


# Academy Week 3



# Week 2 - Agenda

- OOP con C#
  - Classi
    - Proprietà e Metodi
    - Costruttori e Finalizzatori
    - Eventi
    - Interfacce
  - Overloading
  - Abstract e Static class
  - Anonymous types
  - Ereditarietà e override
- Generics
-  Esercitazione



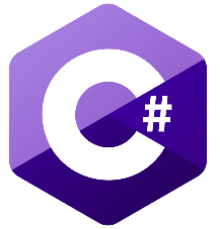
# Convenzioni sul codice

- **Notazione ungherese:** al nome dell'identificatore viene aggiunto un prefisso che ne indica il tipo (es. `intNumber` identifica una variabile intera)
- **Notazione Pascal:** l'inizio di ogni parola che compone il nome dell'identificatore è maiuscola, mentre tutte le altre lettere sono minuscole (es. `FullName`)
- **Notazione Camel:** come la notazione Pascal, a differenza del fatto che la prima iniziale deve essere minuscola (es. `fullName`)

# Convenzioni sul codice

Elemento/i	Notazione
Namespace	Notazione Pascal
Classi	Notazione Pascal
Interfacce	Notazione Pascal
Strutture	Notazione Pascal
Enumerazioni	Notazione Pascal
Campi privati	Notazione Camel, eventualmente preceduta dal carattere di sottolineatura (esempio: <code>_fullName</code> )
Proprietà, metodi ed eventi	Notazione Pascal Parametri dei metodi e delle funzioni in Camel
generale	Notazione Camel
Variabili locali	Notazione Camel

# Classi



Una classe è come un costruttore di oggetti o un "blueprint" per la creazione di oggetti.

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

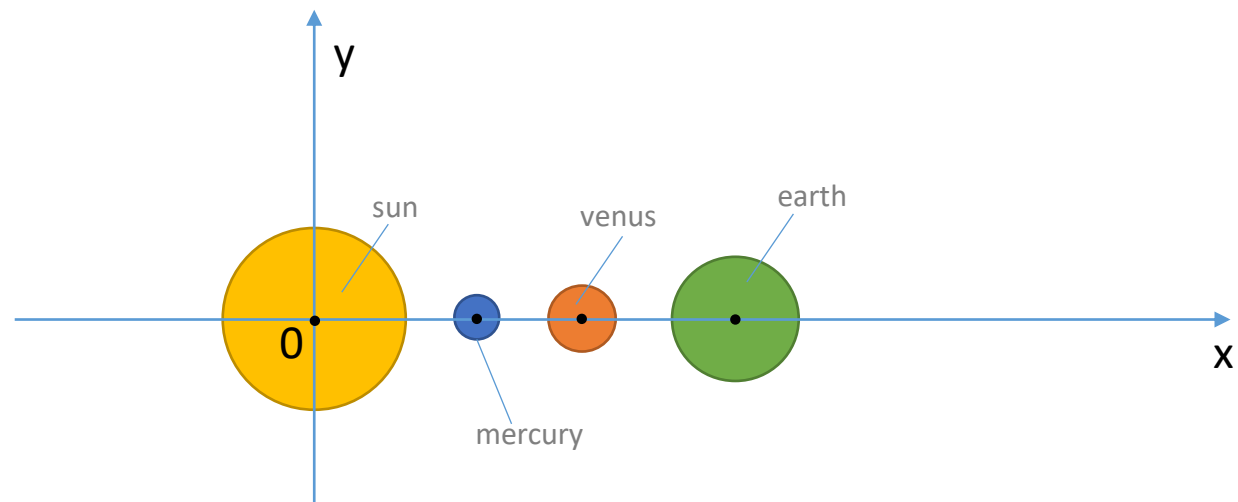
Una classe può contenere ed eventualmente esporre una sua interfaccia:

- Dati (**campi** e **proprietà**)
- Funzioni (**metodi**)

# Esercitazione – Solar System

- Implementare un'applicazione console che modelli un sistema solare, leggendo i diversi pianeti da un file di testo correttamente formattato
- Formato: NOME | MASSA | RAGGIO | DISTANZASOLE
- I vari pianeti devono essere letti dal file e inseriti in una lista/array
- Calcolare la forza di gravità di un pianeta rispetto un altro pianeta
- Calcolare la forza di gravità del sole subita da tutti gli altri pianeti
- Semplificazione:
  - Il file rappresenta un'istantanea del sistema solare, in cui tutti i Pianeti sono allineati lungo uno stesso asse.







# La classe Object

Tutto in .NET deriva dalla **classe Object**

- Se non specifichiamo una classe da cui ereditare, il compilatore assume automaticamente che stiamo ereditando da Object

## System.Object

- Tutto ciò che deriva da Object **ne eredita anche i metodi**
- Questi metodi sono disponibili **per tutte le classi** che definiamo

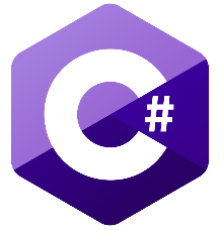




# La classe Object

- **ToString:** converte l'oggetto in una stringa
- **GetHashCode:** ottiene il codice hash dell'oggetto
- **Equals:** permette di effettuare la comparazione tra oggetti
- **Finalize:** chiamato in fase di cancellazione da parte del garbage collector
- **GetType:** ottiene il tipo dell'oggetto
- **MemberwiseClone:** effettua la copia dell'oggetto e ritorna una reference alla copia

# Classi, campi e proprietà

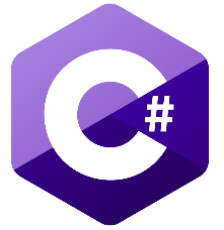


## Campo

```
public class MyClass
{
    public string Name;
}

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

# Classi e proprietà



## Proprietà

```
public class MyClass
{
    public string Name { get; set; }
}

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

# Classi e proprietà



## Proprietà 'condensata'

```
public class MyClass
{
    public string Name { get; set; }
}

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

## Proprietà tradizionale

```
public class MyClass
{
    private string _name;

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

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

# Classi e proprietà



## Proprietà in sola Lettura

```
public class MyClass
{
    public string Name { get; }
}

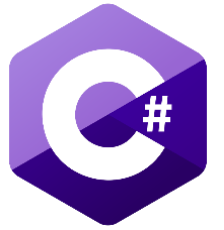
MyClass c = new MyClass();
c.Name = "C#"; // KO
Console.WriteLine(c.Name); // OK
```

## Proprietà in sola Scrittura

```
public class MyClass
{
    public string Name { set; }
}

MyClass c = new MyClass();
c.Name = "C#"; // OK
Console.WriteLine(c.Name); // KO
```

# Classi e proprietà



## Proprietà calcolata

```
public class MyClass
{
    public string FullName {
        get { return $"{FirstName} {LastName}"; }
    }
}

MyClass c = new MyClass();
c.FullName = "C#"; // KO
Console.WriteLine(c.FullName); // OK
```



# I metodi

Sono **funzioni associate** ad una particolare classe

Possibilità di associare **modificatori di accesso** (`public`, `private` ...)

Definizione di un **metodo**:

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

# 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



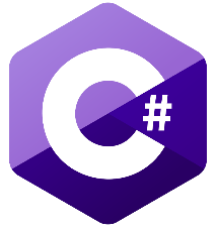
- Valore predefinito di un parametro (parametro opzionale):

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

- Numero variabile di parametri:

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

# Passaggio parametri ad un metodo



Dichiarazione del parametro di tipo array di interi

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

## Utilizzo della keyword params

La keyword params deve essere **sempre utilizzata**

ParametriVariabili(1) ma anche ParametriVariabili(1,2,3,4)

Posso anche utilizzare tipi diversi, dichiarando  
params object[] 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** (C# 7.0)
- 3) Utilizzare la parola chiave **out**



# Valori di ritorno da un metodo

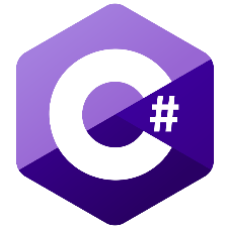
## Esempio di possibile applicazione di out

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

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

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

# Overloading di un metodo



L'overloading mi permette di utilizzare lo stesso nome per un metodo, purché il numero e / o il tipo di parametri siano diversi.

```
void MyMethod(string str)
{
    // ...
}
```

```
int MyMethod(string str, int val)
{
    // ...
}
```

# Esercitazione 1

- All'interno di un Progetto Class Library, creare una classe `ComplexNumber` per gestire i numeri complessi.

$$a + ib$$

- La classe conterrà i seguenti membri:
  - un costruttore con 2 parametri (parte reale ed immaginaria)
  - Le proprietà Parte Reale e Parte Immaginaria
  - Le proprietà calcolate Modulo e Coniugato
  - I metodi per le 4 operazioni aritmetiche fondamentali tra 2 numeri complessi
- Realizzare una Console app di test che
  - Richieda di inserire due numeri complessi (inserire distintamente parte reale e parte complessa)
  - Richieda di inserire una operazione da effettuare (+, -, \*, /) e calcoli il risultato utilizzando la libreria realizzata al punto precedente



# Numeri complessi – Proprietà

$$(a + ib) + (c + id) = (a + c) + (b + d)i$$

$$(a + ib) - (c + id) = (a - c) + (b - d)i$$

$$(a + ib) \cdot (c + id) = (ac - bd) + (ad + bc)i$$

$$\frac{a + ib}{c + id} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

$$\text{Coniugato: } z = a + ib \rightarrow \bar{z} = a - ib$$

$$\text{Modulo: } z = a + ib \rightarrow |z| = \sqrt{a^2 + b^2}$$