

Capitolo 9

Le stringhe

Le stringhe (parola chiave **string**, alias per la classe **System.String**) sono definibili come **sequenze finite e immutabili di caratteri Unicode**. Tale sequenza è delimitata da una **coppia di doppi apici**. La definizione è chiara, il range dei singoli caratteri è il solito, da U+0000 a U+FFFF, mentre il concetto di **immutabilità** è dovuto al fatto che ogni modifica apportata alla stringa ne crea in realtà una nuova, non cambia quella esistente, almeno restando nell'ambito del codice safe. Questo fatto non penalizza in realtà l'efficienza in alcuni casi, in altri come vedremo, esiste il modo per migliorare le cose. Esempi di stringhe sono banali:

```
string s1 = "aabbnn";  
string s2 = "W la Juventus";  
string s3 = "1234";  
string s4 = "##è}+";
```

Si tratta di un tipo di dati estremamente potente ed importante, basti pensare alla presenza del metodo **ToString()** che praticamente tutti gli oggetti .Net hanno a disposizione e implementano secondo la loro natura. Le stringhe sono reference type, come noto, il che significa che la creazione di una stringa equivale alla definizione di un puntatore ad una zona di memoria nella quale sono contenuti i dati. La classe **System.String** è tale per cui da essa non è possibile ereditare. Vedremo cosa significa esattamente quando parleremo delle classiper ora specifico che il motivo di ciò è proprio nella importanza e nella diffusione delle stringhe, ereditare dalla classe originaria, **System.String**, potrebbe portare alla diffusione di varianti delle stringhe caratterizzate da importanti modifiche nella semantica mentre si vuole che questa struttura sia coerente e monolitica nella sua implementazione. Inoltre una classe **sealed**, (dalla quale appunto non si può ereditare) come è **System.String**, gode di certe ottimizzazioni a run time.

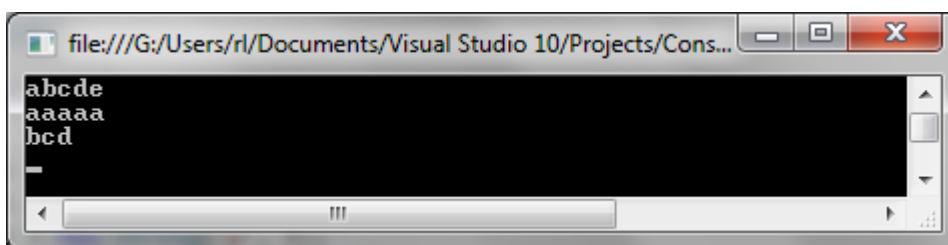
Le righe di codice presentate qui sopra costituiscono il tipico modo in cui viene costruita e definita una stringa. Si usa l'alias **string** seguito dall'identificatore poi, dopo l'operatore di assegnamento si definisce il valore della stringa. Esistono tuttavia dei costruttori (anche qui vedremo con le classi di cosa si tratta esattamente) che, in breve, inizializzano una istanza della classe. **String** ha ben 7 tipi di costruttore ma 4 sono utilizzabili in modalità unsafe e non li trattiamo per ora. Gli altri 3 sono nella forma:

```
String s1 = new String(char[])  
String s1 = new String(char[], int32, int32)  
String s1 = new String(char, int32)
```

L'esempio che segue applica questi costruttori: per quanto molto probabilmente userete la forma presentata in alto il più delle volte vale comunque la pena ricordarsi anche di queste possibilità.

C#	Esempio 6.1
1	using System;
2	public class Class1
3	{
4	public static void Main()
5	{
6	char[] a1 = { 'a', 'b', 'c', 'd', 'e'};
7	String s1 = new String(a1);
8	String s2 = new String('a', 5);
9	String s3 = new String(a1, 1, 3);
10	Console.WriteLine(s1);
11	Console.WriteLine(s2);
12	Console.WriteLine(s3);
14	}
15	}

Con il seguente output:



La prima forma attribuisce alla stringa i caratteri costitutivi di un array of char. La seconda, che è la terza nel codice, riga 9, fa lo stesso ma partendo da un indice nell'array e proseguendo per il numero di caratteri indicati (ecco spiegato il "bcd", ovvero 3 caratteri partendo dall'indice uno) mentre la terza forma (riga 8 nell'esempio) prende un carattere e lo riporta nella stringa tante volte quante indicate dall'intero (nell'esempio risulta "aaaaa").

Ogni stringa che viene creata è memorizzata in una tabella interna detta **intern pool** che viene interrogata dal programma per verificare ad ogni nuova creazione se la stringa esiste già e quindi eventualmente agganciarvi un ulteriore riferimento. Lo scopo è la riduzione della occupazione di memoria. Esiste un metodo per interrogare tale tabella, come vedremo. Importante è evidenziare che in C# le stringhe sono tipi nativi non dei semplici (si fa per dire) array di caratteri come in C. Tuttavia hanno moltissime similitudini con gli array. Vediamo un esempio di partenza:

C#	Esempio 6.1
1	using System;
2	class Test
3	{
4	public static void Main()
5	{
6	string s1 = "Prova"; // definizione della stringa
7	for (int i = 0; i<s1.Length; i++)
8	Console.WriteLine(s1[i]+ " ");
9	}
10	}

La riga 6 ci presenta un metodo di creazione, il più comune, di una stringa come già visto. La riga 8 e la 9 prendono in input ogni singolo carattere della stringa che verrà riscritto in modo da ricreare la stringa “*P r o v a*”. Il programmino in se stesso è molto semplice ma ci illustra alcuni concetti di base. Ad esempio è importante la riga 8 dove troviamo l’operatore `[]` che ci consente di accedere ad un singolo elemento della stringa in modo del tutto analogo a quanto visto per gli array. Sempre analogamente agli array ogni elemento è individuabile attraverso un indice che inizia da sinistra con il valore 0. Anche le stringhe sono pertanto 0 based. Non sono ammessi invece valori negativi come indici. Inoltre è anche interessante vedere l’output se utilizziamo l’istruzione

`Console.WriteLine(s1[n].GetType());` dove ovviamente `n` deve essere sostituito da un valore intero compatibile con la lunghezza della stringa; ebbene il risultato sarà

System.Char.

Questo ci conferma chiaramente che una stringa in realtà è una sequenza di caratteri, di char insomma, come detto nell’introduzione. Anche l’immutabilità di una stringa è un concetto presto spiegato. Sia ad esempio:

```
string s1 = "prova"; scriviamo ora:  
s1 = s1 + " numero 1";
```

Il risultato sarà rispettivamente

P	R	O	V	A
---	---	---	---	---

P	R	O	V	A		N	U	M	E	R	O		1
---	---	---	---	---	--	---	---	---	---	---	---	--	---

La stringa `s1` assumerà i valori indicati prima conterrà i caratteri della parola “PROVA” e poi saranno aggiunti gli altri. Tuttavia il concetto di base è che non viene modificato lo spazio, la regione di memoria in cui si trovano i caratteri della prima stringa ma viene cambiato il riferimento ovvero `s1` punterà ad una zona dove sono contenuti i caratteri del secondo passaggio, mentre la zona che ancora contiene i caratteri “PROVA” sarà liberata celermente dal garbage collector. Questo fatto ci mette però un po’ al riparo da un noto problema; vediamo il codice relativo:

C#	Esempio 6.1
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	string s1 = "aaa";
7	string s2 = s1;
8	s1 = s1 + "bbb";
9	Console.WriteLine(s1);
10	Console.WriteLine(s2);
11	}
12	}

L'output è il seguente;



```
Visual Studio 2010 Command Prompt
aaabbb
aaa
```

Che è logico se si ragiona nei termini appena visti. Tuttavia questo significa anche che non si presentano, in questi casi, quelle criticità viste con la copia degli array. Infatti la stringa `s1` non viene modificata in loco ma, come detto, viene ricreata e pertanto la sua copia `s2` resta non interessata dalle modifiche. Questo sistema non è particolarmente efficiente in molti casi. Supponiamo di avere un programma che partendo da una o più stringhe le modifichi in maniera intensiva. Ci vuole poco a capire che, molto presto, lo heap sarà affollato di parecchi dati che nessuno utilizza più e restano lì ad attendere che il garbage collector li ripulisca. Ovviamente questo conduce ad un probabile e significativo degrado di prestazioni. Come vedremo esiste un costrutto più efficace per fare queste transazioni.

All'interno delle stringhe possono essere presente le classiche sequenze di escape.

C#	Esempio 6.1
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	string s1 = "ciao\nmondo";
7	Console.WriteLine(s1);
8	}
9	}

Ha come output



```
Visual Studio 2010 Command Prompt
ciao
mondo
```

Per effetto di quel `\n` inserito all'interno. Esempio ancora più classico, anche perchè in effetti forse può essere utile è il seguente:

```
string s1 = "C:\\Windows\\System32";
```

inserito nel programma precedente al posto della stringa `s1` esistente e restituisce come output

C:\Windows\System32

Un modo per far sì che le sequenze di escape non abbiano effetto è quello di utilizzare le stringhe **verbatim**, precedute cioè dal carattere `@` come nel caso seguente:

```
string s1 = @"C:\Windows\System32";
```

che ha lo stesso output precedente mentre se nell'esempio xx avessimo scritto:

```
string s1 = @"ciao\nmondo";
```

l'output sarebbe stato un imbarazzante:

ciao\nmondo.

In qualche modo il discorso relativo alle stringhe è più semplice di quello relativo agli array in quanto non abbiamo problemi di multidimensionalità, le stringhe sono sequenze lineari e monodimensionali pure.

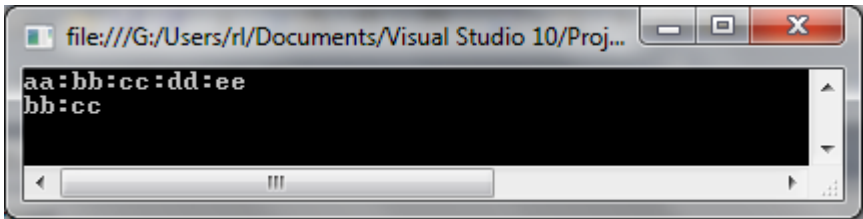
Le stringhe dispongono di un gran numero di metodi devoluti alla loro manipolazione. Vediamo in dettaglio i più utili con i soliti piccoli esempi per quelli più usati:

Clone	<p>Restituisce un riferimento all'istanza corrente della stringa. In pratica ne fa una copia.</p> <pre>using System; class program { public static void Main() { string s1 = "ciao mondo"; string s2 = (string)s1.Clone(); Console.WriteLine(s2); } }</pre>
Compare	<p>Statico, confronta un due stringhe e restituisce un intero che indica il risultato del confronto. Il risultato è 0 se le due stringhe sono identiche, -1 se la seconda è maggiore 1 se la prima è maggiore. Esistono molte varianti di questa istruzione. Qui presento le due versioni più comuni:</p> <pre>using System; class program { public static void Main() { string s1 = "aaaaa"; string s2 = "aaaaaa"; string s3 = "aaaa"; string s4 = "aaaaA"; Console.WriteLine(String.Compare(s1, s2)); Console.WriteLine(String.Compare(s1, s3)); Console.WriteLine(String.Compare(s1, s4)); Console.WriteLine(String.Compare(s1, s4, true)); } }</pre> <p>I primi 3 confronti sono banali. Il quarto invece è nel formato <code>Compare(s1, s2, bool)</code> dove se viene messo il valore <code>true</code> viene ignorato il problema della maiuscola / minuscola.</p>
CompareOrdinal	<p>Statico, confronta due stringhe attraverso il confronto dei valori numerici dei singoli char componenti.</p>
CompareTo	<p>Anche qui viene effettuato un confronto da due stringhe con un output del tipo -1 se la seconda stringa è maggiore, 0 in caso di uguaglianza e 1 se è maggiore la prima</p>

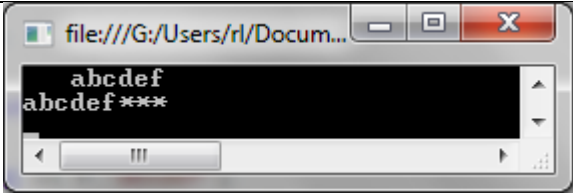
	<pre>using System; class program { public static void Main() { string s1 = "aaaaa"; string s2 = "aaaaa"; string s3 = "aaaa"; string s4 = "aaaaA"; Console.WriteLine(s1.CompareTo(s2)); Console.WriteLine(s1.CompareTo(s3)); Console.WriteLine(s1.CompareTo(s4)); } }</pre> <p>L'output è 0 1 -1</p>
Concat	<p>Statico. Concatena due stringhe. Vale esattamente come il +, al massimo è più descrittivo ma non vi è vantaggio alcuno nell'usare una forma piuttosto che l'altra. Ci sono varie forme nel segnale due: la prima concatena due stringhe, la seconda tutti gli elementi di un array di stringhe.</p> <pre>using System; class program { public static void Main() { string s1 = "Ciao,"; string s2 = " mondo"; string[] ar1 = {"Ciao, ", "mondo"}; Console.WriteLine(string.Concat(s1, s2)); Console.WriteLine(string.Concat(ar1)); } }</pre> <p>L'output è sempre la stringa "Ciao, mondo"</p>
Concat(T)	Generalizzazione del caso precedente provvede una concatenazione di una implementazione di <code>IEnumerable(T)</code>
Contains	<p>Restituisce true o false a seconda che una stringa sia contenuta in un'altra oppure no.</p> <pre>using System; class program { public static void Main() { string s1 = "Ciao,"; string s2 = "ao"; bool b = s1.Contains(s2); if (b) Console.WriteLine("Ok!"); } }</pre>
Copy	<p>Statico, crea una nuova istanza della stringa</p> <pre>using System; class program { public static void Main()</pre>

	<pre> { string s1 = "Ciao"; string s2 = string.Copy(s1); Console.WriteLine(s2); } </pre>
CopyTo	<p>Copia un certo numero di carattere da una stringa partendo da una certa posizione in un array di caratteri a partire anche qui da un dato indice. Il formato e: CopyTo(indice di partenza, array di destinazione, indice di partenza nell'array, numero di caratteri)</p> <pre> using System; class program { public static void Main() { string s1 = "XXXCiao"; char[] ar1 = {'x','x','x','x',' ','2','0','0','9'}; s1.CopyTo(3, ar1, 0, 4); foreach (char c in ar1) Console.Write(c); Console.ReadLine(); } } </pre> <p>Il risultato è: Ciao 2009</p>
EndsWith	<p>Nella sua forma basilare verifica se una stringa termina con una certa sottostringa</p> <pre> using System; class program { public static void Main() { string s1 = "abcde"; if (s1.EndsWith("de")) Console.WriteLine("ok"); } } </pre>
Equals	<p>Statico, booleano, può essere usato per confrontare l'eguaglianza di due stringhe.</p>
Format	<p>Ne parleremo in apposita sezione</p>
GetEnumerator	<p>Recupera un oggetto in grado di iterare tra i caratteri della stringa.</p>
GetHashCode	<p>Restituisce il codice hash per istanza della stringa</p>
GetType GetTypeCode	<p>Restituiscono il tipo. Ad esempio se s1 è una stringa l'output di</p> <pre> Console.WriteLine(s1.GetType()); Console.WriteLine(s1.GetTypeCode()); </pre> <p>sarà:</p> <p><i>System.String</i> <i>String</i></p>
IndexOf	<p>Utile istruzione che ci restituisce la prima occorrenza di un carattere o di una stringa di caratteri.</p> <pre> using System; </pre>

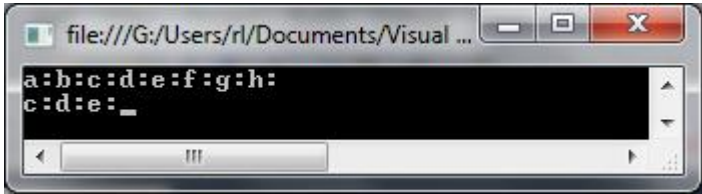
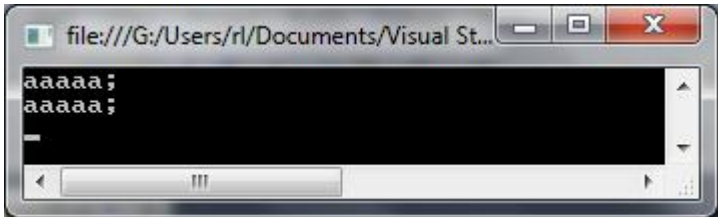
	<pre> class program { public static void Main() { string s1 = "abcde"; Console.WriteLine(s1.IndexOf("cd")); Console.WriteLine(s1.IndexOf('e')); Console.WriteLine(s1.IndexOf("aa")); } } </pre> <p>Questo programma ha come output il numero 2, ovvero di partenza della stringa “cd” all’interno di “abcde”, il numero 4, cioè l’indice di ‘a’ e infine -1 perchè la stringa “aa” non esiste in “abcde”. In questo senso IndexOf può essere usato per verificare l’esistenza di un carattere o di una sottostringa; se il risultato è -1 quanto cercato non c’è.</p>
IndexOfAny	<p>Restituisce l’indice della prima occorrenza degli elementi di un array of char. Detto così, come da definizione, non si capisce niente. Vediamo un esempio:</p> <pre> using System; class program { public static void Main() { string s1 = "abcde"; char[] a1 = { '4', 'r', 'b', 'd' }; char[] a2 = { '4', 'r', 'k', 'd' }; Console.WriteLine(s1.IndexOfAny(a1)); Console.WriteLine(s1.IndexOfAny(a2)); } } </pre> <p>L’output è il seguente: 1 3</p> <p>Ovvero l’indice di b in s1, b è il primo elemento di a1 che si trova in s1 e l’indice di d sempre in s1 che è 3. Utile istruzione per effettuare delle ricerche in una stringa. Abbiamo 3 sintassi differenti:</p> <ol style="list-style-type: none"> 1) IndexOfAny(array) 2) IndexOfAny(array, intero) in cui intero indica il punto di partenza della ricerca nell’ambito <u>della stringa</u> 3) IndexOfAny(array, intero1, intero2) come al punto 2 solo che la ricerca parte da intero1 e coinvolge intero2 caratteri partendo da lì.
Insert	<p>Inserisce in una stringa un’altra stringa a partire da una certa posizione.</p> <pre> using System; class program { public static void Main() { string s1 = "Jutus"; s1 = s1.Insert(2, "ven"); Console.WriteLine(s1); } } </pre>

	<pre> } } </pre>
Intern	<p>Statico, recupera il riferimento interno per la stringa nell'ambito della intern-pool. Utile per recuperare una stringa dalla tabella interna anche qualora non sia stata memorizzata tramite una stringa ma, ad esempio, via StringBuilder.</p>
IsInterned	<p>Recupera il riferimento interno ad una certa stringa</p>
IsNormalized	<p>Indica se la stringa è in una particolare forma di normalizzazione Unicode</p>
IsNullOrEmpty	<p>Statico, verifica se una stringa è null o vuota.</p> <pre> using System; class program { public static void Main() { string s1 = "abcde"; string s2 = ""; string s3 = null; Console.WriteLine(String.IsNullOrEmpty(s1)); Console.WriteLine(String.IsNullOrEmpty(s2)); Console.WriteLine(String.IsNullOrEmpty(s3)); } } </pre> <p>Ovviamente s1 non è nè null nè empty al contrario delle altre due.</p>
IsNullOrWhiteSpace	<p>Statico, verifica se una stringa è null o è costituita solo da blanks, molto similmente a IsNullOrEmpty.</p>
Join	<p>Statico, consente di concatenare gli elementi di un array o di una collezione inserendo eventualmente un separatore. Il formato è duplice:</p> <p><i>Join(separatore, array)</i> <i>Join(separatore, array, indice di partenza nell'array, numero elementi)</i></p> <pre> using System; class program { public static void Main() { string s1 = ""; string s2 = ""; string sep = ":"; string[] a1 = {"aa", "bb", "cc", "dd", "ee"}; s1 = String.Join(sep, a1); s2 = String.Join(sep, a1, 1, 2); Console.WriteLine(s1); Console.WriteLine(s2); } } </pre> <p>L'output è il seguente:</p> 

	<p>La variante Join(T) concatena gli elementi di una certa collezione che, ovviamente, deve implementare IEnumerable..</p>
LastIndexOf	<p>Restituisce l'indice dell'ultima occorrenza di un carattere o il primo indice dell'ultima occorrenza di una sottostringa. Consiglio di dare uno sguardo su MSDN alle diverse varianti di questa istruzione.</p> <pre>using System; class program { public static void Main() { string s1 = "aabbaabbaa"; Console.WriteLine(s1.LastIndexOf('a')); Console.WriteLine(s1.LastIndexOf("aa")); } }</pre> <p>L'output è cosuito dal numero 9 (indice dell'ultima 'a' in s1) e 8 (primo indice dell'ultima occorrenza della sottostringa "aa" in s1)</p>
LastIndexOfAny	<p>Restituisce l'indice dell'occorenza dell'ultimo elemento di un array presente nella stringa.</p> <pre>using System; class program { public static void Main() { string s1 = "abcdefgh"; char[] a1 = { 'x', 't', 'a', 'e' }; Console.WriteLine(s1.LastIndexOfAny(a1)); } }</pre> <p>Il risultato è 4, indice di 'e' nella stringa</p>
MemberWiseClone	Restituisce una shallow copy della stringa
Normalize	Restituisce la stringa in formato binario normalizzato. Ne riparleremo
PadLeft - PadRight	<p>Permettono di allineare una stringa a sinistra o a destra per tramite di spazi o di un carattere predefinito.</p> <pre>using System; class program { public static void Main() { string s1 = "abcdef"; Console.WriteLine(s1.PadLeft(9)); Console.WriteLine(s1.PadRight(9, '*')); Console.ReadLine(); } }</pre> <p>L'output è il seguente:</p>

	 <p>Tramite PadLeft la stringa è allineata verso sinistra di 3 posizioni (9 – 6, che è la lunghezza della stringa fa 3)., mentre PadRight allinea a destra inserendo 3 caratteri “*” come indicato.</p>
Remove	<p>Rimuove tutti i caratteri, o un certo numero prefissato, da una stringa a partire da una certa posizione Pertanto ci sono due formati: Remove(<i>posizione di partenza</i>) Remove(<i>posizione di partenza, numero di elementi da togliere</i>)</p> <pre>using System; class program { public static void Main() { string s1 = "abcdefgh"; Console.WriteLine(s1.Remove(2)); Console.WriteLine(s1.Remove(0, 2)); } }</pre> <p>L'output è costituito dalle stringhe <i>ab</i> <i>cdefgh</i></p> <p>Come si può notare il comando non è distruttivo sulla stringa originaria. Per indicare posizione di partenza e numero di elementi da cancellare bisogna usare gli Int32.</p>
Replace	<p>Utile istruzione che sostituisce tutte le occorrenze di un carattere o di una stringa con un altro carattere o un'altra stringa.</p> <pre>using System; class program { public static void Main() { string s1 = "aaabbbccc"; s1 = s1.Replace('a', 'b'); Console.WriteLine(s1); Console.WriteLine(s1.Replace("bb", "cc")); Console.ReadLine(); } }</pre> <p>La stringa finale sarà composta di sole 'c'.</p>
Split	<p>Interessante e utile istruzione che permette di suddividere una stringa in una array di sottostringhe generate basandosi su di un separatore. Un primo semplice esempio è il seguente:</p> <pre>using System; class program { public static void Main()</pre>

	<pre> { string s1 = "cuori,quadri,fiori,picche"; string[] a1 = s1.Split(','); foreach (string s in a1) Console.WriteLine(s); Console.ReadLine(); } } </pre> <p>Questo programma partendo dalla stringa s1 costruisce un array di 4 elementi ciascuno dei quali è una stringa, precisamente “cuori”, “quadri”, “fiori”, “picche”. L’istruzione è in realtà molto duttile e su Internet si trovano vari esempi i utilizzo più avanzati. L’esempio seguente fa un piccolo passo in più permettendo la separazione in parti di una stringa contenente più separatori diversi. Questi sono definiti tramite l’array di caratteri a1.</p> <pre> using System; class program { public static void Main() { string s1 = "cuori,quadri;fiori picche"; char[] a1 = { ' ', ':', ';', ',', ' ' }; string[] a2 = s1.Split(a1); foreach (string s in a2) Console.WriteLine(s); } } </pre>
StartsWith	<p>Restituisce True se una stringa inizia con un certo carattere o una certa stringa.</p> <pre> using System; class program { public static void Main() { string s1 = "cuori,quadri,fiori,picche"; Console.WriteLine(s1.StartsWith("cuori")); Console.ReadLine(); } } </pre>
SubString	<p>Restituisce una sottostringa a partire da un certo indice alla fine, come nel primo caso dell’esempio che segue o a partire da un certo indice per un numero di posizioni predefinito (si usano sempre Int32).</p> <pre> using System; class program { public static void Main() { string s1 = "abcdefgh"; Console.WriteLine(s1.Substring(2)); Console.WriteLine(s1.Substring(2, 3)); } } </pre> <p>L’output è costituito dalle stringhe <i>cdefgh</i> <i>cde</i></p>
ToCharArray	<p>Converte tutta la stringa in un array di caratteri Unicode. In</p>

	<p>alternativa converte in array di caratteri la porzione indicata a partire da un certo indice e per un numero definito di caratteri.</p> <pre>using System; class program { public static void Main() { string s1 = "abcdefgh"; char[] a1 = s1.ToCharArray(); char[] a2 = s1.ToCharArray(2, 3); foreach (char c in a1) Console.Write(c + ":"); Console.WriteLine(); foreach (char c in a2) Console.Write(c + ":"); Console.ReadLine(); } }</pre> <p>L'output è:</p> 
ToLower	<p>Restituisce una copia della stringa con i caratteri convertiti tutti in minuscolo. "aBcD".ToLower() -> "abcd"</p>
ToUpper	<p>Restituisce una copia della stringa con i caratteri convertiti tutti in maiuscolo "aBcD".ToUpper() -> "ABCD"</p>
Trim	<p>Elimina all'inizio e alla fine di una stringa i caratteri specificati. Senza nulla specificare elimina gli spazi.</p> <pre>using System; class program { public static void Main() { string s1 = " aaaaa "; Console.WriteLine(s1.Trim()+""); String s2 = "****aaaaa****"; Console.WriteLine(s2.Trim('*')+""); Console.ReadLine(); } }</pre>  <p>È anche possibile specificare un array di carattere da eliminare.</p>
TrimStart e	<p>Identici a Trim solo che agiscono rispettivamente solo all'inizio ed alla</p>

TrimEnd	fine della stringa.
---------	---------------------

Da segnalare poi due importanti proprietà, la seconda poi assolutamente fondamentale per la sua frequenza d'uso:

Chars	Restituisce il carattere ad una data posizione, tramite l'operatore []
Length	Restituisce il numero di caratteri che costituisce la stringa "aaa".Length restituisce il numero 3. Questa proprietà come è facile capire è estremamente utile.

Le istruzioni che abbiamo presentato consentono di fare moltissime operazioni sulle stringhe, questo è evidente. Alcune linee guida molto semplici:

- La comparazione di due stringhe può essere fatta normalmente in maniera efficiente e descrittiva tramite Equals
- La copia di una stringa in un'altra è normalmente eseguita, sempre in maniera descrittiva, tramite Clone.

Ma dei tanti metodi si possono fare gli usi più dispratati. Ad esempio il seguente programma permette di rovesciare una stringa senza far uso dei soliti, lenti cicli:

C#	Esempio 6.1
1	using System;
2	using System.Text;
3	public class Class1
4	{
5	public static void Main()
6	{
7	string s1 = "chi ha paura del 2012?";
8	char[] a1 = s1.ToCharArray();
9	Array.Reverse(a1);
10	s1 = new string(a1); ;
11	Console.WriteLine(s1);
12	}
13	}

In questo caso ci siamo giovati del metodo Reverse tipico degli array. Una forma di collaborazione, insomma tra classi diverse.

L'uso delle stringhe nell'ambito del framework è assai importante e diffuso per cui è buona norma conoscere i dettagli della loro manipolazione. Esiste ad esempio una lunga letteratura relativa alla comparazione di due stringhe, pratica superficialmente parlando molto banale ma ricca invece di insidie e di potenziali perdite di performance, soprattutto vi sono complicazioni legate talvolta alla "cultura" nella quale le stringhe sono esposte; su MSDN troverete molto materiale su questo delicato argomento.

Una questione diversa sorge quando è necessario effettuare molteplici operazioni di modifica su una stringa. Per esempio poniamo il caso di dover costruire una stringa attraverso una certa sequenza di modifiche su di essa:

“a”
“ab”
“abc”
“abcd”
ecc....

Sappiamo che le stringhe sono entità immutabili ed ogni operazione di aggiunta di un carattere, come nel caso precedente che è uno ovviamente dei tanti che possono presentarsi, occuperebbe nuovo spazio nello heap dereferenziando quello usato in precedenza. Pertanto, alla fine, rischieremmo di trovarci, non nel caso specifico ma in presenza di una proliferazione di operazioni simili e di più grandi dimensioni, con lo heap occupato da un gran numero di “relitti” che il garbage collector dovrà eliminare in considerazione del fatto che ad ogni iterazione viene creata in realtà in memoria una stringa nuova mentre quella vecchia viene dereferenziata. Questo fatto porta a problemi di efficienza e a rischi di memory leak. Pertanto viene fatto uso di una classe nuova che è contenuta nel namespace (che è necessario pertanto richiamare) **System.Text**: parliamo della classe **StringBuilder**. Come dice il nome stesso si tratta di una classe specializzata nella costruzione di stringhe, cosa che svolge in maniera molto efficiente. Il punto principale è che i costrutti ottenuti da questa classe sono modificabili per così dire in loco; ogni modifica non ricrea nello heap un’area diversa dedicata ai contenuti modificati ma il lavoro avviene nella stessa zona originariamente occupata. Cominciamo quindi a usare la nostra classe:

```
StringBuilder sb1 = new StringBuilder ("Ciao");
```

Poi espandiamo la stringa usando il metodo **Append**

```
sb1.Append(", mondo!");
```

sb1 ora ha il valore “Ciao, mondo”. Come detto sb1 viene modificato in loco non viene creato per esso un nuovo riferimento nello heap. StringBuilder alloca lo spazio necessario per contenere la stringa spazio che cresce quando è necessario inserire i caratteri. Conoscendo a priori il numero dei caratteri che sarà definito è possibile impostare questa quantità in modo diretto nell’ambito della inizializzazione stessa:

```
StringBuilder sb1 = new StringBuilder("Ciao", 20);
```

In questo modo viene definito uno spazio di 20 posizioni, comunque ulteriormente ampliabile se necessario. La quantità di spazio disponibile è interrogabile tramite la proprietà **MaxCapacity**:

```
Console.WriteLine(sb1.MaxCapacity);
```

Una volta che la costruzione di una istanza di StringBuilder è terminata il metodo ToString ci permette di riversarla in una stringa. Vediamo l’esempio che segue, alla riga 2 viene richiamato il namespace **System.Text**, come detto necessario per poter usare la classe StringBuilder:

C#	Esempio 6.1
1	using System;
2	using System.Text;
3	class program
4	{
5	public static void Main()
6	{
7	StringBuilder sb1 = new StringBuilder("a");
8	sb1.Append("b");
9	sb1.Append("c");
10	sb1.Append("d");
11	string s1 = sb1.ToString();
12	Console.WriteLine(s1);
13	}
14	}

Alla fine viene creata la stringa s1 che ha come contenuto “abcd”. Come anticipato, alla riga 11 viene mostrato come è semplice riversare il contenuto di una istanza StringBuilder in una stringa al termine delle elaborazioni seguite su detta istanza. Proprio questo è lo schema operativo da utilizzare: lavorare sulle più duttili istanze StringBuilder e, solo alla fine, passare il risultato del lavoro svolto in una stringa.

E’ possibile definire o modificare la quantità di spazio che la nostra variabile di tipo StringBuilder utilizzerà anche tramite la proprietà **Capacity**. Ad esempio scrivere:

```
sb1.Capacity = 10;
```

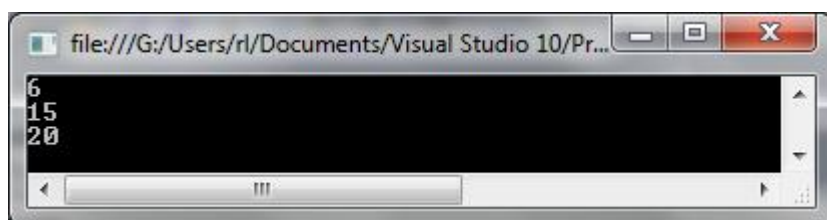
con sb1 ovviamente istanza della classe StringBuilder, equivale a predisporre una capacità contenitiva di 10 caratteri per sb1. Questo evidentemente aiuta a livello prestazionale in quanto non sono necessarie operazioni di ridimensionamento qualora si aggiungano caratteri fino appunto alla capacità massima. Se questa dovesse essere superata la capacità sarà ampliata di altri 10 caratteri e così via. Ad esempio al termine del programma seguente sb1 avrà una capacità complessiva pari a 20:

C#	Esempio 6.1
1	using System;
2	using System.Text;
3	class program
4	{
5	public static void Main()
6	{
7	StringBuilder sb1 = new StringBuilder("abcdef");
8	sb1.Capacity = 10;
9	sb1.Append("ghilmn");
10	Console.WriteLine(sb1.Capacity);
11	Console.ReadLine();
12	}
13	}

Altra proprietà utile è Length, similmente a quanto visto per gli array e le stringhe. Questa proprietà è interrogabile e settabile.

C#	Esempio 6.1
1	using System;
2	using System.Text;
3	class program
4	{
5	public static void Main()
6	{
7	StringBuilder sb1 = new StringBuilder("abcdef");
8	sb1.Capacity = 10;
9	Console.WriteLine(sb1.Length);
10	sb1.Length = 15;
11	Console.WriteLine(sb1.Length);
12	Console.WriteLine(sb1.Capacity);
13	}
14	}

Con il seguente output:



Il primo numero, 6, è la capacità di sb1 dopo che è stata inserita la stringa “abcdef”. Invece 10 è la capacità di sb1. Dopo che la proprietà Length è stata fissata a 15 la capacità, come prevedibile sale a 10 + 10. Length è utilizzabile anche per azzerare il contenuto di una istanza.

sb1.Length = 0; in pratica ripulisce sb1, in maniera più semplice di quanto sia con Remove, che vedremo tra breve. Questo sistema è abbastanza efficiente. Al contrario Length è readonly per le istanze di System.String.

Per quanto non si usi molto in questi casi (almeno io raramente ne ho fatto uso manipolando un'istanza StringBuilder) l'operatore [] è utile per accedere al singolo elemento di una variabile StringBuilder. Ad esempio nel caso precedente:

sb1[0] è il carattere 'a' mentre sb1[2] è il carattere 'c'.

Anche StringBuilder ha interessanti proprietà e metodi.

Append	Come visto nell'esempio precedente appende contenuto in coda a una istanza StringBuilder.
AppendFormat	<p>Piuttosto complesso, accoda una stringa formattata in base a zero o più specifiche istanze di formattazione. Ogni formato viene sostituito dalla rappresentazione in stringa di un oggetto che ha la funzione di argomento.</p> <pre> using System; using System.Text; using System.Globalization; class program { public static void Main() { </pre>

	<pre> StringBuilder sb1 = new StringBuilder("# "); StringBuilder sb2 = new StringBuilder("* "); float f = 3.333F; CultureInfo ci = new CultureInfo("de-DE", true); sb1.Append(f); sb2.AppendFormat(ci, f.ToString()); Console.WriteLine(sb1.ToString()); Console.WriteLine(sb2.ToString()); } } </pre>
AppendLine	<p>Appende in coda la stringa e un fine riga.</p> <pre> using System; using System.Text; class program { public static void Main() { StringBuilder sb1 = new StringBuilder("aa"); sb1.AppendLine("bb"); Console.Write(sb1.ToString()); } } </pre>
Clear	<p>Ripulisce l'istanza a cui viene applicata.</p> <pre> sb1.Clear(); </pre> <p><u>NB – Al momento in cui scrivo questa istruzione è inserita nella documentazione MSDN ma in realtà non compila.</u></p> <p>Per ripulire una istanza, almeno è possibile ad esempio ricorrere alla proprietà Length, come visto in precedenza.</p>
CopyTo	<p>Copia una porzione in una array of char. Il format del comando è: <i>sb.CopyTo(posizione iniziale nella stringa, array di caratteri, indice iniziale nell'array, numero di caratteri)</i></p> <pre> using System; using System.Text; class program { public static void Main() { StringBuilder sb1 = new StringBuilder("abcdef"); char[] ar1 = {'*', '*', '*', '*'}; sb1.CopyTo(1, ar1, 2, 2); foreach (char c in ar1) Console.WriteLine(c); } } </pre>
EnsureCapacity	Garantisce che la capacità minima dello StringBuilder sia quella stabilita.
Equals	<p>Stabilisce l'uguaglianza tra due istanze</p> <pre> sb1.Equals(sb2); </pre>
Insert	<p>Inserisce la rappresentazione in formato stringa di un certo oggetto nell'istanza corrente. Il formato tipico è: <i>Insert(indice, oggetto);</i></p> <pre> using System; using System.Text; public class Class1 </pre>

	<pre> { public static void Main() { StringBuilder sb1 = new StringBuilder("aaaaaa"); sb1.Insert(2, '2'); Console.WriteLine(sb1.ToString()); } } </pre>
Remove	<p>Rimuove a partire da un certo indice x un numero n di elementi</p> <p>sb1.Remove(x, n)</p>
Replace	<p>Rimpiazza tutte le occorrenze di un carattere o di una stringa. Quattro formati diversi:</p> <pre> sb1.Replace(char, char) sb1.Replace(string, string) sb1.Replace(char, char, int32, int32) sb1.Replace(string, string, int32, int32) </pre> <p>I primi due sono chiari mentre come al solito I due interi indicano un indice di inizio in cui cercare il carattere o la stringa e quanto lungo deve essere l'intervallo di ricerca.</p> <pre> using System; using System.Text; public class Class1 { public static void Main() { StringBuilder sb1 = new StringBuilder("aaaaaabbbbbbb"); sb1.Replace('a', 'c', 2, 2); Console.WriteLine(sb1.ToString()); Console.ReadLine(); } } </pre>

Infine sono utili 4 proprietà:

Capacity	Che ottiene o setta la capacità di una istanza StringBuilder
Chars	Che ottiene il carattere ad un certo indice
Length	Come visto restituisce o setta la lunghezza della istanza corrente
MaxCapacity	Restituisce la massima capacità possibile.

E' evidente che la classe StringBuilder è disegnata allo scopo di compiere le operazioni elementari e più comuni sulle sequenze di caratteri in modo efficiente e rapido. La sua forza è proprio quella di svolgere egregiamente compiti ripetuti e massivi. Un errore in cui però non bisogna cadere è quello di usare sempre e comunque questa classe. Va considerato infatti che la sua efficienza è frutto di una logica implementativa abbastanza corposa che può risultare penalizzante in situazioni in cui non vi sia necessità di molte elaborazioni. Tra l'altro la classe StringBuilder non ha la stessa quantità di membri di cui invece dispone la classe String e non è certamente una buona idea saltare da una all'altra per applicare i metodi o le proprietà che

ci servono... Qualcuno afferma anche che l'uso di `StringBuilder` appesantisce la lettura del codice (chissà poi perchè... ma ognuno giudichi da se). Per cui se dovete concatenare due stringhe o gestire un semplice input usate tranquillamente le classe `String` nella sua forma naturale.

FORMATTAZIONE (cenni)

E' un'altra questione abbastanza importante e utile. In breve `C#` permette di esprimere l'output in maniera che sia più facilmente leggibile e immediatamente comprensibile. L'argomento è in realtà piuttosto complesso e sarà ampliato a completato più avanti, quando saranno chiari altri concetti.

In questa sezione darò solo qualche concetto introduttivo, più che altro per stimolare la curiosità sull'argomento, nella sezione relativa gli esempi provvederò qualche dettaglio in più, mentre un po' di questo argomento sarà coperto in altri comparti, come quello relativo alla gestione delle date e delle ore .

Come abbiamo detto è possibile scrivere l'output come segue;

```
Console.WriteLine("x = {0}", x);
```

Questa è una forma molto utile che può essere resa ancora più user friendly agli occhi dell'utente in xaso di valori numerici attraverso opportune formattazione. Allo scopo esiste una tabella che ci aiuta in questo senso:

c = currency, la valuta corrente
d = decimal
e = scientifico
f = fixed point
g = generale
n = con virgola
r = con arrotondamento
x = esadecimale

Ovviamente non tutte le varianti si possono applicare a tutti i numeri in alcuni casi possono essere sollevate delle eccezioni dovute ad un formato non corretto. D esempio il numero 2,1234 non potrà essere convertito in esadecimale. Analogamente la gestione della valuta corrente può dare qualche problema in videate di tipo DOS. Detto questo l'esempio seguente può essere adattato e modificato a vostro piacere per compiere qualche prova:

C#	Esempio 6.1
1	using System;
2	
3	public class Class1
4	{
5	public static void Main()
6	{
7	int i = 20;
8	double d = 2.3456;
9	Console.WriteLine("{0:c}", i);
10	Console.WriteLine("{0:g}", i);
11	Console.WriteLine("{0:d}", i);

```

12         Console.WriteLine("{0:e}", i);
13         Console.WriteLine("{0:f}", i);
14         Console.WriteLine("{0:n}", i);
15         Console.WriteLine("{0:x}", i);
16         Console.WriteLine("{0:c}", d);
17         Console.WriteLine("{0:g}", d);
18         Console.WriteLine("{0:e}", d);
19         Console.WriteLine("{0:f}", d);
20         Console.WriteLine("{0:n}", d);
21         Console.WriteLine("{0:r}", d);
22     }
23 }

```

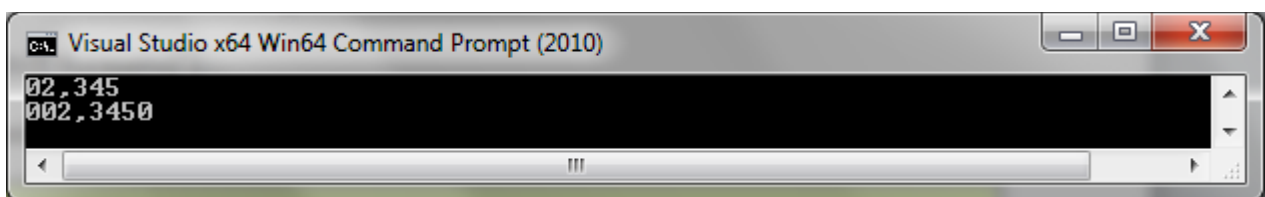
In generale il formato che viene usato segue la definizione

{“indice, [allineamento][:formato]”}

E in questo senso possiamo scrivere cose del genere, utilizzando lo zero per la formattazione:

C#	Esempio 6.1
1	using System;
2	
3	class Test
4	{
5	public static void Main()
6	{
7	float value = 2.345f;
8	Console.WriteLine("{0:00.000}", value);
9	Console.WriteLine("{0:000.0000}", value);
10	}
11	}

Con il relativo output:



Provate a vedere cosa accade aggiungendo una riga del tipo:

```
Console.WriteLine("{0:111.11111}", value);
```

Per utilizzare un placeholder generico si utilizza il simbolo #, quindi ad esempio:

```
Console.WriteLine("{0:##.####}", value);
```

Esistono poi possibilità di customizzare l'esposizione delle date in molti formati diversi, cosa che vedremo quando parleremo della classe apposita di gestione di giorni, mesi, anni, ore ecc...