# I tipi di dati base e passaggio dei parametri

I tipi di dati base derivano direttamente dai tipi di dato di Java, dei quali conservano anche il nome, anche la conversione tra tipi di dato, il cosiddetto «casting» funziona alla maniera di Java. Ci sono invece **novità introdotte da C#** per quanto riguarda i tipi di dato definiti dall'utente: le struct e le enum.
Il **tipo struct** permette di definire nuovi tipi di dati a partire da quelli esistenti:

```
public struct Persona
{
  public string Nome;
  public string Cognome;
}
```

Questa dichiarazione definisce un tipo di dato Persona composto da due variabili di tipo stringa (i cosiddetti membri della struttura). Notiamo che, a differenza del C++, da cui deriva il concetto di struct, ogni membro deve avere un **modificatore di accesso** (public, private, ecc.).
Questo perché il tipo struct è molto potente: può essere dotato di un **costruttore** (che si comporta esattamente come il costruttore di una classe), di proprietà e di metodi. Modifichiamo la struttura Persona aggiungendovi un costruttore e una proprietà che restituisce il nome completo, quindi riprendiamo il nostro primo programma ed aggiorniamolo per mostrare l'uso della **struct**:

```
public class HelloWorld
{
  public struct Persona
  {
    public string Nome;
    public string Cognome;

    public Persona(string Nome, string Cognome)
    {
      this.Nome = Nome;
      this.Cognome = Cognome;
    }

    public string NomeCompleto
    {
```

```
      get {return Nome + " " + Cognome;}
    }
   }

   public static void Main()
   {
     Persona p = new Persona("Marco, "Minerva");
     System.Console.WriteLine("Ciao " + p.NomeCompleto + "!");
     System.Console.ReadLine);
   }
 }
```

All'interno del metodo Main la struttura Persona viene utilizzata come se si trattasse di una classe. Notiamo che, invece di usare il costruttore, avremmo potuto scrivere:

```
Persona p;
p.Nome = "Marco";
p.Cognome = "Minerva";
```

Ottenendo lo stesso risultato, ovvero la stampa del messaggio "Ciao Marco Minerva!". Questo è possibile perché i membri Nome e Cognome della struttura sono stati dichiarati public; se, invece, fossero stati definiti private, non sarebbe stato consentito utilizzare le tre istruzioni sopra riportate (si sarebbe ottenuto un errore in fase di compilazione, come già spiegato a proposito delle classi), e l'unico modo lecito di impostare i membri Nome e Cognome sarebbe stato quello di utilizzare il costruttore.
A questo punto è naturale chiedersi quali siano le **differenze tra struct e class**. La spiegazione è fornita dalla guida in linea: "Il tipo struct è adatto alla rappresentazione di oggetti leggeri […] Se ad esempio si dichiara una matrice di 1.000 oggetti [classi] Point, si allocherà ulteriore memoria per i riferimenti a ciascun oggetto. In questo caso la struttura risulta meno onerosa". Inoltre una struct può implementare interfacce ma non può ereditare.

Le **enum** consentono di associare nomi simbolici a costanti numeriche. Il costrutto deriva dal C++ dove i membri delle enum sono sempre di tipo «int», mentre in C# è possibile indicarne esplicitamente il tipo:

```
public enum Stati : byte
{
  Italia, America, Francia, Inghilterra //,…
}
```

Questa dichiarazione definisce una enumerazione i cui membri sono di tipo byte. Richiamando il **metodo ToString()** di una variabile di tipo enum, è possibile ottenere la stringa contenente il nome simbolico del valore corrente. Ad esempio:

```
Stati s = Stati.America;
System.Console.WriteLine("Stato selezionato: " + s.ToString());

System.Console.WriteLine((int)s); // La stampa a video sarebbe = 1
```

Questo codice stampa a video il messaggio "Stato selezionato: America".

Concludiamo questa Lezione con qualche cenno sul **passaggio dei parametri** ai metodi. I parametri in C# possono essere passati in tre modi: in, out e ref.
Queste parole chiave determinano cosa accade alla variabile passata come argomento quando si esce dal metodo.

- **in** è il tipo di passaggio predefinito: la routine non può modificare la variabile, o, meglio, qualunque modifica della variabile interna al metodo viene persa quando si esce dal suo scope (il tipico passaggio "per valore").
- **ref** e **out** sono i classici passaggi di parametro per riferimento. Viene passato il riferimento (il puntatore) alla variabile e non il suo valore, così anche nello scope del metodo si fa riferimento allo stesso oggetto in memoria.
  La **differenza tra ref e out** è che ref richiede che si inizializzi la variabile prima della chiamata del metodo, mentre con out è sufficiente dichiarare la variabile e passarla senza inizializzazione. In pratica con out è possibile inizializzare intere strutture dati dall'interno di un metodo.

Vediamo ora alcuni semplici esempi per chiarire quanto illustrato:

```
// Se non diversamente specificato il parametro è di tipo <<in>>

public void Calcola(int N) {N = 10;}

public void CalcolaOut(out int N) {N = 10;}

public void CalcolaRef(ref int N) {N += 1;}
```

Un possibile utilizzo di questi metodi è il seguente:

```
int i = 7;
Calcola(i);
```

//All'uscita, i vale ancora 7

```
int i = 7;
CalcolaRef(ref i);
```

//All'uscita, i vale i + 1 = 8

```
// i è solo dichiarata non inizializzata
int i;
```

// la passiamo con «out»

```
CalcolaOut(out i);
```

//All'uscita, i vale 10

Osserviamo che, per utilizzare un parametro out oppure ref nella chiamata ad un metodo, è necessario indicare esplicitamente che si tratta di un argomento passato out o ref.