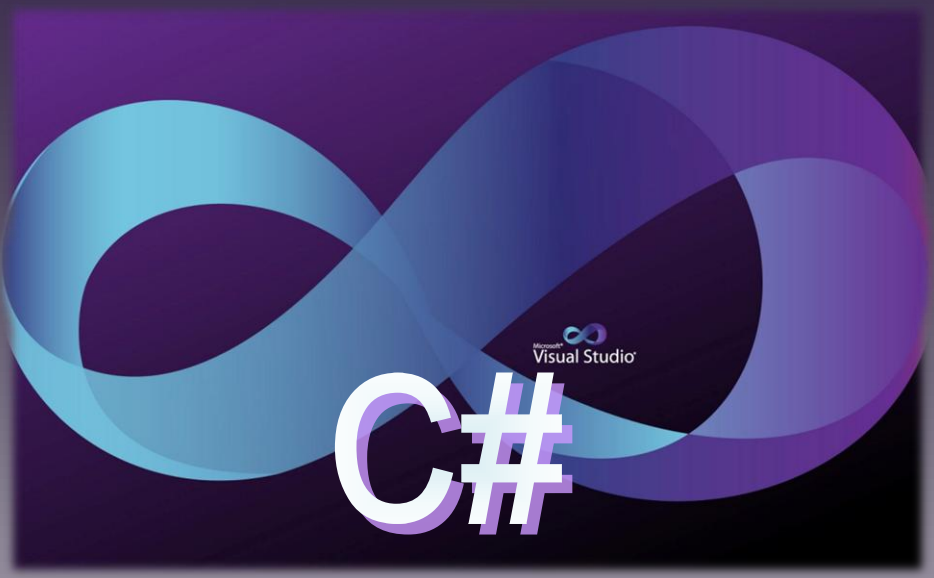
**ISTITUTO TECNICO INDUSTRIALE**
**G. M. ANGIOY**
**SASSARI**




# CORSO DI PROGRAMMAZIONE

## ALTRE CLASSI E PARTICOLARITÀ SUGLI OGGETTI

### DISPENSA 15.06

15-06_OOP_Rifiniture_[06]




Versione del: 07/11/2015
Revisione numero: 15

Prof. Andrea Zoccheddu
Dipartimento di Informatica

**DIPARTIMENTO**
**INFORMATICA E TELECOMUNICAZIONI**

# CLASSI PARTICOLARI

## CLASSI PARTICOLARI

### CLASSI NIDIFICATE (NESTED)

Per tipo nidificato si intende un tipo definito all'interno di una classe o di una struttura.

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms173120(v=vs.80)

Esempio:

VC#

```
class ClasseEsterna
{
    class ClasseInterna
    {
        ClasseInterna() { }        //costruttore della classe Interna
    }
}
```

Per impostazione predefinita i tipi nidificati sono privati, ma possono essere forzati in **public**, **protected internal**, **protected**, **internal** o **private**. Nell'esempio precedente ClasseInterna non è accessibile a tipi esterni, ma può essere reso public con uno specificatore di accesso, nel seguente modo:

VC#

```
class ClasseEsterna
{
    public class ClasseInterna
    {
        ClasseInterna() { }
    }
}
```

Il tipo interno o nidificato può accedere al tipo esterno o che lo contiene. Per accedere al tipo che lo contiene, passarlo come costruttore al tipo nidificato.

Esempio:

VC#

```
public class ClasseEsterna
{
    public class ClasseInterna
    {
        private ClasseEsterna figlio;
        public ClasseInterna()
        {
        }
        public ClasseInterna (ClasseEsterna padre)
        {
            figlio = padre;
        }
    }
}
```

I tipi nidificati possono accedere a membri privati e protetti del tipo che li contengono, inclusi tutti i membri privati o protetti ereditati.

Nella dichiarazione precedente il nome completo della classe ClasseInterna è ClasseEsterna.ClasseInterna. Questo nome viene utilizzato per creare una nuova istanza della classe nidificata, come illustrato di seguito:

VC#

```
ClasseEsterna.ClasseInterna nido = new ClasseEsterna.ClasseInterna();
```

## CLASSI ASTRATTE (ABSTRACT)

### SCOPO E FINALITÀ DELLE CLASSI STRATTE

LINK

Fonte: http://msdn.microsoft.com/it-it/library/sf985hc5(v=vs.80).aspx

Il modificatore abstract può essere utilizzato per definire classi, metodi, proprietà, indicizzatori ed eventi.

Il modificatore abstract è usato in una dichiarazione di metodo per indicare che il metodo non è definito nella classe stessa, ma impone che sia necessariamente definito in qualsiasi classe discendente. In pratica il metodo astratto è dichiarato ma non definito ed impone a qualsiasi classe discendente di implementarlo necessariamente.

Il modificatore abstract è usato in una dichiarazione di classe per indicare che la classe può essere utilizzata soltanto come classe base di altre classi. In pratica la classe astratta è definita solo per consentire di definire altre classi discendenti, la cui struttura è già suggerita.

I membri contrassegnati come astratti o inclusi in una classe astratta devono essere implementati da classi che derivano dalla classe astratta.

Le classi astratte presentano le caratteristiche descritte di seguito.

- Non è possibile creare un'istanza di una classe astratta.
- Una classe astratta può contenere metodi e funzioni di accesso astratti.
- Non è possibile modificare una classe astratta con il modificatore sealed (Riferimenti per C#), il che significa che la classe non può essere ereditata.
- Una classe non astratta derivata da una classe astratta deve includere le implementazioni effettive di tutti i metodi e le funzioni di accesso astratti ereditati.

- Utilizzare il modificatore abstract nella dichiarazione di un metodo o di una proprietà per indicare che il metodo o la proprietà non contengono implementazioni.

I metodi astratti presentano le caratteristiche descritte di seguito.

- È errato utilizzare il modificatore static o virtual per la dichiarazione di un metodo astratto. Un metodo astratto è in modo implicito un metodo virtuale.
- Le dichiarazioni dei metodi astratti possono essere utilizzate solo per le classi astratte.
- Poiché la dichiarazione di un metodo astratto non fornisce alcuna implementazione effettiva, non esiste un corpo del metodo. La dichiarazione del metodo termina semplicemente con un punto e virgola e dopo la firma non sono presenti parentesi graffe per racchiudere un ipotetico corpo.

Di seguito è proposto un esempio:

VC#
```
abstract class Poligono      //classe astratta
{
    abstract public int Area();    //metodo astratto
}
```

### SINTASSI DELLA CLASSE ASTRATTA

Per definire una classe astratta si deve usare il descrittore **abstract** nella posizione dei consueti descrittori. Per definire una caratteristica astratta si deve usare il descrittore **abstract** nella posizione dei consueti descrittori.

VC#
```
abstract class Poligono
{
    abstract public int Area();
}
```

Nell'esempio proposto di seguito, la classe Rettangolo è obbligato a fornire un'implementazione del metodo Area poiché deriva da Poligono, dove è previsto un metodo Area:

VC#
```
class Rettangolo : Poligono
{
    int x, y;
    // Se non si definisce un metodo Area si solleva un errore di
compilazione
    override public int Area()
    {
        return x * y;
    }
}
```

Il metodo implementato nella classe discendente è definito con la parola chiave override, come per i metodi virtuali.

### COSTRUTTORI DI CLASSI ASTRATTE

Una classe astratta impone di non poter creare istanze. A causa di questa visione si desume che la definizione di costruttori per tali classi abbiano una attenzione particolari. Microsoft propone delle linee guida che aiutano a definire classi astratte progettate correttamente, in modo che funzionino adeguatamente quando siano implementate.

**Regola 1**) Conviene non definire costruttori pubblici per una classe astratta. I costruttori con una tale visibilità sarebbero rivolti alla creazione di istanze, evento impossibile per una classe astratta.

**Regola 2**) È possibile definire un costruttore privato o protetto in classi astratte. Se si definisce un costruttore privato o protetto, la classe derivata può sfruttare il costruttore della classe base per eseguire attività di preparazione di elementi ereditati anche dalla classe derivata.

**Regola 3**) È utile provare a implementare almeno un caso specifico di una classe discendente dalla classe astratta definita. Sperimentare concretamente un esempio di classe discendente permette di verificare il funzionamento dei meccanismo previsti.

### ESEMPI DI CLASSI ASTRATTE

**Classe CommonDialog**

LINK

Fonte:

http://msdn.microsoft.com/it-it/library/system.windows.media.textformatting.textparagraphproperties(v=vs.90).aspx
http://msdn.microsoft.com/it-it/library/microsoft.win32.commondialog(v=vs.90).aspx

**Classe Package**

LINK

Fonte:

http://msdn.microsoft.com/it-it/library/system.windows.media.textformatting.textparagraphproperties(v=vs.90).aspx
http://msdn.microsoft.com/it-it/library/system.io.packaging.package(v=vs.90).aspx

**Classe Shape**

LINK

Fonte:

http://msdn.microsoft.com/it-it/library/system.windows.media.textformatting.textparagraphproperties(v=vs.90).aspx
http://msdn.microsoft.com/it-it/library/system.windows.shapes.shape(v=vs.85).aspx

**Classe TextParagraphProperties**

LINK

Fonte:

http://msdn.microsoft.com/it-it/library/system.windows.media.textformatting.textparagraphproperties(v=vs.90).aspx

# CLASSI SIGILLATE (SEALED)

## SCOPO E FINALITÀ DELLE CLASSI SIGILLATE

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms228362.aspx

Il modificatore sigillato può essere applicato alle classi, metodi istanza e proprietà. Una classe **sealed** non può essere ereditata. Un metodo sigillato sostituisce un metodo in una classe base, ma in sé non può essere ignorato ulteriormente in qualsiasi classe derivata. Quando viene applicato a un metodo o una proprietà, la tenuta modificatore deve essere sempre utilizzato con **override** (Riferimenti per C #) .

Una classe **sealed** non può essere ereditata. È un errore usare una classe **sealed** come classe base. Utilizzare il sigillata modificatore in una dichiarazione di classe per evitare l'ereditarietà della classe.

Non è consentito l'utilizzo del astratta modificatore con una classe **sealed**.

Le strutture sono implicitamente sigillate, quindi, non può essere ereditata.

VC#

```
// cs_sealed_keyword.cs
// Sealed classes
using System;
sealed class MyClass
{
   public int x;
   public int y;
}

class MainClass
{
   public static void Main()
   {
      MyClass mC = new MyClass();
      mC.x = 110;
      mC.y = 150;
      Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
   }
}
```

## CONSIDERAZIONI SULLE CLASSI SIGILLATE

Per certi versi le due specificazioni **abstract** e **sealed** sono complementari e contrapposte.

La parola chiave **abstract** consente di creare classi e membri di classi unicamente ai fini dell'ereditarietà, ossia per definire le funzionalità di classi derivate non astratte. La parola chiave **sealed** consente di impedire l'ereditarietà di una classe o di determinati membri di classi in precedenza contrassegnati come virtuali.

Le classi possono essere dichiarate come astratte inserendo la parola chiave **abstract** prima della parola chiave **class** nella relativa definizione.

Le classi possono essere dichiarate come **sealed** inserendo la parola chiave **sealed** prima della parola chiave **class** nella relativa definizione.

In pratica gli elementi astratti costringono a una discendenza specifica, mentre gli elementi sigillati impediscono una discendenza specifica.

Una classe **sealed** non può essere utilizzata come classe base. Per questo motivo non può neanche essere una classe astratta. Le classi **sealed** vengono utilizzate principalmente per evitare derivazioni. Poiché non possono mai essere utilizzate come classe base, in alcune ottimizzazioni in fase di esecuzione la chiamata ai membri delle classi **sealed** può risultare leggermente più rapida.

Un membro, un metodo, un campo, una proprietà o un evento di una classe derivata che esegue l'**override** di un membro virtuale della classe base può dichiarare questo membro come **sealed**. In questo modo viene impedito l'aspetto virtuale del membro per qualsiasi ulteriore classe derivata. A questo scopo è necessario inserire la parola chiave **sealed** prima della parola chiave **override** nella dichiarazione del membro della classe. Esempio:

VC#
```
public class Discendente : Antenato
{
    public sealed override void Metodo() { }
}
```

## CLASSI IMMUTABILI

### LA CLASSE IMMUTABILE DELLE STRINGHE

http://msdn.microsoft.com/it-it/library/ms228362.aspx

Questo paragrafo è da fare.

Questo paragrafo è da fare.

Questo paragrafo è da fare.

Questo paragrafo è da fare.

Questo paragrafo è da fare.

### LA CLASSE IMMUTABILE DEI NUMERI

Mutabilità e la struttura dei tipi numerici

Questo paragrafo è da fare.

Questo paragrafo è da fare.

Questo paragrafo è da fare.

Questo paragrafo è da fare.

Nell'esempio riportato di seguito viene creata un'istanza di un oggetto BigInteger e quindi ne viene incrementato il valore di uno.

```
BigInteger number = BigInteger.Multiply(Int64.MaxValue, 3);
number++;
Console.WriteLine(number);
```

Questo esempio modifica solo all'apparenza il valore dell'oggetto esistente. In realtà gli oggetti numerici sono immutabili ovvero in realtà il Common Language Runtime crea internamente un nuovo oggetto dello

stesso tipo e sarà questo nuovo oggetto che sarà restituito al chiamante. Se nessuna variabile punta al vecchio oggetto, questo sarà affidato al Garbage Collector.

Sebbene questo processo sia trasparente (nascosto) al chiamante, esso comporta un calo delle prestazioni. In alcuni casi, specialmente quando si eseguono operazioni ripetute ciclicamente su valori BigInteger molto grandi, tale riduzione delle prestazioni può essere significativa.

# CONCETTO DI PROPRIETÀ

## BUONE PRASSI E PROGETTAZIONE

### DEFINIZIONE DI PROPRIETÀ

LINK

Fonte: http://msdn.microsoft.com/it-it/library/w86s7x04.aspx

In Visual C# sono previste proprietà che riescono a combinare insieme alcuni aspetti dei campi e dei metodi. Dal punto di vista dell'utilizzatore di un oggetto, una proprietà appare come un attributo e l'accesso in scrittura e lettura alla proprietà richiede esattamente la stessa sintassi.

Dal punto di vista del responsabile dell'implementazione di una classe, una proprietà è costituita da uno o due blocchi di codice che rappresentano una funzione di accesso get e/o set. Il blocco di codice relativo alla funzione di accesso get viene eseguito quando la proprietà viene letta, mentre il blocco di codice relativo alla funzione di accesso set viene eseguito quando viene assegnato un nuovo valore alla proprietà.

Una proprietà in cui non è definita la funzione di accesso set è considerata in sola lettura. Una proprietà in cui non è definita la funzione di accesso get è considerata in sola scrittura. Una proprietà con entrambe le funzioni di accesso è considerata di lettura/scrittura.

Al contrario dei campi, le proprietà non sono classificate come variabili. Pertanto, non è possibile passare una proprietà come parametro per riferimento né per risultato.

Le proprietà possono essere utilizzate per diversi scopi, ad esempio per convalidare i dati prima di consentire una modifica, per esporre in modo trasparente i dati in una classe in cui tali dati vengono in realtà recuperati da un'altra origine, ad esempio un database, per eseguire un'azione in caso di modifica dei dati, ad esempio la generazione di un evento, o per modificare il valore di altri campi.

Le proprietà vengono dichiarate all'interno del blocco della classe specificando il livello di accesso del campo, seguito dal tipo e dal nome della proprietà e da un blocco di codice in cui sono dichiarate le funzioni di accesso get e/o set. Di seguito è riportato un esempio:

### SINTASSI DELLA PROPRIETÀ

La sintassi generale di una proprietà assomiglia alla seguente:

VC#
```
public class MiaClasse
{
    private Tipo attributo ;
    public int Attributo
    {
        get
        {
            // corpo del metodo
            return Valore;
        }
        set
        {
            // corpo del metodo
            attributo = Valore;
        }
    }
}
```

Per poter operare sul valore passato come argomento del metodo Set, è possibile sfruttare la variabile predefinita **value** che rappresenta il valore passato come nuovo valore dell'attributo legato alla proprietà. Per esempio:

VC#
```
public class Recipiente
{
    private int capacità;
    private int contenuto;
    public int Contenuto
    {
        get
        {
            return contenuto;
        }
        set
        {
            if ( ( value >= 0 ) && ( value <= capacità ) )
            contenuto = value;
        }
    }
}
```

### METODO GET PER LE PROPRIETÀ

Il corpo della funzione di accesso get è simile a quello di un metodo. Tale funzione deve restituire un valore del tipo di proprietà. L'esecuzione della funzione di accesso get equivale alla lettura del valore del campo. La proprietà non prevede parametri (e non prevede l'uso delle parentesi tonde) così come non li prevede il metodo get.

Ad esempio, durante la restituzione della variabile privata dalla funzione di accesso get, se le ottimizzazioni sono attivate, la chiamata al metodo della funzione di accesso get viene resa inline dal compilatore, in modo da evitare un sovraccarico per la chiamata al metodo. Tuttavia, un metodo virtuale della funzione di accesso get non può essere reso inline poiché il compilatore non conosce, in fase di compilazione, quale metodo può essere effettivamente chiamato in fase di esecuzione.

#### METODO SET PER LE PROPRIETÀ

La funzione di accesso set è simile a un metodo che restituisce un valore di tipo void. Utilizza un parametro implicito denominato value, il cui tipo corrisponde al tipo della proprietà.

#### OSSERVAZIONI SULLE PROPRIETÀ

Le proprietà possono essere contrassegnate come public, private, protected, internal o protected internal.

Questi modificatori di accesso definiscono il modo in cui gli utenti della classe possono accedere alla proprietà. Le funzioni di accesso get e set per la stessa proprietà possono avere modificatori di accesso differenti. Ad esempio, la funzione di accesso get può essere public per consentire l'accesso in sola lettura dall'esterno del tipo, mentre la funzione di accesso set può essere private o protected.

Una proprietà può essere dichiarata statica utilizzando la parola chiave static. In questo modo la proprietà sarà sempre disponibile ai chiamanti, anche se non esiste alcuna istanza della classe.

Una proprietà può essere contrassegnata come virtuale utilizzando la parola chiave virtual. In questo modo le classi derivate possono eseguire l'override del comportamento della proprietà utilizzando la parola chiave override.

Una proprietà che esegue l'override di una proprietà virtuale può anche essere contrassegnata come sealed, per indicare alle classi derivate che la proprietà non è più virtuale. Infine, una proprietà può essere dichiarata abstract. Ciò significa che non vi è implementazione nella classe e che le classi derivate devono scrivere la propria implementazione.

# GERARCHIA WPF

## GERARCHIA DI OGGETTI PREDEFINITI

Dopo aver studiato il significato delle classi e il meccanismo dell'ereditarietà, è opportuno approfondire adesso alcuni aspetti della gerarchia delle classi predefinite in Visual Studio. Come è naturale aspettarsi, Visual Studio mette a disposizione del programmatore molte classi e queste sono organizzate in una gerarchia di discendenze che formano un ampio albero.

#### LA CLASSE OBJECT ANTENATO DI TUTTI GLI OGGETTI

LINK

Fonte: http://msdn.microsoft.com/it-it/library/system.object.aspx

La classe Object è la radice della gerarchia delle classi di Visual Studio; essa costituisce la base di tutte classi della gerarchia di classi del .NET Framework e implementa i servizi di basso livello per tutte le classi derivate.

Quando si dichiara un parametro di tipo Object questo è in grado di accettare come argomento qualsiasi classe. Per esempio:

VC#

```
private void button1_Click(object sender, EventArgs e)
{
    //corpo del gestore; qui sender è il controllo che ha scatenato l'evento
}
```

Nell'esempio qui sopra il parametro sender (mittente) del gestore Click è di tipo object e quindi è ammesso un qualsiasi elemento (componente, controllo o altro) come sorgente dell'evento.

Qualsiasi classe è anche un Object, persino senza dichiararla esplicitamente.

VC#

```
public class Pippo
{
    //blocco della classe Pippo
}
```

```
public class Pippo : Object
{
    //blocco della classe Pippo
}
```

Le due dichiarazioni della classe Pippo sopra illustrate, sono equivalenti e identiche.

Secondo Microsoft, sebbene sia a volte necessario sviluppare classi generiche che accettano e restituiscono tipi Object, è possibile migliorare le prestazioni fondandosi su una classe specifica del tipo per gestire un tipo utilizzato di frequente.

### GERARCHIA DEGLI OGGETTI WPF

| | |
|---|---|
| Object | classe base per qualsiasi oggetto |
| MarshalByRefObject | classe fondamentale per oggetti che interagiscono |
| Component | classe elementare per oggetti visuali |
| Control | classe elementare per oggetti dotati di focus |

## CLASSE COMPONENT (CLASSE BASE PER I OGGETTI VISUALI)

Una classe di base molto vicina a Object è la classe Component che fornisce l'implementazione di base per collegare e consentire all'oggetto di condivisione tra applicazioni.

Component è la classe base per tutti i componenti in CLR (Common Language Runtime) che eseguono il azioni di marshalling. Il Marshalling è, senza addentrarci nei dettagli, l'insieme delle tecniche di gestione dei modi di comunicazione tra metodi (chiamante e chiamati) ovvero della gestione della memoria che funge da canale di comunicazione quando si invoca un metodo.

### GERARCHIA

| | |
|---|---|
| Object | classe base per qualsiasi oggetto |
| MarshalByRefObject | classe fondamentale per oggetti che interagiscono |
| Component | classe elementare per oggetti visuali |
| Control | classe elementare per oggetti dotati di focus |

LINK

Fonte: http://msdn.microsoft.com/it-it/library/eaw10et3.aspx

## CLASSE CONTROL (CLASSE BASE PER OGGETTI CON FOCUS)

LINK

Fonte: http://msdn.microsoft.com/it-it/library/system.windows.controls.control.aspx

La classe Control è un discendente abbastanza prossimo alla classe Component. Tra Component e Control ci sono altre classi intermedie utili per definire metodi basilari per la gestione dei messaggi e della visualizzazione grafica.

La classe Control costituisce la classe primaria per costruire molti dei controlli aggiunti a un'applicazione. La classe Control definisce molti dei comportamenti fondamentali sebbene nessuno sia immediatamente rilevante; sebbene sia possibile aggiungere ad una applicazione un oggetto di tipo Control, è molto più comune e semplice aggiungere un controllo già derivato da Control, come per esempio un oggetto di tipo Button o di tipo ListBox.

Se si desidera creare un controllo con un comportamento personalizzato e per concederne altri che caratterizzano il suo aspetto, si può definire un controllo derivato da Control e riclassificare e ridefinire la proprietà ControlTemplate.

## CLASSE CONTENTCONTROL (CLASSE BASE PER CONTROLLI CON CONTENUTO)

Rappresenta un controllo con una sola parte di contenuto. Una classe ContentControl ha uno stile predefinito limitato. Se si desidera migliorare l'aspetto del controllo.

Un uso tipico di questa classe è rivolto alla gestione delle informazioni su un elemento selezionato in un controllo ItemsControl (per esempio una lista di controlli visuali). Da questa classe discendono molti dei controlli usuali o controlli base per quelli usuali. Per esempio:

### *GERARCHIA*

| | |
|---|---|
| ContentControl | classe base per i controlli visuali classici |
| Frame | controllo con contenuto che supporta lo spostamento |
| GroupItem | usato come contenitore di altri elementi |
| Label | è l'etichetta di testo per un controllo con tasti di scelta |
| ListBoxItem | elemento selezionabile in un controllo ListBox. |
| Primitives.ButtonBase | libreria Primitives, classe base per i pulsanti |
| Primitives.StatusBarItem | libreria Primitives, classe base per la barra di stato |
| ToolTip | classe per i suggerimenti «al volo» |
| UserControl | è la base più semplice per creare nuovi controlli |

### CLASSE BUTTONBASE (CLASSE BASE PER I PULSANTI)

Questo paragrafo è da finire.

Rappresenta la classe base per tutti i controlli Button. È la classe che si preoccupa della gestione dell'evento Click per reagire quando l'utente fa clic su un ButtonBase. È anche la classe che gestisce la possibilità per un controllo di essere attivo.

La classe non discende direttamente da Control, ma esistono alcune classi intermedie.

#### *GERARCHIA DI EREDITARIETÀ*

| | |
|---|---|
| Control | classe elementare per oggetti dotati di focus |
| ContentControl | classe base |
| ButtonBase | classe base per qualsiasi pulsante cliccabile |

Come si nota nella gerarchia il ButtonBase discendete da ContentControl.

### CLASSE BUTTON (CLASSE CONCLUSIVA PER I PULSANTI)

Un classico controllo visuale è il pulsante della classe Button.

È il controllo più frequentemente usato per gestire input con clic del mouse oppure, se il pulsante si trova nello stato attivo, con il tasto INVIO o la BARRA SPAZIATRICE.

L'aspetto del pulsante può essere modificato (ad esempio, impostare la proprietà FlatStyle) e sfrutta i metodi di visualizzazione ereditati dalle classi antenate.

#### *GERARCHIA DI EREDITARIETÀ*

| | |
|---|---|
| Object | classe base per qualsiasi oggetto |
| MarshalByRefObject | classe fondamentale per oggetti che interagiscono |
| Component | classe elementare per oggetti visuali |
| Control | classe elementare per oggetti dotati di focus |
| ButtonBase | classe base per qualsiasi pulsante cliccabile |
| Button | classe finale per il classico pulsante |

Come si nota nella gerarchia il Button deriva da ButtonBase, a sua volta discendente di Control, che discende da Component.

### CLASSE LABEL (CLASSE CONCLUSIVA PER LE ETICHETTE)

È l'etichetta di testo. Questa classe implementa il supporto funzionale e visivo per i tasti di scelta rapida (scorciatoie). In genere è utilizzata per consentire l'accesso rapido da tastiera ad altri controlli come il TextBox.

Per associare un tasto di scelta, è sufficiente aggiungere un carattere di sottolineatura prima del carattere riconosciuto come tasto di scelta.

### GERARCHIA

| | |
|---|---|
| ContentControl | classe base per i controlli visuali classici |
| Label | è l'etichetta di testo per un controllo con tasti di scelta |

#### IL PARAMETRO SENDER (PARAMETRO ASSOCIATO A EVENTI)

Un gestore eventi è un metodo che viene associato a un evento. Quando viene generato l'evento, viene anche eseguito il codice all'interno del gestore eventi. Ciascun gestore eventi dispone di due parametri che consentono di gestire correttamente l'evento.

Il primo parametro è generalmente un oggetto sender; questo parametro è un riferimento all'oggetto che ha generato l'evento. È dichiarato di tipo Object per poter accogliere qualsiasi controllo come mittente dell'evento generato, per esempio Button, Label, Form1.

Il secondo parametro invece è spesso di tipo differente e specifico per riferirsi a elementi di contesto all'evento generato. In genere fornisce informazioni su posizioni del mouse, su tasti della keyboard, su opzioni di dialogo, ecc.

Un gestore che coincide come parametri può essere usato anche per gestire altri eventi. Ad esempio gli eventi MouseDown e MouseUp, avendo lo stesso tipo di parametri, possono utilizzare lo stesso gestore.

## NUOVI CONTROLLI

LINK

Fonte: http://msdn.microsoft.com/it-it/library/cc460686(v=vs.71).aspx

Visual C# consente di creare controlli personalizzati sfruttando l'ereditarietà. L'ereditarietà permette di definire controlli nuovi che non solo mantengono tutte le funzionalità proprie dei controlli Windows Form da cui si deriva il nuovo controllo, ma possono includere anche caratteristiche specifiche.

Per creare un nuovo controllo si crea un nuovo progetto per il quale è necessario specificare il nome da impostare nello spazio dei nomi di primo livello, il nome dell'assembly e il nome del progetto e assicurarsi che il componente di base sia inserito nello spazio dei nomi corretto.

Quando si crea il nuovo progetto si deve selezionare il modello denominato **Libreria di controlli Windows** dall'elenco dei possibili progetti di C#, e immettere MioNomeLibreria nella casella Nome.

#### ASSEGNAZIONE DI UNO SFONDO TRASPARENTE AL CONTROLLO

Per impostazione predefinita, i controlli non supportano sfondi trasparenti. È tuttavia possibile assegnare a un controllo un colore di sfondo opaco trasparente o parzialmente trasparente utilizzando il metodo Control.SetStyle nel costruttore. Il metodo SetStyle della classe Control consente di impostare particolari preferenze di stile per i controlli e può essere utilizzato per attivare o disattivare il supporto per gli sfondi trasparenti.

Per assegnare al controllo uno sfondo trasparente, nell'editore del codice del controllo, individuare il costruttore e fargli invocare il metodo SetStyle del form nel costruttore.

VC#

```
SetStyle(ControlStyles.SupportsTransparentBackColor, true);
```

Mediante queste operazioni il controllo sarà in grado di supportare uno sfondo trasparente.

Sotto la riga di codice aggiunta nel passaggio 1, aggiungere la riga riportata di seguito. Con questa operazione la proprietà BackColor del controllo verrà impostata su Transparent.

VC#

```
this.BackColor = Color.Transparent;
```

È anche possibile creare colori parzialmente trasparenti utilizzando il metodo Color.FromArgb.

## DEFINIZIONE DI COMPONENTI

LINK

Fonte: http://msdn.microsoft.com/it-it/library/00912yx7(v=vs.71).aspx

Ci sono molti modi per avvicinarsi a un problema di programmazione e la creazione di propri componenti è solo un altro stile di progettazione. Avendo questo in mente, i seguenti passi rappresentano un procedimento agevolmente comprensibile. Per definire un componente:

1. Determinare lo scopo del componente da realizzare e quale parte giochi nell'applicazione;
2. Se si ha un componente complesso e si desidera un oggetto modello, si deve delineare il modello.
3. Se necessario, si scompone la funzionalità in più componenti differenti, eventualmente con sottoclassi, o classi nidificate e si struttura il modello oggetto.
4. Si individua la migliore classe o componente che possa fungere da classe base da cui derivare le classi nuove.
5. Se ci sono sottoclassi, si deve individuare la migliore classe o componente da usare.
6. Determinare quali funzionalità possa fornire incorporando altre classi del frame work.
7. Se si desidera usare classi come attributi del nuovo componente, si devono individuare le classi che possano agire da elementi.
8. Si devono esprimere funzionalmente le proprietà, i metodi e gli eventi del componente e delle strutture sussidiarie.
9. Si devono assegnare gli appropriati livelli di accesso (privati, protetti, interni, pubbliche e analoghe) per ciascuna caratteristica.
10. Si deve testare e correggere il componente. Le caratteristiche aggiunte devono essere testate opportunamente.
11. Ripetere e rifinire il progetto.

### *OSSERVAZIONI*

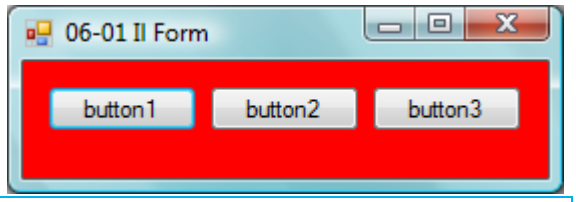
Per estendere la funzionalità di un controllo esistente, è possibile creare un controllo derivato da un controllo esistente mediante ereditarietà. Da un controllo esistente si ereditano tutte le funzionalità e le proprietà visive del controllo. Se si crea, ad esempio, un controllo che eredita da Button, esso avrà lo stesso aspetto e la stessa funzione di un controllo Button standard. La funzionalità del nuovo controllo potrà essere estesa o modificata mediante l'implementazione di metodi e proprietà personalizzati. In alcuni controlli è inoltre possibile modificare l'aspetto visivo del controllo ereditato mediante l'override del relativo metodo OnPaint.

Per creare un controllo completamente personalizzato da utilizzare in Windows Form, ereditarlo dalla classe Control. Benché richieda pianificazione e implementazione più ampie, l'eredità dalla classe Control offre la più vasta gamma di opzioni. Dalla classe Control si eredita la funzionalità di base che consente di attivare i controlli. La funzionalità intrinseca alla classe Control consente di gestire l'input dell'utente tramite tastiera e mouse, definire i limiti e le dimensioni del controllo, fornire un handle di finestre nonché la gestione e la protezione dei messaggi. Non consente di incorporare alcun disegno, ossia il rendering effettivo dell'interfaccia grafica del controllo, né alcuna funzionalità di interazione utente specifica. L'autore dovrà fornire questi aspetti tramite codice personalizzato.

Per combinare la funzionalità di uno o più controlli Windows Form con codice personalizzato, è necessario creare un controllo utente. I controlli utente combinano lo sviluppo rapido dei controlli con la funzionalità standard dei controlli per Windows Form e la versatilità ottenuta dall'aggiunta di proprietà e metodi personalizzati. Quando si crea un controllo utente, viene visualizzata una finestra di progettazione in cui è possibile collocare i controlli standard per Windows Form, che conservano la propria funzionalità naturale nonché l'aspetto dei controlli standard, ma, una volta incorporati nel controllo utente, non sono più disponibili allo sviluppatore nel codice. Il controllo utente esegue il proprio disegno e gestisce inoltre tutta la funzionalità di base associata ai controlli.

## PROGETTO GUIDATO

- Normale Passo Guidato
- Normale Passo Guidato
- Normale Passo Guidato **`parola chiave`** del linguaggio
-




```
private void Form1_Load(object sender, EventArgs e)
{
     Visual C# (Ctrl+Maiusc+#)
}
```

# ESERCIZI

## ESERCIZI SU NUOVI CONTROLLI

Esercizio 1) COMPONENTE PULSANTE LAMPEGGIANTE (TIMER)

Esercizio 2) COMPONENTE PULSANTE CROMO-CANGIANTE

Esercizio 3) COMPONENTE ETICHETTA CROMO-CANGIANTE

Esercizio 4) COMPONENTE PULSANTE ROTONDO

Esercizio 5) COMPONENTE RADIOGROUP (GRUPPO DI RADIOBUTTON)

# SOMMARIO