C#

Alice Colella

Junior Developer @icubed

Alice.Colella@icubed.it

Tipi di dato

- **Value Type**
  - contengono il dato direttamente.
  - Sono memorizzati all’interno della memoria detta **stack**

- **Reference Type**
  - Contengono un link al valore
  - Sono memorizzati all’interno di un’area denominate **heap**
  - Vengono creati utilizzando la parola chiave new()

Tipi di dato

- **Value Type**
  - NON possono essere null
  - Vengono rimossi dalla memoria quando il loro scope termina
- **Reference Type**
  - Possono essere null
  - Garbage collector per liberare risorse

Value Type predefiniti

Integer Types

NAME	.NET TYPE	DESCRIPTION	RANGE (MIN:MAX)
sbyte	System.SByte	8-bit signed integer	-128:127 (-2^7:2^7–1)
short	System.Int16	16-bit signed integer	-32,768:32,767 (-2^{15}:2^{15}–1)
int	System.Int32	32-bit signed integer	-2,147,483,648:2,147,483,647 (-2^{31}:2^{31}–1)
long	System.Int64	64-bit signed integer	-9,223,372,036,854,775,808: 9,223,372,036,854,775,807 (-2^{63}:2^{63}–1)
byte	System.Byte	8-bit unsigned integer	0:255 (0:2^8–1)
ushort	System.UInt16	16-bit unsigned integer	0:65,535 (0:2^{16}–1)
uint	System.UInt32	32-bit unsigned integer	0:4,294,967,295 (0:2^{32}–1)
ulong	System.UInt64	64-bit unsigned integer	0:18,446,744,073,709,551,615 (0:2^{64}–1)

Value Type predefiniti

Floating point Types

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
`float`	`System.Single`	32-bit, single-precision floating point	7	$\pm 1.5 \times 10^{245}$ to $\pm 3.4 \times 10^{38}$
`double`	`System.Double`	64-bit, double-precision floating point	15/16	$\pm 5.0 \times 10^{2324}$ to $\pm 1.7 \times 10^{308}$

Value Type predefiniti

Decimal point Types

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
`decimal`	`System.Decimal`	128-bit, high-precision decimal notation	28	$\pm 1.0 \times 10^{228}$ to $\pm 7.9 \times 10^{28}$

Value Type predefiniti

Boolean Types

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE
bool	System.Boolean	Represents true or false	NA	true or false

Character Type

NAME	.NET TYPE	VALUES
char	System.Char	Represents a single 16-bit (Unicode) character

Reference Types predefiniti

Due tipologie:

- **Object - System.Object**
  - È l'oggetto da cui tutti gli altri oggetti **ereditano**
  - Prevede un **numero limitato di metodi** (comuni a tutte le classi):
    - Equals
    - GetHashCode
    - GetType
    - ToString
- **String - System.String**

Reference Types predefiniti

String Type

- Sono definiti mediante l'utilizzo della **keyword string**
- Un oggetto stringa viene allocato **all'interno dell'heap** e **NON nello stack**

Reference Types predefiniti

Comportamento inusuale

```
string s1 = "a string";
string s2 = s1;
Console.WriteLine("s1 is " + s1);
Console.WriteLine("s2 is " + s2);
s1 = "another string";
Console.WriteLine("s1 is now " + s1);
Console.WriteLine("s2 is now " + s2);
```

La variazione di s1 produce la **generazione di un nuovo oggetto string**. Viene allocato un nuovo oggetto string nell'heap

Reference Types predefiniti

- In C# 6 è possibile utilizzare il **carattere $** per effettuare la sostituzione all'interno di stringhe

```
$"Hello!  My name is {person.FirstName} and I am years old.";
```

- E' un **metodo rapido**, rispetto all'utilizzo di String.Format(…)

Inizializzazione di variabili

Il compilatore richiede che ciascuna variabile sia **inizializzata prima di poter essere utilizzata**

C# implementa **due metodi** per inizializzare le variabili:

- **Campi di classi o strutture**, vengono azzerate (se non valorizzate direttamente)
- **Le variabili** che all'interno di un metodo devono essere inizializzate in maniera esplicita prima di poter essere utilizzate. Il compilatore verifica dove le variabili sono utilizzate, e l'eventuale mancanza di inizializzazione.

Inizializzazione di variabili

Type inference

- Viene effettuata utilizzando la **parola chiave var**
- Il compilatore “capisce” il **tipo delle variabili** in base al valore con cui viene inizializzata
  - Necessario che le variabili siano inizializzate
  - L’inizializzazione non può essere nulla

Enumerazioni

- Un’enumerazione è un tipo definito dall’utente **basato su interi**
- Un’enumerazione viene utilizzata perchè **rende il codice più semplice da mantenere**
- Accediamo ad un intero **tramite il suo nome**
- Pieno supporto di **Intellisense** (Visual Studio)

- **Limite**: il valore è di tipo intero. Possibile estendere le enumerazioni con attribuiti per ottenere anche il nome

Enumerazioni

- Possiamo **accedere ad un valore** utilizzando:

TimeOfDay.Morning

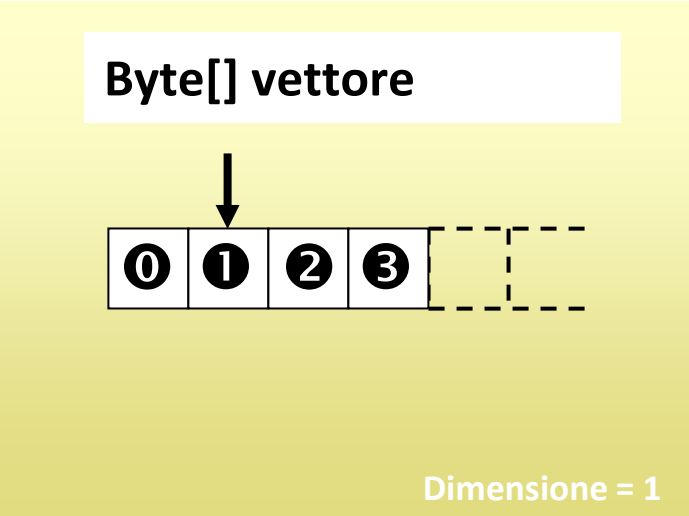
```
public enum TimeOfDay
{
  Morning = 0,
  Afternoon = 1,
  Evening = 2
}
```

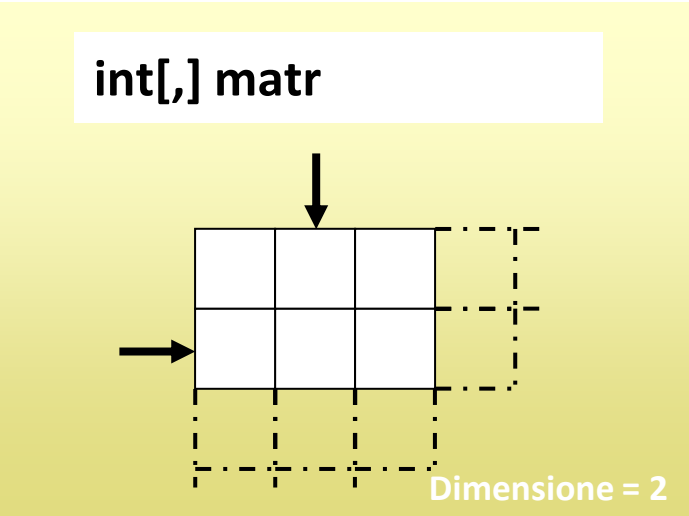
Array

- Gli *array* memorizzano delle **sequenze finite** di elementi, la cui lunghezza deve essere nota a priori
- Gli elementi vengono richiamati in base ad un indice numerico che parte sempre da **zero**
- Gli array non sono direttamente ridimensionabili

Array multidimensionali

- Il metodo **GetLength** permette di conoscere il rank della n-esima dimensione dell'array.






```
Byte[] vettore = new Byte[5];
Byte[] vettore = new Byte[] {1, 2, 3, 4, 5}
```

```
int[,] matrice = new int[2, 3]
int[,] matrice = new int[,] {{1, 2, 3}, {4, 5, 6}}
```

Demo

Lavorare con i tipi

I metodi

Definizione di un **metodo**:

```
[modifiers] return_type MethodName([parameters])
{
  // Method body
}
```

- Possibilità di effettuare **overloading sulla chiamata** del metodo

Overloading di metodi e proprietà

- Possono esistere metodi e proprietà con lo stesso nome.
- È possibile perché il vero “nome” è rappresentato dalla **firma**: nome, numero e tipi dei parametri, ivi inclusi i modificatori come **ref** o **out**.
- Non possono esistere due metodi che differiscono del solo parametro di ritorno (non fa parte della firma).

```
public int Sum(int a, int b) {
    return a + b;
}
```

```
public decimal Sum(decimal a, decimal b) {
    return a + b;
}
```

```
public int Sum(int a, int b, int c) {
    return a + b + c;
}
```

```
public decimal Sum(Decimal a,  Decimal b,  Decimal c) {
    return a + b + c;
}
```

Passaggio parametri ad un metodo

Il passaggio dati ad un metodo può avvenire:

- Per **valore**: passaggio dati di default
- Per **riferimento**: viene utilizzata la parola chiave **ref**

Attenzione alla **keyword out**

Passaggio parametri ad un metodo

1) Valore predefinito di un parametro (parametro opzionale):

```
public void Prova(int nonOpzionale, int opzionale = 32)
```

2) Numero variabile di parametri:

```
public void ParametriVariabili(params int[] data)
```

Valori di ritorno da un metodo

Un metodo può ritornare solo **un valore**

Possiamo estenderne il comportamento per **ritornare più valori**:

1) Ritornando un **classe/struttura** con tutti i valori necessari
2) Utilizzare **tuple**
3) Utilizzare la parola chiave **out**

Valori di ritorno da un metodo

Esempio di possibile applicazione di out

Utilizzo della **funzione TryParse** la cui firma è la seguente:

```
public static bool TryParse(string s, out int result);
```

Ritorna un **bool** se la conversion stringa-> intero è andata a **buon fine**. Ritorna il **valore dell'intero** nella variabile result.

Le tuple

Mentre un array contiene elementi dello stesso tipo, **le tuple possono contenere oggetti di tipo diverso**

- Sono disponibili in tutti i linguaggi del framework .NET
- Classe Tuple: i parametri sono di tipo generic

```
var tuple = Tuple.Create<string,string,string,string,int,int,int>
```

E’ possibile utilizzare fino ad 8 parametri. Eventualmente si può estendere utilizzando una nuova tupla come parametro

```
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

Programmazione Procedurale

Organizzazione e suddivisione del codice in funzioni e procedure.

- Un’operazione è corrispondente a una routine, che accetta parametri iniziali e che produce eventualmente un risultato.

- Separazione tra logica applicativa e dati

Object Oriented Programming

OOP – Object Oriented Programming

- È un paradigma di programmazione

- Si basa sulla definizione e uso di diverse entità, collegate e interagenti, caratterizzate da un'insieme di informazioni di stato e di comportamenti
- Tali entità vengono denominate *Oggetti*

Oggetti

Gli oggetti possono contenere:

- Dati
- Funzioni
- Procedure

Funzioni e procedure possono sfruttare lo stato dell’oggetto per ricavare informazioni utili per la rispettiva elaborazione.

Classi

- Una classe può contenere ed eventualmente esporre sulla sua interfaccia:
  - Dati (**campi** e **proprietà**)
  - Funzioni (**metodi**)

- Una classe può essere **Partial**, ossia può essere definita in file differenti (versione 2.0)

```
// File part1.cs
partial public class MyClass {
    //...
}
```

```
// File part2.cs
partial public class MyClass {
    //...
}
```

Tipi, classi e oggetti

- Un **tipo** è una rappresentazione concreta di un concetto. Per esempio, il tipo built-in *float* fornisce una rappresentazione concreta di un numero reale. (*)

- Una **classe** è un tipo definito dall'utente. (*)

- Un **oggetto** è l'istanza di una classe caratterizzato da:
  - un'identità (distinto dagli altri);
  - un comportamento (compie elaborazioni tramite i metodi);
  - uno stato (memorizza dati tramite campi e proprietà).

```
MyClass c = new MyClass();

public class MyClass {
    //...
}
```

(*) The C++ Programming Language, Third Edition. Bjarne Stroustrup

icubed

Accessibilità

- I tipi definiti dall'utente (classi, strutture, enum) e i membri di classi e strutture (campi, proprietà e metodi) possono avere accessibilità diversa (***accessor modifier***):
  - **public** — Accessibile da tutte le classi
  - **protected** — Accessibile solo dalle classi derivate
  - **private** — Non accessibile dall'esterno
  - **internal** — Accessibile all'interno dell'assembly
  - **internal protected** — Combinazione delle due
- Differenziare l'accessibilità di un membro è fondamentale per realizzare l'*incapsulamento*.
- **L'insieme dei membri esposti da un classe rappresenta la sua *interfaccia*.**

Classi e proprietà

- È il modo migliore per soddisfare uno dei pilastri della programmazione OOP: *incapsulamento*.
- Una proprietà può provvedere accessibilità in lettura (**get**) scrittura (**set**) o entrambi.
- Si può usare una proprietà per ritornare valori calcolati o eseguire una validazione.

Classi e proprietà

Proprietà tradizionale

```
public class MyClass
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

MyClass c = new MyClass();
c.Name = "C#";
```

ReadOnly / WriteOnly

```
public class MyClass
{
    private string _name = "C#";

    public string Name
    {
        get { return _name; }
    }
}

MyClass c = new MyClass();
c.Name = "C#"; // non si può fare
Console.WriteLine(c.Name); // si può fare
```

Dati statici e di istanza 1/2

- I dati relativi ad una classe sono marcati con la keyword **static** e descrivono le informazioni comuni a tutti gli oggetti dello stesso tipo.
- Ciò che è marcato **static** può essere utilizzato senza la necessità di istanziare oggetti.

```
public class MyClass {
    public static int One;
    public int Two;
}

MyClass c1 = New MyClass();
MyClass c2 = New MyClass();

MyClass.One = 3;
c1.Two = 5;
c2.Two = 7;
```

I metodi

- Particolari tipi di metodi: **Costruttori delle Classi**
- Differenze tra metodi statici e non
- Possibilità di **richiamare costruttori da altri costruttori** (interni alla stessa classe)

```
public Costruttore1(string descrizione, int valore)
public Costruttore2(string descrizione)
public Costruttore3(string descrizione) : this(descrizione, 4)
```

I metodi all’interno delle classi

Pattern Injection

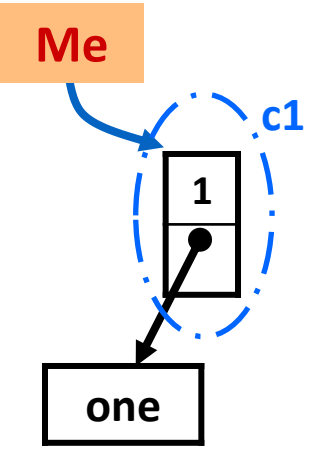
- Dipendenza tra classi
- Utilizzo di Interface per la separazione delle dipendenze
- Utilizzo di framework specifici (es. Ninject)

Keyword this

- **this** è un riferimento che punta all'istanza della classe stessa.
- È usabile solo in relazione ai membri non statici.

```
public class MyClass {
    int one;

    public void MyMethod(int one) {
        this.one = one;
    }
}
```



Costruttore

- Il costruttore è fondamentale perché permette di dare all'oggetto uno stato iniziale stabile e congruo.
- Il costruttore di default non ha argomenti.

```
public class Person {
    private string _name;
    private int _age;

    public Person(string name, int age) {
        _name = name;
        _age = age;
    }
}
```

Classi annidate

- Le classe annidata è semplicemente un tipo definito all'interno di un'altra classe.

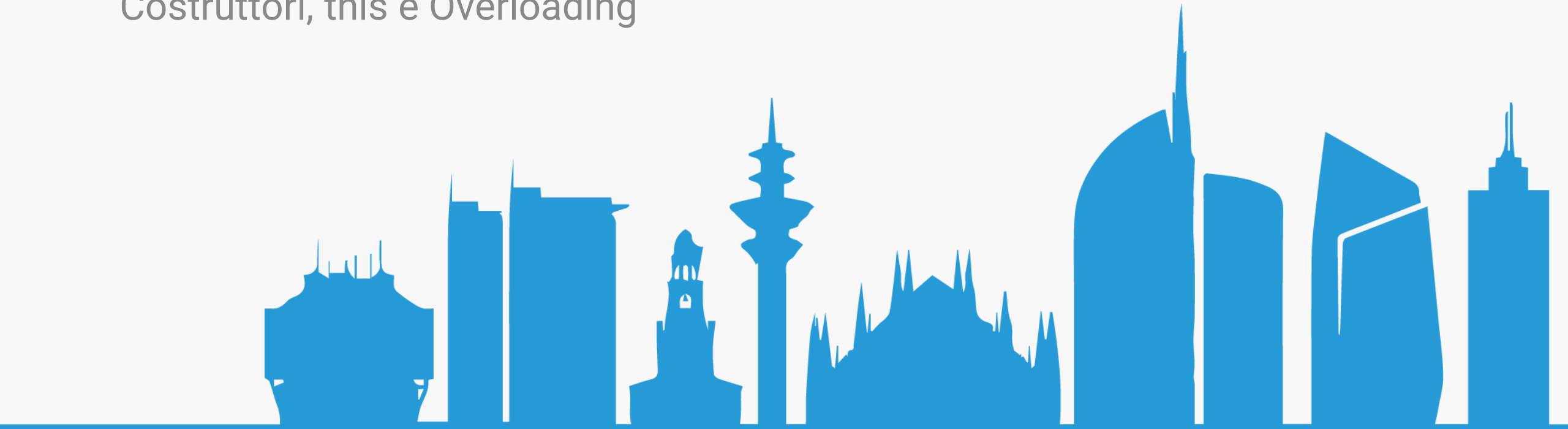
```
public class MyClass {
    public class MyNestedClass {
        //...
    }
}

// occorre specificare il nome della classe container
MyClass.MyNestedClass nested = New MyClass.MyNestedClass();
```

Demo

Classi

Costruttori, this e Overloading



Strutture

- Quanto visto finora per le classi è tipicamente valido anche per la definizione di **strutture** (keyword **struct**).
- Il compilatore genera sempre un costruttore di default che inizializza tutti i membri della struttura al valore di default.
- È possibile dichiarare unicamente costruttori con parametri.

```
struct MyStructure {
    public MyStructure(string a) {
        ValueOne = int.Parse(a);
        ValueTwo = false;
    }

    public int ValueOne;
    public bool ValueTwo;
}
```

Classi vs Strutture

Classi	Strutture
Possono definire data member, proprietà, metodi.	Possono definire data member, proprietà, metodi.
Supportano costruttori e l'inizializzazione dei membri.	Non supportano costruttori di default e l'inizializzazione dei membri.
Supportano il metodo **Finalize**.	Non supportano il metodo **Finalize**.
Supportano l'ereditarietà.	Non supportano l'ereditarietà.
È un Reference Type.	È un Value Type.

Demo

Classi e Strutture

Anonymous types

- Utilizzando la Type Inference possiamo utilizzare la parola chiave var per **definire una variabile senza dichiararne il tipo**
- Un **anonymous type** è semplicemente una classe che eredita direttamente da object
- La **definizione di una classe** avviene durante la sua dichiarazione

Anonymous types

La classe viene definita **senza specificarne** il tipo:

```
var captain = new
{
  FirstName = "James",
  MiddleName = "T",
  LastName = "Kirk"
};
```

```
var doctor = new
{
  FirstName = "Leonard",
  MiddleName = string.Empty,
  LastName = "McCoy"
};
```

Sono ammesse **operazioni di assegnazione** del tipo:

```
captain = doctor
```

Ma solo quando **il tipo** delle proprietà **corrisponde**

Demo

Anonymous Type

Nullable

- Le variabili di tipo Type Reference **possono essere** null
- Le variabili di tipo Value **non possono essere** null

- Problema tipico: mappatura tra classi e tabelle del database
  - I valori nel db possono essere **null**
  - I valori dei campi delle classi non lo possono essere (es. Int, double)

Nullable

- Con i reference types è necessario il Garbage Collector per **liberare la memoria**, mentre con i tipi a valore questa operazione non è necessaria

- I tipi a valore vengono liberati dalla memoria non appena **non sono più all'interno del loro scope**

Nullable

- In C# possiamo utilizzare I **tipi Nullable**
- Come dice il nome, sono tipi che **possono essere null**
- E' sufficiente utilizzare il **carattere "?"** dopo un tipo per renderlo nullabile:

```
int x1 = 1; // intero "tradizionale"
int? x2 = null; // intero "nullabile"
```

- L'assegnazione x1 a x2 non genera problemi
- L'assegnazione da x2 a x1 non è possibile. E' necessario il cast:

```
X1 = (int) X2
```

Nullable

- Il tipo nullabile ha anche associato il **metodo .HasValue()** che ritorna true o false a seconda che il tipo abbiamo un valore o meno.
- Nella **proprietà .Value** possiamo recuperare il valore:

```
int x5 = x3.HasValue ? x3.Value : -1;
```

Ma anche:

```
int x6 = x3 ?? -1;
```

Demo

Nullable