituisce un int; richiede come argomento il carattere da cercare. Se la ricerca fallisce, come IndexOf(), restituisce -1.

```
int lastIndex = string.LastIndexOf(a);

cerca la lettera a nella stringa string, ed assegna un valore ad index
 in base al successo o meno della ricerca, ed all'ultima posizione del
 carattere nella stringa.
[source, C#]
```

- SubString(start, length)
    - copia una parte di una stringa in un'altra stringa. Richiede come argomenti due numeri, il primo è la posizione da cui iniziare la copia, ed il secondo è la lunghezza della stringa da copiare.

```
string ciao = "ciao";

string substring = ciao.SubString(0, 2);

Il metodo copia due caratteri dalla stringa ciao e li inserisce nella
 stringa substring. Il risultato sarà

substring = "ci";
```

- Remove(start, length)
    - rimuove parte di una stringa. Richiede come argomenti due numeri, il primo è la posizione da cui iniziare la copia, ed il secondo è la lunghezza della stringa da eliminare.

```
string falso = "questa non è una stringa";
```

```
string tmp = falso;
string vero = tmp.Remove(6, 4);

Il metodo rimuove 4 caratteri dalla stringa tmp, partendo dalla posizione
 6, e va ad inserire il risultato nella stringa vero. In questo caso ed
 anche con il metodo Replace(), è consigliato usare una stringa temporanea
 per evitare perdita di dati accidentali.
```

- Replace("target", "replacement")
    - sostituisce parte di una stringa con la stringa desiderata.

```
string falso = "questa non è una stringa";
string tmp = falso;
string vero = tmp.Replace(" non", "");

Il metodo va a sostituire " non" nella stringa tmp con una stringa vuota.

Il risultato è

vero = "questa è una stringa";
```

- TryParse(string, out variable)
    - Il metodo cerca di interpretare il contenuto di una stringa, traducendolo in numero. Richiede come argomenti la stringa da interpretare e la variabile che andrà a contenere il valore numerico. Si usa la parola chiave out per indicare che la variabile può essere usata anche se non è stata inizializzata. Il metodo restituisce al caller un valore true se la conversione ha avuto successo, altrimenti da un valore false. Perché riesca a convertire la stringa in valore numerico, la stringa deve essere composta solo da numeri. In genere si usa una variabile booleana per chiamare il metodo, in modo da utilizzare poi la variabile come controllo dei dati inseriti. Viene chiamato usando il data type che ci si aspetta di ottenere. Ad esempio se si vuole che la stringa contenga un numero decimale, la sintassi sarà `float.TryParse(args)` .

```
bool flag;
int var;
string stringa "131ab";

flag = int.TryParse(stringa, out var);
```

```
In questo caso `flag` sarà `false`, in quanto `stringa` contiene due
 caratteri non numerici.
```

## 4.6. Escape sequences

Alcuni caratteri se inseriti in una stringa possono causare errori o effetti collaterali, per inserire questi caratteri, o per ottenere formattazioni particolari vengono usate delle escape sequences

| Simbolo | Effetto |
|---|---|
| \" | inserisce i doppi apici |
| \' | inserisce un apice singolo |
| \\ | mostra una backslash |
| \0 | null (non stampa) |
| \a | alarm (beep del sistema) |
| \b | backspace |
| \f | formfeed (pagina successiva) |
| \n | newline (nuova riga) |
| \r | carriage return (sposta a sinistra) |
| \t | tab |
| \v | tab verticale |

In alternativa con certi IDE si può usare @ davanti ad una stringa. La @ dice all'IDE di mostrare tutto il contenuto della stringa così com'è, senza cercare di interpretarlo.

## 4.7. Register value e reference value

Ogni variabile viene conservata in memoria, e per farlo viene assegnato ad ogni variabile un indirizzo di memoria, detto *lvalue*. L'indirizzo di memoria è una parte di memoria dalle dimensioni variabili che indica dove è conservato nella memoria il valore di una variabile. Le dimensioni dipendono dal data type in uso, mentre per le stringhe, dato che possono avere qualsiasi dimensione, l'indirizzo di memoria è sempre di 4 byte. L'lvalue è la memoria allocata per la variabile, ed esiste alla dichiarazione della variabile: `int i;` dice al sistema operativo di assegnare 4 byte di memoria alla variabile `i`. Il sistema operativo a sua volta risponderà all'IDE con l'indirizzo di memoria dove la variabile conterrà il valore, anche se non è stato assegnato per ora.

Quando viene assegnato un valore alla variabile `i`, allora i 4 byte di memoria allocati per la variabile conterranno un valore, e la *rvalue* avrà anch'essa quel valore.

```
int i;
```

Lo statement qui sopra dichiara una variabile i, e che è un int data type, quindi ha bisogno di 4 byte di memoria. Dato che non è stata definita la variabile, i avrà una lvalue precisa ad esempio 900,000, che indica dove sono i 4 byte di memoria necessari alla variabile, ma la rvalue è sconosciuta.

```
i = 10;
```

Con l'assegnazione di un valore alla variabile, adesso i 4 byte di memoria all'indirizzo 900,000 avranno un valore di 10.

In breve la lvalue indica dove è conservato in memoria il valore della variabile, mentre la rvalue contiene il valore vero e proprio.

Per quanto riguarda le stirnghe invece, la rvalue non va intesa come register value ma come *reference value*. Al momento della dichiarazione della stringa, vengono allocati 4 byte di memoria ed assegnato l'indirizzo di questi 4 byte alla lvalue della variabile. La rvalue però non è sconosciuta in questo caso, ma ha un valore: *null*: ciò significa che non sono contenuti dati utili all'interno della stringa. Quando verrà assegnato un valore alla stringa, allora la rvalue conterrà i singoli caratteri. Ogni carattere occupa 2 byte di memoria, e siccome vengono allocati 4 byte per la stringa, sarebbe possibile conservare sono 2 caratteri. La reference value però, a differenza della register value indica uno spazio di memoria flessibile, che può variare di dimensioni a seconda delle necessità. Questo significa che se per conservare la stringa servono più di 4 byte, verranno allocatii byte necessari.

**Cast**

Il cast si usa per "trasformare" un data type in un altro.

```
char c;
int val;
c=(char) val;
```

Nello snippet riportato qui sopra `val` passa dall'essere un `int` data type ad un `char` data type.

# 5. Metodi

Un metodo è una serie di istruzioni che possono variare in lunghezza, complessità e risultato, ma raggruppate in un'unico elemento. Tramite i metodi si vanno a fare operazioni di modifica, controllo, scrittura, lettura sulle proprietà dell'oggetto.

## 5.1. Definizione di un metodo

Per poter utilizzare un metodo, è necessario definirlo, ovvero scrivere la serie di istruzioni che vogliamo che esegua quando viene chiamato. Per farlo, le istruzioni sono ragruppate tra parentesi graffe. Le operazioni che il metodo deve compiere devono tutte essere comprese tra le due parentesi graffe del metodo, altrimenti verranno eseguite subito dopo la dichiarazione del metodo stesso. È possibile che il metodo restituisca qualcosa. in quel caso, viene specificato alla definizione, e prima del nome del metodo, cosa deve restituire.

```
int addendo1;
int addendo2;
int risultato;

public int somma() {
      risultato = addendo1 + addendo2;

      return risultato;
      }
Il metodo fa una semplice somma delle variabili addendo1 ed addendo2,
 assegnando il valore ottenuto a risultato.
```

## 5.2. Chiamata

Per poter effettivamente utilizzare un metodo definito, dobbiamo chiamarlo nel codice. Per chiamata si intende dirigere il programma ad eseguire le istruzioni contenute nel metodo.

```
flag = int.TryParse(string out var);
```

Nell'esempio qui sopra, la variabile `flag` è chiamata *caller*, in quanto richiama il metodo `TryParse`. Il metodo a sua volta, una volta completate le istruzioni al suo interno, assegna un valore alla variabile `flag`; quest'assegnazione è chiamata *return to the caller*.

## 5.3. Overloading dei metodi

Un metodo è *overloaded* quando ci sono più metodi che hanno lo stesso nome, ma hanno una *signature* diversa. La signature comprende tutto ciò che c'è tra il nome del metodo e la chiusura della parentesi tonda del metodo stesso. Il principale cambiamento tra un metodo e un altro è il numero o il tipo di argomenti da inserire richiamando il metodo.

## 5.4. Costruttori

Un metodo che ha lo stesso nome della classe a cui appartiene è chiamato *class constructor*, e serve per creare un nuovo oggetto di quella classe. Il costruttore è creato di default, e la sintassi per richiamarlo e creare un oggetto di una classe è la seguente

```
clsFattura miaClasse = new clsFattura();
```

La riga di codice riportata qui sopra crea l'oggetto `miaClasse` ed, essendo il costruttore di default, inizializza le variabili come sono state dichiarate nella classe `clsFattura`, oppure, qualora non siano state dichiarate, le inizializza con in valore di default in base al loro data type.

> Per molti valori numerici, il valore di default è 0.

In genere un costruttore deve sottostare a delle regole precise: * Deve avere sempre lo stesso nome della classe a cui fa riferimento. * Se si vuole istanziare una classe deve avere sempre la parola chiave `public`. * Non deve restituire dati al caller.

Il costruttore di default non ha argomenti tra le parentesi. È possibile tuttavia definire dei costruttori che accettino degli argomenti e che inizializzino delle variabili con un valore specifico; tale definizione va fatta all'interno della definizione della classe, ed il costruttore "overloaded" deve comunque sottostare alle regole degli altri costruttori.

```
public clsFattura(arg a, arg b) : this()
```

La sintassi riportata qui sopra è quella per definire un costruttore non default. I `:` e la parola chiave `this()` indicano che il costruttore non default inizializza tutte le variabili come il costruttore default, tranne quelle specificate nei suoi argomenti, in questo caso `arg a` e `arg b`, che avranno un vaolre diverso.

## 5.5. Metodi delle proprietà

Se le proprietà di un oggetto sono *private* non possono essere alterate al di fuori dell'oggetto stesso. Se non si possono alterare, lo stato dell'oggetto non cambia, e la flessibilità dell'oggetto e di tutta la programmazione ad oggetti viene meno.

La soluzione a questo problema si ha con dei metodi che vengono utilizzati per specificamente per accedere alle proprietà private di un dato oggetto. Tali metodi si chiamano *property methods* e sono composti da due metodi particolari: il *getter* ed il *setter*.

Il property getter viene usato per ottenere il valore di una proprietà mentre il property setter assegna ad una proprietà un valore.

Per convenzione il property method ha lo stesso nome della variabile, non ha parentesi e si distingue dalla variabile stessa perché inizia con la lettera maiuscola.

I property method devono essere sempre public, e devono lavorare con data type uguali a quelli della proprietà a cui fanno riferimento.

```
public int Var{

get {
  return Var;
 }
set {
  Var = value;
 }
}
```

value è una parola chiave che indica un valore non ancora specificato.

La sintassi qui sopra mostra la dichiarazione del property method per la variabile di tipo int var. È public, e restituisce un int, come la variabile a cui fa rifermento.

```
int externalVar;

clsName clsA = new clsName();
```

```
externalVar = clsA.Var;
```

Il codice qui sopra mostra l'utilizzo di un get: semplicemente si richiama il metodo della classe `clsA`, assegnando il valore di `var` alla variabile esterna alla classe `externalVar`. Per utilizzare invece il set, il codice è quanto segue:

```
int externalVar = 5;

clsName clsA = new clsName();

clsA.Var = externalVar;
```

A differenza del get, questa volta è il valore di `externalVar` ad essere assegnato a `var`.

È buona norma inserire nei property method, soprattutto nel set, delle righe che permettano la validazione dei dati.

## 5.6. Cohesion e Coupling

La cohesion ed il coupling dei metodi sono degli obiettivi che bisognerebbe porsi alla definizione dei metodi. Per cohesion si intende la possibilità di descrivere un metodo ed il suo funzionamento in una o due frasi. Ciò implica la creazione di metodi molto semplici, che in genere adempiono ad un incarico ed uno soltanto. I metodi multitasking tendono ad essere meno riutilizzabili, più complessi e con possibilità di errori più alte. Il coupling dei metodi invece indica il grado di dipendenza di un metodo da un altro. Il *decoupling* fa riferimento alla capacià di usare un metodo e cambiarne il codice senza dover alterare il codice di altri metodi. L'obiettivo è quello di avere metodi indipendenti tra loro, e quindi facilmente modificabili e soprattutto riutilizzabili.

# 6. Scope

Lo scope indica il livello di visibilità e la "vita" di una variabile, ovvero dove possono essere utilizzate. Lo scope serve a limitare interazioni indesiderate tra variabili, ed a facilitare la risoluzione di eventuali bug. In base a dove viene dichiarata la variabile, lo scope e la sua visibilità varia.

- Block scope
    - La variabile è dichiarata all'interno di un blocco di statement. Alla chiusura del blocco la variabile non è più utilizzabile

- Local scope
    - La variabile è dichiarata all'interno di un metodo, ma fuori da un blocco di statement. È utlilizzabile ovunque ma solo all'interno del metodo
- Class scope
    - La variabile è dichiarata all'interno di una classe, ma fuori da un metodo. È utilizzabile ovunque nella classe.
- Namespace scope
    - La variabile è dichiarata all'interno di un namespace, e può essere utilizzata ovunque in quel namespace.

Finché una variabile è *in scope*, la variabile è visibile e può essere utilizzata. Se si prova ad utilizzare una variabile che ha un block scope al di fuori del blocco in cui è dichiarata, si avrà un out of scope error.

# 7. Prendere decisioni nel codice

È inevitabile che prima o poi in un programma ci sia da prendere delle decisioni, come validare un input, o per quanto ancora eseguire un ciclo. Per prendere queste decisioni, si usano gli if e switch statement.

## 7.1. If/else statement

l'if è uno statement che esegue dei comandi se l'espressione tra le parentesi tonde risulta true. La sintassi dell'if è la seguente:

```
if(a==b) {
 c=d+e*f;
}
```

L' `if` controlla tramite operatori relazionali, ed eventualmente logici per verificare che l'espressione o espressioni al suo interno restituisca true. Se è così allora lo statement procede ad eseguire i comandi all'interno delle parentesi graffe. Se invece l'espressione o espressioni restituiscono false, semplicemente non esegue niente.

Se si vuole aggiungere altre istruzioni, nel caso la prima condizione non si sia verificata, viene utilizzato l' `else`.

```
if(a==b) {
 c=d+e*f;
}
else {
 c=g-h;
}
```

Lo statement dell' `else` , non richiede condizioni, in quanto lo statement viene eseguito se la condizione nell'if non viene soddisfatta.

## If nesting

Se fosse necessario verificare diverse condizioni, è possibile usare una concatenazione di if ed else statement.

```
if(a==b) {
 c=d+e*f;
}
else if(a<b) {
 c=g-h;
}
else if(a>b) {
 c=f-d;
}
else {
 return c;
}
```

Questa tecnica, pur funzionando, può risultare complesso da scrivere e leggere, ed inoltre inefficente. Per questo motivo, viene preferito lo *switch* statement.

È disponibile una versione abbreviata dell' `if` .

```
(a==b)? c=d+e*f : c=g-h;
```

# 7.2. Switch statement

Lo `switch` serve a risolvere il problema della concatenazione di if statement, che come detto sopra possono risultare particolarmente lunghi e diffici da scrivere ed interpretare. Lo switch esamina un'unica espressione, e pone dei casi a seconda del risultato dell'espressione.

```
switch (a + b) {
 case 1: {c = b; break;}

 case 5: {c = a; break;}

 case 140: {c = a / b; break;}

 case default: {b++; continue;}

}
```

Nell'esempio sopra vediamo che lo switch prende in considerazione l'espressione `a + b`, ed in base al risultato compie operazioni differenti: nel primo caso, se `a + b = 1`, assegnerà a `c` il valore della variabile `b`, nel secondo caso, se il risutato è 5, allora `c` prenderà il valore di `a` e così via. La parola chiave `case` specifica un possibile risultato dell'espressione compresa tra le parentesi. Non è necessario che i vari casi siano in ordine, o numerati. Nell'esempio, i numeri 1, 5, e 140, sono semplicemente i risultati possibili dell'espressione. Il `case default` indica un caso che viene eseguito se non dovessero verificarsi gli altri. Il break in questo caso e nei cicli, serve a passare il controllo al primo statement fuori dal ciclo. Il continue invece serve a rimanere ne loop; lo statement rimanda il controllo al contatore: nel for passa il controllo allo statement che incrementa la variabile contatore e nel do while rimanda al contarore che verifica il criterio di terminazione

# 8. Loop e ripetizione degli statement

Può capitare che sia necessario eseguire un comando diverse volte, oppure un numero sconosciuto di volte finché una condizione si verifica. In questi casi, invece di scrivere irghe di codice inutili, vengono usati i cicli.

I cicli, devono obbedire ad alcune regole per evitare che il controllo del programma rimanga nel ciclo e quindi si crei un loop infinito. Per evitare ciò, vengono utilizzati dei contatori, ovvero delle variabili che tengono conto dello stato di avanzamento nel loop, e che ne controllano l'esecuzione, e se uscire o meno dal ciclo.

- Lo stato del loop va inizializzato:
    - La variabile che controlla il loop, ovvero il contatore, deve avere un valore definito. Solitamente è 0.
- Fornisce un'espressione di controllo

    - Il contatore deve avere una condizione per la quale il ciclo non si ripete
- Lo stato della variabile va alterato
    - L'espressione che controlla la variabile e la sua condizione di terminazione, deve anche alterare lo stato della variabile affinché questa, ad ogni passaggio nel ciclo vada ad avvicinarsi alla sua condizione di terminazione

## 8.1. For loop

Il ciclo for esegue lo statement o gli statement al suo interno finché il contatore non raggiunge il suo criterio di terminazione, quindi in genere viene utilizzato per eseguire uno o più statement un numero specifico di volte.

```
for(int i = 0; i < 10; i++) {
 a = b + c;
}
```

La sintassi riportata sopra mostra un ciclo for. Tra le parentesi tonde è contenuta l'espressione di controllo: viene definita una variabile `i` come un int, e le assegna un valore di 0. Il secondo statement mostra il criterio di terminazione: finché `i` rimane minore di 10, il ciclo continua ad eseguire lo statement contenuto nelle parentesi graffe. Una volta eseguito lo statement, la variabile `i` viene incrementata, per poi controllare di nuovo se il criterio di terminazione è verificato o meno: se è verificato esce dal ciclo ed il controllo del programma torna all'entità che ha chiamato il metodo; se non è verificato, il ciclo si ripete.

## 8.2. While loop

Il while, a differenza del for, richiede che il contatore sia inizializzato prima del loop, e che l'incremento al contatore venga effettuato all'interno delle parentesi graffe.

```
int i = 0;
while(i < 10) {
 a = b + c;
 i++;
}
```

Il for ed il while sono molto simili, infatti, quasi tutti i while sono riscrivibili come for e viceversa. In genere si usa il for quando si è a conoscenza del numero di volte in cui eseguire il ciclo.

## 8.3. Do-while loop

Il do while fa sempre almeno un passaggio nel loop, e poi valuta se ne deve fare un altro o meno. Questo è dovuto al fatto che il criterio di terminazione è posto alla fine del ciclo.

```
int i = 0;
do{
 a = b + c;
 i++;
}while(i < 10)
```

# 9. Array

Un array è un gruppo di dati identici con un nome in comune. E utilizzato per tenere grandi quantità di dati dello stesso tipo, ad esempio una mailing list o una lista di numeri di telefono. Per effettuare operazioni sugli array vengono utilizzato molto i cicli. La sintassi per definire un array è la seguente:

```
int[] nuovoArray = new int[10]
```

Il codice riportato qui sopra mostra la creazione di un array di numeri interi, che contiene 10 elementi. Non vengono specificati i singoli elementi, in questo caso l'array è vuoto, ovvero contiene 10 valori `null`. Alla creazione dell'array è possibile specificare il contenuto dell'array stesso con la sintassi seguente:

```
int[] nuovoArray = new int[10] {21, 154, 32, 1, 54, 26, 97, 76, 67, 93};
```

La posizione di ogni singolo elemento nell'array è chiamata *indice*: l'indice indica dove nell'array si trova l'elemento. Negli array, per numerare gli indici, viene applicata la **regola n-1**: in un array, l'indice massimo è sempre il numero di elementi -1; questo perché il primo indice di un array è sempre 0. Ciò significa che nell'array `nuovoArray`, gli indici vanno da 0 a 9.

Per indicare un elemento di un array, quindi si utilizzano gli indici:

```
a = nuovoArray[9];
```

Lo statement va ad assegnare alla variabile `a` l'elemento in posizione 9 dell'array `nuovoArray`, quindi 93.

## 9.1. Array multidimensionali

Gli array multidimensionali sono matrici, ovvero insiemi di valori organizzati in righe e colonne. La dichiarazione di un array multidimensionale è simile a quella di un array, ma viene usata una virgola per differenziare le varie dimensioni.

```
int[,] matrix = new int[5,4];
```

Questo statement dichiara un array a 2 dimensioni, con 5 righe e 4 colonne. La regola n-1 si applica anche al numero di virgole per definire altre dimensioni, quindi `int[,,]` indica un array a 3 dimensioni, `int[,,,]` ne avrà 4 e così via.

In base al numero di dimensioni, un array avrà un rank: rank 2 per due dimensioni, 4 per 4 dimensioni e così via.

## 9.2. Array dinamici

Gli array sono oggetti di dimensioni fisse, ciò li rende poco flessibili perché bisogna sapere esattamente quanti elementi devono essere contenuti nell'array. Un array quindi va definito con il worst-case design, che può portare ad allocare uno spazio eccessivo rispetto al necessario.

```
ArrayList dynArray = new ArrayList();
```