

## Capitolo 2 Hello, World

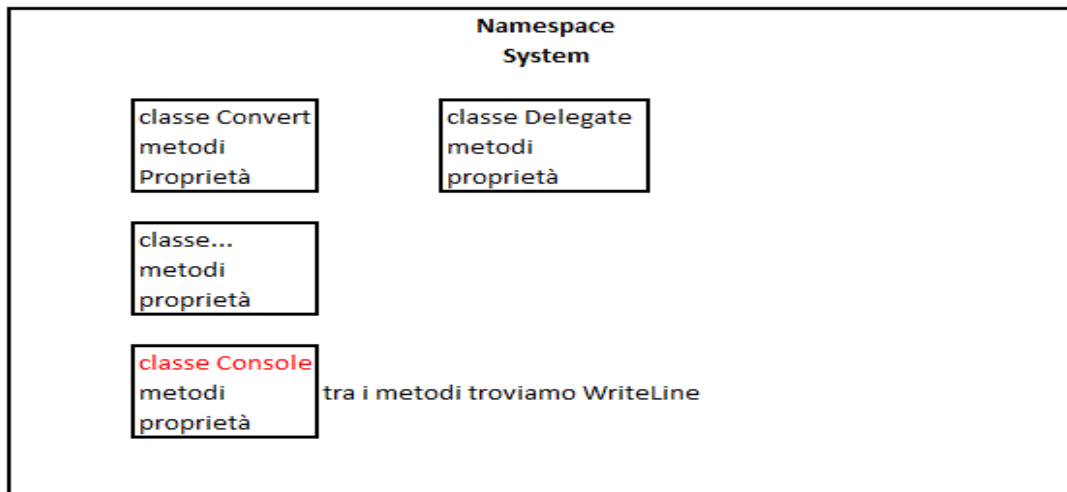
Ci togliamo subito il pensiero del classico Hello World, che fa da delicato apripista per tutti i linguaggi di programmazione, il quale ci fornirà subito alcuni utili spunti:

```

C#   Esempio 2.1
1   using System;
2   class Test
3   {
4       public static void Main()
5       {
6           Console.WriteLine("##### Salutiamo #####");
7           Console.WriteLine("Hello, World!");
8       }
9   }

```

La riga 1 introduce la parola chiave **using** che ci consente di utilizzare il **namespace System**. Che cosa è un **namespace**? Esso in pratica definisce un ambito, una sezione, all'interno del quale possiamo definire entità di livello più specifico (una classe ad esempio) oppure anche un altro namespace. In pratica siamo di fronte ad un guscio che definisce il perimetro del codice interno e lo delimita. Si realizza quindi una sorta di gioco di scatole cinesi ove all'interno di un namespace possiamo trovare le classi che a loro volta espongono metodi e proprietà. Per ora è importante capire che la funzione di questa entità è prevalentemente di tipo organizzativa proprio perchè al suo interno possiamo mettere praticamente di tutto garantendo nel contempo al contenuto una precisa, stabile ed univoca identità. Anche la libreria di classi di .Net, per fare un esempio nobile, è livellata gerarchicamente in namespace. Nell'esempio 2.1 occorre scrivere sullo standard output e questo è possibile utilizzando il metodo **WriteLine** che appartiene alla classe **Console** contenuta nel namespace System. Visivamente potrebbe essere così:



Ovviamente System, come è evidente, contiene molte altre classi, qui ne sono state indicate solo alcune a campione e a sua volta Console ha svariati altri metodi e proprietà. Vedremo tra breve come si accede a questi metodi e proprietà e come si naviga tra i namespace. Anche **using** è una keyword interessante. Essa ci consente di usare per le vie brevi, questa frase ha un senso come vedremo, i namespace del framework ma anche quelli eventualmente creati dai programmatori. In realtà si tratta di una istruzione più flessibile di quanto qui presentato ma è prematuro parlarne. Diciamo, in generale, che come dice il nome stesso ci mette a disposizione qualche cosa che altrimenti non potremmo usare, in questo caso il namespace System con tutto il suo contenuto. Se non specifichiamo al compilatore che intendiamo usare un certo namespace non potremo usare il suo contenuto, almeno in maniera immediata. Per adesso quindi ci basta dire che **using** ci apre un ambito, un perimetro, all'interno del quale usare un "qualcosa" che, al di fuori di tale ambito, non è più disponibile a meno di non andarlo specificamente a cercare (esempio 2.2).

## C# - Capitolo 2 – Hello, World

La riga 2 introduce la definizione di **classe**. C# è un linguaggio a oggetti e tutto deve essere definito tramite classi che a loro volta possono appartenere a namespace definiti dall'utente esattamente come nella figura precedente. Delle classi parleremo ampiamente a suo tempo l'importanza dell'argomento nella programmazione odierna è nota anche ai non esperti. La riga 3 ci dimostra che, come avviene in C/C++ e Java, anche in questo linguaggio vengono utilizzate le **parentesi graffe** per aprire e chiudere le "frasi", o meglio i blocchi di codice del programma. La graffa che chiude il blocco che si apre a riga 3 si trova alla riga 9, mentre il blocco aperto alla riga 5 termina alla 8 come suggerisce anche l'indentazione. E' cosa buona indentare i propri programmi. Per quanto il compilatore non lo richieda nella maniera più assoluta l'occhio umano ringrazia sempre quando la leggibilità è migliore. Si evitano anche degli errori. Gli editor più comuni (no, Notepad no) supportano l'indentazione automatica della sintassi di C#. E la comodità di questo aspetto vale a maggior ragione quando, come nel pur banale esempio di sopra, si trovano blocchi di codice innestati in altri (i blocchi per l'appunto possono essere innestati uno nell'altro definendo una sorta di gerarchia a livelli). Ogni riga o meglio ogni singolo comando di un blocco deve terminare con il **punto e virgola**, vedasi ad esempio le righe 6 e 7, in assenza del quale sarà il compilatore a lamentarsi facendo chiaramente capire cosa si aspetta.

Una parola la meritano anche i commenti che sono introdotti da `//` se su singola riga o racchiusi in una sequenza `/* ... */` se sono invece espansi in più righe. Li vedremo all'opera nell'ultimo esempio di questo paragrafo.

La riga 4 introduce il concetto una serie di parole chiave importanti (**public** è generalmente opzionale in casi come il 2.1) tra cui fondamentale è l' **entry-point** del programma che è costituito dal metodo **Main()**. Il JIT cerca sempre quel particolare punto nel codice in cui è presente Main() quando deve mandare in esecuzione un programma. L'assenza di Main() viene salutata in maniera molto scorbutica dal compilatore (invece in linea teorica è possibile che più di una classe abbia un metodo che sia chiama Main ma in tal caso bisogna usare la direttiva `/main` del compilatore per avvisarlo su qual è il Main principale). L'entry point costituisce anche il punto di partenza per l'intero assembly a cui il programma appartiene. In questo senso, ovvero in presenza dell'entry point, è corretto parlare di **applicazione** e in particolare di **dominio dell'applicazione che viene creato nel momento in cui questa viene eseguita**. Questo concetto è importante perchè permette di capire come possano esistere anche più istanze della medesima applicazione; infatti per ciascuna di queste viene creato un **proprio spazio isolato** e protetto nel quale vivono le proprie istanze delle variabili, dei tipi, insomma tutto il corredo necessario all'applicazione. Tale dominio viene generato automaticamente ma è possibile interagire con esso anche per via programmatica. Come si vede il nostro Main ha un davvero un ruolo di una certa importanza... A proposito: è proprio Main() e non main(), errore che si fa spesso all'inizio, in quanto C# è un linguaggio **case-sensitive**, ovvero usare le maiuscole o le minuscole non è indifferente. Per questo motivo quindi è necessario scrivere in maniera assolutamente corretta i nomi di classi, variabili, namespace ecc... provate a compilare il programma precedente scrivendo, che so, `writeLine` piuttosto che `WriteLine` e osservate il risultato. A questo proposito in C# convenzionalmente abbiamo:

- le parole chiave sono minuscole (static, for o che altro)
- i tipi, i namespace e le classi iniziano con la maiuscola (System, Console)
- metodi e proprietà iniziano con la maiuscola così come eventuali parole mediane. es `WriteLine` o `ReadLine`

La riga 6 così come la 7, oltre a presentarci il metodo `WriteLine` che scrive una stringa (che è delimitata da una coppia di doppi apici) sullo standard output e va a capo, ci consente di capire come navigare nella gerarchia architeturale di .Net il che avviene attraverso il **.** (**punto**). Ovvero è possibile ad esempio arrivare ad un metodo o una proprietà di una classe in questo modo:

```
namespace.classe.metodo oppure
namespace.classe.proprietà
```

Questo passaggio è estremamente importante: è **il punto l'operatore che ci permette di navigare quindi nella gerarchia che dal namespace arriva al singolo metodo o alla singola proprietà**; in generale il punto segna il passaggio da qualcosa di più "grosso" di più "alto livello" a qualche cosa che in questo è contenuto con un significato più preciso, più limitato, meno generico insomma. Questo è ciò che avviene anche nel nostro semplice Hello World alla riga 6 ed alla 7. Nel caso in questione inoltre abbiamo usato la citata parola chiave `using` che mette a disposizione il namespace `System` senza doverlo invocare ogni volta, il che è senza dubbio comodo ed è una prassi consolidata. Tuttavia avremmo potuto anche farne a meno e scrivere il programma come segue, evidenziando tra l'altro il percorso di navigazione che parte dal namespace `System` ed arriva al metodo `WriteLine`, come abbiamo visto qualche riga qui sopra:

C#	Esempio 2.2
1	class test
2	{
3	public static void Main()
4	{
5	System.Console.WriteLine("##### Salutiamo #####");
6	System.Console.WriteLine("Hello, World!");
7	}
8	}

Ecco quindi spiegata l'utilità di using che ci consente invece di usare immediatamente le classi di System senza dover anteporre ogni volta il nome del namespace. In programmi complessi, come vedremo anche nel nostro piccolo, è necessario utilizzare più namespace del framework ed è molto comodo poter richiamare il loro contenuto per le vie brevi. Per completezza sull'utilizzo di using almeno in questa fase introduttiva, si può anche aggiungere il seguente esempio, un sistema utilizzato da alcuni programmatori per migliorare la leggibilità del codice (miglioramento opinabile a parer mio):

C#	Esempio 2.3
1	using sulVideo = System.Console;
2	class test
3	{
4	public static void Main()
5	{
6	sulVideo.WriteLine("##### Salutiamo #####");
7	sulVideo.WriteLine("Hello, World!");
8	}
9	}

Utilizzando quindi un sistema di alias. Come ho detto, per me, è inutile e forse anche controproducente, nel senso che non essendo convenzionale può rendere più difficoltosa la fase di manutenzione specie se chi vi deve provvedere non è l'estensore originale del codice, in linea di massima. Quanto meno ho sempre visto una pratica molto limitata di questa "finezza" e per lo più ridotta ad ambiti operativi chiusi dove, probabilmente, avevano studiato una nomenclatura interna. Per il resto ben poche altre volte.

Come si può vedere Main è preceduto da una serie di altre parole riservate evidentemente non messe lì a caso ma di esse riparleremo. Subito vicino ad esse però c'è quel **void** che merita qualche spiegazione. In effetti esso significa che il metodo in questione, qui è il Main ma potrebbe essere come vedremo qualunque altro, non presenta valori di ritorno quindi non restituisce nulla al chiamante ovvero a chi ne ha invocato l'esecuzione. Questo è un caso comune ma è altrettanto sovente necessario definire un valore di ritorno. A questo scopo possiamo ad esempio modificare l'esempio 2.1 di modo che presenti una risposta come:

C#	Esempio 2.4
1	using System;
2	class test
3	{
4	public static int Main()
5	{
6	Console.WriteLine("##### Salutiamo #####");
7	Console.WriteLine("Hello, World!");
8	return 0
9	}
10	}

dove hanno fatto la loro comparsa quel **int** (intero) prima del Main e quel **return** alla riga 8 che in pratica restituisce uno 0. L'utilità di restituire dei valori in programmi semplici come questo è assolutamente nulla mentre può essere necessaria laddove vi siano dei codici particolari che devono essere comunicati in uscita o meglio verso l'esterno nel caso di sistemi cooperanti più complessi. Va ricordato ad esempio che, nel mondo Windows, esiste la variabile di sistema %ERRORLEVEL% che può essere utilizzata per memorizzare un codice di uscita da un programma. Tipico ad esempio è l'uso di questa variabile in batch script (i famosi files .bat che tutti più o meno conosciamo).

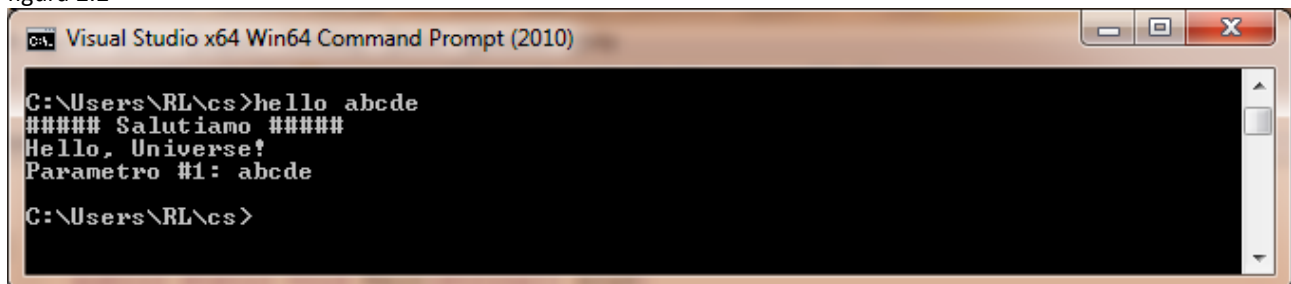
Ho parlato di “metodo”. Ma che roba è un “metodo”? Ne parleremo ovviamente più avanti (lo so che è un po’ ripetitivo questo “più avanti” ma siamo solo al capitolo introduttivo), per ora ci limitiamo a dire che è un blocco di codice rigidamente delimitato che contiene da 0 (caso limite, naturalmente, ma il compilatore lo permette) a  $n$  istruzioni con  $n$  grande a piacere, teoricamente, e che può essere richiamato attraverso il suo nome, sia pure con alcune regole da seguire; in generale è buona norma non esagerare con le dimensioni dei metodi per questioni di manutenibilità e leggibilità. Il nostro Main è un metodo come gli altri da un punto di vista costituzionale anche se, abbiamo visto, il suo ruolo è estremamente importante.

Torniamo appunto adesso a parlare proprio di Main al quale possono essere passati dei parametri direttamente dalla riga di comando. Per quanto in ambiti GUI questa cosa sia spesso bypassata essa esiste comunque ed è parecchio utile. Modifichiamo il nostro esempio 2.1 come segue, anche se non è necessario che tutto sia chiaro fin da adesso (ma si allarghiamo anche l’ambito del saluto ☺ ) già nel prossimo capitolo tante cose si chiariranno:

```
C# Esempio 2.5
1 using System;
2 class test
3 {
4     public static void Main(string[] args)
5     {
6         Console.WriteLine("##### Salutiamo #####");
7         Console.WriteLine("Hello, Universe!");
8         Console.WriteLine("Parametro #1: " + args[0]);
9     }
10 }
```

La riga 4 ci presenta dopo il Main la specifica di un array di stringhe che si chiama **args** (nome convenzionale, potreste metterci un altro nome a scelta); `args[0]` indica il primo elemento dell’array. Il tutto sarà più chiaro quando parleremo degli array e delle stringhe, per intanto ci basta sapere che, subito dopo aver richiamato il programma è possibile (nel caso specifico obbligatorio) mettere una stringa (che è una sequenza di caratteri alfanumerici) la quale sarà richiamata e stampata a video come richiesto alla riga 8. Si veda l’output di esempio:

figura 2.2



```
C:\Users\RL\cs>hello abcde
##### Salutiamo #####
Hello, Universe!
Parametro #1: abcde
C:\Users\RL\cs>
```

Come si vede la stringa, ovvero la sequenza di caratteri "abcde" riportata dopo aver richiamato il programma (dal poco originale nome `hello.exe`) viene stampata nella terza riga di output. Questa stringa è il parametro che viene memorizzato nella prima locazione disponibile dell’array `args` ovvero `args[0]`. Non vi preoccupate se non è chiaro, caso è utile tornare sopra questo esempio tra qualche tempo. L’assenza del parametro in questo programma causa un crash (come è facile testare) non gestibile con le pochissime cognizioni fin qui apprese. Da notare che abbiamo messo un parametro solo in questo caso ma possono essere  $n$  a piacere in quanto `args` è in grado di accettare un numero molto grande di valori.

Una domanda che possiamo a questo punto farci è: ma alla fine come è fatto un programma in C#? Da un punto di vista costituzionale esso, normalmente, si compone di una serie di classi cooperanti tra di loro che espongono metodi e proprietà (ovvero li rendono disponibili verso l’esterno). Una di queste classi deve prevedere l’entry point ovvero il nostro `Main()`. Insomma qualche cosa che, grossolanamente, somiglia alla seguente struttura:

## C# - Capitolo 2 – Hello, World

```
using.....

class 1
{
.....
}

class 2
{
.....
}

class 3
{
public static void Main()
.....
}

.....

class n
{
.....
}
```

E' possibile poi racchiudere tutto in un namespace o più namespace, volendo, a seconda di come vogliamo organizzare logicamente la nostra applicazione. Questi namespace, come detto all'inizio, hanno una funzione organizzativa logica molto importante e aiutano sia la leggibilità e la manutenzione del codice sia in un certo qual modo la sicurezza diminuendo la possibilità di collisioni di nomi in progetti di grandi dimensioni. I programmi presentati in questo paragrafo sono tutti mono classe.

In modo molto semplice possiamo modificare l'esempio iniziale come segue:

C#	Esempio 2.6
1	namespace sample01
2	{
3	using System;
4	class test
5	{
6	public static void Main()
7	{
8	Console.WriteLine("##### First C# Program #####");
9	Console.WriteLine("Hello, Universe");
10	}
11	}
12	}

Il codice è racchiuso all'interno del namespace, anche se in questo caso è del tutto superfluo. In un programma possono essere definiti quanti namespace vogliamo, dipende solo da noi ovviamente in base a una logica. Già che ci siamo introduciamo una prima semplice variante sul tema mostrando un modo diverso di usare Console.Write e Console.WriteLine, ovvero in via parametrica.

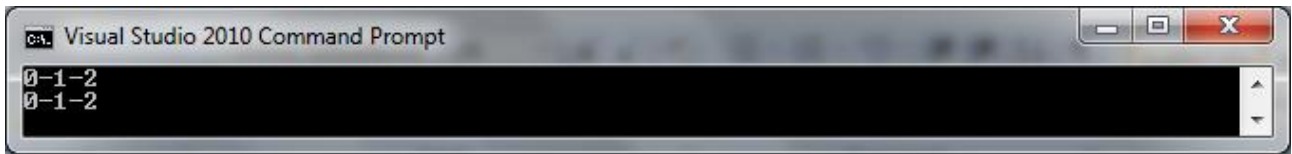
C#	Esempio 2.7
1	using System;
2	class program
3	{
4	public static void Main()
5	{
6	int x = 0;
7	int y = 1;
8	int z = 2;
9	Console.WriteLine(x + "-" + y + "-" + z);
10	Console.WriteLine("{0}-{1}-{2}", x, y, z);

```

11     }
12 }

```

L'output è il seguente:



```

Visual Studio 2010 Command Prompt
0-1-2
0-1-2

```

Ovvero la riga 9 e la 10 stampano in modo identico. Se la riga 9 è, tutto sommato, in linea con quanto già visto, la 10 ci presenta, racchiusi tra parentesi graffe, 3 parametri ai quali, dopo la chiusura della stringa, corrispondono 3 variabili, x, y e z. Esse si collocano in ordine così' come sono posizionati i parametri all'interno della stringa, compresa nella coppia dei doppi apici. Il primo parametro deve essere sempre 0, gli altri a seguire in sequenza aritmetica. Questa è già una particolarità, la parametrizzazione accettata in forma "variabile", per quanto riguarda il numero degli elementi da trattare, da Console.WriteLine, che discuteremo in generale nel capitolo 6.

Per chiudere questo paragrafo del tutto introduttivo vediamo un modo di salutarci un po' più interattivo e gradevole, visto che non si vive di solo command prompt e intanto vediamo all'operano i commenti che qui troviamo all'inizio del codice e alla riga 14. La riga 13 ci consente invece di catturare una stringa in input memorizzandola nella variabile nome (= è il simbolo di assegnazione ed attribuisce alla variabile alla sua sinistra il valore di ciò che si trova a destra purchè siano soddisfatti quei criteri, li vedremo, che rendono possibile l'attribuzione).

```

C#   Esempio 2.8
1   /*
2   Siamo nel mondo delle finestre
3   o no???
4   */
5
6   using System;
7   using System.Windows.Forms;
8   class test
9   {
10  public static void Main(string[] args)
11  {
12      Console.Write("Scrivi il tuo nome: ");
13      string nome = Console.ReadLine();
14      MessageBox.Show("Hello, " + nome); // la nostra prima finestra!!
15  }
16  }

```

Ovviamente la programmazione specifica di forms e relativa alla GUI la vedremo molto più avanti anche perchè si preferisce un approccio più moderno a quello basato sui forms da quando è entrata in gioco una sigla nuova da qualche anno ovvero WPF col suo linguaggio a marcatori XAML. Ne parleremo perchè entrambi questi nuovi attori sono ottimi amici di C#.

Abbiamo toccato vari argomenti in questo paragrafo, abbiamo parlato di stringhe, di classi ecc. Ovviamente tante cose non sono per nulla chiare, lo so. Vedremo tutto con calma per ora basta focalizzarsi sui concetti immediati e in particolare:

- I namespace
- la natura a classi dei programmi C#
- la navigazione gerarchica tra le varie entità in gioco, estremamente importante e per la quale bisognerà subito "fare l'occhio"
- il fatto che si tratta di un linguaggio case-sensitive
- le poche keyword che abbiamo presentato
- qualche concetto come i metodi, le classi stesse

## C# - Capitolo 2 – Hello, World

oltre naturalmente a prendere un po' di confidenza con la sintassi. Questi sono i pochi punti chiave su cui concentrarsi. Di tutto il resto parleremo ampiamente poco alla volta cominciando, nel prossimo paragrafo, a parlare delle variabili e dei tipi.

Concludiamo questo capitolo con l'elenco della parole riservate in C#. Una tabella forse un po' arida, certamente astrusa a questo punto, ma che varrà forse la pena riguardare man mano che ci procede nello studio per vedere che cosa si conosce e cosa manca ancora.

abstract	event	New	struct
as	explicit	Null	switch
base	extern	Object	this
bool	false	Operator	throw
break	finally	Out	true
byte	fixed	Override	try
case	float	Params	typeof
catch	for	Private	uint
char	foreach	Protected	ulong
checked	goto	Public	unchecked
class	if	ReadOnly	unsafe
const	implicit	Ref	ushort
continue	in	Return	using
decimal	int	Sbyte	var (non sempre)
default	interface	Sealed	virtual
delegate	internal	Short	volatile
do	is	Sizeof	void
double	lock	Stackalloc	while
else	long	Static	
enum	namespace	String	

NB: var non sempre è usato come keyword, come vedremo, quindi non sarebbe tale in senso stretto.