

Capitolo 8

Gli array e System.Collections

Anche gli array sono una struttura dati molto conosciuta, direi quasi amichevole, nel mondo della programmazione. Il suo successo è dovuto a due fattori: la potenza, intesa come campo di utilizzo e la semplicità concettuale, soprattutto in linguaggi come C#. **Si tratta semplicemente di una collezione di dati, dello stesso tipo, che sono localizzabili ed accessibili attraverso un indice ad andamento sequenziale.** In particolare in C# gli array sono zero-based nel senso che il primo elemento avrà sempre indice zero. L'indice, come detto, segue un andamento sequenziale ad incrementi unitari. Pertanto se un array è composto di n elementi avendo il primo indice 0 l'ultimo avrà indice n-1. Gli elementi di un array possono essere di qualunque tipo (anche classi, interfacce o delegati) purchè, come detto dianzi, siano tutti di uno stesso tipo. In altri linguaggi è possibile che un array contenga tipi diversi (un carattere, poi un numero, poi una stringa), in C# no, anche se come vedremo c'è un escamotage per emulare questo comportamento. Non va inoltre dimenticato che gli array appartengono ai reference-types per cui il nome che si attribuisce ad uno di essi è in realtà un puntatore ad una zona di memoria occupata dall'inizio della sequenza di elementi mentre l'indice è un offset a partire dal quell'indirizzo. Gli array sono efficienti ed intuitivi nell'accesso agli elementi in quanto essi sono memorizzati in maniera sequenziale e questo li rende ideali nella soluzione di tanti problemi. Di contro sono un po' rigidi e in alcuni casi questo li penalizza nel confronto con altre strutture dati.

Gli array possono essere dichiarati in modalità statica o dinamica. Nel primo caso assegneremo subito una dimensione mentre nel secondo potremo farlo a run-time. Una volta attribuita una dimensione all'array questa non può essere cambiata. Nel caso in cui ci serva una struttura dati più flessibile in questo senso come vedremo queste esistono in .Net. Per la individuazione dei singoli elementi **si usano le parentesi quadre**, ovvero l'operatore di indicizzazione []. **Concludendo, per essere utilizzabili, per gli array devono essere specificati il tipo dei suoi elementi, il rank, ovvero il numero di dimensioni e l'indice di partenza e quello finale per ciascuna di dette dimensioni. Queste ultime specifiche possono essere anche determinate in via deduttiva, come vedremo nel primo esempio qui di seguito. Gli indici devono essere interi col segno (positivi).**

`int[] ar1 = {1,2,3}` indica un array composto da 3 elementi. da notare che non è corretto far seguire le parentesi al nome dell'array ma bisogna metterle dopo il tipo. Quindi non può essere `int ar1[] = {1,2,3}`. In questo caso il numero di dimensioni e gli indici sono dedotti dalla inizializzazione. Siamo di fronte ad un array monodimensionale avente indice più basso = 0 e quello più alto = 2.

`int[] ar2 = new int[3]` qui viene definito un array che contiene 3 interi non specificati, e vedremo tra breve cosa c'è dentro subito dopo la creazione dello stesso.

`int[] ar3 = new int[3] {1,2,3}` e anche in questo caso inizializziamo anche i singoli elementi.

`int[] ar4 = new int[] {1,2,3}` `ar4` sarà uguale ad `ar3` e `ar1` e come per questo la sua dimensione è fornita tramite la sua stessa inizializzazione.

E' peraltro possibile scrivere genericamente

```
int[] ar5
```

riservandosi in un momento successivo l'inizializzazione senza la quale l'array non può essere usato.

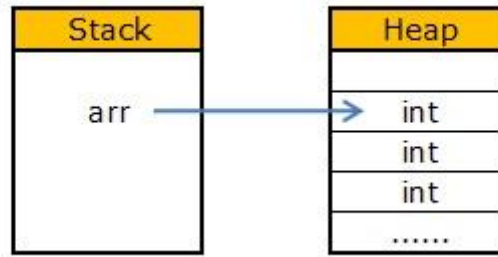
Ma come sono fatti `ar1` `ar2` e `ar3` e `ar4`?

<code>ar2 - dati</code>	0	0	0
<code>indici</code>	0	1	2

<code>ar1, ar3, ar4 - dati</code>	1	2	3
<code>indici</code>	0	1	2

ar2 ha una dimensione specificata ma nessuna inizializzazione: ne viene allora fornita una di default legata al tipo degli elementi interni. Per gli interi 0 è il default così come il carattere blank lo è per i char, la stringa vuota per le stringhe ecc... Invece ar3, come ar4, ha i suoi elementi già assegnati. Incontriamo inoltre un altro importantissimo operatore, avrete già capito, si tratta di **new** la cui funzione in casi come questo è quello di caricare nello heap un nuovo elemento di tipo reference, in questo caso l'array. L'operatore new ha altri due utilizzi come vedremo più avanti il più comune è lanciare le operazioni di caricamento nello heap di un nuovo reference type. Nel caso di ar1 è il CLR che si occupa, **implicitamente**, di chiamare in causa new. Da notare infine, per quanto ovvio, che il numero tra parentesi quadre indica quanti sono gli elementi dell'array, non il massimo indice.

Relativamente alla occupazione di memoria un array può essere così visualizzato



In pratica lo stack contiene il nome del nostro array con un puntatore che rimanda laddove nello heap c'è il punto di inizio per la sequenza dei dati. Interessante notare la presenza di value types all'interno dello heap e non dello stack, loro collocazione naturale come ricorderete dal capitolo precedente. Un array è sempre un reference type anche se i suoi elementi costitutivi sono value types.

C#	Esempio 8.1
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	int[] ar1;
7	int[] ar2 = new int[3];
8	int[] ar3 = new int[] {1,2,3};
9	Console.WriteLine(ar2[1]);
10	Console.WriteLine(ar3[1]);
11	// Console.WriteLine(ar1[1]);
12	}
13	}

Questo programma ha come output il numero 0 (riferito alla riga 9) ed il numero 2 (riga 10) esattamente come atteso. Come si può vedere i singoli elementi sono individuati attraverso la parentesi quadra [] vedansi sempre le righe 9 e 10. La riga 11 non può essere compilata in quanto ar1 non è in alcun modo inizializzato quindi, come detto, non può essere usato. Gli indici hanno un loro range che come detto parte da 0 ed arriva alla lunghezza, cioè il numero di elementi dell'array, meno uno. Il fatto che l'indice più basso sia 0 è una restrizione imposta dal CLS. Se si tenta di accedere ad un elemento al di fuori del range ammesso origina un chiaro errore a run.time. Ad esempio se la riga 10 fosse:

`Console.WriteLine(ar3[3]);` otterremmo in esecuzione:

*Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
at program.Main()*

viene sollevata una eccezione, `IndexOutOfRangeException`, come logica conseguenza della non esistenza dell'elemento avente indice 3 in ar3. Questo controllo è una utile feature del runtime; come fanno i programmatori C/C++ senza l'intervento

di questa eccezione accederemmo ad una zona di memoria dal contenuto random con conseguenze ben immaginabili (anzi... forse inimmaginabili...).

C#, al contrario di altri linguaggi, **non consente l'uso di indici negativi**, di fronte ai quali il compilatore emette un warning e se si insiste è sicuro un errore a run-time.

I singoli elementi di un array sono modificabili accedendo direttamente ad essi e procedendo con un assegnamento del nuovo valore desiderato. Ovviamente sono da seguire le consuete regole relative alla tipizzazione. Così è possibile ad esempio scrivere nel programma precedente:

```
ar2[1] = 10; mentre
ar2[1] = "pippo"; non è possibile perchè una stringa non può essere implicitamente convertita in intero. Invece una istruzione tipo:
ar2[1] = 'a'; è lecita? sì, perchè esiste la conversione implicita da char a int. In questo caso ar2[1] assume valore 97 cioè la posizione di 'a' nella tabella ASCII. Bisogna quindi fare attenzione perchè il compilatore ci difende fino ad un certo punto ma i risultati inapparenza "strani" sono lì ad aspettarci, questa è una considerazione avente comunque una valenza generale.
```

Abbiamo detto che gli elementi di un array devono essere tutti dello stesso tipo. In altri linguaggi c'è una gestione più permissiva per cui nello stesso array possono convivere un intero e una stringa ad esempio, o meglio, generalizzando, uno o più elementi possono essere di un tipo ed uno o più di altri tipi. Caratteristici in questo senso sono i linguaggi dinamici tipo Ruby o Python. Se vogliamo emulare in qualche modo questo comportamento degli array che troviamo quindi in altri linguaggi più "rilassati" relativamente alla tipizzazione possiamo definire un array con tipo base **object**. Ad esempio:

C#	Esempio 8.2
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	object[] ar1 = new object[3];
7	ar1[0] = 1;
8	ar1[1] = "ciao";
9	ar1[2] = 'a';
10	}
11	}

Come già osservato in casi analoghi object è il progenitore di tutti i tipi in C# e pertanto in quanto tale può assumere le vesti di qualsiasi cosa. Di qui la possibilità di mettere qualunque tipo di variabile negli elementi di un array di oggetti. A livello prestazionale credo che si paghi un certo prezzo, ma non ho mai fatto test specifici per quantificare.

Anche gli array godono della possibilità di una **tipizzazione implicita**. Vediamo subito un esempio in merito

C#	Esempio 8.3
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	var ar1 = new[] {1, 11.5, 3};
7	var ar2 = new[] {1, "aa"}; // non compila!!
8	}
9	}

La riga 6 ci presenta quanto detto. Viene definito un array il cui tipo base è **var**. Il compilatore deve scegliere il tipo "migliore" per identificare l'appartenenza dei singoli elementi e qui sarà evidentemente il tipo double dato che il primo e il terzo elemento sono degli interi ma il secondo ha la virgola quindi solo il tipo double può comprendere tutti gli elementi. D'altronde basta provare aggiungendo, dopo la 6, una istruzione tipo

C# - Capitolo 8 – Gli array e System.Collections

```
Console.WriteLine(arl[0].GetType());
```

per verificare. La riga 7 invece non compila in quanto non è possibile determinare un tipo compatibile con tutti gli elementi dell'array. L'errore non lascia adito a dubbi:

error CS0826: No best type found for implicitly-typed array

Però, direte voi, ci sarebbe object che va sempre bene per tutto. E' vero ma il team di C# ha bloccato questa possibilità in quanto sarebbe stata soggetta ad operazioni di boxing molto pesanti in termini prestazionali in presenza di array di grosse dimensioni. Insomma, va bene la libertà ma non fino ad arrivare all'autolesionismo. La stessa problematica si può incontrare nel caso, raro, in cui abbiamo due elementi di tipo diverso che possono però essere riversati l'uno nel tipo dell'altro, situazione che si può verificare in presenza di tipi custom creati dal programmatore. Come vedremo gli array implicitamente tipizzati verranno utili quando parleremo di **LINQ**. Di essi, è intuitivo, non deve essere fatto un uso smodato, ovviamente, si tratta di una soluzione per casi particolari. Non ho mai effettuato personalmente, anche in questo caso, dei test di efficienza ma credo che non sia una soluzione particolarmente performante.

Fino a questo punto abbiamo visto gli array come entità monodimensionali. Ma in C# essi possono essere anche definiti su n dimensione con n variabile a nostro piacere. La sintassi è la seguente:

```
int[ , ] multiarr1 = new int[3,3] definisce un array 3 x 3.  
int[ , , ] multiarr2 = new int[3,3,3] definisce un array 3 x 3 x 3  
int[ , ] multiarr3 = new int[] { {1,2}, {3,4} } qui abbiamo un array 2 x 2 inizializzato
```

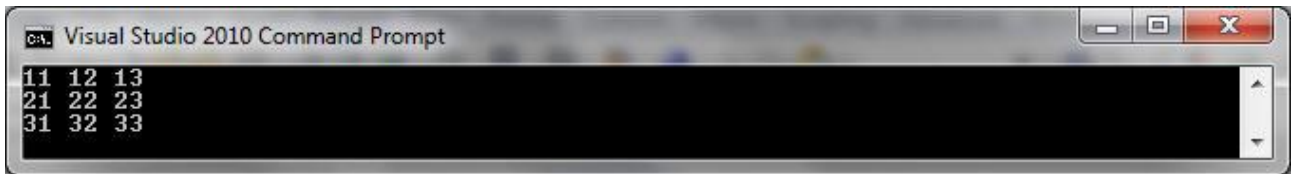
Gli array definiti così si dicono rettangolari, il motivo è evidente. L'accesso al singolo elemento è subordinato all'indicazione di tutte le sue coordinate.

```
int [2,2,2] = 5; ad esempio individua ed assegna un valore ad una singola cella in un array a 3 dimensioni.
```

Vediamo un piccolo programma esplicativo:

C#	Esempio 8.4
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	int [,] multil = new int[3,3];
7	for (int i = 0; i <= 2; i++)
8	{
9	for (int j = 0; j <= 2; j++)
10	{
11	multil[i, j] = ((i + 1) * 10) + j;
12	}
13	}
14	for (int i = 0; i <= 2; i++)
15	{
16	for (int j = 0; j <= 2; j++)
17	{
18	Console.Write(multil[i,j] + " ");
19	}
20	Console.WriteLine();
21	}
22	}
23	}

Avente il seguente output:



Tipicamente il ciclo for assume una grande importanza quando si lavora con gli array, ancora di più nel caso multidimensionale, in quanto risulta lo strumento più comodo per navigare tra gli elementi.

Prima di andare oltre è interessante fare una osservazione. Se proviamo a scrivere un programma che abbia semplicemente il seguente corpo:

```
public static void Main()
{
    int[] ar1 = new int[1];
    int[ , ] ar2 = new int[1,1];
}
```

E diamo quindi uno sguardo col nostro fido Ildasm vediamo quanto segue, tra l'altro:

```
IL_0001: ldc.i4.1
IL_0002: newarr      [mscorlib]System.Int32
IL_0007: stloc.0
IL_0008: ldc.i4.1
IL_0009: ldc.i4.1
IL_000a: newobj      instance void int32[0...,0...]::ctor(int32,
                                                    int32)
```

le due righe evidenziate corrispondono alla definizione dell'array ar1 (monodimensionale alla riga che inizia con IL_0002) e ar2 (bidimensionale). Ebbene è evidente che nel primo caso abbiamo il semplice uso di una istruzione nativa del IL mentre nel secondo sia un vero processo di istanziazione. Analoga osservazione è possibile farla, ve la lascio perchè non da problemi, nel caso si cerchi di accedere ad un elemento, nel primo caso il codice vedrete è molto più semplice. Ad esempio, poi potrete approfondire la cosa, io vi do giusto una traccia, ponendo:

```
ar1[0] = 5;
ar2[0,0] = 5;
```

le istruzioni relative saranno:

```
IL_0012: ldc.i4.5
IL_0013: stelem.i4
```

nel primo caso e

```
IL_0017: ldc.i4.5
IL_0018: call      instance void int32[0...,0...]::Set(int32,
                                                    int32,
                                                    int32)
```

nel secondo. Infatti **stelem** è istruzione nativa del CLR al pari di newarr, altre sono **ldelem**, **ldelema**, **ldlen**, ne vedremo alcune in un paragrafo più avanti.

La morale è che, se possibile, sarebbe bene usare gli array monodimensionali perchè significativamente più performanti. So che non è sempre possibile ma è un consiglio da tenere presente visto che, in pratica, gli array multidimensionali vengono trattati dal CLR come fossero no 0-based, per tutta una serie di ragioni..

Oltre agli array rettangolari C# ammette anche gli array cosiddetti **jagged**. Si tratta, in definitiva di array di array in cui i vari rami possono assumere lunghezze diverse. La costruzione di questi array è un po' particolare. Ovviamente gli array rettangolari possono essere visti come caso particolare dei jagged. La definizione di questi ultimi è la seguente

C# - Capitolo 8 – Gli array e System.Collections

```
int[][] jagged01 = new int[n, ];
```

questa definizione ci presenta un array composto da altri n array al momento non noti. Prima di usare tale array tuttavia dovremo esplicitare tutti i componenti. Il solito esempio chiarirà le cose:

C#	Esempio 8.5
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	int [][] jagged = new int[3][];
7	jagged[0] = new int[3] {3, 5, 123};
8	jagged[1] = new int[6];
9	jagged[2] = new int[2];
10	for (int i = 0; i < 3; i++)
11	{
12	for (int j = 0; j < jagged[i].Length; j++)
13	Console.Write(jagged[i][j].ToString() + ' ');
14	Console.WriteLine();
15	}
16	Console.WriteLine("jagged[0][1]: " + jagged[0][1]);
17	}
18	}

le righe 7, 8 e 9 inizializzano i componenti dell'array jagged, la riga 7 forza dei valori diversi dal default inserendone altri specifici. Dalla riga 12 alla 14 abbiamo le istruzioni per la stampa a video dell'array completo, tra di esse molto importante perchè molto usato è il metodo **Length** che restituisce la lunghezza dell'array e di cui ripareremo tra brevissimo. per intanto vediamo l'output:



la prima è jagged[0] la seconda jagged[1] e la terza jagged[2]. La riga 16 invece ci spiega come accedere ad un singolo elemento di un array jagged, quindi con la forma

```
arrayjagged[x][y].
```

E' possibile complicare ancora un po' le cose mischiando array multidimensionali ad array jagged. Si guardi la seguente definizione:

`int [] [,] mixjagged = new int[3][,];` che dichiara un array jagged costituito da 3 array bidimensionali. Esempio:

C#	Esempio 8.6
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	int[][] [,] mixjagged = new int[3][,];
7	mixjagged[0] = new int[,]{{1,2}, {3,4}};
8	mixjagged[1] = new int[,]{{10,20},{30,40}};
9	mixjagged[2] = new int[,]{{100, 200}, {300,400}};
10	Console.WriteLine("mixjagged[1][0,1]: " + mixjagged[1][0,1]);
11	}
12	}

Il lavoro di inizializzazione, svolto dalla riga 7 alla 9 non è dei più agevoli. La riga 10 invece ci insegna come accedere ad un singolo elemento. Non so quante volte nella vostra vita vi imbatterete in costrutti simili.

Finora abbiamo visto array i cui elementi erano tipi base ma è naturalmente possibile costruirne anche con tipi definiti dall'utente, siano esse classi, strutture, enumerativi o che altro. Bisogna fare attenzione alle inizializzazioni.

C#	Esempio 8.7
1	using System;
2	class program
3	{
4	struct foo
5	{
6	public int x;
7	public string s;
8	}
9	public static void Main()
10	{
11	foo[] ar = new foo[5];
12	ar[0] = new foo();
13	ar[0].x = 5;
14	ar[0].s = "a";
15	}
16	}

La riga 11 definisce un nuovo array ogni elemento del quale è una struct. Le 3 righe successive invece inizializzano un elemento dell'array. Vediamo un esempio basato sulle classi:

C#	Esempio 8.8
1	using System;
2	
3	class foo
4	{
5	public void scrivi(int x)
6	{
7	Console.WriteLine(x);
8	}
9	}
10	
11	class program
12	{
13	public static void Main()
14	{
15	foo[] ar = new foo[5];
16	ar[1].scrivi(5);
17	}
18	}

Questo codice compila ma origina una eccezione a runtime di tipo `NullReferenceException`. La causa, come accennato è che gli elementi di `ar` sono in realtà null, nel momento in cui vengono chiamati. Questo perchè null è il default assegnato dal compilatore alle classi. L'inizializzazione avviene con elementi null principalmente per ragioni di performance ma questo non agevola certamente il processo di utilizzo di quanto memorizzato nell'array. Propongo un paio di soluzioni che al momento possono non essere chiare ma lo diventeranno in seguito. La prima fornisce un costruttore:

C#	Esempio 8.9
1	using System;
2	
3	class foo
4	{
5	public static foo scrivi(int x)
6	{
7	foo f = new foo();
8	Console.WriteLine(x);
9	return f;

```

10     }
11 }
12
13 class program
14 {
15     public static void Main()
16     {
17         foo[] ar = new foo[5];
18         ar[1] = foo.scrivi(5);
19     }
20 }

```

La seconda, forse più comprensibile, è la seguente che costringe però alla istanziazione di tutti gli elementi (riga 16) dopo avere creato l'array. Più sicura ma anche più pesante specialmente in presenza di array contenenti molti elementi:

C#	Esempio 8.10
1	using System;
2	
3	class foo
4	{
5	public void scrivi(int x)
6	{
7	Console.WriteLine(x);
8	}
9	}
10	
11	class program
12	{
13	public static void Main()
14	{
15	foo[] ar = new foo[5];
16	for (int i = 0; i < ar.Length; i++) ar[i] = new foo();
17	ar[1].scrivi(5);
18	}
19	}

Ovviamente dopo aver visto le classi le cose saranno un po' più chiare.

FOREACH.

Abbiamo in precedenza citato, nel capitolo 5, questa istruzione presentandola anche formalmente. E' giunto il momento di usarla. Come si era detto essa viene usata su array e collections (che ancora non sappiamo cosa sono) per eseguire delle operazioni ripetute ed è una importante alternativa all'uso di for. Più in generale la definizione da manuale è che esso si applica su tutti gli oggetti che implementano le interfacce (di cui parleremo in dettaglio più avanti) **System.Collection.IEnumerable** e **System.Collection.Generic.IEnumerable**. Come avevamo anche anticipato si tratta di una istruzione sicura in quanto lavora in modalità readonly sugli elementi che incontra, al contrario di for.

C#	Esempio 8.11
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	int[] ar1 = new int[100];
7	for (int i = 0; i < 100; i++)
8	ar1[i] = i + 1;
9	foreach(int i in ar1)
10	Console.WriteLine("i = " + i);
11	}
12	}

L'esecuzione di foreach può essere terminata anticipatamente da break, come avviene per l'istruzione for e le altre legate alla esecuzione di un ciclo.

Questa istruzione è senza dubbio comoda ed espressiva. Sulla sua efficienza tuttavia ci sono alcuni dibattiti sulla rete. In generale si ritiene che `foreach` non introduca particolari penalizzazioni laddove, dicono invece alcuni autori, il codice generato in presenza di esso parrebbe affetto da qualche ridondanza. A voi la scelta, personalmente, qualora non mi sia servito un qualche controllo sugli indici, ho quasi sempre preferito `foreach`, per ragioni di leggibilità del codice, senza ravvisare handicap in qualche modo significativi in base ai (per la verità pochi) test eseguiti. Un piccolo vantaggio che `foreach` possiede è quello di poter governare senza modifiche anche eventuali casi di array non 0-based, laddove il ciclo `for` necessita per forza di cose di un aggiustamento dell'indice iniziale. Per finire notiamo che, ovviamente, data la natura **readonly** del suo operato, `foreach` non può essere usato per operazioni di inizializzazione di un array.

Parleremo ancora di `foreach` quando introdurremo gli iteratori.

REFERENCE COPY, DEEP COPY E SHALLOW COPY

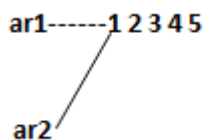
Un problema con il quale ci si può facilmente scontrare con i reference type è quello segnalato dal titolo di questo paragrafo e del quale avevamo già parlato nel capitolo 3 relativamente alla differenziazione tra value e reference types. Qui ripetiamo buona parte dei concetti già espressi a suo tempo ampliandoli perchè bisogna fare attenzione a queste situazioni. Consideriamo il seguente breve programma:

C#	Esempio 8.12
1	<code>using System;</code>
2	<code>class program</code>
3	<code>{</code>
4	<code> public static void Main()</code>
5	<code> {</code>
6	<code> int[] ar1 = new int[] {1,2,3,4,5};</code>
7	<code> int[] ar2 = ar1;</code>
8	<code> ar1[3] = 5;</code>
9	<code> foreach(int x in ar2)</code>
10	<code> Console.Write(x + " ");</code>
11	<code> }</code>
12	<code>}</code>

Il risultato è

1 2 3 5 5

La cosa può sembrare curiosa in prima istanza in quanto abbiamo definito (riga 6) l'array `ar1`, abbiamo poi alla riga 7 creato `ar2` al quale abbiamo in pratica passato i valori di `ar1` con l'assegnazione alla stessa riga. Alla 8 abbiamo poi modificato il valore all'indice 3 di `ar1` ma l'output ci dice chiaramente che lo stesso valore è stato identicamente modificato anche per `ar2`. Perchè? E' abbastanza semplice, lo sappiamo già. Quando creiamo `ar1` e poi `ar2` col simbolo di assegnazione, riga 7, ci veniamo a trovare in una situazione siffatta:



In pratica, come sappiamo, ragioniamo per riferimenti. Creare una copia come abbiamo fatto vuol dire creare un nuovo riferimento alla zona di memoria già puntata da `ar1`. Gli interi 1, 2, 3, 4, 5 sono contenuti in una certa regione di memoria ed è a quella che puntano sia `ar1`, l'originale, sia `ar2`, la sua copia. Quindi ogni modifica indotta su quella zona, sia partendo da `ar1` che da `ar2` ha effetto **anche sull'array apparentemente non coinvolto**. Questo è un effetto che bisogna tenere in conto molto attentamente ed è un esempio di **reference copy**. Abbiamo semplicemente creato un altro riferimento all'array `ar1`. Assolutamente legati l'uno all'altro. Quando invece parliamo di **shallow copy** e **deep copy** la situazione cambia radicalmente. C# ci offre vari modi di effettuare delle copie. **Clone**, **Copy** e **CopyTo**. In nessun caso viene effettuata una deep copy in forma diretta. Esse si affiancano al solito **MemberWiseClone** ereditata da **Object**. Per quanto lavorino diversamente dalla semplice assegnazione tramite "=" non siamo davanti ad

una deep copy anche se accorgercene è un po' più complicato. Vediamo ora il seguente esempio che ci chiarirà la natura, a mio avviso un po' insidiosa, della cosa. Riprendiamo l'esempio precedente solo con qualche marginale modifica:

```
(1)  int[] ar1 = new int[] {1,2,3,4,5};

    int[] ar2 = ar1;
    ar2[3] = 5;
    foreach(int x in ar1) Console.Write(x + " ");
    Console.WriteLine();
    foreach(int x in ar2) Console.Write(x + " ");
    Console.WriteLine();
```

L'output di questo frammento di codice, facilmente richiudibile in un main ovviamente, è quello che segue:

```
1 2 3 5 5
1 2 3 5 5
```

Giusto, come abbiamo visto. Usiamo invece ora **Clone**, tenendo buona la riga 1:

```
int[] ar3 = (int[])ar1.Clone();
ar3[3] = 6;
foreach(int x in ar1) Console.Write(x + " ");
Console.WriteLine();
foreach(int x in ar3) Console.Write(x + " ");
```

e qui avremo come output:

```
1 2 3 4 5
1 2 3 6 5
```

Che strano.... abbiamo variato ar3 però ar1, come dimostra la prima linea di output è invariato. Ma allora è una deep copy. Prima di andare oltre vediamo cosa combina **CopyTo**:

```
int[] ar4 = new int[5];
ar1.CopyTo(ar4, 0);
ar4[3] = 7;
foreach(int x in ar1) Console.Write(x + " ");
Console.WriteLine();
foreach(int x in ar4) Console.Write(x + " ");
Console.WriteLine();
```

e qui riceviamo:

```
1 2 3 4 5
1 2 3 7 5
```

Anche qui stesso comportamento riscontrato in Clone. Insomma è una deep copy o no?. No almeno non in forma completa. Finchè gli elementi sono value types essi vengono ricopiati "bit x bit" nella destinazione. Ma se gli elementi sono reference types? Cambia la cosa? Altrochè. Ricorriamo questa volta ad un esempio completo:

C#	Esempio 8.13
1	using System;
2	
3	public class Stringa
4	{
5	public string s;
6	}
7	
8	class program

```

9  {
10     public static void Main()
11     {
12         Stringa[] x1 = new Stringa[1];
13         x1[0] = new Stringa();
14         x1[0].s = "ciao";
15         Stringa[] x2 = new Stringa[1];
16         x1.CopyTo(x2, 0);
17         Stringa[] x3 = (Stringa[])x1.Clone();
18         Console.WriteLine(x1[0].s);
19         Console.WriteLine(x2[0].s);
20         Console.WriteLine(x3[0].s);
21         x2[0].s = "mamma";
22         Console.WriteLine(x1[0].s);
23         Console.WriteLine(x2[0].s);
24         Console.WriteLine(x3[0].s);
25     }
26 }

```

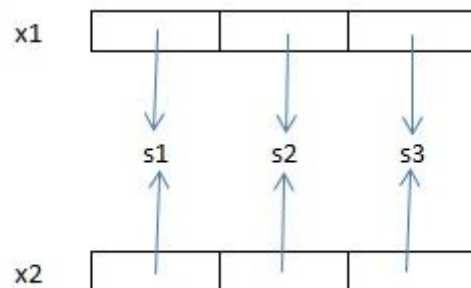
Qui come output abbiamo:

```

ciao
ciao
ciao
mamma
mamma
mamma

```

ovvero la variazione introdotta alla riga 21 si è fatta sentire sia su x1 sia su x3 oltre che naturalmente su x2. Perché questo? La cosa si propone in questi termini validi sempre quando il tipo base dell'array è un reference



In pratica viene creato sì un nuovo array, sia con Clone che con CopyTo ma la copia passa i riferimenti alle stringhe (per ogni singola cella e indicate con s1 s2 ed s3 nella figura) non attribuisce dei valori indipendenti. Questo non avviene come abbiamo visto con i value types come tipi base per l'array in quanto la copia bit a bit fornisce dei valori indipendenti. Non ci riferiamo mai ai tipi valori tramite dei riferimenti.

La differenza quindi è che il simbolo "=" crea solo un nuovo riferimento mentre le altre istruzioni creano un nuovo array e copiano valori e **riferimenti**. Attenzione a questo distinguo. Prima di procedere vediamo un esempio con una semplice classe custom:

C#	Esempio 8.14
1	using System;
2	
3	class Test
4	{
5	public int x = 0;
6	}

```

7
8 class program
9 {
10     public static void Main()
11     {
12         Test[] t1 = new Test[1];
13         t1[0] = new Test();
14         t1[0].x = 5;
15         Test[] t2 = new Test[1];
16         t1.CopyTo(t2, 0);
17         Console.WriteLine(t1[0].x);
18         Console.WriteLine(t2[0].x);
19         t2[0].x = 100;
20         Console.WriteLine(t1[0].x);
21         Console.WriteLine(t2[0].x);
22     }
23 }

```

A voi guardare l'output e comprendere cosa è successo (deja vu, ovviamente)

Per risolvere il problema ed effettuare in santa pace una **deep copy** si può procedere ad esempio come segue:

C#	Esempio 8.15
1	using System;
2	
3	public class Stringa
4	{
5	public string s;
6	}
7	
8	class program
9	{
10	public static void Main()
11	{
12	Stringa[] x1 = new Stringa[1];
13	Stringa[] x2 = new Stringa[1];
14	x1[0] = new Stringa();
15	x2[0] = new Stringa();
16	x1[0].s = "ciao";
17	x2[0].s = x1[0].s;
18	Console.WriteLine(x1[0].s);
19	Console.WriteLine(x2[0].s);
20	x2[0].s = "mamma";
21	Console.WriteLine(x1[0].s);
22	Console.WriteLine(x2[0].s);
23	}
24	}

In questo caso, come è facile verificare, la modifica introdotta alla riga 20 non ha influenza su x1 che conserva il suo valore originale. I due array x1 e x2 sono del tutto indipendenti.

Il problema della shallow copy non interessa i value types, ovviamente. Esiste tuttavia qualche situazione dove i due tipi si mischiano, per così dire. Succede ad esempio nel caso in cui una struct, che è un value type, contiene tra i suoi campi un array, che è appunto un reference type. Ora la domanda che ci possiamo porre è come si presenta la situazione se effettuiamo una copia di una struttura nell'altra.

C#	Esempio 8.16
1	using System;
2	
3	struct Punto
4	{
5	public int x;
6	public int y;
7	public int z;

```

8      public int[] ar1;
9    }
10
11    class Program
12    {
13        public static void Main(string[] args)
14        {
15            Punto p1 = new Punto(); //istanziazione tramite l'operatore new()
16            p1.x = 0; //inizializzazione, così come le due righe successive
17            p1.y = 1;
18            p1.z = 2;
19            p1.ar1 = new int[] {1,2,3};
20
21            Punto p2 = new Punto(); //altra istanziazione
22            p2 = p1;
23            p2.x = 10;
24            p2.ar1[1] = 20;
25            Console.WriteLine(p1.ar1[1]);
26            Console.WriteLine(p1.x);
27        }
28    }

```

La riga 15 così come la 21 istanziano tipo generico Punto creato alle righe che vanno dalla 3 alla 9 e questo lo sapevamo già. L'output è esposto in conseguenza delle righe 25 e 26 ed è costituito dal numero 20 e poi dal numero 0. Quest'ultimo non ci stupisce; il dato x all'interno di p1 vale 0 e non incide il fatto che p2.x si è diventato di valore 10. I value types non interferiscono tra di loro e modificarne non incide su una sua eventuale copia. Il valore 20 invece che ritorna da ar1[1] (riga 25) ci fa capire che la modifica subita dall'array all'interno di p2 (riga 24) ha avuto effetto anche su ar1 contenuto in p1 perchè comunque all'array ci si è arrivati per riferimento. Quindi, nonostante l'array sia contenuto in una struttura la duplicazione di questa non ci risparmia dall'effetto shallow copy legato all'array. Anche questo è un esempio banale ma il concetto espresso è molto importante.

Lavorare con gli arrays

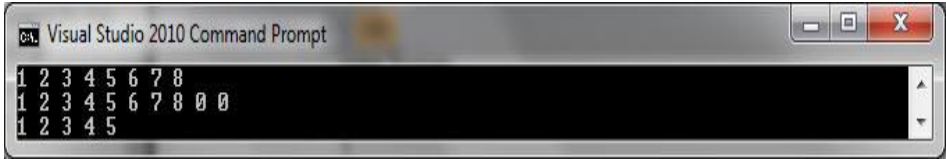
Una delle cose migliori del framework .Net è l'abbondanza di strumenti che ci mette a disposizione. Lavorare con gli array è teoricamente non così terribile ma avere sotto mano quanto ci serve per operare su di essi senza dovercelo creare ex novo ogni volta e soprattutto potendo disporre di un qualche cosa di standard avente una valenza per così dire "universale" non è poi una brutta cosa. Di seguito presente praticamente tutti i più comuni metodi unendovi qualche esempio per presentare quelli che ritengo più utili ripetendone anche qualcuno già visto nel caso in cui si usasse questa breve guida come reference.

Metodo	Spiegazione
Array.AsReadOnly	Rimappa l'array in una collection read-only.
BinarySearch	In una array monodimensionale ordinato cerca un elemento usando un algoritmo di ricerca binaria. Restituisce l'indice dell'elemento trovato
Clear	<p>Pone a zero, o false, o null, o blank a seconda del caso un gruppo di elementi di un array definito da un indice iniziale e da un numero indicante quanti elementi sono oggetto della modifica</p> <pre> using System; class program { public static void Main() { int[] x = {1,2,3,4}; Array.Clear(x,0,2); foreach(int i in x) Console.Write(i + " "); } } </pre> <p>Questo programma ha come output 0 0 3 4</p>

Clone	<p>Crea una <u>shallow</u> copy di un array. Può essere necessaria una operazione di cast</p> <pre>using System; class program { public static void Main() { int[] x = {1,2,3,4}; int[] y = (int[])x.Clone(); } }</pre>
ConstrainedCopy	<p>Effettua una copia di un array in un altro partendo da un dato indice nell'array sorgente e da un altro dato indice nell'array destinazione garantendo che nel caso in cui la copia non andasse a buon fine nessuna modifica verrà memorizzata</p>
ConvertAll	<p>Converte un array di un tipo in un array di un altro tipo.</p>
Copy	<p>Copia un range di elementi da un array ad un altro, eseguendo le conversioni ed i cast richiesti. E' una istruzione che può risultare utile e ha due formati: Array.Copy(sorgente, destinazione, numero di elementi) oppure Array.Copy(sorgente, indice di partenza, destinazione, indice di partenza, numero di elementi). Un esempio che illustra il primo caso:</p> <pre>using System; class program { public static void Main() { int[] x = {1,2,3,4}; int[] y = new int[3]; Array.Copy(x, y, 3); foreach(int i in y) Console.Write(i + "-"); } }</pre>
CopyTo	<p>Effettua una copia di di tutti gli elementi di un array monodimensionale in un altro. Contrariamente a quanto detto su molti forum e siti questa non è una deep copy. Una deep copy, come abbiamo già detto, può soltanto essere implementata manualmente in maniera esplicita.</p> <pre>using System; class program { public static void Main() { int[] x = {1,2,3,4}; int[] y = new int[4]; x.CopyTo(y, 0); // 0 è l'indice di partenza foreach(int i in y) Console.Write(i + "-"); } }</pre>
CreateInstance	<p>Crea una nuova istanza della classe Array. Ne parliamo di seguito a questa tabella</p>
Equals	<p>Determina l'uguaglianza di due array, funzionalità ereditata da object.</p>
Exists	<p>Specifica se nell'array ci sono elementi che specificano un certo predicato. Di seguito un esempio banale:</p> <pre>using System; class program { public static void Main() { int[] x = {1,2,3,4}; Console.WriteLine(Array.Exists(x, major)); } }</pre>

	<pre>private static bool major (int i) { if (i > 2) return true; else return false; } }</pre>
Find	<p>Cerca il primo elemento che risponde ad una certa regola e ne restituisce l'elemento stesso. Rivedendo l'esempio precedente:</p> <pre>using System; class program { public static void Main() { int[] x = {1,2,8,4}; Console.WriteLine(Array.Find(x, major)); } private static bool major (int i) { if (i > 2) return true; else return false; } }</pre>
FindAll	<p>Trova tutte le occorrenze che soddisfano una certa condizione</p> <pre>using System; class program { public static void Main() { int[] x = {1,2,8,4}; int[] s = (Array.FindAll(x, major)); foreach (int y in s) Console.Write(y + " "); } private static bool major (int i) { if (i > 2) return true; else return false; } }</pre>
FindIndex	<p>Restituisce l'indice del primo elemento che soddisfa la condizione. Come esempio basta modificare quello relativo a Find mettendo la funzione FindAll.</p>
FindLast	<p>Restituisce l'ultimo elemento che soddisfa la condizione. Anche in questo caso basta modificare l'esempio precedente.</p>
FindLastIndex	<p>Restituisce l'indice dell'ultimo elemento che soddisfa il predicato.</p> <pre>using System; class program { public static void Main() { int[] x = {1,2,8,1.6,0}; int s = (Array.FindLastIndex(x, major)); Console.Write(s); } private static bool major (int i) { if (i > 2) return true; else return false; } }</pre> <p>restituisce 4 ovvero l'indice di 6 che è l'ultimo elemento maggiore di 2 come imposto dalla regola.</p>

foreach	Ne abbiamo parlato sopra
GetEnumerator	Restituisce IEnumerator per l'array. Foreach ne fa le veci in quasi tutti i casi
GetHashCode	Derivato da Object restituisce un codice hash per l'array
GetLength	Restituisce il numero degli elementi di un array in una data dimensione che deve essere compresa tra 0 e rank (vedi più avanti)
GetLongLength	Come GetLength ma il numero di elementi è espresso tramite un intero 64 bit anzichè 32.
GetLowerBound GetUpperBound	<p>Individuano il minore e maggior indice in una data dimensione di un array. L'uso è Array.GetLowerBound(n-1) dove n è il numero di dimensioni dell'array. Lo stesso va per GetUpperBound.</p> <pre>using System; class program { public static void Main() { int[] arr = new int[] {1,2,3,4,5}; Console.WriteLine(arr.GetLowerBound(0)); Console.WriteLine(arr.GetUpperBound(0)); } }</pre> <p>Il risultato è 0 4. Ovviamente il limite minore è molto spesso 0 ma se prendono in esame array creati con CreateInstance o con altri linguaggi potrebbe non essere vero.</p>
GetValue	<p>restituisce il valore all'indice indicato da GetValue</p> <pre>using System; class program { public static void Main() { int[] arr = new int[] {1,2,3,4,5}; Console.WriteLine(arr.GetValue(0)); } }</pre> <p>per array a due dimensioni si usa GetValue(x, y) per array a 3 dimensioni GetValue(x, y, z) ecc...</p>
Initialize	Inizializza gli elementi di un array per tramite di un costruttore, per i tipi che lo ammettono.
IndexOf	<p>Restituisce la prima occorrenza di un dato elemento in un array o -1 se questo non esiste. <u>Utile quindi per sapere dell'esistenza o meno di un elemento in un array</u></p> <pre>using System; class test { public static void Main() { int[] x = new int[5]{1,2,3,4,5}; Console.WriteLine(Array.LastIndexOf(x, 3)); Console.WriteLine(Array.LastIndexOf(x, 9)); } }</pre> <p>questo programma ha come output 2 e -1 ovvero l'indice del numero 3 e nell'array e l'indice di "non esistenza" di 9.</p>

LastIndexOf	<p>Recupera e restituisce l'ultimo indice di un elemento cercato.</p> <pre>using System; class program { public static void Main() { int[] arr = new int[] {1,1,2,2,2,2}; Console.WriteLine(Array.LastIndexOf(arr, 2)); } }</pre> <p>il risultato è 5, ovvero l'indice più alto al quale troviamo un 2.</p>
MemberwiseClone	<p>Ereditata da Object crea una shallow copy</p>
Resize	<p>Dal momento che non è possibile cambiare le dimensioni di un array dinamicamente una strada percorribile, se non si vuole usare un'altra struttura dati più flessibile, è usare Resize che rialloca gli elementi di un array in una sua copia di diverse dimensioni.</p> <pre>using System; class program { public static void Main() { int[] arr = new int[] {1,2,3,4,5,6,7,8}; foreach (int i in arr) Console.Write(i + " "); Console.WriteLine(); Array.Resize(ref arr, 10); foreach (int i in arr) Console.Write(i + " "); Console.WriteLine(); Array.Resize(ref arr, 5); foreach (int i in arr) Console.Write(i + " "); } }</pre> <p>L'output è il seguente:</p>  <p>Il primo rimangiamento dell'array introduce due nuovi elemento che vengono inseriti ed inizializzati a 0. Il secondo Resize ne riduce le dimensioni ai soli primi 5 elementi. Questo è un modo efficace per ridimensionare gli array nel caso monodimensionale. A testimonianza del fatto che viene effettivamente creato un nuovo array si può far notare come eventuali altri riferimenti all'array originale restino invariati. Esempio:</p> <pre>using System; class program { public static void Main() { int[] ar1 = {1,2,3,4}; int[] ar2; ar2 = ar1; Array.Resize(ref ar1, 10); foreach (int x in ar1) Console.Write(x + " "); Console.WriteLine(); foreach (int x in ar2) Console.Write(x + " "); } }</pre>

	}
Reverse	Inverte gli elementi di un array quindi il primo diventa l'ultimo e viceversa; è possibile ordinare solo range specifici. Array.Reverse(arr) inverte un array in modo completo Array.Reverse(arr, 1,4) inverte solo 4 elementi a partire da quello avente indice 1 compreso.
SetValue	Assegna un valore nella locazione di un dato indice. mioArray.SetValue(elemento, indice). E possibile specificare più indici per array a più dimensione (fino a 3) mioArray.SetValue(elemento, indice1, indice2, indice3)
Sort	Ordina gli elementi di un array. Questa istruzione ha diverse varianti. Vediamo un esempio di ordinamento completo e limitato di un array. Non si possono sortare array multidimensionali (ne risulta una rank exception) <pre>using System; class Program { public static void Main() { int[] arr1 = new int[] {1,18, 7, 33, 18, 17, 44, 0}; int[] arr2 = new int[] {1,18, 7, 33, 18, 17, 44, 0}; Array.Sort(arr1); foreach (int x in arr1) Console.Write(x + " "); Console.WriteLine(); Array.Sort(arr2, 2, 5); // ordina 5 elementi a partire da quello all'indice 2 foreach (int x in arr2) Console.Write(x + " "); } }</pre> ed ecco l'output: 0 1 7 17 18 18 33 44 1 18 7 17 18 33 44 0
ToString	Ereditato da Object, non serve per stampare un array, ma solo il tipo degli elementi dello stesso, se applicato all'array "intero". <pre>int[] x1 = new int[5]; Console.WriteLine(x1.ToString());</pre> Ci restituisce System.Int32[]
TrueForAll	Specifica se gli elementi di un array soddisfano un certo predicato

Come si vede i metodi sono tanti e coprono una vasta tipologia di operazioni sugli array. Credo che una rapida occhiata ad esse non faccia poi così male. Il sito MSDN è a disposizione per ulteriori dettagli.

Di particolare interesse, a mio avviso, il metodo **CreateInstance** in pratica un sistema alternativo per istanziare un array direttamente a livello di classe (System.Array è una classe astratta e non permette costruttori) anche senza conoscere il tipo degli elementi, che può essere passato per via parametrica. Per esempio è possibile scrivere codice siffatto:

```
Type arrayType = typeof(int);
Array arr1 = Array.CreateInstance(arrayType , 100);
arrayType = typeof(char);
Array arr2 = Array.CreateInstance(arrayType , 10);
```

Come si vede arrayType cambia ed è passato per parametro.

Nell'esempio che segue noi useremo **typeof** legato agli interi, in questo esempio l'uso è solo ed esclusivamente didattico. Una volta creata l'istanza possiamo e dobbiamo usare i metodi **SetValue** e **GetValue** per mettere e ricavare dei valori. Non è infatti possibile operare in modo chiamiamolo classico su di un array creato in questo modo usando l'operatore [] proprio per la natura in qualche modo indefinita dell'array.

C#	Esempio 8.17
1	using System;
2	
3	class program
4	{
5	public static void Main()
6	{
7	Array ar = Array.CreateInstance(typeof(int), 3);
8	for (int x = 0; x < 3; x++)
9	ar.SetValue(x, x);
10	for (int x = 0; x < 3; x++)
11	Console.WriteLine(ar.GetValue(x));
12	}
13	}

CreateInstance permette ovviamente di creare array aventi più dimensioni ma anche, qui sta la novità, non zero based, ovvero aventi il primo indice diverso da zero. E' l'unico modo per avere questo tipo di array. L'esempio seguente, sfruttando uno dei numerosi overload di CreateInstance, consiglio di andarli a sbirciare sul solito sito MSDN, crea un array a due dimensioni di due elementi ciascuna, definite alla riga 6, il primo indice della prima dimensione è 3 mentre la seconda dimensione parte dall'indice 50. Di primo acchito non è così semplice raccapezzarsi, soprattutto con la gestione degli indici ma dopo qualche prova le cose diventano più facili. Il problema maggiore sta nell'evitare di andare al di là dei valori consentiti per gli indici consentiti il che solleverebbe una eccezione di tipo OutOfRange. Consideriamo comunque che array non 0-based non sono CLS compliant; in aggiunta si ha un certo handicap anche da un punto di vista prestazionale, visto che il discorso relativo agli array è ottimizzato a livello di CLR per quelli "classici". Detto questo si tratta di una possibilità che forse avrete raramente bisogno di usare ma che può comunque essere utile conoscere:

C#	Esempio 8.18
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	int[] d = {2, 2};
7	int[] i = {3, 50};
8	Array ar = Array.CreateInstance(typeof(int), d, i);
9	ar.SetValue(3, 3, 50);
10	Console.WriteLine(ar.GetValue(3, 50));
11	Console.WriteLine(ar.GetValue(4, 50));
12	}
13	}

Ad essere sincero non ricordo di avere mai utilizzato CreateInstance se non a scopo esemplificativo per cui ritengo che anche a voi capiterà raramente. Probabilmente viene utilizzato internamente al framework, dovrei verificare.

Esistono poi una serie di utili proprietà, alcune già viste:

Proprietà	Spiegazione
IsFixedSize	True se l'array ha una lunghezza fissa
IsReadOnly	True se l'array è read-only
IsSynchronized	Interessante proprietà ci dice se l'accesso all'array è sincronizzato. Utile in funzione della gestione dei thread
Length	Restituisce la lunghezza dell'array <pre>using System; class Test { public static void Main() { int[] arr1 = new int[5]; int[,] arr2 = new int[4,3];</pre>

	<pre> Console.WriteLine(arr1.Length.ToString()); Console.WriteLine(arr2.Length.ToString()); } </pre>
LongLength	Restituisce la lunghezza dell'array usando un long anziché un Int32, ovviamente si deve usare per array particolarmente lunghi. Potete adattare senza problemi l'esempio precedente.
Rank	Restituisce il numero di dimensioni dell'array <pre> using System; class Test { public static void Main() { int[] arr = new int[5]; int[,] arr2 = new int[4,3]; Console.WriteLine(arr.Rank.ToString()); Console.WriteLine(arr2.Rank.ToString()); } } </pre>
SyncRoot	Restituisce un oggetto che può essere usato per sincronizzare l'accesso all'array. Anche questo verrà utile quando parleremo dei thread.

Length e Rank sono sicuramente i più utilizzati, in particolare il primo.

CONVERSIONI

E' possibile effettuare conversioni tra due array esattamente come per le variabili. Perchè ciò sia possibile, dati due array, chiamiamoli ar1 e ar2 devono

- avere le stesse dimensioni,
- essere basati su tipi compatibili, esplicitamente o implicitamente
- essere basati su reference type.

Ecco un esempio:

C#	Esempio 8.19
1	using System;
2	
3	class program
4	{
5	public static void Main()
6	{
7	string[] ar1 = new string[] { "aa", "bb" };
8	object[] ar2 = new object[2];
9	ar2 = ar1;
10	}
11	}

Oppure, con un cast

C#	Esempio 8.20
1	using System;
2	
3	class program
4	{
5	public static void Main()
6	{
7	object[] ar1 = new object[] { "aa", "bb" };
8	string[] ar2 = new string[2];
9	ar2 = (string[])ar1;
10	}
11	}

Inoltre è possibile inserire il concetto di covarianza per gli array, un po' una formalizzazione di quanto visto, che, detto semplicemente, rimappa su di essi i rapporti di conversione implita ed esplicita dei reference types (il concetto non è applicabile ai value types). In base ad esso è possibile ad esempio che, dato un array di tipo A, si possano in essi inserire elementi di tipo B derivato da A. La regola espressa in modo "ufficiale" recita:

se fra due tipi A e B esiste una conversione esplicita o una conversione implicita allora la stessa esiste tra due array di tipi A[R] e B[R] dove R indica il rank degli array.

Un esempio che dà l'idea del concetto, in forma molto semplice, è il seguente che darò senza commenti in quanto si **consiglia vivamente di rivederlo dopo la lettura del capitolo 11** in quanto è indispensabile una conoscenza almeno basilare delle classi..

C#	Esempio 8.21
1	using System;
2	
3	class A
4	{
5	public int x = 0;
6	}
7	
8	class B : A
9	{
10	public int y = 0;
11	}
12	
13	class program
14	{
15	public static void Main()
16	{
17	A[] ar1 = new A[5];
18	B b = new B();
19	B[] ar2 = new B[5];
20	ar2 = (B[])ar1; // si noti il cast
21	ar1[3] = b;
22	}
23	}

CONCLUSIONI.

Non sono certo io a poter evidenziare l'importanza degli array. Basta osservare qualsiasi altro linguaggio di programmazione e troverete strutture del tutto analoghe; questo accade in quanto gli array sono alla base di tanti importanti algoritmi, sono utili, espressivi e molto potenti. Li userete moltissimo, per cui è bene prendere confidenza col loro utilizzo, iniziando dalle operazioni più comuni, la copia, la selezione e la sostituzione di elementi e così via. Si tratta di un compito relativamente semplice visti gli strumenti a disposizione e la già citata semplicità concettuale degli array. Detto questo direi che possiamo calare il sipario sugli array e buttarci sulla loro naturale evoluzione: le collezioni.

SYSTEM.COLLECTIONS

Le collezioni costituiscono un passo successivo naturale rispetto agli array. Il namespace System.Collections contiene molte classi, interfacce e una strutture atte a gestire collezioni, appunto, di dati in qualche modo coerenti. Di questo namespace parleremo ancora a suo tempo quando tratteremo la sua controparte "generica" che ritengo più moderna ed interessante, per ora ci limitiamo ad un approccio di integrazione rispetto a quanto visto nel paragrafo 8. Va detto che molto spesso gli appartenenti a questo namespace sono un po' messi da parte dai programmatori, vuoi per la grande efficienza degli array e per la loro semplicità concettuale, vuoi per una sorta di pigirizia mentale che tende a sottovalutarne i vantaggi. Questo capita anche a me, sia chiaro.

Vediamo subito il contenuto del namespace poi analizzeremo le classi più interessanti:

Classe	Uso
ArrayList	In pratica un array dinamico
BitArray	E' un array di bit espressi in forma booleana (true = 1, false = 0)
CasInsensitiveComparer	Compare due elementi ignorando le maiuscole/minuscole
CollectionBase	E' una classe base astratta per collezioni fortemente tipizzate
DictionaryBase	E' una classe base astratta per collezioni di coppie chiave-valore
Hashtable	Coppie chiave-valore organizzate tramite il codice hash della chiave
Queue	Lista FIFO di elementi
ReadOnlyCollectionBase	Classe base astratta per collezioni fortemente tipizzate, read-only e non generiche
SortedList	Rappresenta una sequenze di coppie chiave-valore ordinata ed accessibile sia tramite la chiave che tramite l'indice.
Stack	Classica struttura LIFO
StructuralComparison	Effettua una comparazione strutturale tra due collezioni di oggetti.

Altrettanto interessanti le interfacce.

Interfaccia	Uso
ICollection	Definisce i metodi da utilizzare per manipolare collezioni generiche
IComparer	Definisce i metodi per la comparazione di due oggetti
IDictionary	Rappresenta una generica collezione aventi come elementi coppie chiave-valore
IDictionaryEnumerator	Enumera gli elementi di un Dictionary non generico
IEnumerator	Permette l'iterazione sugli elementi di una collezione. Molto importante.
IEnumerable	Restituisce l'interfaccia IEnumerator di una collezione
IEqualityComparer	Definisce dei metodi per la comparazione degli oggetti con ugualianza
 IList	Definisce una lista di oggetti accessibili tramite indice
IStructuralComparer	Fornisce metodi per la comparazione strutturale di collezioni
IStructuralEquatable	Fornisce metodi che permettono la comparazione strutturale.

La struttura invece è **DictionaryEntry** che definisce una coppia chiave valore da recuperare o inizializzare.

Bene, ora che abbiamo il nostro elenco cominciamo ad analizzare la classi più interessanti iniziando ovviamente da **ArrayList**. Si tratta, come da descrizione, di un qualche cosa di molto vicino agli array ma con caratteristiche di maggior dinamismo. In particolare è in grado di accrescere le proprie dimensioni in caso di necessità. L'aumento è di un certo numero fisso di locazioni disponibili una volta che viene superato il limite di quelle prefissate.

C#	Esempio 8.22
1	using System;
2	using System.Collections;
3	
4	class program
5	{
6	public static void Main()
7	{
8	ArrayList AL = new ArrayList();
9	Console.WriteLine(AL.Capacity);
10	AL.Add(5);
11	Console.WriteLine(AL.Capacity);
12	for (int x = 0; x < 4; x++)
13	AL.Add(x);
14	Console.WriteLine(AL.Capacity);
15	}
16	}

L'output è quello che segue:

0
4
8

Semplicemente aggiungendo un elemento vengono rese disponibili 4 allocazioni. Una volta che abbiamo richiesto una quinta cella la capacità di AL viene portata a 8. Quindi abbiamo incrementi di 4 chiamiamole celle ogni qual volta sia necessario. E' necessario notare che per aggiungere elementi ad una ArrayList è necessario usare un metodo specifico ovvero **Add**, come alle righe 10 e 13. E' possibile anche fornire al nostro arraylist una dimensione iniziale, ad esempio modifichiamo la riga 8 come segue:

```
ArrayList AL = new ArrayList(8);
```

dove il numero tra parentesi deve essere un Int32. Quest'ultimo accorgimento può essere utile per guadagnare qualche cosa in termini prestazionali se siete sicuri di avere un minimo di dati da inserire in quando vengono evidentemente ridotti gli arrangiamenti dimensionali dell'arraylist. Per accedere ad un singolo elemento è possibile usare il classico operatore **[]**. ArrayList è quindi un utile strumento pur se non dispone delle ottimizzazioni a livello prestazionale tipiche degli array. Anch'esso presenta comunque numerose proprietà e metodi, di seguito i più interessanti (sul sito MSDN troverete la lunga lista completa) con qualche esempio partendo dalla seguente base (se non avete voglia di scrivere troppo ☺):

```
ArrayList arl01 = new ArrayList();
for (int x = 0; x < 5; x++)
    arl01.Add(x);
```

Capacity – lo abbiamo visto

Count – esprime quanti elementi sono presenti nell'arraylist

```
Console.WriteLine(arl01.Count);
```

Add() – aggiunge un elemento in coda all'arraylist, come abbiamo visto

Clear() – elimina tutti gli elementi dell'arraylist

```
arl01.Clear();
```

Contains() – metodo booleano indica se un elemento esiste nell'arraylist

```
Console.WriteLine(arl01.Contains(4));
```

restituisce *true*.

Insert() – inserisce in elemento all'indice specificato. L'uso è: insert(indice, elemento). Molto utile

```
arl01.Insert(3, 100);
```

Remove(), RemoveAt() – rimuovono rispettivamente la prima occorrenza di un elemento e l'elemento ad una certa posizione nell' arraylist

Sort() – ordina gli elementi dell'arraylist.

ToArray() – converte gli elementi dell'arraylist in un array.

Un altro elemento interessante che troviamo nel namespace System.Collections è **Queue**. Si tratta di una struttura che implementa la logica first-in first-out ed è utile per l'appunto in situazione nelle quali si abbia bisogno di memorizzare e recuperarli dati secondo questa modalità.

Creare una Queue è semplice così come aggiungervi un elemento tramite il metodo **Enqueue**:

C#	Esempio 8.23
1	using System;
2	using System.Collections;
3	

```

4  class program
5  {
6      public static void Main()
7      {
8          Queue q1 = new Queue(16);
9          q1.Enqueue(10);
10     }
11 }

```

Anche in questo caso, come per gli arraylist, si può dare o non dare una dimensione iniziale, riga 8, ma è bene conferirne una per limitare i ridimensionamenti man mano che si inseriscono elementi. Dopo l'istruzione alla riga 9 si trova in coda alla Queue. Per recuperare il dato si usa l'istruzione **Dequeue()** mentre con **Count()** abbiamo il numero di elementi presenti nella coda.

C#	Esempio 8.24
1	using System;
2	using System.Collections;
3	
4	class program
5	{
6	public static void Main()
7	{
8	Queue q1 = new Queue(16);
9	q1.Enqueue(10);
10	Console.WriteLine(q1.Count);
11	Console.WriteLine(q1.Dequeue());
12	Console.WriteLine(q1.Count);
13	}
14	}

Il metodo **Peek()** ci fornisce invece una ispezione dell'elemento in rampa di lancio, ovvero quello che sarà espulso alla prima operazione di Dequeue operando senza scrivere nulla, ovvero in modalità readonly, interessante per finalità "ispettive".

C#	Esempio 8.25
1	using System;
2	using System.Collections;
3	
4	class program
5	{
6	public static void Main()
7	{
8	Queue q1 = new Queue(16);
9	q1.Enqueue(10);
10	q1.Enqueue(13);
11	q1.Enqueue(21);
12	Console.WriteLine(q1.Count);
13	Console.WriteLine(q1.Peek());
14	Console.WriteLine(q1.Dequeue());
15	Console.WriteLine(q1.Count);
16	Console.WriteLine(q1.Peek());
17	}
18	}

Anche per questa classe esiste il metodo **Clear()** che elimina in un colpo solo tutti gli elementi presenti, l'uso è del tutto analogo a quanto visto per gli arraylist. Infine, invito ad una visita del solito sito MSDN per vedere altri interessanti metodi e proprietà disponibili per Queue.

La controparte che implementa la logica LIFO (Last In First Out) è invece la classe **Stack**. E' del tutto analoga a Queue, dal punto di vista operativo e le istruzioni di inserimento e recupero dei dati sono le "classiche" **Push()** e **Pop()**. Darò solo l'esempio finale:

C#	Esempio 8.26
1	using System;
2	using System.Collections;
3	
4	class program
5	{
6	public static void Main()
7	{
8	Stack s1 = new Stack(16);
9	s1.Push(10);
10	s1.Push(13);
11	s1.Push(21);
12	Console.WriteLine(s1.Count);
13	Console.WriteLine(s1.Peek());
14	Console.WriteLine(s1.Pop());
15	Console.WriteLine(s1.Count);
16	Console.WriteLine(s1.Peek());
17	}
18	}

Ovviamente si osservi bene l'output: in questo caso il Peek alla riga 13 ci mostra un elemento diverso che sarà il primo ad uscire dallo stack, ovvero quello che nella Queue starebbe all'ultimo posto nell'ordine di eliminazione.

Abbastanza utile è anche la classe [SortedList](#). Si tratta di una struttura dati piuttosto particolare in quanto basata ed accessibile su una coppia di composta da chiave e valore. Anche in questo caso, come nei precedenti stiamo parlando di una entità dinamica in grado di aumentare lo spazio quando la sua proprietà [Count](#) eguaglia quella indicata da [Capacity](#). Esattamente come per quanto visto in precedenza. Facile, per analogia, costruire una SortedList ed attaccarvi un elemento (si usa il metodo [Add\(\)](#)), ricordando che questo deve essere una coppia.

C#	Esempio 8.27
1	using System;
2	using System.Collections;
3	
4	class program
5	{
6	public static void Main()
7	{
8	SortedList SL = new SortedList(16);
9	SL.Add(1, "uno");
10	SL.Add(2, "due");
11	SL.Add(3, "tre");
12	}
13	}

In questo esempio il numero 1 è la chiave mentre la stringa "uno" è il valore. Una chiave non può essere "null" mentre per un valore ciò è ammissibile. Vediamo ora un esempio, estensione del precedente, che mostra come recuperare i dati da una SortList:

C#	Esempio 8.28
1	using System;
2	using System.Collections;
3	
4	class program
5	{
6	public static void Main()
7	{
8	SortedList SL = new SortedList(16);
9	SL.Add(1, "uno");
10	SL.Add(2, "due");
11	SL.Add(3, "tre");
12	for (int i = 0; i < SL.Count; i++)
13	Console.Write(SL.GetKey(i) + " " + SL.GetByIndex(i) + " ");
14	}
15	}

Anche in questo caso **Count** come detto indica il numero degli elementi contenuti nella SortedList, mentre **GetKey** e **GetByIndex** restituiscono rispettivamente la chiave e il valore corrispondente all'indice di scorrimento. In questo caso abbiamo usato un approccio molto simile a quello di un array (tenete presente che, internamente, una SortedList è costituita da due array, uno per le chiavi ed uno per i valori) mentre per accedere ad un valore tramite la corrispondente chiave la cosa non cambia molto, ad esempio provate ad aggiungere la seguente riga:

```
Console.WriteLine(SL[2]); // 2 non è un indice ma la chiave definita alla riga 10
```

e vi verrà restituito il valore "due". Quindi abbiamo visto le due modalità di accesso: per indice e per chiave. Ovviamente, ancora in maniera del tutto simile agli array, l'operatore **[]** può anche essere usato per definire un valore:

```
SL[2] = "two";
```

SortedList è dotato di molti operatori interessanti; rimandandovi al solito sito di MSDN (lo so, è monotono ma parliamo di una miniera di informazioni di prima qualità), ne segnalo qualcuno, si tratta di metodi, a mio avviso particolarmente utile:

Clear – rimuove tutti gli elementi

ContainsKey – true se la sortedlist contiene una certa chiave

ContainsValue – true se la sortedlist contiene un certo valore

IndexOfKey – restituisce l'indice di una certa chiave (partendo da zero)

IndexOfValue – restituisce l'indice di un certo valore (partendo da zero)

Remove – rimuove l'elemento ad una certa chiave

RemoveAt – rimuove l'elemento ad un certo indice

L'ultimo elemento appartenente a System.Collections su cui ci soffermiamo, per ora, è la classe **Hashtable**. Anche in questo caso siamo di fronte ad una coppia fissa composta da una chiave e da un valore. Esattamente come accade nel caso della SortedList un valore al suo interno può essere null ma una chiave non può.

Mi soffermerò altrove in maniera più organica sulle hashtable, elemento abbastanza importante e probabilmente più noto rispetto a quelli presentati, per ora propongo solo un esempio di base che mostra qualche elementare manipolazione.

C#	Esempio 8.29
1	using System;
2	using System.Collections;
3	
4	class program
5	{
6	public static void Main()
7	{
8	Hashtable ht = new Hashtable();
9	ht.Add(1, "uno");
10	ht.Add(2, "due");
11	ht.Add(3, "tre");
12	IDictionaryEnumerator iden = ht.GetEnumerator();
13	while (iden.MoveNext())
14	{
15	string s1 = iden.Value.ToString();
16	Console.WriteLine(s1);
17	}
18	ht.Remove("due");
19	}
20	}

Interessante credo risulti in modo particolare il metodo di navigazioni tra gli elementi della hashtable definito dalla

riga 12 alla 17; in pratica è possibile riversare gli elementi della hashtable in un enumeratore del quale poi si possono leggere i componenti.

E' impossibile non notare la similitudine tra questa classe e SortedList . Tuttavia vi sono profonde differenze, ad esempio il fatto che SortList mantiene l'ordine dell'inserimento, mentre i nuovi elementi in una hashtable vengono ordinati sulla base del codice hash della chiave, non esattamente la stessa cosa, logicamente e concettualmente un po' meno trasparente, mentre da un punto di vista prestazionale le hashtable avrebbero un certo vantaggio in molte situazioni di uso pratico, per esempio nelle operazioni di inserimento e di ricerca. Sortedlist è forse un po' più parco per quanto riguarda il consumo di memoria.

Come detto quest'ultima classe è importante, per cui dedicherò ad essa un approfondimento in una sezione appositamente studiata.