Reference types

Approfondimento sui “tipi riferimento”

Linguaggio C#

Anno 2017/2018

Indice generale

1 Introduzione ai *"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 valori interi e che su tali valori sono ammesse solo determinate operazioni.***

È una definizione valida, ma incompleta; quando ci si riferisce a un linguaggio di programmazione 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 riferimento*** e ***tipi valore*** L'appartenenza all'una o all'altra categoria produce delle conseguenze:

1. nella modalità di memorizzazione e accesso ai dati;
2. nell'operazione di assegnazione;
3. nell'operazione di confronto (operatore ==);
4. nell'inizializzazione;

Di seguito analizzerò il modello di memorizzazione utilizzato per i *reference types*.

2 Reference *types*

Appartengono alla categoria dei **tipi riferimento** i record e gli oggetti; tra questi vi sono moltissimi tipi predefiniti: gli **array**, `string`, `List<>`, `Stack<>`, `Queue<>`, etc.

Poiché da questo punto di vista non vi è alcuna differenza tra record e oggetti, d'ora in avanti userò soltanto il termine oggetto.

2.1 Variabili di tipo riferimento

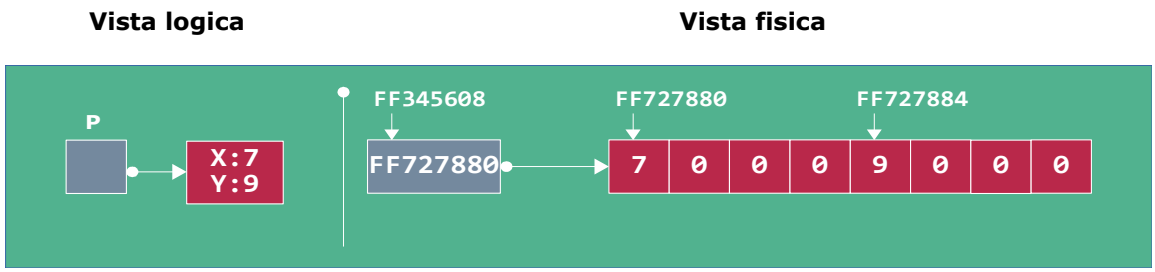
Quando si pensa a una variabile la si immagina come una "scatola" che contiene un valore (precedentemente assegnato). Ebbene, una variabile (di tipo) riferimento non contiene il valore (l'oggetto), ma soltanto un riferimento alla zona di memoria nel quale è memorizzato.

In sostanza: *qualsiasi variabile riferimento contiene semplicemente un indirizzo di memoria* che "punta" all'oggetto vero e proprio.

Considera il seguente codice, che crea un oggetto di tipo `Punto` e lo assegna alla variabile `p`:

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

class Program
{
    static void Main(string[] args)
    {
        Punto p = new Punto();
        p.X = 7;
        p.Y = 9;
    }
}
```

Ecco come può essere rappresentata la situazione in memoria[1]:



L'operatore `new` crea l'oggetto, allocando una zona di memoria di dimensioni adeguate per memorizzarne i dati. Dopodiché, l'indirizzo di memoria viene assegnato alla variabile `p`.

Il fatto che la variabile contenga un indirizzo di memoria ha delle precise implicazioni, che riguardano le operazioni di assegnazione e confronto, i concetti di stato iniziale delle variabili, di *shared reference* e di *null reference*.

1 Gli indirizzi di memoria usati nello schema sono stati scelti a caso ed espressi in notazione esadecimale.

2.2 Assegnazione

Il seguente codice contiene quattro assegnazioni:

```
Punto p1 = new Punto();
p1.X = 7;
p1.Y = 9;
Punto p2 = p1;
```

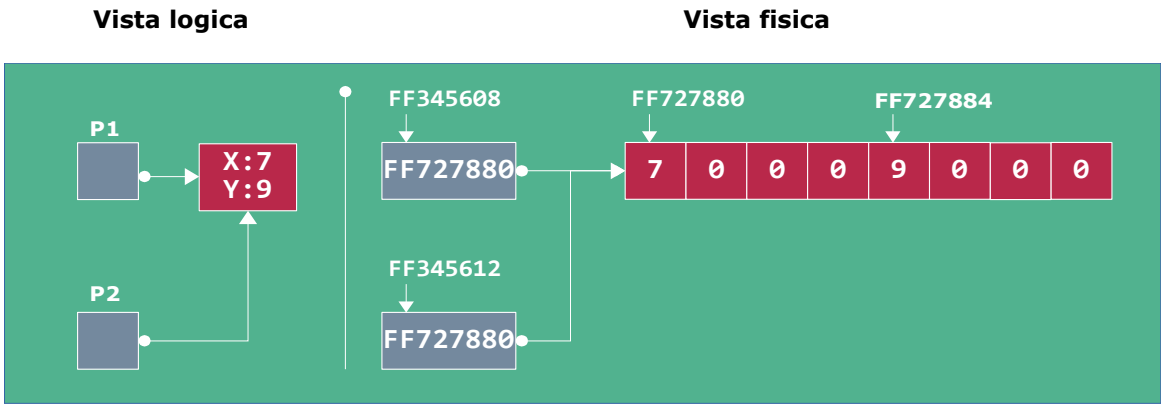
Considera la prima e l'ultima (le assegnazioni centrali non sono rilevanti, poiché riguardano variabili di tipo `int`).

```
Punto p1 = new Punto();
```

Viene creato un nuovo oggetto, allocato in una determinata zona di memoria; i suoi campi vengono azzerati. L'indirizzo dell'oggetto viene memorizzato nella variabile `p1`, che diventa dunque un *reference* all'oggetto stesso.

```
Punto p2 = p1
```

Come accade per qualsiasi assegnazione, il contenuto della variabile `p1` viene copiato nella variabile `p2`. Questa operazione non influenza l'oggetto referenziato da `p1`; semplicemente: dopo l'assegnazione, le due variabili referenziano lo stesso oggetto.



2.3 "*shared*" *references* (riferimenti condivisi)

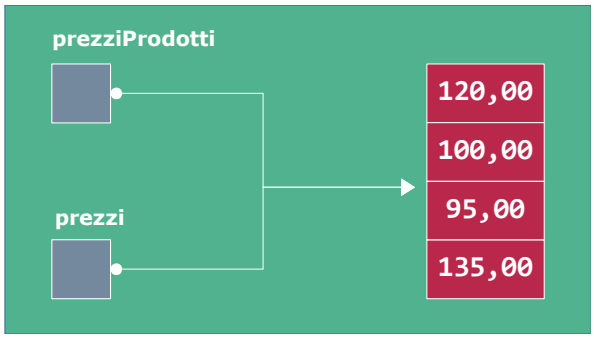
Si usa il termine *shared references* per indicare che più variabili referenziano lo stesso oggetto. Questo concetto è molto importante, soprattutto per capire come è possibile che un metodo sia in grado di modificare un oggetto creato altrove nel programma.

Si consideri il seguente esempio, nel quale il metodo `ApplicaSconto()` modifica i valori memorizzati in una lista:

```
static void Main(string[] args)
{
    List<double> prezziProdotti = new List<double>() { 120, 100, 95, 135 };
    ApplicaSconto(prezziProdotti, 0.1);
}
```

```
static void ApplicaSconto(List<double> prezzi, double sconto)
{
    for (int i = 0; i < prezzi.Count; i++)
    {
        prezzi[i] = prezzi[i] - (prezzi[i] * sconto);
    }
}
```

L'esecuzione di `ApplicaSconto()` implica l'assegnazione della variabile `prezziProdotti` al parametro `prezzi`:



Lo schema mostra come vi siano due variabili in gioco, ma un solo oggetto, che nel metodo viene modificato attraverso la variabile `prezzi`. Dunque, `prezzi` e `prezziProdotti` condividono il riferimento allo stesso oggetto.

2.3.1 Fraintendere la natura dei reference types

Se non si comprende appieno il concetto introdotto in precedenza, si può essere portati a scrivere codice come il seguente:

```
static void Main(string[] args)
{
    List<double> prezziProdotti = new List<double>() { 120, 100, 95, 135 };
    prezziProdotti = ApplicaSconto(prezziProdotti, 0.1);
}

static List<double> ApplicaSconto(List<double> prezzi, double sconto)
{
    for (int i = 0; i < prezzi.Count; i++)
    {
        prezzi[i] = prezzi[i] - (prezzi[i] * sconto);
    }
    return prezzi;
}
```

L'intento è chiaro:

- Si passa al metodo la lista dei prezzi.
- Il metodo la modifica, applicando lo sconto.
- Infine, il metodo restituisce la lista modificata, che viene riassegnata alla variabile di partenza, la quale conterrà ora i prezzi modificati.

Alla base di questo ragionamento sta il presupposto che, per ottenere i prezzi scontati, sia necessario restituire la lista modificata e riassegnarla alla variabile originale. Ma questo implica supporre che al metodo sia passata una copia dell'oggetto, e che il metodo restituisca la copia modificata, che andrà a sostituire la versione originale.

Come mostrato in precedenza, si tratta di un presupposto errato, poiché al metodo viene passato un riferimento alla lista; qualunque modifica fatta attraverso quel riferimento, sarà fatta all'oggetto originale (e unico), referenziato dalla variabile `prezziProdotti`.

In conclusione: non esiste alcun bisogno di restituire la lista modificata, poiché la modifica effettuata all'interno del metodo viene già applicata alla lista originale.

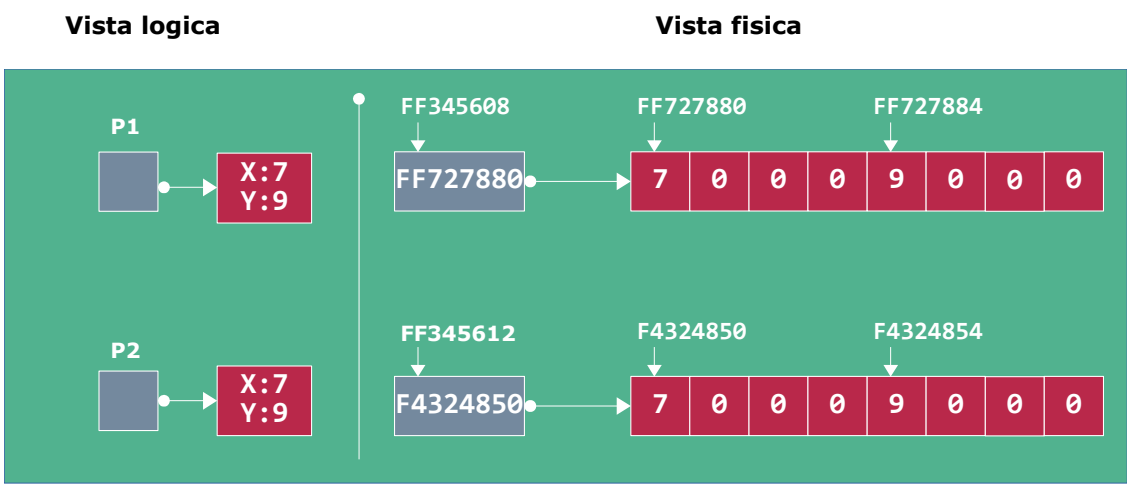
2.4 Confronto (operatore ==)

Il confronto tra due variabili riferimento funziona esattamente come il confronto tra due variabili di tipo valore (`int`, `double`, etc) : viene stabilito se i loro contenuti sono uguali oppure no. Ma, poiché il contenuto di una variabile riferimento è un indirizzo, ad essere confrontati sono, appunto, i due *reference* e non gli oggetti referenziati.

Si consideri il seguente codice:

```
Punto p1 = new Punto();
p1.X = 7;
p1.Y = 9;

Punto p2 = new Punto();
p2.X = 7;
p2.Y = 9;

if (p1 == p2)
{
    Console.WriteLine("Sono uguali");
}
```

Lo schema riassume la situazione:



Le variabili `p1` e `p2` sono uguali? Ebbene, gli oggetti referenziati dalle due variabili sono

indubbiamente uguali, ma nella condizione sono confrontati gli indirizzi e non gli oggetti; pertanto il confronto produce `false`.

Il confronto stabilisce dunque l'uguaglianza tra i riferimenti, e non tra gli oggetti. In sostanza: *stabilisce se due variabili referenziano lo stesso oggetto*.

Si parla di ***object identity***. Questo termine designa l'idea che ogni oggetto rappresenta un'entità distinta da tutti gli altri oggetti dello stesso tipo, *anche da quelli che contengono esattamente gli stessi dati*. Dunque, due variabili dello stesso tipo *sono uguali soltanto se rappresentano lo stesso oggetto*.

2.5 Valore iniziale e inizializzazione delle variabili

Quando si parla di valore iniziale di una variabile occorre innanzitutto distinguere tra *variabili globali* (variabili di classe, statiche o non statiche) e *variabili locali*, dichiarate nei metodi (anche i parametri sono variabili locali).

2.5.1 Variabili globali

Tutte le variabili globali vengono automaticamente azzerate durante la dichiarazione; dunque, le variabili riferimento hanno come valore iniziale l'indirizzo `0`. Per il linguaggio, l'indirizzo zero corrisponde a *nessun indirizzo*.

Conclusione: *una variabile globale appena dichiarata non referenzia alcun oggetto*.

```
class Program
{
    static Punto p;  // p contiene 0 e dunque non referenza alcun oggetto!
    static void Main(string[] args)
    {

    }
}
```

2.5.2 Riferimenti nulli e constante null

Una variabile riferimento azzerata si definisce *null reference* (riferimento nullo). Ma, nonostante i *reference* siano dei valori interi, per il linguaggio non possono essere trattati come tali. Per questo non possiamo scrivere:

```
Punto p = 0;  // illegale!
```

per azzerare una variabile riferimento. Né possiamo scrivere:

```
if (p == 0)   // illegale!
{

}
```

per sapere se una variabile referenzia un oggetto oppure no.

A questo scopo il linguaggio definisce la costante `null`, che rappresenta un *null reference*.

Dunque:

- Una variabile globale appena dichiarata vale `null`.
- Una variabile che vale `null` non referenzia alcun oggetto.
- Per sapere se una variabile referenzia un oggetto, occorre confrontarla con `null`.

2.5.3 NullReferenceException

Il tentativo di accedere a un oggetto mediante una variabile che vale `null` provoca l'errore `NullReferenceException`. Ad esempio:

```
class Program
{
    static Punto p1;  // p1 contiene null
    static void Main(string[] args)
    {
        Punto p2 = null;          // p2 contiene null
        Punto p3 = new Punto();   // p3 referenzia un oggetto (X:0, Y:0)
        p1.X = 10;                // -> NullReferenceException
        int x = p2.X;             // -> NullReferenceException
        Console.WriteLine(p3.X)   // -> OK (visualizza 0)

    }
}
```

2.5.4 Variabili locali

Le variabili locali non vengono azzerate durante la dichiarazione e dunque possono contenere qualsiasi valore; per questo motivo il linguaggio ne impedisce l'uso fintantoché non viene loro assegnato un valore. Si può inizializzare una variabile locale (o globale):

- Assegnandole il valore `null`:

```
Punto p = null;
```

- Assegnandole un'altra variabile (globale, locale o parametro):

```
Punto p1 = p2;
```

- Assegnandole il riferimento a un oggetto appena creato.

```
Punto p = new Punto();
```

2.6 Un tipico errore programmazione con i *reference types*

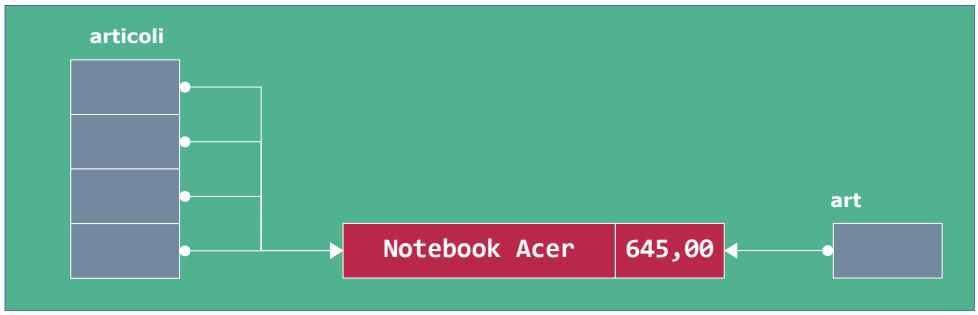
Di seguito mostro uno scenario nel quale la mancata comprensione dei *reference types* produce un programma errato.

Supposta l'esistenza di un file di testo contenente i dati di un elenco di articoli, si vuole caricarli in memoria in una lista di record di tipo `Articolo`.

```
public class Articolo
{
    public string Descrizione;
    public double Prezzo;
}
...
public static List<Articolo> CaricaArticoli()
{
    string[] righeFile = File.ReadAllLines("Articoli.txt");

    Articolo art = new Articolo();

    List<Articolo> articoli = new List<Articolo>();
    foreach (var riga in righeFile)
    {
        string[] campiRiga = riga.Split();

        art.Descrizione = campiRiga[0].Trim();
        art.Prezzo = double.Parse(campiRiga[1]);
        articoli.Add(art);
    }
    return articoli;
}
```

Il codice descrive correttamente l'intento del programmatore, ma non produce il risultato richiesto; infatti, crea un solo oggetto, referenziato da `art`, e lo "riutilizza" per memorizzare la descrizione e il prezzo di ogni articolo, aggiungendolo alla lista.

In figura viene schematizzato il risultato prodotto in memoria (si ipotizzano quattro articoli, l'ultimo dei quale un notebook Acer):



Nota bene: i quattro elementi della lista (nonché la variabile `art`) referenziano lo stesso (e unico) oggetto, creato prima del ciclo, ma modificato dentro di esso.

2.6.1 Soluzione corretta

Per ogni articolo deve essere creato un oggetto distinto:

```
public static List<Articolo> CaricaArticoli()
{
    string[] righeFile = File.ReadAllLines("Articoli.txt");

    List<Articolo> articoli = new List<Articolo>();
```

```
    foreach (var riga in righeFile)
    {
        string[] campiRiga = riga.Split();

        Articolo art = new Articolo();
        art.Descrizione = campiRiga[0].Trim();
        art.Prezzo = double.Parse(campiRiga[1]);
        articoli.Add(art);
    }
    return articoli;
}
```

Nota bene: il fatto di riutilizzare la stessa variabile `art` non è rilevante; infatti, questa contiene di volta in volta un riferimento a un oggetto diverso.