Capitolo 3
Variabili e cenni sui tipi

Quando si programma si hanno alcuni punti fissi, praticamente in tutti i linguaggi di programmazione. Uno di questi è senza dubbio la presenza delle **variabili** che rappresentano dei valori che vengono memorizzati, manipolati ed eventualmente modificati a seguito delle operazioni applicate su di essi. Il nostro linguaggio non fa eccezione.

Nella forma più semplice le variabili in C# hanno, nell'ordine un **tipo**, un **nome** e possono avere anche un **valore iniziale**, mancando il quale si dicono "**unassigned**", ovvero non hanno un valore assegnato (ma è necessario attribuirgliene uno prima o poi se devono essere usate). Come detto la tipizzazione è di tipo statico quindi una variabile non può cambiare tipo nel corso del programma anche se in realtà sono permesse delle conversioni tra tipi compatibili. C# inoltre è un linguaggio type-safe e pertanto siamo sicuri che nello spazio allocato per una certa variabile non entrerà mai un valore incompatibile con il suo tipo. Vediamo alcuni esempi di dichiarazione:

int x = 0; inizializza la variabile x con il valore 0. La quale x, inoltre, è un intero in quanto int è la parola (shortcut) usata per definire quella tipologia numerica
string s = "ciao"; ci dice che s è una stringa e precisamente la sequenza di caratteri che compone la parola "ciao"
int z = 3 + 4; ovvero va bene anche una espressione per inizializzare una variabile.
int y; in questo caso y è una variabile unassigned. Come detto prima dell'uso andrà inizializzata
int i = “c”; questo invece non va bene per incompatibilità tra il tipo della variabile ed il valore iniziale assegnato
int x = metodo(); anche un metodo (purchè restituisca il tipo corretto) può essere assegnato ad una variabile.

Definire una variabile significa pertanto allocare lo spazio di memoria necessario per contenerla. Questo lavoro ovviamente lo fa il compilatore, dal punto di vista utente è tutto trasparente. La definizione di una variabile è propedeutica al suo uso, ovvero non potrete usare una variabile che non sia dichiarata ed inizializzata.

A ciascuna variabile poi può essere assegnato un certo grado di **visibilità** nei suoi riguardi dall'esterno, ad esempio:

```
(1) private int x = 1;
```

quella parola private è uno dei possibili **modificatori** il cui uso spiegheremo nel prossimo paragrafo.

Quanto segue pertanto può essere considerata la definzione formale completa di una variabile nel nostro linguaggio:

visibilità tipo nome = valore iniziale;

solo i campi in rosso sono necessari affinchè il parser del compilatore non si lamenti. Come detto il valore iniziale è tuttavia necessario sia assegnato prima di utilizzare la variabile. Questa definizione ricalca in modo preciso la espressione (1) posta qualche riga sopra.

Se la visibilità, e il tipo sono parole chiave e non modificabili e i valori iniziali devono seguire ovvie limitazioni, il nome delle variabili è a completa scelta dell'utente. Anche in C# abbiamo poche semplici regole per l'attribuzione dei nomi alle variabili, regole che non dovrebbero infastidire ne limitare troppo l'uso comune. Sono ammessi (quasi) tutti i caratteri **UNICODE**, in particolare vengono seguite le regole **Unicode Standard Annex 15**; in aggiunta, conformemente a quanto usato nel linguaggio C è ammesso l'uso del carattere _ (underscore) in prima posizione (molto spesso viene usato per variabili di uso speciale). E inoltre possibile trovare all'inizio il carattere @ che introduce un identificatore verbatim che permette anche ad una parola chiave di essere usata come identificatore, diversamente come è naturale, tali parole riservate sono di proprietà del linguaggio; questa caratteristica è utile per l'interazione con altri linguaggi aventi parole chiave differenti. Di norma comunque è molto meglio non usare parole chiave precedute da @ come identificatore. Si è già detto, ed è importante ricordare, che C# è case sensitive per cui x = 0 è diverso da X = 0; e xY è diverso da xy; ovviamente un nome di variabile può essere usato solo una volta nel programma nel'ambito dello stesso **scope**, vedremo tra breve di cosa si stratta. Bisogna invece tenere presente che due identificatori sono considerati uguali se risultano identici una volta compiute le seguenti operazioni:

- vengono tolti i caratteri @
- vengono sostituiti alle sequenze unicode i corrispondenti caratteri
- vengono eliminate eventuali formattazioni.

Quindi ad esempio @aa e aa sono considerati eguali e collidono se nello stesso scope. Vediamo ora alcuni esempi di nome di variabile (ovviamente vale la generica raccomandazione di usare nomi significativi e descrittivi nelle vostre applicazioni):

x è un nome accettabile
_xyz è anche accettabile
va?r non va bene perchè contiene un carattere non valido, il punto interrogativo, al suo interno.
x01 va bene
nome_utente è ok anche questo.
@array è corretto.
_@aaa non va bene, il carattere @ deve stare in prima posizione
@_aaa invece è ok.
3w inizia con un numero e non va bene
a33w invece è corretto in quanto inizia con un carattere
a#01 contiene il carattere # che non è valido.
int /u005a va bene ed equivale a int _a; questo conferma che anche le sequenze UNICODE sono accettabili usate nella forma /uNNNN

Una domanda che a volte viene posta è se esiste un limite alla lunghezza degli identificatori utilizzabili in C#. Per quanto ne so tale limite è di 511 caratteri, pur non essendo una richiesta del linguaggio by design. Se qualcuno vuole fare dei test, da parte mia l'unico tentativo fatto in merito mi ha confermato tale valore che mi sembra ampio a sufficienza per qualsiasi scopo.

Infine, cosa che risulta abbastanza comoda a livello di pura digitazione, è possibile dichiarare in un colpo solo più variabili dello stesso tipo usando la virgola come separatore, purchè siano tutte dello stesso tipo, ad esempio:

```
int x, y, z; oppure
int y = 0, t = 1, v = 2;
```

Problematica strettamente legata al discorso relativo alle variabili è quello della sua **visibilità** o ambito, si parla anche di area di validità, o"**scope**", a cui accennavamo, usando il termine inglese corrispondente che userò anche nel prosieguo in quanto più diffuso nella letteratura anche di lingua italiana, ed è relativo in termini pratici all'individuazione della zona nella quale una variabile può essere usata o meglio è visibile semplicemente tramite il suo nome, quindi senza ulteriori specificazioni. Detto così può apparire un po' complesso, in realtà questo problema è legato strettamente al concetto di blocco visto nel paragrafo precedente. Una variabile in pratica vive e muore nel suo blocco all'interno del quale può essere utilizzata mentre non è visibile e tantomeno utilizzabile all'esterno dello stesso. L'esempio che segue è chiarificatore e ci fa apprezzare i vantaggi visivi della indentazione:

C# Esempio 3.1

```
1  using System;
2  class program
3  {
4    public static void Main()
5    {
6      {
7      int a1 = 0;
8      Console.WriteLine(a1 + 1);
9      }
10   Console.WriteLine(a1 + 2);
11   }
12 }
```

Se cercate di compilare questo programma ricevere un messaggio dal compilatore con un chiaro riferimento alla riga 10 dell'esempio 3.1:

error CS0103: The name 'a1' does not exist in the current context

Per chiarire ulteriormente la cosa possiamo anche fare riferimento alla definizione ufficiale di “scope” nella definizione di C#: "The scope of a local variable declared in a local-variable-declaration **is the block** in which the declaration occurs. It is a error to refer to a local variable in a textual position that precedes the variable declarator of the local variable." Quindi nell’esempio 3.1 semplicemente la variabile a1 esiste solo all'interno del blocco di codice che inizia alla riga 6 e termina alla 9. All'esterno di questo blocco siamo ad un livello "superiore", o più “esterno” se volete, come l'indentazione suggerisce visivamente e quindi a1 non è visibile. La segnalazione del compilatore è molto esplicita in tal senso. Invece, se modifichiamo il programma come segue:

C# Esempio 3.2

```
1  using System;
2  class program
3  {
4    public static void Main()
5    {
6      {
7        int a1 = 0;
8        Console.WriteLine(a1 + 1);
9        {
10         Console.WriteLine(a1 + 2);
11       }
12     }
13 // string a1 = "";
14   }
15 }
```

allora tutto funziona in quanto la richiesta di scrittura alla riga 10 è più interna rispetto alla dichiarazione di a1 e quindi la può utilizzare. Ovviamente non è possibile togliere il commento che blocca l'esecuzione del codice alla riga 13. Se lo facessimo la variabile a1 ivi definita, essendo ad un livello più alto di quella alla riga 7 collliderebbe con la stessa in quanto teoricamente visibile dal codice a livello più basso. Insomma non avviene quello shadowing, o hiding, quel fare ombra da parte della variabile più interna rispetto a quella più esterna che si ha in altri linguaggi come ad esempio il C++. Questo a parer mio è un vantaggio in quanto senza perdere nulla in potenzialità vengono evitati bug la cui individuazione non è sempre banale. Il codice seguente invece funziona:

C# Esempio 3.3

```
1  using System;
2  class program
3    {
4    public static void Main()
5      {
6        {
7          int a1 = 0;
8          Console.WriteLine(a1 + 1);
9          {
10           Console.WriteLine(a1 + 2);
11         }
12       }
13       {
14         int a1 = 9;
15         Console.WriteLine(a1);
16       }
17     }
18   }
```

perchè evidentemente le due variabili a1 sono l'una fuori dalla visibilità dell'altra.

Una particolarità che vale la pena sottolineare, a mio avviso non immediatamente intuibile in base alle definizioni formali, è che in effetti lo scope di una variabile è il suo blocco ma l’utilizzo della stessa è inibito fino alla linea di codice, nel blocco, che esplicita la definizione. Sembra complicato ma non lo è, come vediamo dal seguente blocco di codice:

```
{
        Console.WriteLine(y);
        int y = 9;
}
```

Y, evidentemente, appartiene al blocco di codice al quale afferisce anche l'operazione di scrittura. Tuttavia il compilatore, se gli passate questo codice si lamenta:

error CS0841: Cannot use local variable 'y' before it is declared

più chiaro di così.... In altri linguaggi, come Java, la variabile è definita solo quando viene incrociata la riga in cui essa è definita. All'atto pratico in realtà non cambia nulla e se il comportamento tra i due linguaggi è diverso internamente questo non impatta in alcun modo all'esterno.

Quando parleremo delle classi incontreremo anche il concetto di variabile di classe e vedremo delle eccezioni a questo comportamento. In generale comunque vale sempre questo concetto di area delimitata entro il quale una variabile esiste, area che può anche essere una classe, un metodo o una struct o un enumerativo, non necessariamente quindi una anonima porzione di codice. Come detto lo scope di una variabile è direzionato verso i livelli più interni del codice nel viene definito mai verso l'esterno, questo è il punto cardine di tutto il discorso che vale la pena ribadire. Il concetto di scope è molto importante e consiglio di fare qualche prova. Lo reincontreremo, anche se nulla di eclatante sarà aggiunto, nel capitolo 5 ove parleremo dei cicli di controllo dell'esecuzione del flusso del programma.

A volte può essere utile, definendo una variabile, fornire ad essa una forma di protezione che la tuteli da futuri possibile modifiche del valore in essa memorizzato. Si può allora fare uso del prefisso **const**. Come è intuibile dal nome stesso quindi una variabile di tipo const rimane costante per tutta la durata del programma e non può essere modificata. I vantaggi che offre l'uso delle costanti è dato dal fatto che, paradossalmente, possono rendere più agevoli le modifiche complessive di un programma nonchè la sua gestione e inoltre, non essendo modificabili, ci possono aiutare a prevenire gli errori nel senso che un valore in una const è in cassaforte. Dal momento che il compilatore sa tutto di una costante è la gestisce in maniera ottimale. Una variabile const è anche static implicitamente tant'è che il compilatore non ci permette di appiccicargli esplicitamente l'etichetta static perchè in pratica c'è già. L'uso delle costanti è molto semplice anche concettualmente ed è soggetto soltanto ad un paio di regole di base:

- una costante deve essere inizializzata al momento della sua dichiarazione
- inoltre non può essere inizializzata tramite una variabile ma deve avere un valore definito e noto già a compile time. Quest'ultima limitazione può essere superata coi campi **readonly** che affronteremo insieme alle classi.

Di seguito un facile esempio:

C# Esempio 3.4

```
1  using System;
2  class program
3  {
4  public static void Main()
5    {
6    int venti = 20;
7    const int dieci = 10;
8    Console.WriteLine(venti + dieci);
9    // dieci = 20; se tolgo il commento il compilatore si lamenta....
10   }
11 }
```

la riga 9 è errata in quanto tenta di modificare un valore dichiarato const. Di questo argomento parleremo ancora.

Una novità interessante ed abbastanza importante invece è la seguente ispirata anche da altri linguaggi di programmazione; in alcuni di essi, infatti, sentirete parlare di inferenza di tipo. Si tratta di un concetto teoricamente abbastanza semplice: **in pratica il compilatore deduce il tipo di una variabile in modo automatico basandosi sul valore ad essa assegnato**. Ad esempio scrivendo:

```
x = 5
```

il compilatore, in alcuni dei linguaggi che seguono, assegnerà ad x il tipo intero in modo automatico. Questa possibilità è offerta da numerosi linguaggi funzionali e si sta estendendo a molti altri. Così Haskell, OCaml, ad esempio, da sempre supportano questa modalità ma anche C++0x avrà questa features e si unirà a F#, Nemerle, Boo, Cobra, Delphi, D, Scala, Falcon, Ruby, Python e così via. In C# questa feature è stata introdotta dalla versione 3.0 in avanti e fa uso della parola **var**. Ecco l'esempio:

C# Esempio 3.5

```
1  using System;
2  class program
3  {
4    public static void Main()
5    {
6      var i = 9;
7      Console.WriteLine(i.GetType()); //esplicita il tipo di x
8      string var = "ciao";
9    }
10 }
```

L'output, determinato dall'istruzione alla riga 7, è semplicemente:

System.Int32

GetType restituisce il tipo della variabile a cui viene applicato, lo vedremo in questo stesso paragrafo. Nel caso in esame la nostra variabile i ha valore 9 e il compilatore gli assegna il tipo standard int.

La riga 8 è interessante perchè ci fa capire come var non sia in realtà una keyword in senso stretto in quanto possiamo definire ad esempio un'altra variabile, ma anche una classe, un array o quel che ci pare, che si chiama appunto "var". L'inferenza si attiva quando var è nella posizione che normalmente sarebbe occupata dal tipo. In quel caso il compilatore è istruito a ricavare il tipo dal valore posto a destra dell'espressione ed assegnarlo alla variabile. Il motivo per cui var non è una keyword personalmente lo ignoro; è possibile che si tratti di questioni di compatibilità con programmi scritti con versioni precedenti di C# in cui la parola var, che non era riservata in alcun modo, poteva essere liberamente utilizzata e ovviamente ci sarebbero dei problemi ricompilando... ma è una mia supposizione.

L'uso dell'inferenza è sottoposto a qualche limitazione.

- Essa può essere applicata solo a variabili locali (quindi ad esempio non a parametri e nemmeno ai campi di una classe e nemmeno a valori di ritorno per un metodo a meno che questi non coincidano con la signature del metodo stesso, vedremo un esempio quando parleremo dell'argomento metodi) evidentemente sempre per non mandare in cofusione il compilatore.
- Va sempre attribuito un valore in fase di dichiarazione, valore che comunque non potrà essere null, questo è ovvio se no il compilatore non sa che fare.

Quindi:

```
var x; // non va bene
var x = 0; // ok
var x; x = 0; // non va neanche così. Il valore va attribuito in fase di dichiarazione
int x = 0; var y = x; // questo è ok
var y = null // non va
```

Va comunque precisato, anche se intuibile, che una volta che il compilatore attribuisce un tipo ad una variabile valgono i principi di tipizzazione forte di questo linguaggio. Ovvero:

```
var s = "aaa"; // s è una stringa a questo punto
s = 10; // non va bene
```

Una nota importante è che in pratica non esistono particolari limitazioni per il tipo su cui si può "inferire" la variabile, ovvero può essere built-in, user defined, anonimo ecc... Per quanto raro può capitare inoltre che un tipo chiamato "var" sia presente a livello di scope quando noi definiamo la nostra variabile da inferire. In quel caso il compilatore fa riferimento al tipo definito non al meccanismo di inferenza. Usate nomi di tipi sensati, siete avvistati.... Come vedremo la keyword var può essere usata anche con gli array, pur se la cosa diventa un attimo più complicata.

A questo punto del nostro excursus questa feature non aggiunge poi molto (magari è utile per limitare in qualche modo la digitazione di tipi dalla descrizione lunga, pur se la leggibilità sarà comunque un po' penalizzata) e sarà più utile, quando parleremo dei tipi anonimi. Per ora basta memorizzare l'esistenza di questa possibilità. Su Internet tra l'altro esistono anche dei dibattiti sui reali vantaggio nell'uso di var... a mio modesto avviso nelle condizioni giuste questi vantaggi ci sono sicuramente.

TIPI

Accenniamo ora genericamente il discorso sui tipi in .Net e C# che verrà comunque ben approfondito in seguito. Sulle variabili abbiamo scritto tante cose ma fondamentale è il fatto che nessuna variabile ha un senso, in un linguaggio fortemente tipizzato, se non ha, appunto, un tipo ad essa associato, come da definizione; la **tipizzazione forte** e la **type-safety**, ovvero la conoscenza da parte del CLR del tipo di ogni variabile a run-time, sono, come detto e ridetto, i cardini del type system di .Net e quindi di C#. I discorsi relativi ai tipi sono molto importanti in .Net, ricordate ad esempio cosa si è detto relativamente al CTS. In questo ambito sono ammesse due famiglie di tipi:

- tipi valore o value types nella dizione inglese che userò nel prosieguo
- tipi di (o per) riferimento o reference types.

Come detto per ora non consideriamo i puntatori unsafe.

Tra i primi troviamo ad esempio gli interi (int) e i caratteri singoli (char) mentre tra i tipi di riferimento abbiamo le classe, disponibili nella BCL o create dall'utente, gli array, le stringhe e così via, come vedremo nel prosieguo del nostro cammino. Evidentemente ogni tipo si porta dietro una certa parte di informazioni che saranno usate dal compilatore per effettuare delle verifiche. Ad esempio i valori massimi e minimi ammissibili, lo spazio di memoria necessario per l'allocazione, le operazioni che sono permesse su quel tipo, in linea con l'impostazione generale del linguaggio e così via. Da un punto di vista pratico i tipi valore sono forse di comprensione più immediata (certamente un intero è concettualmente più semplice di una classe, anche se nella realtà vedremo che non è poi tanto vero) e rappresentano elementi immutabili (come natura, non in termini di valore); d'altronde si chiamano value types quindi è comprensibile che, a logica, descrivano dei valori. In assoluto esso si dividono in due categorie:

1. strutture (a loro volta sono distinguibili in interi, floating point, decimal, bool e user defined)
2. enumeratori

da qui si evince (si evincerà, forse è meglio) che anche gli interi o i caratteri sono strutture in C# e dichiarare una variabile di quel tipo significa istanziare una struttura. Gli enumeratori sono un'altra cosa ancora e vvoiamente ne parleremo più avanti.

I reference types si possono invece dividere in:

1. object types
2. interface types
3. pointer types (managed e unmanaged)
4. built-in reference types

non preoccupatevi per ora di approfondire di cosa si tratta ogni cosa sarà ampiamente sviluppata nel corso dei prossimi paragrafi. I reference types sono un po' più complicati e sfuggenti in prima battuta e complessivamente sono forse meno intuitivi per cui andranno studiati con calma. Analogamente non vale la pena di fissarsi troppo sulla differenziazione tra dati built-in e user-defined. Tutto sommato, fatta salva l'origine, lavorano allo stesso modo e non ci sono particolarità comportamentali eclatanti in genere.

Tecnicamente parlando possiamo differenziare le due tipologie come segue:

- una variabile value type altro non è che una cella di memoria che contiene un valore coerente con la sua natura.
- Una variabile appartenente ai reference types invece è in pratica un puntatore ad una zona di memoria in cui sono contenuti (o iniziano) i suoi dati. Una secondo me brillante analogia li paragona agli URL, ai link nelle pagine Web che puntano ad altre locazioni sulla rete.
- I value types sono allocati nello stack qualora si tratti di variabili locali o parametri di metodi. Non date retta a chi vi dice che tutti i value types vivono sempre nello stack perchè non è assolutamente vero. Esempio facile sono i campi delle classi che sono rappresentati da tipi valore; nonostante la loro natura, proprio per il fatto di appartenere ad una classe, pascolano beatamente nello heap.
- I reference types sono sempre allocati nello heap, loro si hanno fissa dimora, e sono realmente interessati dall'operato del garbage collector, meccanismo automatico del quale parleremo in seguito. Normalmente non ci importa in alcun modo sapere dove sono collocati nello heap i vari tipi.
- Si vuole che value types siano normalmente più leggeri e prestazionalmente preferibili, laddove i reference types sono più potenti e duttili. In realtà le cose non stanno esattamente così sia perchè anche i value types si portano dietro un bel po' di informazioni essi stessi, come vedremo, quindi tanto leggeri non lo sono, sia perchè, in realtà, in alcuni casi è vero il contrario anche per quanto concerne le performance, alcune operazioni richiedono pochi bytes rispetto a trasferimenti dati più pesanti se realizzate tramite value types. Insomma si tratta di concezioni un po' da rivedere per quanto diffuse.
- I value types non possono avere eredi, nel senso che sono come le foglie di un albero, se mi passate l'immagine, mentre i reference-types possono avere dei discendenti.
- A loro volta i value-types possono discendere solo da System.ValueType mentre è evidente che i reference type possono avere progenitori di diversa origine
- I reference types accettano il valore null i value types no.

Un concetto importante legato ad entrambe le tipologie è quello di istanziazione. Partiamo dalla considerazione che quando definiamo una classe (reference types) una struct o un enumerativo (value types) stiamo in pratica definendo un nuovo "tipo" di dati; il processo di istanziazione è, in soldoni, la definizione di una variabile basata sul nuovo tipo definito. La cosa bella però è che mettiamo praticamente sempre in moto il meccanismo di istanziazione anche quando definiamo un intero o un carattere e quindi vale, anche se meno evidentemente, anche per i tipi predefiniti. Lo vedremo più avanti ma per il momento, teniamoci stratta questa idea che il legame tra una variabile ed un tipo è il processo di istanziazione.

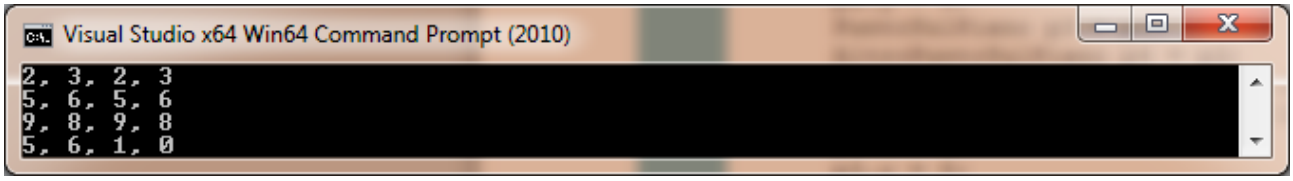
Abbiamo appena spiegato che **i tipi valore sono valori effettiv**i mentre i **reference lavorano per riferimenti**. Vediamo adesso in pratica questa cosa. Prendiamo il seguente esempio, senza aver la pretesa che sia tutto chiaro ora (bisogna avere una comprensione di base delle classi e delle strutture per avere padronanza dell'esempio per cui sarà cosa buona rivederlo dopo aver appreso quegli argomenti) e premettendo da subito che ribadiremo l'argomento, in maniera puntigliosa, anche in altri paragrafi, ad esempio quello relativo agli array (capitolo 8):

C# Esempio 3.6

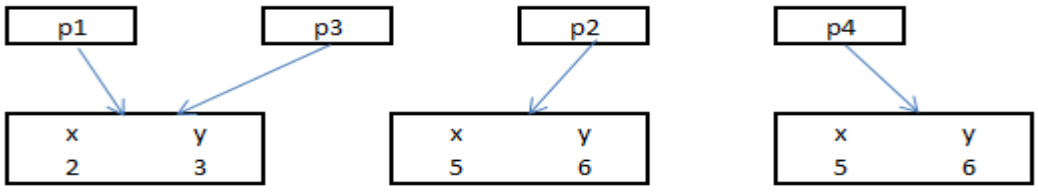
```
1  using System;
2
3  class PuntoSulPiano
4  {
5      public PuntoSulPiano(int x, int y)
6      {
7          this.x = x;
8          this.y = y;
9      }
10     internal int x;
11     internal int y;
12 }
13
14 struct AltroPuntoSulPiano
15 {
16     public int x;
```

```
17      public int y;
18  }
19
20  class program
21  {
22    public static void Main()
23    {
24        PuntoSulPiano p1 = new PuntoSulPiano(2, 3);
25        AltroPuntoSulPiano p2 = new AltroPuntoSulPiano();
26        p2.x = 5;
27        p2.y = 6;
28        PuntoSulPiano p3 = p1;
29        AltroPuntoSulPiano p4 = p2;
30        Console.WriteLine("{0}, {1}, {2}, {3}", p1.x, p1.y, p3.x, p3.y);
31        Console.WriteLine("{0}, {1}, {2}, {3}", p2.x, p2.y, p4.x, p4.y);
32        p3.x = 9;
33        p3.y = 8;
34        p4.x = 1;
35        p4.y = 0;
36        Console.WriteLine("{0}, {1}, {2}, {3}", p1.x, p1.y, p3.x, p3.y);
37        Console.WriteLine("{0}, {1}, {2}, {3}", p2.x, p2.y, p4.x, p4.y);
38    }
39  }
```

che ci presenta il seguente output:



Andiamo con ordine. Abbiamo creato una classe dalla riga 3 alla 12 (un reference type quindi) e una struct dalla riga 14 alla 18 (un value type). Alle righe 24 e 25 ne abbiamo creato delle istanze, ossia delle variabili basate sui due tipi e alle righe 28 e 29 una copia per ciascuna istanza. Le righe 30 e 31 mostrano, espondendo l'output relativo ai valori dei rispettivi campi, che le istanze principali e le loro copie sono perfettamente identiche in termini di detti valori. Le righe 32 e 33 procurano delle alterazioni nelle copie p2 e p4. L'output alle righe 36 e 37 dimostrano che la variazione introdotta su p2 causa un cambiamento dei valori anche in p1 mentre i mutamenti in p4 non intaccano p2 che mantiene i valori 5 e 6 rispettivamente per x ed y. Questo perchè? Perchè le i reference type (la classe, nell'esempio) lavorano come detto per riferimento i value type (la struct, nell'esempio) vanno per valore. Insomma la situazione è la seguente:



p1 e p3 puntano per riferimento alla **stessa** zona di memoria mentre p2 e p4 hanno una totale indipendenza l'una dall'altra. Ecco perchè le modifiche su p3 interessano anche p1 mentre quelle su p4 lasciano inalterati i valori su p2. Questo è un concetto assolutamente fondamentale e costituisce la differenza maggiore ma paradossalmente meno visibile tra le due tipologie; almeno finchè non ci si scontra con strani risultati... Quindi ricordiamoci bene: i reference types potrebbero puntare "in massa" ai loro valori ;-)

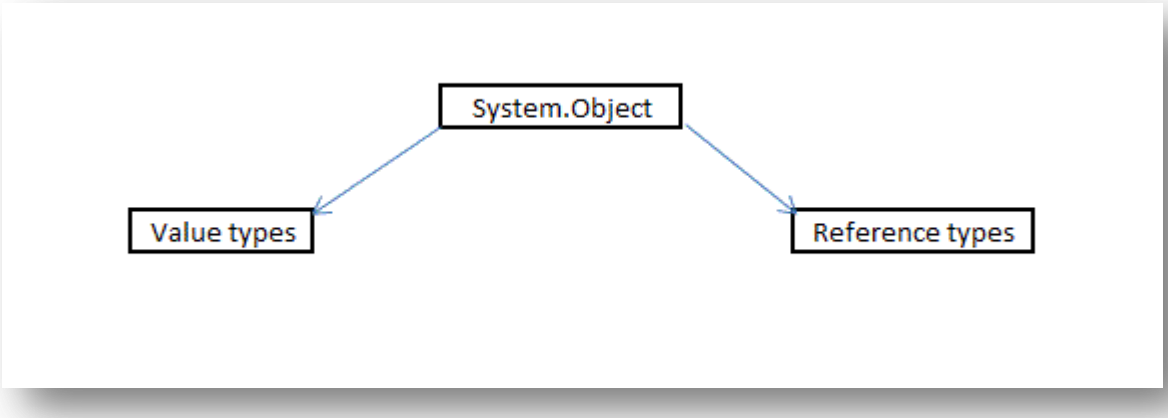
Due parole le dedichiamo alle "abitazioni" delle due famiglie di tipi. Lo **stack** è un ben nota struttura a pila del tipo LIFO (Last In First Out) semplice concettualmente e tipicamente efficiente; in esso vengono allocate variabili, metodi e parametri delle funzioni che "appoggiano"queste entità una sopra l'altra; man mano che le funzioni terminano lo stack, che normalmente è di piccole dimensioni, si svuota, partendo, evidentemente, dall'alto. Quindi cresce e

decresce dinamicamente a seconda che entrino o escano elementi. In pratica reca una traccia di cosa viene eseguito nel nostro codice. E' una struttura compatta composta, in un certo modo di "strati" o di scatole se preferite l'immagine (tecnicamente ciascuno di essi si chiama frame) ognuno dei quali corrisponde ad un qualcosa che vi viene appoggiato sopra. Lo **heap** invece, che è responsabile della gestione degli oggetti, è abbastanza vicino al concetto di memoria libera e, contrariamente allo stack, può soffrire di problemi di frammentazione a causa della natura dinamica delle variabili in esso allocate e deallocate che lasciano dei "buchi" dopo la fase di deallocazione i quali possono anche creare dei problemi di saturazione. Come vedremo il fatto che il nome di un reference type sia in pratica un puntatore ad una zona di memoria porta a qualche complicazione aggiuntiva. Nello stack insomma si può immaginare il contenuto ordinatamente impilato mentre lo heap può apparire più caotico e casuale per quanto riguarda la distribuzione interna degli oggetti. Stack e Heap sono limitati dalla memoria virtuale disponibile per il processo cui si riferiscono. I tipi valore che vengono caricati nello heap sopravvivono in esso finchè non cessa il dominio dell'applicazione.

In aiuto allo heap c'è il già citato garbage collector, che è un meccanismo che si occupa di liberare lo heap da oggetti non più referenziati, come abbiamo detto anche nel paragrafo dedicato a .Net. E' molto utile e solleva il programmatore da una buona parte delle problematiche gestione degli oggetti stessi, purchè si rimanga nell'ambito managed. Per quanto come detto si tratti di un automatismo è tuttavia possibile interagire con esso per via programmatica, pur se, almeno in base alle mie esperienze, capita raramente.

Va da sè che sia lo heap che lo stack sono in realtà astrazioni costruite dal compilatore sulla base della memoria fisica del computer.

Può essere sorprendente la rivelazione che value-types e reference-types hanno **una radice comune.** Ebbene si, ed è **System.Object**. Si tratta dell'oggetto archetipo, progenitore e supporto per tutte le classi in .Net, indipendentemente dal linguaggio che utilizzate. Per questo motivo, forse, soprattutto all'inizio quando altri meccanismi come i generics non erano disponibili, si faceva e a volte si doveva fare, un uso un po' pesante di object il che poteva causare un qualche gravame sulle prestazioni. Non è necessario in genere esplicitare il legame di un oggetto con System.Object in quanto questo è implicito. Esso come detto è, restando al nostro discorso attuale, anche l'antenato da cui discendono entrambe le famiglie di tipi come mostra la semplice ma fondamentale immagine qui di seguito, realmente esprimente un pilastro per quanto concerne .Net:



Le due tipologie poi prendono ciascuna la sua strada ma l'origine comune lascia una impotante traccia costituita da una serie di metodi che in ogni caso sono applicabili ad ogni tipo di dato quale che ne sia l'origine. Precisamente si tratta dei seguenti:

- **Equals** – pubblico - supporta il confronto
- **Finalize** – protetto - viene adoperato per eliminare l'oggetto dalla memoria ed è richiamato dal garbage collector
- **GetHashCode** – pubblico - genera un valore per il tipo nell'ambito di una hash table, argomento su cui torneremo
- **GetType** – pubblico - restituisce il tipo dell'oggetto. Lo abbiamo incontrato nell'esempio 3.5

- **MemberwiseClone()** – protetto - crea una shallow copy dell'oggetto, In pratica in caso di oggetti contenenti dati reference-type viene creato un riferimento che punta alla stessa zona di memoria dell'oggetto origine. Il concetto di shallow copy sarà meglio illustrato parlando dei reference-types.
- **ToString()** – pubblico - di default restituisce il nome completo del tipo anche se spesso si usa per altri scopi. Il nome comunque è abbastanza "parlante" e indica chiaramente il trasferimento di un certo contenuto in una stringa.
- **ReferenceEquals** – pubblico – determina se le istanze di un certo oggetto sono la medesima istanza.

Ogni tipo potrà poi aggiungere, o avrà già predefiniti, altri metodi specifici ovviamente ma questi appena citati sono un corredo comunque presente. Object, proprio a causa del fatto che costituisce il punto di incontro tra tipi di natura molto diversa, è utile come elemento generico di memorizzazione dei dati, ne parleremo ad esempio nel capitolo degli array, ed è anche alla base del potente ed importante meccanismo di boxing e unboxing che spiegheremo più avanti e che costituisce un ponte virtuale tra value e reference types.

Siamo giunti alla fine di questo paragrafo di pura presentazione; per il momento basta ricordare le poche nozioni fondamentali qui espresse, che sono però vitali per una piena comprensione degli argomenti che seguiranno. Per fare qualche utile test è almeno necessario tenere presenti:

- La necessità di dichiarare correttamente in termini grammaticali e, preferibilmente anche dal punto di vista delle convenzioni, ed inizializzare le variabili che vogliamo tenendo d'occhio il discorso relativo alla loro visibilità (o scope)
- La scelta e l'assegnazione del tipo corretto (o la creazione della classe adatta) in base ai contenuti che nella variabile vogliamo memorizzare.

Sembra poco ma è la base senza la quale non si fa nulla.

Nei prossimi paragrafi impareremo quindi quali sono nel dettaglio i tipi che abbiamo a disposizione in C#.