OOP con C#

Alessandra Degan Di Dieco

Analyst@icubedsrl

alessandra@icubed.it

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*

Object Oriented Programming

Principi fondamentali della programmazione ad oggetti

- Incapsulamento

  Principio di base per cui una classe può mascherare la struttura interna e proibire ad altri oggetti di accedere ai suoi dati o le sue funzioni che non siano direttamente accessibili dall'esterno

- Ereditarietà

  Si basa sul legame di dipendenza di tipo gerarchico tra classi diverse. Una classe deriva da un'altra se ne eredita il comportamento.

- Polimorfismo

  Il polimorfismo rappresenta il principio in funzione del quale diverse classi derivate possono implementare uno stesso comportamento definito da una classe base in modo differente

- Astrazione

  Modellazione degli attributi e delle interazioni delle entità come classi per definire una rappresentazione astratta di un sistema.

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.

Classe

Gli oggetti sono istanze di una classe.

Una classe:

- È un reference type
- È composta da membri

I membri di una classe sono:

- Campi
- Proprietà
- Metodi
- Eventi

```
MyClass c = new MyClass();

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

Tipi, classi e oggetti

- Un **tipo** è una rappresentazione concreta di un concetto.

- 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).

Istanze delle classi

- La creazione dell'istanza di una classe (ovvero un oggetto) può avvenire utilizzando la keyword ***new***

```
MyClass c = new MyClass();

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

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

Forma estesa

Forma compatta

```
public class MyClass
{
    private string _name;

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

    public string Surname
    { get; set; }
}
```

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
```

Stack e Managed Heap

Value Type

- Dichiarati all’interno di una funzione: Stack
- Parametri di una funzione: Stack
- *All’interno di una classe: Heap*

Reference Type

- Sempre: Heap

Metodi

Definisce un comportamento o un'elaborazione relative all'oggetto.

Si definisce come una routine, quindi ha una firma in cui si definiscono eventuali parametri d'ingresso e valori di ritorno.

```
int MyMethod(string str) {
    int a = int.Parse(str);
    return a;
}
```

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
}
```

- 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, 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;
}
```

Metodi

- Particolari tipi di metodi : Costruttori
- 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)
```

Costruttore

Costruttore: metodo con stesso nome del tipo relativo. La firma di questo metodo include solo il nome del metodo e l'elenco dei parametri, ma NON include un tipo di ritorno.

Se non si specifica un costruttore per la classe, C# ne crea di default uno vuoto che crea un'istanza dell'oggetto e imposta le variabili ai valori di default

→ ***Costruttore senza parametri***

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

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

Distruttori o finalizzatori

Per eseguire pulizia finale quando un'istanza di classe viene raccolta dal Garbage Collector.

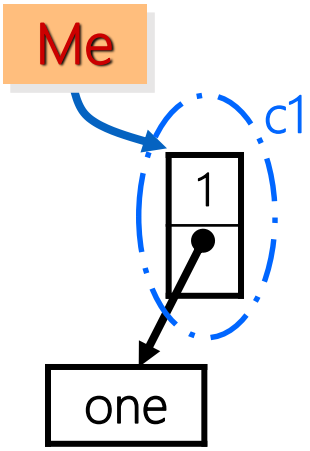
- Vengono usati solo con le classi.
- Uno per classe.
- Un finalizzatore non accetta modificatori e non ha parametri.
- Il Garbage Collector decide quando chiamarlo → chiama metodo ***Finalize***

```
class Person {
   ~Person() //finalizzatore
   {
   //espressioni di cleanup...
   }

}
```

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;
    }
}
```



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();
```

Membri statici e classi statiche

- 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 MyStaticProperty;
    public int MyNotStaticProperty;
}

MyClass c1 = new MyClass();
MyClass c2 = new MyClass();

MyClass.MyStaticProperty = 3;
c1.MyNotStaticProperty = 5;
c2.MyNotStaticProperty = 7;
```

Membri statici e classi statiche

```
public static class MyClass {
    public static int MyStaticProperty;
    public static string MyStaticProperty2;
}

//MyClass c1 = new MyClass(); //Errore!

MyClass.MyStaticProperty = 3;
MyClass.MyStaticProperty2 = “abc”;
```

Classe statica

- I membri devono essere static.
- Non si può istanziare la classe
- È sealed
- Accedere ai membri tramite il nome della classe.

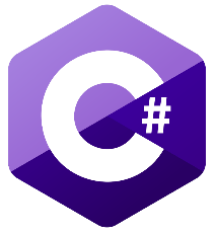
Membri e Classi statiche

- Gli elementi direttamente associati al tipo e condivisi con tutte le istanze vengono detti membri ***statici.***

- Una classe che contiene unicamente membri statici viene anch'essa indicata come statica.

- Per indicare che una classe o un membro è statico si usa la *keyword* ***static.***

La classe Object



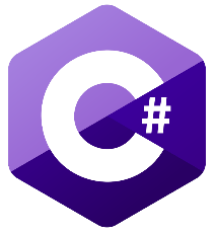
Tutto in .NET deriva dalla **classe Object**

- 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

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 minuscule (es. FullName)

- **Notazione Camel:** come la notazione Pascal, a differenza del fatto che la prima iniziale deve essere minuscola (es. fullName)

Convenzioni sul codice

Elementi	Notazione
Namespace	Notazione Pascal
Classi	Notazione Pascal
Interfacce	Notazione Pascal
Strutture	Notazione Pascal
Enumerazioni	Notazione Pascal
Campi privati	Notazione Camel
Proprietà, metodi e eventi	Notazione Pascal
Parametri di metodi e funzioni	Notazione Camel
Variabili locali	Notazione Camel

© 2019 iCubed Srl

iCubed s.r.l. • Piazza Durante, 8 – 20131, Milano
• Phone: +39 02 57501057 • P.IVA 07284390965