**Programmazione orientata a oggetti**

https://msdn.microsoft.com/it-it/library/dd460654.aspx#Classes

Tutti i linguaggi gestiti in .NET Framework, ad esempio Visual Basic e C#, forniscono il supporto completo per la programmazione orientata a oggetti, inclusi incapsulamento, ereditarietà e polimorfismo.

L'incapsulamento indica che un gruppo di proprietà, metodi e altri membri correlati vengono considerati come una singola unità o un singolo oggetto.

L'ereditarietà indica la capacità di creare nuove classi sulla base di una classe esistente.

Il polimorfismo indica la capacità di utilizzare più classi in modo intercambiabile, anche se in ognuna di esse le stesse proprietà o gli stessi metodi sono implementati in modi diversi.

In questa sezione vengono descritti i concetti seguenti:

* [Classi e oggetti](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Classes)
  + [Membri della classe](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Members)

[Proprietà e campi](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Properties)

[Metodi](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Methods)

[Costruttori](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Constructors)

[Distruttori](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Destructors)

[Eventi](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Events)

[Classi annidate](https://msdn.microsoft.com/it-it/library/dd460654.aspx#NestedClasses)

* + [Modificatori di accesso e livelli di accesso](https://msdn.microsoft.com/it-it/library/dd460654.aspx#AccessModifiers)
  + [Creazione di istanze di classi](https://msdn.microsoft.com/it-it/library/dd460654.aspx#InstantiatingClasses)
  + [Classi e membri statici (condivisi)](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Static)
* [Ereditarietà](https://msdn.microsoft.com/it-it/library/dd460654.aspx" \l "Inheritance)
  + - [Override di membri](https://msdn.microsoft.com/it-it/library/dd460654.aspx#Overriding)

[**Classi e oggetti**](javascript:void(0))

I termini classe e oggetto vengono talvolta utilizzati in modo intercambiabile. Di fatto, però, le classi descrivono il tipo degli oggetti, mentre gli oggetti sono istanze utilizzabili delle classi. L'atto di creare un oggetto viene pertanto chiamato creazione di istanze. Rifacendoci all'analogia precedente, la classe corrisponde al progetto iniziale e l'oggetto all'edificio realizzato in base a tale progetto.

Per definire una classe:

class SampleClass

{

}

C# fornisce anche una versione ridotta delle classi chiamate strutture, utili quando è necessario creare una matrice di grandi dimensioni di oggetti e non si desidera utilizzare a tale scopo una quantità eccessiva di memoria.

Per definire una struttura:

struct SampleStruct

{

}

Per ulteriori informazioni, vedere **Appendice: class**

**Membri di classe**

Ogni classe può disporre di membri della classe diversi che includono proprietà che descrivono i dati della classe, i metodi che definiscono il comportamento della classe e gli eventi che forniscono la comunicazione tra classi e oggetti diversi.

**Proprietà e campi**

I campi e le proprietà rappresentano le informazioni contenute in un oggetto. I campi sono simili a variabili in quanto possono essere letti o impostati direttamente.

Per definire un campo:

Class SampleClass

{

public string sampleField;

}

Le proprietà dispongono di routine Get e Set, che forniscono un maggiore controllo sul modo in cui i valori vengono impostati o restituiti.

C# consente di creare un campo privato per archiviare il valore della proprietà o utilizzare le cosiddette proprietà implementate automaticamente che creano automaticamente questo campo e forniscono la logica di base per le routine della proprietà.

Per definire una proprietà implementata automaticamente:

class SampleClass

{

public int SampleProperty { get; set; }

}

Se è necessario eseguire alcune operazioni aggiuntive per la lettura e la scrittura del valore della proprietà, definire un campo per archiviare il valore della proprietà e fornire la logica di base per archiviarlo e recuperarlo:

class SampleClass

{

private int \_sample;

public int Sample

{

// Return the value stored in a field.

get { return \_sample; }

// Store the value in the field.

set { \_sample = value; }

}

}

La maggior parte delle proprietà dispone di metodi o di routine per impostare e ottenere il valore della proprietà. È possibile, tuttavia, creare proprietà di sola lettura o di sola scrittura per impedirne la modifica o la lettura. In C#, è possibile omettere il metodo della proprietà get o set. Le proprietà implementate automaticamente non possono essere di sola lettura o di sola scrittura.

Per ulteriori informazioni, vedere **Appendice: get, set**

**Metodi**

Un metodo è un'azione che può essere eseguita da un oggetto.

Per definire un metodo di una classe:

class SampleClass

{

public int sampleMethod(string sampleParam)

{

// Insert code here

}

}

Una classe può disporre di diverse implementazioni, o overload, dello stesso metodo che differiscono per il numero di parametri o per i tipi di parametro.

Per essere in rapporto di overload con un metodo:

public int sampleMethod(string sampleParam) {};

public int sampleMethod(int sampleParam) {}

Nella maggior parte dei casi si dichiara un metodo all'interno di una definizione della classe.

Per ulteriori informazioni, vedere **Appendice: metodi**

**Costruttori**

I costruttori sono metodi di classe che vengono eseguiti automaticamente durante la creazione di un oggetto di un tipo specifico. I costruttori in genere inizializzano i membri dati del nuovo oggetto. Un costruttore può essere eseguito solo una volta alla creazione di una classe. Inoltre, il codice nel costruttore viene sempre eseguito prima di qualsiasi altro codice in una classe. Tuttavia, è possibile creare più overload del costruttore esattamente come per qualsiasi altro metodo.

Per definire un costruttore per una classe:

public class SampleClass

{

public SampleClass()

{

// Add code here

}

}

Per ulteriori informazioni, vedere **Appendice: costruttori**

**Distruttori**

I distruttori sono utilizzati per distruggere istanze di classi. In .NET Framework, il Garbage Collector gestisce l'allocazione e il rilascio di memoria per gli oggetti gestiti di un'applicazione. Potrebbero, tuttavia, essere necessari distruttori per pulire eventuali risorse non gestite create dall'applicazione. Può esistere un solo distruttore per classe.

**Eventi**

Tramite gli eventi una classe o un oggetto sono in grado di segnalare ad altre classi o oggetti una situazione di interesse. La classe che invia (o genera) l'evento è chiamata autore e le classi che ricevono (o gestiscono) l'evento sono chiamate sottoscrittori. Per ulteriori informazioni sugli eventi e sulla corrispondente modalità di generazione e gestione, vedere [Gestione e generazione di eventi](https://msdn.microsoft.com/it-it/library/edzehd2t.aspx).

* Per dichiarare un evento in una classe, utilizzare la parola chiave [event (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/8627sbea.aspx).
* Per generare un evento, richiamare il delegato dell'evento.
* Per sottoscrivere un evento, utilizzare l'operatore +=. Per annullare la sottoscrizione a un evento utilizzare l'operatore -=:

**Classi annidate**

Una classe definita all'interno di un'altra classe è denominata annidata. Per impostazione predefinita, la classe annidata è privata.

class Container

{

class Nested

{

// Add code here.

}

}

Per creare un'istanza della classe annidata, utilizzare il nome della classe dei contenitori seguita dal punto, quindi dal nome della classe annidata:

Container.Nested nestedInstance = new Container.Nested()

**Modificatori di accesso e livelli di accesso**

Tutte le classi e i membri della classe possono specificare quale livello di accesso forniscono alle altre classi utilizzando i modificatori di accesso.

Sono disponibili i seguenti modificatori di accesso:

|  |  |
| --- | --- |
| **Modificatore di C#** | **Definizione** |
| [public](https://msdn.microsoft.com/it-it/library/yzh058ae.aspx) | Il tipo o il membro è accessibile da altro codice nello stesso assembly o in un altro assembly che vi fa riferimento. |
| [private](https://msdn.microsoft.com/it-it/library/st6sy9xe.aspx) | Il tipo o il membro è accessibile solo dal codice nella stessa classe. |
| [protected](https://msdn.microsoft.com/it-it/library/bcd5672a.aspx) | Il tipo o il membro è accessibile solo dal codice nella stessa classe o in una classe derivata. |
| [internal](https://msdn.microsoft.com/it-it/library/7c5ka91b.aspx) | Il tipo o il membro è accessibile dal codice nello stesso assembly ma non da un altro assembly. |
| protected internal | Il tipo o il membro è accessibile dal codice nello stesso assembly o da una classe derivata in un altro assembly. |

**Creazione di istanze di classi**

Per creare un oggetto, è necessario creare un'istanza di una classe.

SampleClass sampleObject = new SampleClass();

Dopo avere creato un'istanza di una classe, è possibile assegnare i valori alle proprietà e ai campi dell'istanza e richiamare i metodi della classe.

// Set a property value.

sampleObject.sampleProperty = "Sample String";

// Call a method.

sampleObject.sampleMethod();

Per assegnare i valori alle proprietà durante il processo di creazione dell'istanza della classe, utilizzare gli inizializzatori di oggetto:

// Set a property value.

SampleClass sampleObject = new SampleClass

{ FirstProperty = "A", SecondProperty = "B" };

**Classi e membri statici (condivisi)**

Un membro statico (condiviso in Visual Basic) della classe è una proprietà, una routine o un campo condiviso da tutte le istanze di una classe.

Per definire un membro statico (condiviso):

static class SampleClass

{

public static string SampleString = "Sample String";

}

Per accedere al membro statico (condiviso), utilizzare il nome della classe senza creare un oggetto di questa classe:

Console.WriteLine(SampleClass.SampleString);

Le classi statiche (condivise) in C# e i moduli in Visual Basic dispongono solo di membri statici (condivisi) e non è possibile crearne un'istanza. I membri statici (condivisi) non possono accedere inoltre a proprietà, campi o metodi non statici (non condivisi)

Per ulteriori informazioni, vedere **Dispensa static**

[**Ereditarietà**](javascript:void(0))

L'ereditarietà permette di creare una nuova classe che riutilizza, estende e modifica il comportamento definito in un'altra classe. La classe i cui membri vengono ereditati è denominata classe base, mentre la classe che eredita tali membri è denominata classe derivata. Tuttavia, tutte le classi sia in C# che in Visual Basic ereditano in modo implicito dalla classe [Object](https://msdn.microsoft.com/it-it/library/system.object.aspx) che supporta la gerarchia di classi .NET e fornisce servizi di basso livello a tutte le classi.

**NotaNota:** I linguaggi gestiti in .NET Framework non supportano l'ereditarietà multipla, ad esempio è possibile specificare una sola classe di base per una classe derivata.

Per ereditare da una classe base:

class DerivedClass:BaseClass{}

Per impostazione predefinita, tutte le classi possono essere ereditate. Tuttavia, è possibile specificare se una classe non deve essere utilizzata come classe base oppure creare una classe utilizzabile solo come classe base.

Per specificare che una classe non può essere utilizzata come classe base:

public sealed class A { }

Per specificare che una classe può essere utilizzata solo come classe base e che non è possibile crearne un'istanza:

public abstract class B { }

**Override di membri**

Per impostazione predefinita, in una classe derivata vengono ereditati tutti i membri della classe base relativa. Se si desidera modificare il comportamento del membro ereditato, è necessario eseguirne l'override. È possibile definire una nuova implementazione del metodo, della proprietà o dell'evento nella classe derivata.

I seguenti modificatori consentono di controllare le modalità di override di proprietà e metodi:

|  |  |
| --- | --- |
| **Modificatore di C#** | **Definizione** |
| [virtual (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/9fkccyh4.aspx) | Consente a un membro della classe di essere sottoposto a override in una classe derivata. |
| [override (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/ebca9ah3.aspx) | Esegue l'override di un membro virtuale (sottoponibile a override) definito nella classe base. |
| [abstract (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/sf985hc5.aspx) | Richiede che un membro della classe venga sottoposto a override nella classe derivata. |
| [Modificatore new (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/435f1dw2.aspx) | Nasconde un membro ereditato da una classe base. |

# Appendice: class

https://msdn.microsoft.com/it-it/library/0b0thckt.aspx

Per la dichiarazione delle classi viene utilizzata la parola chiave class, come illustrato nell'esempio seguente:

class TestClass

{

// Methods, properties, fields, events, delegates

// and nested classes go here.

}

Solo l'ereditarietà singola è consentita in c. In altre parole, una classe può ereditare l'implementazione da una sola classe base.

Le classi dichiarate direttamente all'interno di uno spazio dei nomi, non annidate all'interno di altre classi, è possibile definirle [pubblico](https://msdn.microsoft.com/it-it/library/yzh058ae.aspx) o [interno](https://msdn.microsoft.com/it-it/library/7c5ka91b.aspx). Le classi sono internal per impostazione predefinita.

I membri della classe, incluse le classi annidate, è possibile dichiararle [pubblico](https://msdn.microsoft.com/it-it/library/yzh058ae.aspx), protected internal, [protetta](https://msdn.microsoft.com/it-it/library/bcd5672a.aspx), [interno](https://msdn.microsoft.com/it-it/library/7c5ka91b.aspx), o [privato](https://msdn.microsoft.com/it-it/library/st6sy9xe.aspx). I membri sono [privato](https://msdn.microsoft.com/it-it/library/st6sy9xe.aspx) per impostazione predefinita.

Una classe può contenere dichiarazioni dei seguenti membri:

* [Costruttori](https://msdn.microsoft.com/it-it/library/ace5hbzh.aspx)
* [Distruttori](https://msdn.microsoft.com/it-it/library/66x5fx1b.aspx)
* [Costanti](https://msdn.microsoft.com/it-it/library/ms173119.aspx)
* [Campi](https://msdn.microsoft.com/it-it/library/ms173118.aspx)
* [Metodi](https://msdn.microsoft.com/it-it/library/ms173114.aspx)
* [Proprietà](https://msdn.microsoft.com/it-it/library/x9fsa0sw.aspx)
* [Indicizzatori](https://msdn.microsoft.com/it-it/library/6x16t2tx.aspx)
* [Operatori](https://msdn.microsoft.com/it-it/library/ms173145.aspx)
* [Eventi](https://msdn.microsoft.com/it-it/library/awbftdfh.aspx)
* [Delegati](https://msdn.microsoft.com/it-it/library/ms173171.aspx)
* [Classi](https://msdn.microsoft.com/it-it/library/x9afc042.aspx)
* [Interfacce](https://msdn.microsoft.com/it-it/library/ms173156.aspx)
* [Strutture](https://msdn.microsoft.com/it-it/library/saxz13w4.aspx)

## [Esempio](javascript:void(0))

Nell'esempio riportato di seguito viene illustrata la dichiarazione di costruttori, metodi e campi di classi. Viene inoltre descritta la creazione di istanze di oggetti e la stampa di dati di un'istanza. In questo esempio vengono dichiarate due classi, la classe Child, che contiene due campi privati (name e age) e due metodi pubblici. La seconda classe, StringTest, viene utilizzata per contenere Main.

class Child

{

private int age;

private string name;

// Default constructor:

public Child()

{

name = "N/A";

}

// Constructor:

public Child(string name, int age)

{

this.name = name;

this.age = age;

}

// Printing method:

public void PrintChild()

{

Console.WriteLine("{0}, {1} years old.", name, age);

}

}

class StringTest

{

static void Main()

{

// Create objects by using the new operator:

Child child1 = new Child("Craig", 11);

Child child2 = new Child("Sally", 10);

// Create an object using the default constructor:

Child child3 = new Child();

// Display results:

Console.Write("Child #1: ");

child1.PrintChild();

Console.Write("Child #2: ");

child2.PrintChild();

Console.Write("Child #3: ");

child3.PrintChild();

}

}

/\* Output:

Child #1: Craig, 11 years old.

Child #2: Sally, 10 years old.

Child #3: N/A, 0 years old.

\*/

**Commenti**

Si noti, nell'esempio precedente, che è possibile accedere ai campi privati (name e age) solo tramite i metodi pubblici della classe Child. Non è quindi possibile stampare il nome del bambino dal metodo Main utilizzando un'istruzione come quella riportata di seguito.

Console.Write(child1.name); // Error

L'accesso ai membri privati di Child da Main è possibile solo se Main è un membro della classe.

Si noti infine che per l'oggetto creato mediante il costruttore predefinito (child3), il campo dell'età è stato inizializzato su zero per impostazione predefinita.

# Appendice: get, set

https://msdn.microsoft.com/it-it/library/ms228503.aspx

https://msdn.microsoft.com/it-it/library/ms228368.aspx

# get

Di seguito è riportato un esempio di una funzione di accesso get in una proprietà denominata Seconds:

class TimePeriod

{

private double \_seconds;

public double Seconds

{

get { return \_seconds; }

set { \_seconds = value; }

}

}

Nell'esempio seguente viene mostrata una funzione di accesso get in una proprietà implementata automaticamente:

class TimePeriod2

{

public double Hours { get; set; }

}

# set

La parola chiave set definisce un metodo della funzione di accesso in una proprietà.

Di seguito è riportato un esempio di una funzione di accesso set per una proprietà denominata Seconds:

class TimePeriod

{

private double \_seconds;

public double Seconds

{

get { return \_seconds; }

set { \_seconds = value; }

}

}

Nell'esempio seguente viene mostrata una funzione di accesso set in una proprietà implementata automaticamente:

class TimePeriod2

{

public double Hours { get; set; }

}

# Appendice: metodi

https://msdn.microsoft.com/it-it/library/ms173114.aspx

Un metodo è un blocco di codice che contiene una serie di istruzioni. Un programma fa in modo che le istruzioni vengano eseguite chiamando il metodo e specificando gli argomenti del metodo obbligatori. In C#, ogni istruzione eseguita viene attuata nel contesto di un metodo. Il metodo principale è il punto di ingresso per ogni applicazione C# e viene chiamato da Common Language Runtime (CLR) quando viene avviato il programma.

## [Firme del metodo](javascript:void(0))

I metodi vengono dichiarati in una [classe](https://msdn.microsoft.com/it-it/library/0b0thckt.aspx) o in una [struct](https://msdn.microsoft.com/it-it/library/ah19swz4.aspx) specificando il livello di accesso, ad esempio public o private, i modificatori facoltativi, ad esempio abstract o sealed, il valore restituito, il nome del metodo e i parametri del metodo. Queste parti costituiscono la firma del metodo.

I parametri del metodo vengono racchiusi tra parentesi e separati da virgole. Le parentesi vuote indicano che il metodo non richiede parametri.

Questa classe contiene tre metodi:

abstract class Motorcycle

{

// Anyone can call this.

public void StartEngine() {/\* Method statements here \*/ }

// Only derived classes can call this.

protected void AddGas(int gallons) { /\* Method statements here \*/ }

// Derived classes can override the base class implementation.

public virtual int Drive(int miles, int speed) { /\* Method statements here \*/ return 1; }

// Derived classes must implement this.

public abstract double GetTopSpeed();

}

## [Accesso ai metodi](javascript:void(0))

Chiamare un metodo su un oggetto è come accedere a un campo. Dopo il nome dell'oggetto aggiungere un punto, il nome del metodo e le parentesi. Gli argomenti vengono elencati tra parentesi e separati da virgole. I metodi della classe Motorcycle possono quindi essere chiamati come nell'esempio seguente:

class TestMotorcycle : Motorcycle

{

public override double GetTopSpeed()

{

return 108.4;

}

static void Main()

{

TestMotorcycle moto = new TestMotorcycle();

moto.StartEngine();

moto.AddGas(15);

moto.Drive(5, 20);

double speed = moto.GetTopSpeed();

Console.WriteLine("My top speed is {0}", speed);

}

}

## [Parametri di metodo e argomenti](javascript:void(0))

La definizione del metodo specifica i nomi e i tipi di tutti i parametri obbligatori. Quando il codice chiamante chiama il metodo, fornisce valori concreti, detti argomenti, per ogni parametro. Gli argomenti devono essere compatibili con il tipo di parametro, ma il nome dell'argomento (se esistente) usato nel codice chiamante non deve essere lo stesso del parametro denominato definito nel metodo. Ad esempio:

public void Caller()

{

int numA = 4;

// Call with an int variable.

int productA = Square(numA);

int numB = 32;

// Call with another int variable.

int productB = Square(numB);

// Call with an integer literal.

int productC = Square(12);

// Call with an expression that evaulates to int.

productC = Square(productA \* 3);

}

int Square(int i)

{

// Store input argument in a local variable.

int input = i;

return input \* input;

}

## [Passaggio per riferimento e passaggio per valore](javascript:void(0))

Per impostazione predefinita, quando un tipo valore viene passato a un metodo, viene passata una copia anziché l'oggetto stesso. Di conseguenza, le modifiche all'argomento non hanno effetto sulla copia dell'originale nel metodo chiamante. È possibile passare un tipo valore per riferimento usando la parola chiave ref.

Quando viene passato un oggetto di un tipo riferimento a un metodo, viene passato un riferimento all'oggetto, ovvero, il metodo riceve un argomento che indica la posizione dell'oggetto, ma non l'oggetto stesso. Se si modifica un membro dell'oggetto usando questo riferimento, la modifica si riflette nell'argomento nel metodo chiamante, anche se si passa l'oggetto per valore.

Per creare un tipo riferimento, usare la parola chiave class, come mostra l'esempio seguente.

public class SampleRefType

{

public int value;

}

Ora, se si passa un oggetto basato su questo tipo a un metodo, si passa un riferimento all'oggetto. Il seguente esempio passa un oggetto di tipo SampleRefType al metodo ModifyObject.

public static void TestRefType()

{

SampleRefType rt = new SampleRefType();

rt.value = 44;

ModifyObject(rt);

Console.WriteLine(rt.value);

}

static void ModifyObject(SampleRefType obj)

{

obj.value = 33;

}

L'esempio fa essenzialmente la stessa operazione dell'esempio precedente in cui ad un metodo veniva passato un argomento per valore. ma, essendo usato un tipo riferimento, il risultato è diverso. La modifica apportata in ModifyObject al campo value del parametro, obj, cambia anche il campo value dell'argomento, rt, nel metodo TestRefType. Il metodo TestRefType visualizza 33 come output.

.

## [Valori restituiti](javascript:void(0))

I metodi possono restituire un valore al chiamante. Se il tipo restituito, il tipo elencato prima del nome del metodo, non è void, il metodo può restituire il valore usando la parola chiave return. Un'istruzione con la parola chiave return seguita da un valore corrispondente al tipo restituito restituirà tale valore al chiamante del metodo. La parola chiave return interrompe anche l'esecuzione del metodo. Se il tipo restituito è void, un'istruzione return senza un valore è tuttavia utile per interrompere l'esecuzione del metodo. Senza la parola chiave return, l'esecuzione del metodo verrà interrotta quando verrà raggiunta la fine del blocco di codice. Per usare la parola chiave return per restituire un valore, sono obbligatori metodi con un tipo restituito non void. Ad esempio, questi due metodi usano la parola chiave return per restituire numeri interi:

class SimpleMath

{

public int AddTwoNumbers(int number1, int number2)

{

return number1 + number2;

}

public int SquareANumber(int number)

{

return number \* number;

}

}

Per usare un valore restituito da un metodo, il metodo chiamante può usare la chiamata al metodo stessa ovunque è sufficiente un valore dello stesso tipo. È inoltre possibile assegnare il valore restituito a una variabile. I due esempi seguenti di codice ottengono lo stesso risultato:

int result = obj.AddTwoNumbers(1, 2);

result = obj.SquareANumber(result);

// The result is 9.

Console.WriteLine(result);

result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));

// The result is 9.

Console.WriteLine(result);

L'utilizzo di una variabile locale, in questo caso, result, per memorizzare un valore è facoltativo. Potrebbe migliorare la leggibilità del codice o potrebbe essere necessario se si desidera archiviare il valore originale dell'argomento per l'intero ambito del metodo.

# Appendice: costruttori

https://msdn.microsoft.com/it-it/library/ace5hbzh.aspx

Ogni volta che viene creata una [classe](https://msdn.microsoft.com/it-it/library/0b0thckt.aspx) o una [struttura](https://msdn.microsoft.com/it-it/library/ah19swz4.aspx) viene chiamato un costruttore. Una classe o una struttura può avere più costruttori che accettano argomenti differenti. Consentono al programmatore di impostare valori predefiniti, limitare la creazione di istanze, nonché scrivere codice flessibile e di facile lettura. Per ulteriori informazioni ed esempi, vedere [Utilizzo di costruttori (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/ms173115.aspx) e [Costruttori di istanze (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/k6sa6h87.aspx).

Se non viene fornito un costruttore per l'oggetto, per impostazione predefinita in C# ne verrà creato uno che crea l'istanza dell'oggetto e imposta le variabili membro sui valori predefiniti elencati nella [Tabella dei valori predefiniti (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/83fhsxwc.aspx). Per ulteriori informazioni ed esempi, vedere [Costruttori di istanze (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/k6sa6h87.aspx).

Anche le classi e le strutture statiche possono disporre di costruttori. Per ulteriori informazioni ed esempi, vedere [Costruttori statici (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/k9x6w0hc.aspx).

[Utilizzo di costruttori (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/ms173115.aspx)

Quando [classe](https://msdn.microsoft.com/it-it/library/0b0thckt.aspx) o [struttura](https://msdn.microsoft.com/it-it/library/ah19swz4.aspx) viene creato, il costruttore viene chiamato. I costruttori hanno lo stesso nome della classe o della struttura e in genere inizializzano i membri dati del nuovo oggetto.

Nell'esempio seguente viene definita una classe denominata Taxi con un costruttore semplice. Viene quindi creata un'istanza di questa classe con l'operatore [new](https://msdn.microsoft.com/it-it/library/51y09td4.aspx). Il costruttore Taxi viene richiamato dall'operatore new immediatamente dopo l'allocazione della memoria per il nuovo oggetto.

public class Taxi

{

public bool isInitialized;

public Taxi()

{

isInitialized = true;

}

}

class TestTaxi

{

static void Main()

{

Taxi t = new Taxi();

Console.WriteLine(t.isInitialized);

}

}

I costruttori predefiniti, ossia che non accettano parametri, vengono richiamati ogni volta che viene creata un'istanza di un oggetto utilizzando l'operatore new e non vengono forniti argomenti a new. Per ulteriori informazioni, vedere [Costruttori di istanze (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/k6sa6h87.aspx).

A meno che non siano [statiche](https://msdn.microsoft.com/it-it/library/98f28cdx.aspx), le classi senza costruttori ricevono un costruttore predefinito pubblico dal compilatore C# per consentire la creazione di istanze. Per ulteriori informazioni, vedere [Classi statiche e membri di classi statiche (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/79b3xss3.aspx).

È possibile impedire la creazione di istanze di una classe rendendo privato il costruttore, come riportato di seguito:

class NLog

{

// Private Constructor:

private NLog() { }

public static double e = Math.E; //2.71828...

}

Per ulteriori informazioni, vedere [Costruttori privati (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/kcfb85a6.aspx).

I costruttori per i tipi [struct](https://msdn.microsoft.com/it-it/library/ah19swz4.aspx) sono simili ai costruttori di classi, con la differenza che i tipi structs non possono contenere un costruttore predefinito esplicito perché ne viene automaticamente fornito uno dal compilatore. Il costruttore inizializza ogni campo di struct in base ai valori predefiniti. Per ulteriori informazioni, vedere [Tabella dei valori predefiniti (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/83fhsxwc.aspx). Questo costruttore predefinito viene tuttavia richiamato solo se viene creata un'istanza di struct con new. Nel codice seguente viene ad esempio utilizzato il costruttore predefinito per [Int32](https://msdn.microsoft.com/it-it/library/system.int32.aspx), in modo da avere la certezza che venga inizializzato l'intero:

int i = new int();

Console.WriteLine(i);

Il codice seguente comporta tuttavia la generazione di un errore del compilatore in quanto non viene utilizzato l'oggetto new e in quanto viene eseguito un tentativo di utilizzare un oggetto non inizializzato.

int i;

Console.WriteLine(i);

In alternativa, gli oggetti basati su structs (inclusi tutti i tipi numerici incorporati) possono essere inizializzati o assegnati e quindi utilizzati, come illustrato nell'esempio seguente:

int a = 44; // Initialize the value type...

int b;

b = 33; // Or assign it before using it.

Console.WriteLine("{0}, {1}", a, b);

In questo caso non è necessario chiamare il costruttore predefinito per ottenere un tipo di valore.

Sia le classi che gli oggetti structs possono definire costruttori che accettano parametri. I costruttori che accettano parametri devono essere chiamati tramite un'istruzione new o un'istruzione [base](https://msdn.microsoft.com/it-it/library/hfw7t1ce.aspx). Le classi e gli oggetti structs possono inoltre definire più costruttori e non devono necessariamente definire un costruttore predefinito. Ad esempio:

public class Employee

{

public int salary;

public Employee(int annualSalary)

{

salary = annualSalary;

}

public Employee(int weeklySalary, int numberOfWeeks)

{

salary = weeklySalary \* numberOfWeeks;

}

}

Per creare questa classe, è possibile utilizzare una delle seguenti istruzioni:

Employee e1 = new Employee(30000);

Employee e2 = new Employee(500, 52);

Un costruttore può utilizzare la parola chiave base per chiamare il costruttore di una classe base. Ad esempio:

public class Manager : Employee

{

public Manager(int annualSalary)

: base(annualSalary)

{

//Add further instructions here.

}

}

In questo esempio il costruttore della classe base viene chiamato prima dell'esecuzione del blocco per il costruttore. La parola chiave base può essere utilizzata con o senza parametri. Gli eventuali parametri del costruttore possono essere utilizzati come parametri per base oppure come parte di un'espressione. Per ulteriori informazioni, vedere [base (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/hfw7t1ce.aspx).

Se in una classe derivata non viene chiamato in modo esplicito il costruttore della classe di base mediante la parola chiave base, verrà chiamato in modo implicito l'eventuale costruttore predefinito disponibile. Le seguenti dichiarazioni di costruttori sono quindi in realtà identiche:

public Manager(int initialdata)

{

//Add further instructions here.

}

public Manager(int initialdata)

: base()

{

//Add further instructions here.

}

Se una classe di base non include un costruttore predefinito, la classe derivata deve effettuare una chiamata esplicita a un costruttore di base utilizzando base.

Un costruttore può richiamare un altro costruttore nello stesso oggetto utilizzando la parola chiave [this](https://msdn.microsoft.com/it-it/library/dk1507sz.aspx). Analogamente a base, è possibile utilizzare this con o senza parametri. Gli eventuali parametri del costruttore sono disponibili come parametri per this o come parte di un'espressione. È ad esempio possibile riscrivere il secondo costruttore dell'esempio precedente utilizzando la parola chiave this:

public Employee(int weeklySalary, int numberOfWeeks)

: this(weeklySalary \* numberOfWeeks)

{

}

Utilizzando la parola chiave this riportata nell'esempio precedente, verrà chiamato questo costruttore:

public Employee(int annualSalary)

{

salary = annualSalary;

}

I costruttori possono essere contrassegnati come [public](https://msdn.microsoft.com/it-it/library/yzh058ae.aspx), [private](https://msdn.microsoft.com/it-it/library/st6sy9xe.aspx), [protected](https://msdn.microsoft.com/it-it/library/bcd5672a.aspx), [internal](https://msdn.microsoft.com/it-it/library/7c5ka91b.aspx) o protectedinternal. Questi modificatori definiscono le modalità di costruzione della classe per gli utenti della classe. Per ulteriori informazioni, vedere [Modificatori di accesso](https://msdn.microsoft.com/it-it/library/ms173121.aspx).

Un costruttore può essere dichiarato statico mediante la parola chiave [static](https://msdn.microsoft.com/it-it/library/98f28cdx.aspx). I costruttori statici vengono chiamati automaticamente immediatamente prima dell'accesso a qualsiasi campo statico e vengono in genere utilizzati per inizializzare i membri di classi statiche. Per ulteriori informazioni, vedere [Costruttori statici](https://msdn.microsoft.com/it-it/library/k9x6w0hc.aspx).

[Costruttori di istanze (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/k6sa6h87.aspx)

I costruttori di istanza sono utilizzati per creare e inizializzare qualsiasi variabile membro di istanza quando si utilizza l'espressione [new](https://msdn.microsoft.com/it-it/library/51y09td4.aspx) per creare un oggetto di una [classe](https://msdn.microsoft.com/it-it/library/0b0thckt.aspx). Per inizializzare una classe [statica](https://msdn.microsoft.com/it-it/library/98f28cdx.aspx) o variabili statiche in una classe non statica, è necessario definire un costruttore statico. Per ulteriori informazioni, vedere [Costruttori statici (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/k9x6w0hc.aspx).

Nell'esempio seguente viene illustrato un costruttore di istanza:

class CoOrds

{

public int x, y;

// constructor

public CoOrds()

{

x = 0;

y = 0;

}

}

|  |
| --- |
| **NotaNota** |
| Per maggior chiarezza, questa classe contiene campi pubblici. L'utilizzo di campi pubblici non è consigliato nella programmazione, perché consente a qualsiasi metodo in un punto qualsiasi di un programma di accedere senza restrizioni e senza alcuna verifica ai meccanismi interni di un oggetto. I membri dati devono in genere essere privati e devono essere accessibili solo tramite metodi e proprietà di classi. |

Questo costruttore di istanza viene chiamato quando viene creato un oggetto basato sulla classe CoOrds. Un costruttore come questo, che non accetta argomenti, viene definito costruttore predefinito. Tuttavia, risulta spesso utile per fornire costruttori aggiuntivi. È ad esempio possibile aggiungere alla classe CoOrds un costruttore che consenta di specificare i valori iniziali per i membri dati:

// A constructor with two arguments:

public CoOrds(int x, int y)

{

this.x = x;

this.y = y;

}

In questo modo sarà possibile creare oggetti CoOrd con valori iniziali predefiniti o specifici, come illustrato di seguito:

CoOrds p1 = new CoOrds();

CoOrds p2 = new CoOrds(5, 3);

Se una classe non dispone di un costruttore, viene generato automaticamente un costruttore predefinito e per inizializzare i campi dell'oggetto vengono utilizzati i valori predefiniti. Ad esempio, un [int](https://msdn.microsoft.com/it-it/library/5kzh1b5w.aspx) viene inizializzato a 0. Per ulteriori informazioni sui valori predefiniti, vedere [Tabella dei valori predefiniti (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/83fhsxwc.aspx). Poiché il costruttore predefinito della classe CoOrds inizializza tutti i membri dati su zero, può pertanto essere rimosso completamente senza modificare il funzionamento della classe. Per un esempio completo sull'utilizzo di più costruttori, vedere l'Esempio 1 più avanti in questo argomento. Per un esempio di costruttore generato automaticamente, vedere l'Esempio 2.

I costruttori di istanza possono inoltre essere utilizzati per chiamare i costruttori di istanza delle classi base. Il costruttore di classi può richiamare il costruttore della classe base mediante l'inizializzatore, ad esempio:

class Circle : Shape

{

public Circle(double radius)

: base(radius, 0)

{

}

}

In questo esempio la classe Circle passa i valori che rappresentano il raggio e l'altezza al costruttore fornito da Shape, da cui deriva Circle. Per un esempio completo sull'utilizzo di Shape e Circle, vedere l'Esempio 3 di questo argomento.

## [Esempio 1](javascript:void(0))

Nell'esempio riportato di seguito viene illustrata una classe con due costruttori di classi, uno senza argomenti e uno con due argomenti.

class CoOrds

{

public int x, y;

// Default constructor:

public CoOrds()

{

x = 0;

y = 0;

}

// A constructor with two arguments:

public CoOrds(int x, int y)

{

this.x = x;

this.y = y;

}

// Override the ToString method:

public override string ToString()

{

return (String.Format("({0},{1})", x, y));

}

}

class MainClass

{

static void Main()

{

CoOrds p1 = new CoOrds();

CoOrds p2 = new CoOrds(5, 3);

// Display the results using the overriden ToString method:

Console.WriteLine("CoOrds #1 at {0}", p1);

Console.WriteLine("CoOrds #2 at {0}", p2);

Console.ReadKey();

}

}

/\* Output:

CoOrds #1 at (0,0)

CoOrds #2 at (5,3)

\*/

## [Esempio 2](javascript:void(0))

Nell'esempio riportato di seguito la classe Person non ha alcun costruttore. In questo caso viene fornito automaticamente un costruttore predefinito e i campi vengono inizializzati sui valori predefiniti.

public class Person

{

public int age;

public string name;

}

class TestPerson

{

static void Main()

{

Person person = new Person();

Console.WriteLine("Name: {0}, Age: {1}", person.name, person.age);

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

// Output: Name: , Age: 0

Si noti che il valore predefinito di age è 0 e il valore predefinito di name è null. Per ulteriori informazioni sui valori predefiniti, vedere [Tabella dei valori predefiniti (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/83fhsxwc.aspx).

## [Esempio 3](javascript:void(0))

Nell'esempio riportato di seguito viene illustrato l'utilizzo dell'inizializzatore della classe base. La classe Circle è derivata dalla classe generale Shape e la classe Cylinder è derivata dalla classe Circle. Il costruttore di ciascuna classe derivata utilizza il relativo inizializzatore della classe base.

abstract class Shape

{

public const double pi = Math.PI;

protected double x, y;

public Shape(double x, double y)

{

this.x = x;

this.y = y;

}

public abstract double Area();

}

class Circle : Shape

{

public Circle(double radius)

: base(radius, 0)

{

}

public override double Area()

{

return pi \* x \* x;

}

}

class Cylinder : Circle

{

public Cylinder(double radius, double height)

: base(radius)

{

y = height;

}

public override double Area()

{

return (2 \* base.Area()) + (2 \* pi \* x \* y);

}

}

class TestShapes

{

static void Main()

{

double radius = 2.5;

double height = 3.0;

Circle ring = new Circle(radius);

Cylinder tube = new Cylinder(radius, height);

Console.WriteLine("Area of the circle = {0:F2}", ring.Area());

Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

/\* Output:

Area of the circle = 19.63

Area of the cylinder = 86.39

\*/

Per ulteriori esempi su come richiamare i costruttori della classe base, vedere [virtual (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/9fkccyh4.aspx), [override (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/ebca9ah3.aspx) e [base (Riferimenti per C#)](https://msdn.microsoft.com/it-it/library/hfw7t1ce.aspx)

[Costruttori privati (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/kcfb85a6.aspx)

Un costruttore privato è un costruttore di istanza speciale e viene generalmente utilizzato in classi contenenti solo membri static. Se una classe dispone di uno o più costruttori privati ma non include alcun costruttore pubblico, alle altre classi, tranne quelle annidate, non sarà consentita la creazione delle istanze della classe. Ad esempio:

class NLog

{

// Private Constructor:

private NLog() { }

public static double e = Math.E; //2.71828...

}

La dichiarazione del costruttore vuoto impedisce la generazione automatica di un costruttore predefinito. Tenere presente che se non si utilizza un modificatore di accesso con il costruttore, questo rimarrà di tipo privato per impostazione predefinita. Tuttavia, il modificatore [private](https://msdn.microsoft.com/it-it/library/st6sy9xe.aspx) viene solitamente utilizzato in modo esplicito per specificare che non è possibile creare un'istanza della classe.

I costruttori privati vengono utilizzati per impedire la creazione di istanze di una classe in assenza di metodi e campi di istanza, ad esempio la classe [Math](https://msdn.microsoft.com/it-it/library/system.math.aspx), oppure quando viene chiamato un metodo per ottenere un'istanza di una classe. Se tutti i metodi della classe sono statici, è consigliabile rendere statica l'intera classe. Per ulteriori informazioni, vedere [Classi statiche e membri di classi statiche (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/79b3xss3.aspx).

## [Esempio](javascript:void(0))

Di seguito è riportato l'esempio di una classe che utilizza un costruttore privato.

public class Counter

{

private Counter() { }

public static int currentCount;

public static int IncrementCount()

{

return ++currentCount;

}

}

class TestCounter

{

static void Main()

{

// If you uncomment the following statement, it will generate

// an error because the constructor is inaccessible:

// Counter aCounter = new Counter(); // Error

Counter.currentCount = 100;

Counter.IncrementCount();

Console.WriteLine("New count: {0}", Counter.currentCount);

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

// Output: New count: 101

Se si rimuove il commento dall'istruzione dell'esempio riportata di seguito, verrà generato un errore poiché non è possibile accedere al costruttore a causa del livello di protezione.

// Counter aCounter = new Counter(); // Error

[Costruttori statici (Guida per programmatori C#)](https://msdn.microsoft.com/it-it/library/k9x6w0hc.aspx)

Un costruttore statico consente di inizializzare gli eventuali dati [statici](https://msdn.microsoft.com/it-it/library/98f28cdx.aspx) oppure di eseguire un'operazione specifica che deve essere effettuata una sola volta. Viene chiamato automaticamente prima che ne venga creata la prima istanza o venga fatto riferimento a qualsiasi membro statico.

class SimpleClass

{

// Static variable that must be initialized at run time.

static readonly long baseline;

// Static constructor is called at most one time, before any

// instance constructor is invoked or member is accessed.

static SimpleClass()

{

baseline = DateTime.Now.Ticks;

}

}

Di seguito sono riportate le proprietà dei costruttori statici:

* Un costruttore statico non accetta modificatori di accesso né parametri.
* Un costruttore statico viene chiamato automaticamente per inizializzare la [classe](https://msdn.microsoft.com/it-it/library/0b0thckt.aspx) prima che ne venga creata la prima istanza o venga fatto riferimento a qualsiasi membro statico.
* Un costruttore statico non può essere chiamato direttamente.
* L'utente non può controllare in alcun modo il momento in cui il costruttore statico viene eseguito nel programma.
* In genere, i costruttori statici sono utilizzati per scrivere voci nel file di log, quando alla classe è associato un file di log.
* I costruttori statici risultano utili anche durante la creazione di classi wrapper per il codice non gestito, quando il costruttore può chiamare il metodo LoadLibrary.
* Se un costruttore statico genera un'eccezione, il runtime non lo richiamerà una seconda volta e il tipo rimarrà non inizializzato per la durata del dominio dell'applicazione in cui il programma è in esecuzione.

## [Esempio](javascript:void(0))

In questo esempio la classe Bus dispone di un costruttore statico. Quando viene creata la prima istanza di Bus (bus1), il costruttore statico viene richiamato per inizializzare la classe. L'output dell'esempio verifica che il costruttore statico venga eseguito una sola volta, anche se vengono create due istanze di Bus, e che venga eseguito prima del costruttore di istanza.

public class Bus

{

// Static variable used by all Bus instances.

// Represents the time the first bus of the day starts its route.

protected static readonly DateTime globalStartTime;

// Property for the number of each bus.

protected int RouteNumber { get; set; }

// Static constructor to initialize the static variable.

// It is invoked before the first instance constructor is run.

static Bus()

{

globalStartTime = DateTime.Now;

// The following statement produces the first line of output,

// and the line occurs only once.

Console.WriteLine("Static constructor sets global start time to {0}",

globalStartTime.ToLongTimeString());

}

// Instance constructor.

public Bus(int routeNum)

{

RouteNumber = routeNum;

Console.WriteLine("Bus #{0} is created.", RouteNumber);

}

// Instance method.

public void Drive()

{

TimeSpan elapsedTime = DateTime.Now - globalStartTime;

// For demonstration purposes we treat milliseconds as minutes to simulate

// actual bus times. Do not do this in your actual bus schedule program!

Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",

this.RouteNumber,

elapsedTime.TotalMilliseconds,

globalStartTime.ToShortTimeString());

}

}

class TestBus

{

static void Main()

{

// The creation of this instance activates the static constructor.

Bus bus1 = new Bus(71);

// Create a second bus.

Bus bus2 = new Bus(72);

// Send bus1 on its way.

bus1.Drive();

// Wait for bus2 to warm up.

System.Threading.Thread.Sleep(25);

// Send bus2 on its way.

bus2.Drive();

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

/\* Sample output:

Static constructor sets global start time to 3:57:08 PM.

Bus #71 is created.

Bus #72 is created.

71 is starting its route 6.00 minutes after global start time 3:57 PM.

72 is starting its route 31.00 minutes after global start time 3:57 PM.

\*/