Tipi valore & riferimento

Approfondimento sui "*value types*" e "*reference types*"

Linguaggio C#

Anno 2011/2012

Indice generale

1 Introduzione: *"value types"* vs *"reference types"*

Partiamo dalla definizione generale di ***tipo***. Da wikipedia:

> ***un tipo di dato (o semplicemente "tipo") è un nome che indica l'insieme di valori che una variabile, o il risultato di un'espressione, possono assumere e le operazioni che si possono effettuare su tali valori.***
>
> ***Dire, ad esempio, che la variabile X è di tipo "intero" significa affermare che X può assumere come valori solo numeri interi e che su tali valori sono ammesse solo certe operazioni.***

Questa definizione è valida in generale, ma quando ci si riferisce ad un linguaggio specifico occorre considerare anche altri aspetti, tra i quali *la modalità di memorizzazione dei valori appartenenti ad un tipo*.

In C# i tipi sono suddivisi in due categorie: ***tipi valore*** e ***tipi riferimento***. L'appartenenza all'una o all'altra categoria produce delle conseguenze:

1. nella modalità di accesso al valore;
2. nell'operazione di assegnazione;
3. nell'operazione di confronto (operatore ==);
4. nell'inizializzazione delle variabili;
5. nel modello di memorizzazione utilizzato.

Di seguito esamineremo i primi 4 punti per entrambe le categorie.

2 Tipi valore (*value types*)

Appartengono alla categoria dei *tipi valore*: **int**, **double**, **bool**, **char**, e tutti i tipi struttura.

2.1 Rappresentazione delle variabili

Una variabile di tipo valore può essere rappresentata come una "scatola" contenente un valore. Sulla scatola c'è il suo nome.

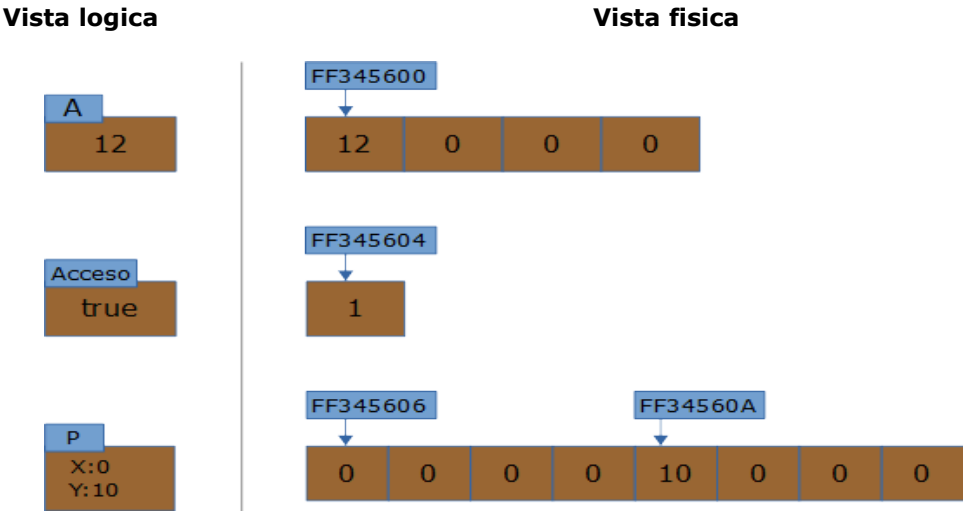
In memoria, la scatola è l'insieme dei byte necessari per memorizzare il valore, mentre il nome è l'indirizzo di memoria del primo byte.

Consideriamo il seguente codice, che dichiara tre variabili di tipo valore:

```
struct Punto
{
    public int X;
    public int Y;
}

class Program
{
    static void Main(string[] args)
    {
        int A = 12;
        bool acceso = true;
        Punto P;
        P.X = 0;
        P.Y = 10;
    }
}
```

Ecco la rappresentazione delle variabili `A`, `acceso` e `P`[1]:



1 Gli indirizzi di memoria sono espresso in notazione esadecimale e sono stati scelti a caso. I byte del valore devono essere letti da sinistra a destra; a sinistra sta byte meno significativo.

2.1.1 Accesso al valore di una variabile

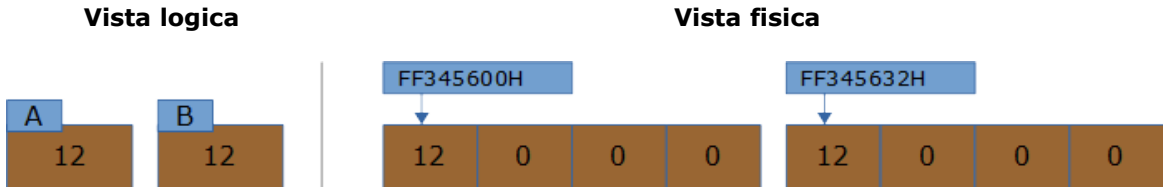
La CPU accede ai byte di memoria referenziati dall'indirizzo della variabile. Viene applicato cioè un ***indirizzamento diretto***.

2.2 Assegnazione

Si consideri la seguente assegnazione:

```
int B = A;
```

Il valore di `A` viene copiato nella variabile `B`. Dopo l'assegnazione la situazione è la seguente:



In sostanza: i byte della variabile `A` vengono copiati all'indirizzo di `B`. Si ha dunque una ***copia del valore*** di `A`.

2.3 Confronto (operatore ==)

Viene eseguito un confronto "bit a bit" tra le zone di memoria delle due variabili. Il risultato è falso se differisce anche un solo bit.

2.3.1 Operazione di confronto sui tipi struttura

L'operazione di confronto può essere applicata soltanto ai tipi predefiniti (**bool**, **char**, **int**, **double**,...). Non è ammesso il confronto tra due variabili appartenenti a un tipo struttura, a meno che questo non definisca un ***operatore di confronto***.

2.4 Inizializzazione delle variabili (valore predefinito)

Sia le variabili globali che quelli locali *hanno sempre un valore*.

2.4.1 Variabili globali

Appena dichiarate, le variabili globali hanno un valore predefinito, che dipende dal tipo:

Tipo	Valore predefinito
bool	false
char	'\0' (valore numerico 0)
int	0
double	0

Per le variabili struttura vale lo stesso: *tutti i campi della struttura sono impostati al loro valore predefinito*.

2.4.2 Variabili locali

Il linguaggio *non imposta il valore predefinito delle variabili locali*. Dunque: *il loro valore iniziale è casuale*, poiché dipende dai bit presenti nella zona di memoria associata alla variabile.

C# proibisce l'uso di una variabile locale prima che le sia stato assegnato un valore (dopo la dichiarazione, la variabile si trova nello stato ***non assegnata***).

Il codice seguente mostra la dichiarazione e l'assegnazione di cinque variabili locali:

```
static void Main(string[] args)
{
    int a;                          // 'a' dichiarata
    int b;                          // 'b' dichiarata
    int n = 0;                      // 'n' dichiarata & assegnata
    string tmp = Console.ReadLine(); // 'tmp' dichiarata & assegnata
    b = int.Parse(tmp);             // 'b' assegnata
    n = n + a;                      // -> errore: uso di 'a' non assegnata!
}
```

L'ultima istruzione non è corretta; infatti, nonostante `a` contenga un valore, questo è del tutto casuale, poiché dipende da cosa è memorizzato nella zona di memoria assegnata alla variabile. Per questo motivo, C# segnala un errore.

> **Linguaggi e uso di variabili non assegnate**
>
> Il divieto di usare variabili *non assegnate* è una caratteristica di C# e altri linguaggi, *ma non di tutti i linguaggi*! Il linguaggio C, ad esempio, consente questa operazione.

3 Tipi riferimento (reference *types*)

Appartengono alla categoria dei tipi riferimento: **string**, **array** e tutte le classi.

3.1 Rappresentazione delle variabili

Una variabile di tipo riferimento può essere rappresentata come una "scatola" contenente un riferimento ad un'altra "scatola" che contiene il valore vero e proprio.

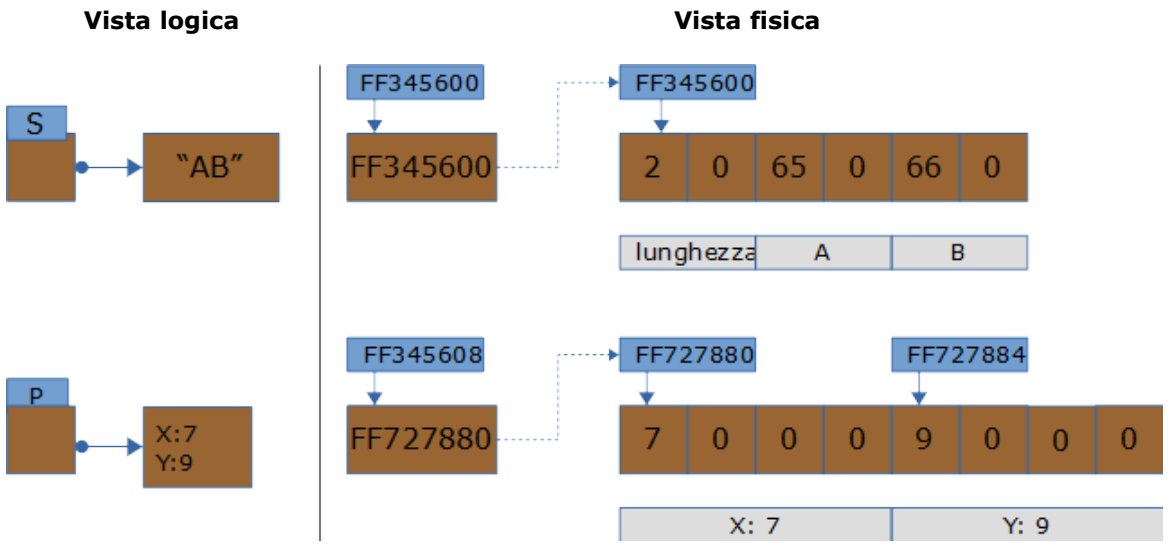
Consideriamo il seguente codice, che dichiara due variabili di tipo riferimento:

```
class Punto
{
    public int X;
    public int Y;
}

class Program
{
    static void Main(string[] args)
    {
    string S = "AB";
    Punto P = new Punto();
    P.X = 7;
    P.Y = 9;

    }
}
```

Ecco come possono essere rappresentate le variabili `S`[2] e `P`:



Le variabili vere e proprie `S` e `P` *non memorizzano un valore*, ma soltanto *l'indirizzo della zona di memoria che contiene il valore*: memorizzano cioè un ***riferimento***.

2 In realtà la rappresentazione in memoria di una stringa è più complessa.

3.1.1 Accesso al valore di una variabile

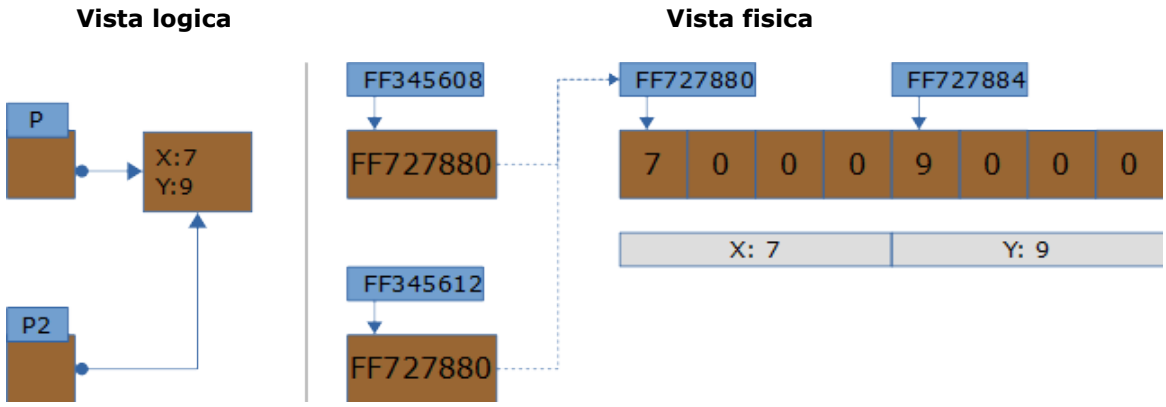
La CPU accede al contenuto della variabile e lo usa come indirizzo per accedere all'oggetto vero e proprio. (Vi è dunque un doppio accesso, chiamato ***indirizzamento indiretto***).

3.2 Assegnazione

Si consideri la seguente assegnazione:

```
Punto P = new Punto();
P.X = 7;
P.Y = 9;
Punto P2 = P;    // copia P in P2
```

Dopo l'assegnazione la situazione è la seguente:



Il contenuto di `P` viene copiato in `P2`. Ma questo non è il valore di `P` ma soltanto l'indirizzo di memoria nel quale si trova tale valore. C'è dunque una *copia del riferimento* e non del valore.

Dopo questa operazione, esiste un solo oggetto di tipo **Punto**, ma due variabili che lo referenziano.

3.3 Confronto (operatore ==)

Viene eseguito un confronto "bit a bit" tra le zone di memoria delle due variabili. Dunque vengono confrontati i riferimenti agli oggetti e non gli oggetti stessi. Il confronto stabilisce se le due variabili puntano allo stesso oggetto oppure no.

(Il discorso cambia se la classe definisce un operatore di confronto.)

3.4 Inizializzazione delle variabili (valore predefinito)

Appena dichiarate, sia le variabili globali che quelli locali *non referenziano alcun oggetto*.

3.4.1 Variabili globali

Appena dichiarate, le variabili globali hanno il valore **null**, qualunque sia il loro tipo. Dunque: *non fanno riferimento ad alcun oggetto*.

3.4.2 Variabili locali

Vale lo stesso discorso fatto per i *tipi valore*.

Prima di usare una variabile è sempre necessario associarla ad un oggetto, creandolo oppure assegnando alla variabile il riferimento ad un oggetto già creato in precedenza.

3.4.3 Costante null

C# definisce la costante **null**, che indica un riferimento nullo, e cioè un riferimento che non punta a nessun oggetto.

È possibile assegnare **null** a una variabile per stabilire che non referenzia alcun oggetto. Ad esempio:

```
Punto p = null;
```

È possibile verificare se una variabile referenzia un oggetto confrontando il suo valore con **null**.

```
if (p != null)
{
    // accedi ai campi di P
}
```

3.4.4 Errore " NullReferenceException"

Il tentativo di accedere ad un oggetto mediante una variabile che contiene **null** provoca l'errore **NullReferenceException**. Ad esempio:

```
Punto P = null;
...
P.X = 10;  // -> errore: 'p' non referenzia alcun oggetto!
```

3.5 Riepilogo "tipi riferimento" vs "tipi valore"

Una variabile di tipo valore non è separabile dal valore che contiene. Le operazioni di assegnazione e confronto avvengono sempre sul valore della variabile.

Per i tipi riferimento esistono due oggetti distinti: la variabile e l'oggetto che questa referenzia. Le operazioni di assegnazione e confronto *avvengono sulla variabile e non sull'oggetto referenziato*.

4 Programmare con i tipi valore e i tipi riferimento

La differenza tra i *tipi valore* e i *tipi riferimento* produce delle conseguenze. Qui ci occuperemo soltanto del seguente aspetto:

> ***le variabili di tipo valore contengono sempre un valore; le variabili di tipo riferimento possono contenere un valore oppure null.***

4.1 Usare i riferimenti null

La possibilità di avere riferimenti **null** (variabili che non referenziano un oggetto) può essere usata per indicare che un certo procedimento non ha prodotto alcun risultato. Un esempio tipico è quello della ricerca.

Ad esempio: si vuole realizzare un metodo che cerchi uno studente in un elenco.

Segue un'implementazione che ritorna la posizione dello studente, oppure -1 se questo non esiste.

```
int IndiceStudenteByNome(string nome, string[] elencoStudenti)
{
    for (int i = 0; i < elencoStudenti.Length; i++)
    {
        if (elencoStudenti[i].Nome == nome)
        {
            return i;
        }
    }
    return -1;
}
```

Se `Studente` è una classe, si può implementare il metodo in modo diverso:

```
Studente StudenteByNome(string nome, string[] elencoStudenti)
{
    foreach (Studente studente in elencoStudenti)
    {
        if (studente.Nome == nome)
        {
            return studente;
        }
    }

    return null; //non esiste uno studente con quel nome!
}
```

Ritornando **null** il metodo comunica che non esiste uno studente con il nome specificato.

Naturalmente, il codice chiamante ha la responsabilità di verificare il valore prodotto dal metodo, come avviene nel seguente esempio:

```
void VisualizzaStudenteByNome()
{
    string nome = Console.ReadLine();
```

```
    Studente studente = StudenteByNome(nome);
    if (studente != null)   // verifica se la variabile punta ad un oggetto
    {
        // visualizza dati studente
    }
    else
    {
        // visualizza messaggio studente non trovato.
    }
}
```

Se Studente fosse un tipo struttura, il metodo **StudenteByNome()** non potrebbe essere implementato in questo modo, *infatti una variabile struttura non può essere nulla*[3].

3 A questo scopo esistono i ***nullable types***, che consentono che portano il concetto di "nullabilità" anche nei tipi valore.