

Appunti di programmazione

Indice

Program design.....	1
Operatori	3
Programmazione ad oggetti	5
Classe.....	5
Proprietà.....	6
Numeric Data	6
Integer Data.....	6
Float Data.....	7
Boolean Data.....	8
Textual Data	8
Escape sequences	11
Register value e reference value	12
Metodi.....	13
Definizione di un metodo	13
Chiamata	13
Overloading dei metodi	14
Costruttori	14
Metodi delle proprietà	14
Cohesion e Coupling	16
Scope	16
Prendere decisioni nel codice	16
If/else statement.....	17

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.

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 generalmente 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		
=	Assegnazione	Assegna il valore dell'operando a destra all'operando a sinistra
Operatori relazionali		

Operatori Matematici		
==	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
&&	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

Operatori Matematici		
/=	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 essere controllato tramite l'uso di parentesi. L'ordine degli operatori è il seguente:

1. ., ++, --, new
2. !
3. *, /, %
4. +, -,
5. <, >, <=, >=
6. ==, !=
7. &&, ||
8. ?:
9. =, *=, /=, %=, +=, -=
10. ,

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**

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.

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

Numeric Data

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

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
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

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 $\pm 1.5 * 10^{-45}$ a $\pm 3.4 * 10^{38}$	7
double	64	da $\pm 5.0 * 10^{-324}$ a $\pm 1.7 * 10^{308}$	15
decimal	128	da $\pm 1.8 * 10^{-28}$ a $\pm 7.9 * 10^{28}$	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 efficiente 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*.

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.

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.

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
<code>\"</code>	inserisce i doppi apici
<code>\'</code>	inserisce un apice singolo
<code>\\</code>	mostra una backslash
<code>\0</code>	null (non stampa)
<code>\a</code>	alarm (beep del sistema)

Simbolo	Effetto
<code>\b</code>	backspace
<code>\f</code>	formfeed (pagina successiva)
<code>\n</code>	newline (nuova riga)
<code>\r</code>	carriage return (sposta a sinistra)
<code>\t</code>	tab
<code>\v</code>	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.

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 stringhe 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 solo 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 allocati byte necessari.

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.

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 raggruppate 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.

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*.

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.

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 il 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 valore diverso.

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 riferimento.

```
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.

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 capacità 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.

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 utilizzabile 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.

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.

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.