

Applicazioni OOP con Net Framework

Rev. Digitale 1.0 del 01/09/2016

Applicazioni Multiform

Form SDI	3
Passaggio dei parametri ad una Form	4
Form secondarie	4
Form modali	5
Form MDI.....	5

XML

Introduzione ad XML	6
Struttura di un file XML	6
File XML well formed	7
Import / Export di dati in formato XML	7
Cenni sui DTD	8
Cenni sui Namespace XML	9
Schemi XML	9
Il linguaggio XSL	11
Esempi di Utilizzo di XML	13
Gli oggetti xmlTextReader e xmlTextWriter	13
Gli oggetti xmlDocument e xmlNode	14
Principali proprietà e metodi	15
Scansione e Manipolazione di un albero XML	15
Creazione di un nuovo albero	16
Collegamento Diretto di un file XML con una Data Grid	17
Scansione ricorsiva di un albero strutturato su più livelli	18

Object Oriented Programming

Le classi	19
Le proprietà	20
I Costruttori	21
Membri Statici	22
Classi Statiche	23
Classi Sealed	23
Classi non direttamente istanziabili	23
Metodi Factory	24
Classi Singleton	24
Classi Indexate	25
Classi Annidate	25
Garbage Collector	26
La generazione degli eventi nelle classi: i delegate	27
I Delegate come strumento di gestione degli Eventi	28
Ereditarietà	30
Classi Astratte	33
Eventi nella classe derivata	34
Ereditarietà applicata ai controlli	36
Interfacce	37
Attributi	41

Espressioni Regolari

Oggetto Regex	43
Esempi	43

Office Automation

Utilizzo embedded di msWord	45
Gestione dei paragrafi	45
Range, Selection, find	46
Parametrizzazione delle formattazioni	47
Creazione di una tabella	47
Creazione di un PDF	48
La gestione dei moduli	48
Utilizzo embedded di msExcel	49
Utilizzo del controllo msChart	51

I Thread	53
-----------------------	----

Ado Net

Modello Oggetti ADO NET	55
Proprietà e Metodi di un DataSet	59
Creazione di una DLL	61
Il Controllo Binding Source	62
Crystal Report	64
Classe ado net	66

HTML Help	68
------------------------	----

Applicazioni Multiform

Per aggiungere una nuova form occorre andare su Solution Explorer e fare Tasto Destro / Add / Windows Form. La **STARTUP FORM** è quella lanciata dal main definito all'interno del file Program.cs.

Applicazioni SDI [single document inteface]

Le applicazioni multiform nella maggior parte dei casi sono applicazioni **SDI** che significa che le varie form sono tutte allo stesso livello e sono completamente indipendenti fra loro. Ognuna può visualizzare / nascondere tutte le altre. L'applicazione termina nel momento in cui viene chiusa la forma principale, azione che comporta la chiusura di tutte le form in quel momento aperte.

Per visualizzare una nuova form occorre :

- Istanziare la nuova form
- Renderla visibile mediante il metodo **.Show()**

```
Form2 form2 = new Form2();
form2.Show();
```





Il metodo **Show()** non interrompe il flusso di esecuzione del chiamante, ma :

- attende la visualizzazione della nuova form (sostanzialmente attende il termine dell'evento formLoad)
- procede ad eseguire le istruzioni del chiamante successive al metodo Show();


Visibilità delle Variabili e Procedure di una Form

Una Form contiene al suo interno :

- Il **Costruttore** che può essere pubblico o privato ma deve essere dichiarato senza il void 
`public Form1() {`
Il Costruttore viene richiamato nel momento in cui il chiamante istanzia la nuova form.
Per prima cosa deve richiamare la procedura **InitializeComponent** scritta automaticamente dal designer e che provvede ad istanziare ed inizializzare tutti i controlli trascinati staticamente sulla form a Design Time. Dopo aver richiamato InitializeComponent il costruttore ha già fin da subito la possibilità di andare a scrivere su questi controlli.
- Le **Variabili globali** dichiarate in testa alla form e che prendono il nome di **Proprietà** della form
- Le **Procedure** di elaborazione dei dati che prendono il nome di **Metodi**
- Le **Procedure** associate agli eventi che prendono il nome di **Event Handler**. 

Qualificatori di Visibilità:

Private = Visibili soltanto all'interno della form

Public = Visibili anche all'esterno della form, (cioè visibili da altre form o da altre classi). 

Cotruttore, Proprietà e Metodi possono essere sia **Pubblici** che **Privati**, mentre gli Event Handler sono sempre privati.

- Le Proprietà dichiarate senza qualificatore di visibilità sono per **default Private**. La presenza di Private rende però la dichiarazione più leggibile.. intese le variabili
- I metodi dichiarati senza Qualificatore di Visibilità sono per **default Public**, ma anche in questo caso è meglio sempre anteporre Private / Public

Eventuali Strutture e Variabili Globali all'intera applicazione vengono di solito dichiarate all'interno di una apposita classe dedicata.

Passaggio di parametri ad una nuova form

Il modo più elegante e strutturato per passare dei parametri ad una nuova form è quello di passarli attraverso il costruttore:

Form1

```
Form2 frm;
int n = 15;
frm = new Form2(n);
frm.Show();
```

Form2

All'interno di Form2 occorre definire un nuovo costruttore che si aspetti un intero come parametro.

Questo costruttore per prima cosa deve richiamare il metodo `InitializeComponent()` per istanziare e disegnare tutti i componenti presenti sulla nuova form.

```
private int n;
public Form2(int n) {
    InitializeComponent();
    this.n = n;
}
```



Passaggio dei controlli come parametri

Se la form chiamata ha necessità di accedere a qualche controllo della form chiamante, la form principale può passare questi controlli come parametri al momento dell'istanza. La sintassi è la stessa di prima:

Form1

```
Form2 frm = New Form2(TextBox1);
frm.Show();
```

FORM2:

```
private TextBox txt; // riferimento al TextBox di Form1
public Form2 (t As TextBox) {
    Me.InitializeComponent()
    txt = t; 'salvo il riferimento al TextBox di form1
    txt.Text = "scrivo sul Text Box di Form1"
}
```

Se non si specifica nessun costruttore, viene automaticamente creato un costruttore senza parametri, che si limita semplicemente a richiamare `InitializeComponent()`.

Form SECONDARIE figlie della form principale

Una Form secondaria è una Form figlia di una Form Principale che continua a rimanere visualizzata davanti a quella principale anche quando perde il focus. Questa soluzione rende le form secondarie ideali per implementare finestre di interazione tipo "Trova e Sostituisci" che deve rimanere in primo piano anche quando perde il Focus.

- Le form secondarie vengono chiuse o ridotte a icona quando l'utente chiude o riduce a icona la form principale
- La Form secondaria non viene rappresentata sulla barra delle applicazioni.

Per aprire una form come secondaria di un'altra form occorre, sulla form principale, prima di visualizzare la form figlia, utilizzare il metodo **AddOwnedForm** prima del metodo `Show()`.

```
Form2 form2 = new Form2();
```

```
this.AddOwnedForm(form2);
```

```
form2.Show();
```



All'interno della form secondaria la proprietà **Owner** contiene un riferimento ad una eventuale form genitore :

```
if (this.Owner != null)    MsgBox("Owned by " + this.Owner.Text)
```

Per vedere l'elenco delle form secondarie possedute da una form si può utilizzare la collection **OwnedForms**:

```
foreach (Form frm in this.OwnedForms)    MsgBox(frm.Text);
```

Form Modali

Una form Modale è una form **bloccante**, cioè una form che mantiene il focus fino a quando non viene chiusa.

- Una qualunque form può essere aperta come modale utilizzando il metodo **ShowDialog()** che, a differenza del metodo Show interrompe il codice chiamante fino a quando la form modale non viene chiusa
- Le form aperte come modali **restituiscono come risultato al chiamante il codice del pulsante che l'utente ha utilizzato per chiudere la form modale (DialogResult.OK o DialogResult.Cancel).**
- Per facilitare la restituzione di un risultato al chiamante all'interno delle form modali è possibile trasformare i Pulsanti in **Pulsanti di Dialogo** impostando per ogni pulsante la Proprietà **DialogResult** ai valori Ok, Cancel, etc. Per default la proprietà DialogResult è impostata al valore **none** (pulsante normale)
- **In corrispondenza del click su un pulsante modale**, viene eseguita la seguente sequenza di operazioni:
 - La proprietà **this.DialogResult** della form viene settata al valore di DialogResult del pulsante
 - Viene eseguita la normale routine di evento associata al pulsante
 - La finestra modale viene automaticamente chiusa restituendo **il DialogResult al chiamante**
- In alternativa è possibile assegnare un valore di DialogResult anche **da codice** al termine della procedura di evento di un pulsante o di qualunque altro controllo. A tal fine occorre assegnare il valore desiderato direttamente alla Proprietà DialogResult della Form: **this.DialogResult = DialogResult.OK**
- Se DialogResult è stato imposto nelle Properties, reimpostandolo da codice = **None** si abbate la chiusura.
- Le finestre di dialogo, come tutte le altre finestre, non vengono deistanziate alla chiusura, ma rimangono istanziate finché permane un riferimento valido. Per cui, grazie al fatto che sono bloccanti, dopo la loro chiusura l'utente può tranquillamente andare a leggere il contenuto di tutti i vari membri pubblici.
- **Sulle Finestre Modali in genere si imposta FormBorderStyle = FixedDialog (form non ridimensionabile) MinimizeBox=False, MaximizeBox=False**

Il chiamante potrà leggere il valore di DialogResult al solito modo:

```
Form2 frm = new Form2();
```

```
if (frm.ShowDialog() == DialogResult.Ok) . . . . .
```



La proprietà **Modal** As Boolean indica se la form corrente è stata aperta come modale oppure no. Può essere interpellata nel form Load per applicare stili diversi (ad esempio Non Ridimensionabile se aperta come Modale).

MDI Form

Una form MDI Figlia “vive” soltanto all'interno dell'area client della form genitore. Per trasformare una form in contenitore MDI occorre impostare la proprietà **IsMdiContainer = True**. Una form MDI Container presenta sfondo grigio. La proprietà IsMdiContainer è incompatibile con AutoScroll: non possono essere entrambe True

Form Figlie

La proprietà **IsMdiChild** restituisce True se la form è attualmente figlia di un Container MDI.

Per dichiarare una form figlia di una form MDI occorre impostare la proprietà **MdiParent**, disponibile solo da codice, che è un puntatore alla form genitore. Quando una form MDI Container intende visualizzare una form figlia, prima dello Show, dovrà settare la proprietà MdiParent nel modo seguente:

```
Form2 frm = new Form2();
```

```
frm.MdiParent = this;    frm.Show();
```

Per scandire le form figlie di un MDI Container si può utilizzare il seguente ciclo:

```
foreach (Form frm in this.MdiChildren) { MsgBox(frm.Text); frm.Select(); }
```

In alternativa al Select, la form Container può attivare una delle figlie utilizzando il metodo **ActivateMdiChild** a cui occorre passare il puntatore alla form da attivare: **ActivateMdiChild(this.MdiChildren(2));**

La form MDI Container può anche contenere controlli e le form figlie appariranno in secondo piano rispetto ad essi

XML

XML = eXtensible Markup Language. Linguaggio Estendibile basato su Marcatori (tag)

Raccomandazione W3C (consorzio di aziende che si occupa della definizione degli standard Internet) del 10 febbraio 1998, (derivato dall'ambizioso progetto SGML) e rilasciato come standard il 24 ottobre 2001.

- Non è un linguaggio di programmazione
- E' un **formato di rappresentazione di dati all'interno di un file di testo**, (file di testo strutturato).
- Ogni dato è racchiuso mediante dei Tag che indicano in tipo di contenuto. Ognuno può utilizzare quei tag che ritiene più adatti per la rappresentazione dei propri dati.
- Trattandosi di files testuali, per la loro lettura non serve nessun motore dbms particolare.

Utilizzo di XML:

- Memorizzare informazioni di configurazione (vecchi files .INI, strutturati però soltanto su uno o due livelli)
- Memorizzare database di piccole dimensioni, in un formato semplice e autodescrittivo (files dati)
- Trasportare dati da un ambiente all'altro (alternativa ai files .csv)
- Formato di comunicazione fra applicativi differenti, come ad esempio il **CSV (Comma Separated Values**, ma quale elemento separatore ?)

Allo stato attuale qualsiasi ambiente applicativo (Excel, Access, VB, ASP) consente di salvare i propri dati in formato XML e allo stesso tempo di caricare dati da un file XML.

L'elevata modularità di XML è pagata con una scarsa economicità in termini di spazio di occupazione. Rispetto ad esempio al formato CSV, XML produce dei file molto più prolissi ma allo stesso tempo molto più leggibili.

Struttura di un file XML

Un file XML memorizza i dati attraverso **TAG** ed **ATTRIBUTI** strutturati **ad albero** in modo **gerarchico**, nel senso che i tag risultano annidati uno dentro l'altro. Gli attributi aggiungono informazione al TAG.

```
<?xml version="1.0" encoding="UTF-8" ?>
<elencoFilm>      <!-- Elemento radice -->
  <genere nome="Thriller">
    <film regista="Quentin Tarantino" > "Reservoir Dogs" </film>
    <film regista="Steven Spielberg" > "Duel" </film>
    <film regista="Brian De Palma" > "Blow Out" </film>
    <film regista="Alfred Hitchcock" > "The Birds" </film>
    <film regista="John Carpenter" > "The Thing" </film>
    <film regista="Rob Reiner" > "Misery" </film>
    <film regista="Alfred Hitchcock" > "Psycho" </film>
  </genere>
  <genere nome="Crimine">
    <film regista="Martin Scorsese" > "GoodFellas" </film>
    <film regista="Bryan Singer" > "The Usual Suspects" </film>
    <film regista="Francis Ford Coppola" > "The Godfather" </film>
    <film regista="Brian De Palma" > "Carlito's Way" </film>
  </genere>
  <genere nome="Azione">
    <film regista="Wachowski Bros." > "The Matrix" </film>
    <film regista="Ridley Scott" > "Thelma & Louise" </film>
    <film regista="James Cameron" > "True Lies" </film>
    <film regista="Walter Hill" > "The Warriors" </film>
  </genere>
</elencoFilm>
```

ElencoFilm, **Genere** e **Film** sono **TAG** del file xml. **ElencoFilm** rappresenta la **RADICE** dell'albero (root) **Nome** (del genere) e **regista** (del film) sono **ATTRIBUTI**. Il **titolo del film** rappresenta il valore del tag film (foglie dell'albero). Notare la struttura gerarchica. **ElencoFilm** è l'**archivio**, **Film** sono i **record**, **ulteriormente suddivisi per genere**. Gli attributi del tag film diventeranno ulteriori campi dei record Film.

Formato dei file XML well formed

- La dichiarazione iniziale `<?xml version="1.0"?>` si chiama **processing instruction** o **prologo** ed è facoltativa. Se omessa il documento viene considerato di versione 1.0. Oltre a version, accetta altri attributi quali **Standalone** = "yes" / "no" indica se il documento utilizza o meno uno schema XSD esterno **Encoding** indica il set di caratteri utilizzato, ANSI, UTF-8, UTF-16 (unicode).
- I commenti sono esattamente come in HTML `<!-- ... -->` e possono essere posizionati ovunque fra i tag XML
- Deve sempre esistere un **elemento radice** (root dell'albero) che contiene tutti gli altri, indicato anche come **DocumentElement**. Le dichiarazioni devono essere esterne all'elemento radice.
- Tutti i Tag devono essere Aperti e Chiusi:
`<Tag> contenuto </Tag>`
Qualora il Tag non contenga alcun valore (ma ad esempio soltanto degli attributi), si può utilizzare la cosiddetta chiusura implicita
`< TagVuoto />`
- I Tag devono essere scritti in maniera gerarchica senza sovrapposizioni, cioè i Tag che si aprono per ultimi devono chiudersi per primi, senza alcuna intersezione.
- I Tag devono essere considerati **case-sensitive**, per cui `<TAG>` e `<tag>` sono da considerarsi differenti.
- Gli attributi associati ai Tag devono essere costituiti da coppie **Nome-Valore**. I valori devono essere sempre racchiusi tra virgolette indifferentemente singole o doppie (anche se numerici)
- I seguenti 5 caratteri speciali devono essere scritti nel modo indicato, punto e virgola compreso

<	<
>	>
&	&
'	'
"	"
	&#x?;

L'ultimo caso consente di rappresentare qualsiasi carattere tramite il suo codice ASCII espresso in esadecimale. Ad esempio `€` rappresenta il simbolo €, mentre `®` rappresenta il simbolo @.

I files XML che soddisfano a tutte queste regole si dicono **well-formed**, e sono leggibili da qualunque parser XML.

Un semplice modo per vedere se un documento è well-formed consiste nell'aprirlo con un browser, che contiene al suo interno anche le funzionalità di parser XML. Documenti non well-formed producono un messaggio di errore.

Import / Export di dati in formato XML

Da Access 2002 in avanti, è possibile, selezionata una qualsiasi tabella, mediante tasto destro dare il comando **Esporta in formato XML**. Allo stesso modo ADO NET consente di salvare le informazioni di un intero dataSet (oppure di una singola tabella del dataSet) in formato XML. Ad esempio le istruzioni successive salvano sul file "Squadre.xml" la tabella "tabSquadre" di un certo dataSet:

```
ado.daTable.WriteXml("Squadre.xml")
ado.daSet.Tables("tabSquadre").WriteXml("Squadre.xml")
```

```
// Risultato :
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <tabSquadre> <!-- record1 -->
    <CodiceSquadra> 1 </CodiceSquadra>
    <Nome> Ferrari </Nome>
    <CodiceNazione> 14 </CodiceNazione>
    <Motore> Ferrari </Motore>
    <Pneumatici> Bridgestone </Pneumatici>
  </tabSquadre> .....
</NewDataSet>
```

Come si può vedere, a differenza del caso iniziale dei film, questo documento **xml non contiene attributi ma soltanto tag**. La scelta se utilizzare **tag** o **attributi** è discezionale. Gli attributi occupano meno spazio rispetto ai tag (non hanno il tag di chiusura) ma non possono avere figli. Gestire una informazione mediante un attributo significa non poterla espandere successivamente senza modificare la struttura del file.

NB: per poter importare un file XML in un applicativo tabellare (Excel, Access), occorre necessariamente che:

- Il file XML sia articolato in due livelli (root, Record, Campo)
- I tag siano privi di attributi (che verrebbero semplicemente ignorati)

Nel documento precedente:

i tag di primo livello (es tabSquadre) indicano il **nome della tabella** a cui il record appartiene.

i tag di secondo livello indicano i **campi del record** con i rispettivi valori.

DTD Document Type Definition – Definizione del tipo di documento

Il **DTD** è una vecchia tecnica che consente di definire in modo chiaro la struttura dei dati che dovrà essere utilizzata all'interno del file XML. Oggi i DTD sono stati sostituiti dagli schemi XML, molto più complessi che però consentono, ad esempio, di distinguere tra tipi di dati numeri, stringhe, date, etc.

Se i dati del file XML sono conformi alle regole DTD, si parla di **documento valido** (che è un passo in più rispetto ad un documento **well-formed**, cioè semplicemente privo di errori sintattici.). Numerosi **Parser XML** consentono di validare la correttezza di un file XML sulla base del suo DTD. In questo modo è possibile eseguire un parsing del file prima di eseguire la sua elaborazione, ed evitare così di incorrere in errori dovuti a dati non validi.

Esistono **Parser validanti** (es STG) che verificano se un file XML è conforme allo schema associato, e **Parser non validanti** che verificano semplicemente se il file XML è well formed.

I browser attuali non forniscono più supporto per la validazione di un documento XML tramite DTD.

Il contenuto del DTD può essere scritto direttamente all'interno del file XML oppure può essere salvato in un file con estensione **.dtd** collegato al file XML. Un file **.dtd** è strutturato mediante i seguenti TAG:

Il tag **<!DOCTYPE []>** definisce la RADICE del file e la relativa struttura.

Il tag **<!ELEMENT>** definisce la struttura di ciascun TAG costituente il file

Il tag **<!ATTLIST>** definisce la struttura di ciascun ATTRIBUTO presente all'interno dei TAG

Si consideri il precedente file XML contenente le squadre di F1:

root	squadre
tag di 1 livello	tabSquadre
tag di 2 livello	campi del record squadra

Il file **.dtd** potrebbe essere semplicemente il seguente:

```
<!DOCTYPE squadre[
  <!ELEMENT squadre (tabSquadre+)>
  <!ELEMENT tabSquadre (CodSquadra, Nome, CodNazione, Motore, Pneumatici)>
  <!ELEMENT CodSquadra (#PCDATA)>
  <!ELEMENT Nome (#PCDATA)>
  <!ELEMENT CodNazione (#PCDATA)>
  <!ELEMENT Motore (#PCDATA)>
  <!ELEMENT Pneumatici ANY>
  <!ATTLIST tabSquadre punti CDATA #REQUIRED> ]>
```

L'elemento radice denominato **squadre** può contenere uno o più tag denominati **tabSquadre**

★ significa che l'elemento può essere presente 0 o più volte

? significa 0 o 1 volta)

Se non si inserisce NESSUN simbolo dopo il nome del campo, significa UNA ed UNA sola volta

- Ciascun elemento denominato **tabSquadre** contiene uno ed un solo elemento di tipo **CodSquadra**, uno ed un solo elemento di tipo **Nome**, e così via.
- **(#PCDATA)** significa che l'elemento attuale racchiude un contenuto testuale (foglia). Non è possibile specificare con maggior dettaglio il tipo di dato. Le alternative possibili sono **EMPTY** (tag vuoto) oppure **ANY** (vuoto oppure #PCDATA). Ad esempio il campo Pneumatici non è obbligatorio.
- Il tag **tabSquadre** contiene anche un attributo **punti** indicante i punti finora totalizzati dalla squadra. Questo attributo può avere come valore una qualsiasi combinazione di caratteri (**CDATA**). In alternativa l'attributo potrebbe essere di tipo **ID** (univoco). L'indicazione **#REQUIRED** indica che la presenza dell'attributo è obbligatoria. L'alternativa potrebbe essere **#REQUIRED** cioè facoltativo. Sintassi completa su : http://www.w3schools.com/dtd/dtd_attributes.asp

Per collegare il documento DTD al file XML occorre utilizzare la seguente direttiva, da scriversi DOPO la direttiva di versione:

```
<!DOCTYPE squadre SYSTEM "mioFile.dtd">
```

Cenni sui Namespace XML

Visto che ognuno può assegnare ai tag XML il nome che vuole, c'è il rischio di incrociare files in cui esistono tag con lo stesso nome. L'attributo **xmlns** (xml name space) applicabile a qualsiasi tag xml, consente di associare ai tag un **namespace che in fase di parsing verrà anteposto al nome di quel tag** e di tutti i suoi figli.

Il namespace rappresenta in sostanza un **contenitore di nomi**, espresso in genere in forma di **URL**, ma ciò non implica che debba necessariamente esistere su Internet un sito con quel namespace. Si tratta semplicemente di un nome univoco di identificazione.

```
<Prodotto xmlns="http://www.RobertoManaProdotti.it/Prodotti">
  <Codice> 20 </Codice>
  <Prezzo> 120 </Prezzo>
</Prodotto>
```

E' anche possibile assegnare namespace differenti a tag differenti di uno stesso documento

```
<Ordine xmlns="http://www.RobertoMana.it/Ordini" // sintassi base
  xmlns:mieiProdotti="http://www.RobertoMana.it/Prodotti">
<mieiProdotti:Prodotto>
  <mieiProdotti:Codice> 23 </mieiProdotti:Codice>
  <mieiProdotti:Prezzo> 120 </mieiProdotti:Prezzo>
</mieiProdotti:Prodotto> // sintassi alternativa
<mieiClienti:Cliente xmlns:mieiClienti="http://www.RobertoMana.it/Clienti">
  <mieiClienti:Codice> 74 </mieiClienti:Codice>
  <mieiClienti:Nome> Mario Serra </mieiClienti:Nome>
</mieiClienti:Cliente>
```

Schemi XML

Gli schemi XML con estensione **.XSD** (Xml Schema Definition) sono una evoluzione delle DTD che consentono di definire la **grammatica utilizzata dal file XML**, in modo da poter consentire un controllo sulla correttezza dei dati.

Lo schema XML è in pratica un modello che definisce il **dominio di un file XML**, definendo ad esempio quali devono essere i nomi di tag e attributi da utilizzare ed il **tipo di dati** dei vari tag ed attributi.

.xdr è una vecchia versione di schema XML utilizzata da Microsoft ma in seguito abbandonata.

Lo schema XML è esso stesso un documento XML che utilizza dei tag specifici che iniziano con **<xs:.....>**.

Il tag principale è **<xs:schema> </xs:schema>** che racchiude l'intero schema.

Visual Studio Net dispone di un editor visuale che consente di salvare come schema XML una struttura creata mediante semplici tabelle relazionali.

Esempio di schema XML

I file di schema iniziano di solito nel modo seguente :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"> dove:
```

schema è il nome dell'elemento radice (root) che racchiude l'intero schema.

xmlns è un attributo che dichiara il namespace che si intende utilizzare.

"http://www.w3.org/2001/XMLSchema" è il namespace standard del W3C per i files XSD.

E' un sito in cui viene presentata un descrizione dello standard XSD

Il seguente schema XSD è ottenuto come esportazione da un dataset di ADO NET, mediante le seguenti istruzioni:

```
ado.daTable("tabSquadre").WriteXmlSchema("Squadre.xsd")
ado.daTable("tabSquadre").ReadXmlSchema("Squadre.xsd")
```

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="NewDataSet" xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSet" msdata:IsDataSet="true" msdata:Locale="it-IT">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="tabSquadre">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CodiceSquadra" type="xs:unsignedByte" minOccurs="0"/>
              <xs:element name="Nome" minOccurs="0">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:maxLength value="30" />
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
              <xs:element name="CodiceNazione" type="xs:unsignedByte" minOccurs="0"/>
              <xs:element name="Motore" type="xs:string" minOccurs="0" />
              <xs:element name="Pneumatici" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1" msdata:PrimaryKey="true">
      <xs:selector xpath="//tabSquadre" />
      <xs:field xpath="CodiceSquadra" />
    </xs:unique>
  </xs:element>
</xs:schema>
```

Associazione di uno schema XSD ad un documento XML

Un file XML che intende far riferimento al file XSD, (e che non contiene la definizione di specifici namespace nel rootTag), deve richiamare il file XSD all'interno del tag ROOT nel modo seguente, dove **xsi** =xml schema instance.

```
<Squadre xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="Squadre.xsd" >
```

dove "http://www.w3.org/2001/XMLSchema-instance" è il namespace di riferimento per i fogli schema-instance, cioè i fogli XML che implementano uno schema XSD. **noNamespaceSchemaLocation** indica in pratica che il file si trova nella cartella corrente. In alternativa è possibile specificare un **NamespaceSchemaLocation** ben preciso che indica il percorso dove andare a recuperare lo schema XSD.

Approfondimento: Il linguaggio XSL

I CSS consentono di definire una fine grafica personalizzata per la presentazione di un documento XML all'interno di un browser, ma presentano dei limiti che possono essere così riassunti:

- Non è possibile estrarre singoli dati da un documento (vengono sempre visualizzati tutti i dati)
- Non è possibile inserire nella presentazione contenuti aggiuntivi rispetto a quelli contenuti nel file XML
- Non è possibile modificare la struttura di un documento

Per ovviare a questi limiti, nel 1997 W3C ha standardizzato il linguaggio **XSL** (eXtensible Stylesheet Language) cioè un **linguaggio di stile** standard, evoluzione dei CSS, per la presentazione dei documenti XML.

Si tratta in pratica di un meta-linguaggio che consente di scandire un documento XML (mediante una tipica scansione ad albero) e trasformarlo in un altro documento con formato differente, tipicamente una pagina HTML con una certa impaginazione, ma eventualmente anche una pagina RTF o PDF. Da questo punto di vista XML diventa un formato intermedio **standard** per la memorizzazione dei dati, dati che possono poi essere facilmente tradotti, mediante XSL, in più formati differenti.

Ad esempio, nel momento in cui si esporta una tabella di Access in XML, Access consente anche la creazione automatica di un file XSL che, legato al file XML dei dati, consente di visualizzare i dati XML mediante una classica TABLE HTML (che è poi quello che fa l'oggetto adoNet dataTable nel momento in cui visualizza in modo tabellare una tabella internamente memorizzata come albero XML).

Si consideri ad esempio il solito file "squadre.xml" e si aggiunga come seconda riga del file il seguente codice:

```
<?xml-stylesheet type="text/xsl" href="squadre.xsl" ?>
```

Il file XSL è un **file XML** in contenente dei tag HTML più le istruzioni XSLT che indicano come trasformare il documento XML a cui XSL si riferisce e **quali dati visualizzare**. Il file XSL viene inviato al Browser insieme al file XML e la trasformazione XSL viene eseguita dal Browser al momento della visualizzazione (ad esempio IE 5.0 in avanti, mediante il noto processore **MSXML**). Notare però che la trasformazione XSL non è necessariamente legata ad un Browser, ma può essere eseguita con un qualunque processore XSLT, cioè un software che sappia interpretare le istruzioni XSLT.

XSL è costituito da due componenti:

XSLT (XSL Transformation) che è il linguaggio vero e proprio per la trasformazione della pagina XML XPath che definisce le espressioni utilizzate da XSLT per l'identificazione dei nodi a cui applicare le trasformazioni

Esempio

Visualizzazione dei nomi delle squadre.

Aggiungere ad un progetto ASP il seguente file XSL ((Add New Item / **XSLT File**) :

```
<xsl:template match="/">
  <html>
  <body>
    <xsl:for-each select="NewDataSet/tabSquadre">
      <h2>
        <xsl:value-of select="Nome"/> &#x20;
        <xsl:value-of select="Pneumatici"/>
        <xsl:value-of select="@Attributo"/>
      </h2>
    </xsl:for-each>
  </body>
</html>
</xsl:template>
```

L'elemento Template

L'attributo `match` del tag `Template` indica il sottoalbero a cui devono essere applicate le regole di trasformazione contenute nel file XSL. Normalmente si utilizza `match="/"` (dove `/` sta per nodo radice) e significa che le regole devono essere applicate a tutti i figli dell'albero. I template eseguono una elaborazione guidata dai dati del documento sorgente (file XML) e sono pertanto abbastanza difficili da utilizzare. Se si imposta ad esempio

```
match = "NewDataSet/SerieA"
```

vengono visualizzati tutti i nodi figli di `NewDataSet`, ma le regole vengono applicate soltanto ai nodi "SerieA". La scelta migliore è quella di utilizzare **sempre** `match = "/"` e poi filtrare i record mediante i cicli successivi.

Il tag <xsl:for-each>

E' il cuore di XSL. E' l'istruzione che "cerca" i dati da visualizzare ed opera su di essi. L'attributo **`select`** consente di impostare una espressione XPath che indica quali nodi devono essere visualizzati (partendo sempre dalla root). All'interno del `for-each` si possono utilizzare tutti i tag HTML. E' possibile annidare più cicli `for-each`.

Il tag <xsl:value-of />

Consente di visualizzare valori e attributi del nodo corrente. Utilizzabile frammisto ai tag HTML

Il tag <xsl:attribute />

Consente di aggiungere un attributo al tag HTML antecedente. Ad esempio

```
<a>
    <xsl:attribute name="href">
        http://<xsl:value-of select="link"/>
    </xsl:attribute>
    <xsl:value-of select="link"/>
</a>
```

dove `link` è un campo che contiene un collegamento ipertestuale (ad esempio `www.google.it`).

Il tag <xsl:if />

Consente di eseguire un test per valutare quali nodi visualizzare rispetto al recordset corrente. L'esempio seguente mostra soltanto le squadre che hanno pneumatici Michelin.

```
<xsl:for-each select="NewDataSet/tabSquadre">
    <xsl:if test="Pneumatici='Michelin'">
```

Il tag <xsl:sort />

Consente di ordinare il recordset restituito da `for-each`. Deve essere specificato immediatamente dopo il `for-each`.

```
<xsl:sort order="ascending" select="Nome"/>
```

Espressioni XPath (utilizzabili all'interno di for-each e value-of)

T	Nome del nodo accessibile attraverso il nodo corrente (es <code>NewDataSet</code>)
T1/T2	Nodi T2 figli di T1 (<code>NewDataSet/tabSquadre</code> significa "tutti i nodi <code>tabSquadre</code> figli di <code>NewDataSet</code> ")
T1//T2	Nodi T2 discendenti anche non diretti di T1 (<code>NewDataSet/tabSquadre</code> significa "tutti i nodi <code>tabSquadre</code> discendenti di <code>NewDataSet</code> ". Funziona anche se tra <code>NewDataSet</code> e <code>tabSquadre</code> ci sono dei nodi intermedi)
T1 T2	Nodi T1 oppure Nodi T2
T[3]	Terzo figlio del nodo T
..	Genitore del nodo attuale
@	Indica un attributo del nodo corrente @nomeAttributo
/	Nodo radice
//	Un nodo discendente dalla radice. In pratica qualsiasi nodo.

```
<xsl:for-each select="NewDataSet/tabSquadre[Nome='Williams']">
<xsl:for-each select="NewDataSet/tabSquadre[@Attributo='FF']">
<xsl:value-of select="count(NewDataSet/tabSquadre)"/> //fuori dai cicli
<xsl:if test="position() != last()">
```

Esempi di utilizzo di XML

```
using System.xml;
```

Lettura testuale di un file XML mediante l'oggetto XmlTextReader

L'oggetto **reader** as XmlTextReader consente di scandire, uno alla volta, tutti i nodi dell'albero xml

Il metodo **reader.Read()**, ad ogni invocazione, legge sequenzialmente il prossimo nodo sul file di testo (tag, valore o attributo) *caricandolo* dentro l'oggetto reader

La proprietà **reader.NodeType** restituisce la tipologia del nodo corrente. Le principali tipologie possibili sono:

- 1 = Element = Tag in apertura
- 2 = EndElement = Chiusura Tag
- 3 = Text = Valore del tag

Notare come il valore di un tag sia anch'esso considerato un **nodo** dell'albero, figlio del tag da cui dipende. Mentre i tag sono nodi *Element*, i valori sono nodi *Text*.

La proprietà **reader.Name** restituisce il contenuto di un nodo di tipo *Element* (in pratica restituisce il nome del tag)

Nel caso di nodi di tipo *Element* (tag) la proprietà Value è stringa vuota.

La proprietà **reader.Value** restituisce il contenuto di un nodo di tipo *Text* (es "Michelin").

Nel caso di nodi di tipo *Text* la proprietà Name è stringa vuota. Name e Value sono mutuamente esclusive.

La proprietà **reader.HasAttributes** indica se il nodo corrente dispone di attributi. Mediante un ciclo da zero a **reader.AttributeCount-1** si può scorrere il valore dei vari attributi dell'elemento. Per accedere all'attributo *i*-esimo si può utilizzare il metodo **reader.GetAttribute(i)**. Si può accedere al valore dell'attributo anche in modo diretto mediante il nome dell'attributo stesso: **reader.GetAttribute** ("NomeAttributo").

Considerando ad esempio un file XML ad un solo livello, il codice seguente visualizza tutti i tag di apertura con il rispettivo valore (ad esempio nel formato *NomeTag=Valore*). A tal fine si può fare un select case sul NodeType:

```
Dim reader As XmlTextReader
reader = New XmlTextReader(Server.MapPath("squadre.xml"))
reader.WhiteSpaceHandling = WhiteSpaceHandling.None 'Disabilito i WhiteSpace
reader.read() 'Leggo l'intestazione
reader.read() 'Leggo il tag root
While (reader.Read())
    Select Case reader.NodeType
        Case XmlNodeType.Element
            Response.Write(reader.Name & "=") 'reader.Value in questo momento è vuoto
        Case XmlNodeType.Text
            Response.Write(reader.Value & "<br>") 'dopo ogni value va a capo
    End Select
End While
reader.Close()
```

L'esempio seguente mostra invece tutti i nodi (indipendentemente dal tipo) contenuti in un file .xml.

```
While (reader.Read())
    R.Write(reader.NodeType.ToString & " -- " & reader.Name & reader.Value & "<br>")
    If (reader.HasAttributes) Then Response.Write(reader.GetAttribute(0) & "<br>")
End While
reader.Close()
```

Scrittura di un file XML mediante l'oggetto XmlTextWriter

```
Dim writer As XmlTextWriter
writer = New XmlTextWriter(Server.MapPath("Squadre2.xml"), Nothing)
writer.WriteStartDocument() 'intestazione xml
writer.Formatting = Formatting.Indented
writer.Indentation = 3
```

```

writer.WriteStartElement("NewDataSet")
'While (. . . . .)
    writer.WriteStartElement("tabSquadre")
    writer.WriteElementString("CodiceSquadra", "1")
    writer.WriteElementString("Nome", "Ferrari")
    writer.WriteElementString("CodiceNazione", "14")
    writer.WriteElementString("Motore", "Ferrari")
    writer.WriteStartElement("Pneumatici")
    writer.WriteAttributeString("tipo", "bagnato")
    writer.WriteString("Bridgestone")
    writer.WriteEndElement()      '</pneumatici>'
    writer.WriteEndElement()      '</tabSquadre>'
'End While
writer.WriteEndElement()          '</NewDataSet>'
writer.Flush()
writer.Close()
End Sub

```

WriteStartDocument scrive la dichiarazione iniziale <?xml version="1.0"?>

Formatting può essere Indented oppure None (nessuna indentazione, tutto sulla prima colonna)

WriteStartElement scrive un tag in apertura

WriteString scrive il valore (text) relativo all'ultimo tag aperto

WriteEndElement chiude l'ultimo tag ancora aperto

WriteElementString genera un nodo completo, dove i tag di apertura e chiusura sono specificati dal primo parametro, mentre il valore è specificato dal secondo parametro.

WriteAttributeString scrive un attributo completo (nome – valore) all'interno dell'ultimo tag aperto.

Flush() forza la scrittura su file. Se omissa la scrittura viene comunque forzata dal close().

Se il file xml non esiste viene automaticamente creato. Se esiste viene sovrascritto. Il 2° parametro del costruttore indica il set di caratteri da utilizzare. Se non viene specificato (nothing) si intende il set UTF-8.

Gli oggetti XmlDocument e XmlNode

Un modo più strutturato per accedere ad un file xml è quello di utilizzare gli oggetti standard XmlDocument e XmlNode previsti dal DOM XML e supportati da tutte le applicazioni che accedono ai file xml. Questi oggetti consentono il caricamento di un file XML all'interno di un **albero** XML e la relativa navigazione dell'albero.

Proprietà e Metodi dell'oggetto XmlDocument

Il metodo **Load** esegue la lettura dell'intero file XML, caricandolo in memoria mediante una struttura ad albero.

Il metodo **LoadXml** consente di leggere la struttura (tag) XML **da una stringa anziché da un file**.

Anziché creare l'albero, può risultare più veloce creare una semplice stringa XML (str = . . . += . . . += . . .) e poi fare xmlDoc.**LoadXml**(str) per caricare l'intera stringa sull'albero.

I metodi **Save** e **SaveXml** salvano la struttura Xml su file o su stringa

La proprietà **DocumentElement** restituisce il primo nodo reale dell'albero (nodo radice), di tipo XmlNode

Il metodo **CreateNode**("NomeNodo") crea un nuovo nodo generico di tipo XmlNode slegato dall'albero.

Il metodo **CreateElement**("NomeNodo") crea un nuovo nodo di tipo XmlElement slegato dall'albero.

Il metodo **CreateTextNode**("valore") crea un nuovo nodo di tipo XmlText sempre slegato dall'albero.

Il metodo **CreateAttribute**("Nome", "Valore") crea un nuovo attributo.

- L'oggetto **XmlNode** indica un nodo generico (Element oppure Text), mentre gli oggetti **XmlElement** e **XmlText** indicano nodi specifici rispettivamente di tipo Tag e di tipo Testo (foglia del tag). Hanno in pratica tutti e tre le stesse proprietà e gli stessi metodi. **XmlAttribute** indica un attributo
- L'istruzione xmlDoc.**Save**(Response.Output) consente di inviare l'albero XML non al file, ma direttamente a chi ha eseguito la richiesta della pagina, che può essere il browser client (che visualizzerà l'albero a video mediante una sintassi predefinita) oppure una applicazione utente come ad esempio Flash.

Proprietà e Metodi dell'oggetto XmlNode

I tag di chiusura non vengono in pratica considerati come appartenenti all'albero.

La proprietà **NodeType** restituisce la tipologia del nodo corrente. Le possibili tipologie sono le solite:

1 = Element = Tag in apertura, 2 = EndElement = Chiusura Tag, 3 = Text = Valore del tag

La proprietà **Name** restituisce il nome di un TAG di tipo *Element*

La proprietà **Value** restituisce il contenuto di un nodo di tipo *Text* (foglia)

La proprietà **InnerText** = "valore" consente di associare un valore ad un nodo di tipo *Element* (in pratica consente di associare direttamente una foglia ad un nodo)

La proprietà **Item**("Nome") restituisce il FirstChild del nodo *Element* corrente che presenta il Nome ricevuto come parametro.

La proprietà **HasChildNodes** indica se il nodo corrente ha nodi figli (notare che, come per xmlTextReader, i valori di un tag sono considerati **nodi figli** del tag stesso).

Il metodo **FirstChild** restituisce il primo nodo figlio del nodo corrente

Il metodo **NextSibling** passa al prossimo fratello del nodo corrente. Nothing se non esistono ulteriori fratelli

Il metodo **LastChild** restituisce l'ultimo nodo figlio del nodo corrente

Il metodo **PreviousSibling** ritorna al precedente fratello del nodo corrente. Nothing se non esistono ulteriori fratelli

Il metodo **ParentNode** ritorna al nodo genitore.

La proprietà **childNodes()** indica la collezione dei figli del nodo corrente.

Per accedere al valore dell'i-esimo figlio occorre scrivere **childNodes(i).value**. **.length** = n di elementi

La proprietà **PreserveWhiteSpace** True/False se impostata a False fa sì che durante il parsing gli "a capo" vengano ignorati. (sostituisce la vecchia proprietà **IgnoreWhite** con significato invertito).

Il metodo **AppendChild(Node)** appende un nodo figlio (già creato di tipo *Element* o *Text*) al nodo corrente

Il metodo **RemoveChild(childNode)** elimina il nodo figlio (ricevuto come parametro) del nodo corrente, lui e tutti gli eventuali nipoti, di qualunque ordine e grado.

La collezione **Attributes[]** restituisce una lista degli attributi del nodo, di cui si possono leggere **Name** e **Value**. **Attributes.count** indica il numero di attributi contenuti in **Attributes**

Il metodo **.SetAttribute**("Nome", "Valore") imposta un attributo al nodo corrente (nodo di tipo *Element*)

Il metodo **.GetAttribute**("Nome") restituisce il valore dell'attributo avente il nome indicato

Esempio di scansione di un albero XML

Caricamento e Scansione dell'albero relativo alle squadre della Formula 1. Questo file, come del resto spesso accade, ha soltanto due livelli (tag "record" e tag "campi"), e può dunque essere scandito mediante 2 semplici cicli annidati (il ciclo esterno scandisce i record, quello interno i campi):

```
Dim s As String = ""
Dim doc As New XmlDocument
doc.Load(Server.MapPath("squadre.xml"))

Dim root As XmlElement
Dim record As XmlElement
Dim campo As XmlElement
root = doc.DocumentElement
record = root.FirstChild

While (Not record Is Nothing)
    campo = record.FirstChild
    While (Not campo Is Nothing)
        s = s & campo.Name & "----" & campo.ChildNodes(0).Value & "<br>"
        campo = campo.NextSibling
    End While
    s = s & "<br>"
    record = record.NextSibling
End While
lblMsg.Text = s
```

root = Elemento radice. Attenzione che se c'è un tag di dichiarazione, FIRST CHILD accede ad esso, mentre DOCUMENT ELEMENT accede al vero nodo radice

Creazione di un nuovo albero xml

La creazione di un nuovo albero xml si rende necessaria ogni volta che si deve esportare in XML (ad esempio per passarli a Flash) una sequenza di dati provenienti da un database che devono essere letti mediante un ciclo ed appesi ad un nuovo albero. L'esempio seguente mostra la creazione diretta di un semplice albero xml, con un la creazione di un solo nodo da appendere alla radice.

```
Sub Page_Load(ByVal obj As Object, ByVal e As EventArgs)
    Dim xmlDoc As New XmlDocument
    Dim root As XmlElement 'XmlElement è comunque un nodo
    root = xmlDoc.CreateElement("tips")
    xmlDoc.AppendChild(root)

    Dim nodo As XmlElement
    nodo = xmlDoc.CreateElement("tip") 'Nome del nodo
    nodo.InnerText = "Valore del nodo"
    nodo.SetAttribute("codice", "1")
    root.AppendChild(nodo)

    xmlDoc.Save("filename")
End Sub
```

Aggiunta di un nuovo record ad un albero XML esistente

```
Sub Page_Load(ByVal obj As Object, ByVal e As EventArgs)
    Dim xmlDoc As New XmlDocument
    xmlDoc.Load(Server.MapPath("squadre.xml"))
    Dim root As XmlNode = xmlDoc.Item("NewDataSet") 'nodo radice

    Dim record As XmlElement
    record = xmlDoc.CreateElement("tabSquadre") 'crea un nuovo tag
    root.AppendChild(record)

    Dim campo As XmlElement
    campo = xmlDoc.CreateElement("CodiceSquadra")
    campo.SetAttribute("nome", "ferrari") 'Aggiunta di un attributo al tag
    record.AppendChild(campo)

    Dim foglia As XmlText
    foglia = xmlDoc.CreateTextNode("77") 'Crea un valore testo
    campo.AppendChild(foglia) 'Associa il valore testo al tag

    'in alternativa si poteva semplicemente utilizzare la proprietà InnerText
    campo.InnerText = "77"

    xmlDoc.Save(Server.MapPath("squadre.xml"))
End Sub
```

Modifica di un nodo esistente

I nodi contenuti all'interno dell'oggetto xmlDoc possono ovviamente essere modificati. Quando si salverà l'albero anche le modifiche verranno salvate.

```
if (node.Value == "Williams") node.Value = "New Williams";
```

Eliminazione di un nodo esistente

I nodi contenuti all'interno dell'oggetto xmlDoc possono essere eliminati nel modo seguente :

```
node1.RemoveChild(node2);
```

dove **node2** è un riferimento ad un nodo di livello 2 figlio del nodo di livello1 puntato da **node1**

Collegamento diretto di un file XML a livello singolo con una Data Grid

Dato un file XML strutturato soltanto con nodi di primo livello e con tutti i campi espressi come attributi

```
<ElencoDipendenti>
  <Dipendente nome="Oliva" eta="25" residenza="fossano"></Dipendente>
  <Dipendente nome="Carlo" eta="26" residenza="torino"></Dipendente>
</ElencoDipendenti>
```

è possibile utilizzare il controllo **XmlDataSource** per caricare l'intero file XML all'interno di una dataTable direttamente collegabile ad un oggetto DataGridView. A tal fine aggiungere alla form il seguente controllo (ovviamente disponibile anche sulla ToolBox) :

```
<asp:XmlDataSource ID="xmlDs" runat="server"
DataFile="Nomi.xml"></asp:XmlDataSource>
```

Aggiungere quindi alla griglia l'attributo **DataSourceID** che specifica il nome del controllo che funge da Data Source per la griglia: `<asp:GridView ID="Grid" runat="server" DataSourceID="XmlDs">`

Ancora più velocemente, direttamente dal Design, si può selezionare col mouse la freccina in alto a destra sulla Griglia e poi fare "Chose Data Source". Selezionando "New Data Source" si apre un wizard che consente di ricercare direttamente il file xml da utilizzare come data source, provvedendo poi a generare automaticamente tutto il codice necessario e impostando già nella griglia la proprietà `AutoGenerateColumns = False` con i relativi Bound Field, in modo che il programmatore possa poi personalizzare la visualizzazione dei vari campi.

Collegamento diretto di un file XML a due livelli con una Data Grid

In modo simile all'esempio precedente, se il file XML è articolato su due livelli senza attributi (cioè nel formato prodotto dal metodo `dataTable.writeXml()`) in cui:

- Tutti i nodi figli diretti del nodo radice contengono il nome di una tabella
- I tag di secondo livello rappresentano i nomi dei campi, senza attributi

allora è possibile caricare l'intero file XML all'interno di un oggetto DataSet / DataTable, a sua volta collegabile ad una griglia. Infatti in DOT NET, **i dati di un DataSet sono internamente memorizzati in formato XML**. Il DataSet, a differenza dei normali oggetti xml, fornisce semplicemente una vista dei dati di tipo differente.

A tal fine occorre utilizzare un oggetto di libreria **xmlDataDocument** (simile al precedente oggetto **xmlDocument**), che in pratica consente di vedere i dati provenienti da un file xml come un vero e proprio DataSet.

```
Sub Page_Load(ByVal obj As Object, ByVal e As EventArgs)
    Dim xmlDoc As New XmlDataDocument
    xmlDoc.DataSet.ReadXml(Server.MapPath("squadre.xml"))
    DataGrid1.DataSource = xmlDoc.DataSet.Tables(0)
End Sub
```

Questa metodologia funziona anche nel caso in cui il file XML contenga più nodi di primo livello con nomi differenti (in pratica più tabelle) e **anche in presenza di attributi** che vengono gestiti come una tabella di memoria indipendente in relazione 1:N con la tabella principale.

Il metodo `DataSet.ReadXml()`, durante la lettura del file xml, trasforma automaticamente il **modello gerarchico** del file xml in **modello relazionale**, creando di conseguenza tutte le chiavi esterne necessarie a realizzare il modello relazionale. Le regole generali seguite per creare il DataSet (che in questo caso sarà composto ovviamente da più tabelle) sono le seguenti:

- Tutti i nodi figli diretti del nodo radice diventano tabelle
- Tutti gli elementi (tag) che contengono altri elementi (tag) diventano tabelle
- Tutti gli elementi (tag) che dispongono di attributi vengono trasformati in tabelle aggiuntive; all'interno della tabella principale viene aggiunta una chiave esterna che punta alla tabella secondaria appena creata.
- Qualsiasi altra cosa diventa una colonna

Scansione ricorsiva di un albero strutturato su più livelli

Se il file XML presenta più livelli di annidamento oppure non si conosce a priori il numero dei livelli di annidamento, occorre necessariamente eseguire un algoritmo ricorsivo tipico delle scansioni di un albero:

```
Private str As String
Sub Page_Load(ByVal obj As Object, ByVal e As EventArgs)
    Dim doc As New XmlDocument
    doc.Load(Server.MapPath("squadre.xml"))
    ShowTree(doc.DocumentElement)
    msg.Text = str
End Sub

Sub ShowTree(ByVal node As XmlNode)
    Visualizza(node)
    If (node.HasChildNodes) Then
        node = node.FirstChild
        While Not IsNothing(node)
            ShowTree(node)
            node = node.NextSibling
        End While
    End If
End Sub

Sub Visualizza(ByVal node As XmlNode)
    Dim lstAttributi As XmlNamedNodeMap
    Dim attr As XmlNode
    If Not (node.HasChildNodes) Then
        ' se non ha figli è sicuramente un valore (nodeType = 3)
        str += " = " & node.Value
    Else
        ' se ha figli è sicuramente un tag (nodeType = 1)
        str += "<br><b>" & node.Name & "</b>"
        If (node.NodeType = XmlNodeType.Element) Then
            lstAttributi = node.Attributes
            If (lstAttributi.Count > 0) Then
                ' gli attributi vengono racchiusi tra parentesi [ ]
                str += " ["
                For Each attr In lstAttributi
                    str += attr.Name & " = " & attr.Value & " "
                Next
                str += "]"
            End If
        End If
    End If
End Sub
```

OOP : Le Classi

Visual Studio supporta tutte le caratteristiche tipiche dei linguaggi object oriented. Una classe può essere definita in qualsiasi file sorgente. Più classi possono essere contenute nello stesso file.

La form è anch'essa una classe, suddivisa però in due file:

- Il file relativo a **Designer**, gestito automaticamente dall'ambiente di sviluppo, in cui vengono istanziati tutti i controlli trascinati sulla Form e le relative inizializzazioni eseguite a Design Time all'interno delle Properties.
- Il file di **Codice**, contenente le procedure di gestione degli eventi, più Proprietà e Metodi personalizzati che l'utente può definire all'interno della classe Form. Il file di codice può contenere in coda anche classe aggiuntive oltre alla form, che però deve essere la classe iniziale.

Qualificatori di visibilità dei membri di una classe

PRIVATE membro visibile solo all'interno della classe stessa e che non viene ereditato dalle sottoclassi.

PROTECTED come private, ma ereditato dalle sottoclassi.

PUBLIC membro pubblico sempre visibile ed ereditabile.

FRIEND internal in C#. membro visibile al di fuori della classe, ereditabile ma visibile solo all'interno dell'Assembly corrente e **non modificabile** dalla classe che eredita.

Il qualificatore di visibilità predefinito per i membri di classe è il seguente:

- **PRIVATE** per i **Campi** (compresi i campi di Struttura). ES: `string nome; //private`
- **PUBLIC** per **Property e Metodi**

Nella dichiarazione dei campi è possibile utilizzare tutto ciò che si utilizza normalmente per le variabili:

- Dichiarazioni multiple sulla riga (poco leggibili)
- Inizializzatori

Qualificatori di visibilità di una classe

- Una classe **Private** è visibile soltanto all'interno del container nel quale è stata dichiarata (al limite il file).
- Il qualificatore di visibilità predefinito per una classe è **Friend**.

L'Overloading dei Metodi

L'Overloading di un metodo consiste nello scrivere **più procedure con lo stesso nome ma con firme differenti** (different signature). Per firma differente si intende un differente numero di parametri, ma anche solo parametri di tipo differente. Non è invece consentito creare due funzioni di Overload che differiscono solamente nel tipo del valore restituito. (il Run Time, sulla base dei parametri, non saprebbe quale delle due funzioni richiamare). Per aggirare questo problema, invece di restituire un valore, si può passare un parametro aggiuntivo per riferimento.

L'overload è molto comodo per poter passare variabili di tipo differente ad una stessa funzione.

Si supponga di voler rintracciare un libro all'interno di un database passando indifferentemente come parametro il **Codice numerico** oppure il **Titolo** (ammesso che sia univoco). A tal fine si può definire il seguente overloads:

```
public Libro trova (int index) { }  
public Libro trova (string titol) { }
```

L'overload può essere applicato anche a funzioni definite all'interno di un modulo o di una Struttura.

Quando si sta richiamando un metodo da codice, la tecnologia **IntelliSense** (tooltip giallo) è in grado di riconoscere correttamente i metodi di overload e mostra un elenco di tutte le firme possibili.

La parola chiave **Overloads** di VB non è prevista in C#.

Nota: L'**overload** è una tecnica alternativa e più strutturata rispetto ai parametri opzionali. I parametri opzionali possono creare problemi di compilazione quando il codice client e la classe richiamata appartengono ad assembly differenti e sono, pertanto, compilati separatamente.

Le PROPERTIES

Una Property è un campo con controllo sugli accessi. Sul campo vengono mappate due funzioni:

- ❑ una funzione get per il controllo degli accessi in lettura
- ❑ una funzione set per il controllo degli accessi in scrittura

La variabile base deve essere dichiarata come PRIVATE e la Property con i controlli sull'accesso deve essere dichiarata nel modo seguente:

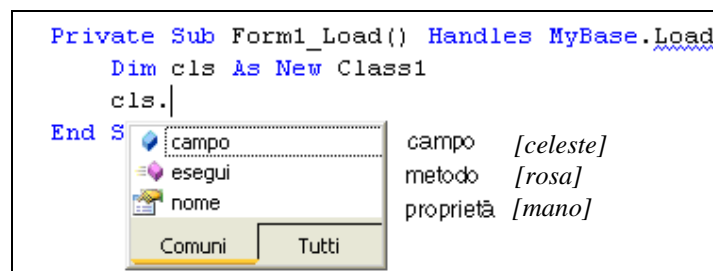
```
private string _nome = string.Empty;
public string nome {
    get {
        return _nome;
    }
    set {
        if (isValid(value))
            _nome = value;
        else
            throw new Exception("Nome NOK");
    }
}
```

value rappresenta il valore che l'utente ha assegnato alla Property. Per valori non validi viene sollevata una eccezione.

Note:

- A differenza di VB, in C# le Properties **NON** supportano l'Overloads (esattamente come i campi).
- Se la classe contiene un campo con lo stesso nome del parametro, il parametro nasconde la variabile esterna; è però possibile accedere alla variabile esterna anteponendo il **this**: `this.nome=nome`
- Il metodo set **NON** può mai modificare il parametro che gli è stato passato.

La figura seguente mostra come Visual Studio rappresenta **Campi, Metodi e Proprietà** :



Campi e Proprietà Read Only

Per definire un **campo** di sola lettura occorre utilizzare la parola chiave **readonly**.

```
public readonly int n;
```

Nel caso delle Property occorre

- omettere il costrutto Set (
- omettere la parola **readonly** che si utilizza SOLO per i campi)

Una Property ReadOnly è simile ad una Costante Pubblica però, a differenza delle costanti, il valore della Property ReadOnly può essere assegnata Run Time dal costruttore della classe.

Campi e Proprietà Write Only

Allo stesso modo è possibile definire Property di sola scrittura omettendo la procedura **get**.

Il valore assegnato alla Property può essere elaborato esclusivamente all'interno della procedura **set**.

Utile ad esempio per l'inserimento e la validazione di una password.

Proprietà vettoriali

Supponiamo di voler gestire come proprietà un **vettore di 10 stringhe**. Si può utilizzare il seguente codice in cui l'**indice** di lettura e scrittura nel vettore viene gestito dal NET Framework in modo **completamente trasparente**:

```
private string [] _vect = new string[10];
public string [] vect {
    get { return _vect; }
    set { _vect = value; }
}
```

Il client richiamerà questa Property nel modo seguente:

```
myClass cls = new myClass();
cls.vect[3] = "pippo";
MessageBox.Show(cls.vect[3]);
```

I Costruttori

Il costruttore è un metodo particolare che viene eseguito nel momento in cui l'utilizzatore provvede ad istanziare la classe. Il costruttore di una classe :

- deve avere lo stesso nome della classe
- Non può restituire un risultato
- Nella dichiarazione **occorre omettere il void**

Consente di eseguire eventuali inizializzazioni sull'oggetto nel momento in cui questo viene creato.

```
class Person {
    private readonly DateTime createTime = Convert.ToDateTime ("1/1/1800")
    private string nazionalita = string.Empty;
    public Person (string nazionalita) {
        MessageBox.Show("Si sta creando una nuova istanza della classe");
        if (nazionalita != string.Empty)
            this.nazionalita = nazionalita ;
        createTime = DateTime.Now ;
    }
}
```

```
Person person = new Person("Italia");
```

- Il Costruttore è l'unica procedura di una classe che può assegnare un valore ad un campo ReadOnly
- Il costruttore viene richiamato **dopo** che tutti gli **inizializzatori sono già stati eseguiti**.
- Se non si specifica nessun costruttore esplicito, C# crea automaticamente un costruttore implicito vuoto.

Overloads dei costruttori

Come per metodi è possibile eseguire l'overloading anche dei costruttori. Per ogni classe si possono cioè definire più costruttori con parametri differenti i quali possono richiamare tutti una stessa procedura privata, passandogli eventuali parametri non ricevuti dal costruttore come valori diretti di default.

Richiamo di un costruttore da un altro costruttore

Un costruttore può richiamare un altro costruttore, però questa chiamata deve essere la prima istruzione del costruttore (esattamente come in VB). Questa tecnica viene spesso utilizzata per scrivere il codice del costruttore soltanto all'interno del costruttore più completo (**quello con più parametri**), mentre gli altri costruttori si limitano ad invocare il costruttore più completo, passandogli eventuali parametri aggiuntivi con dei valori di default.

Per richiamare un costruttore dall'interno di un altro costruttore, in C# si utilizza una sintassi molto particolare :

Subito dopo la dichiarazione del costruttore occorre inserire il carattere : e poi, tramite this, richiamare subito il nuovo costruttore, prima ancora dell'aperta graffa.

```
public Person()  
    : this("Italia") {  
    MessageBox.Show("!!");  
}
```

Membri Statici di una classe

Proprietà e Metodi definiti in precedenza sono indicati come **Membri di Istanza** della classe, nel senso che sono riferiti ad una singola istanza della classe. In contrapposizione ai Membri di Istanza, all'interno di una classe è possibile dichiarare, tramite la parola chiave **static**, dei **Membri Statici** o **Membri di Classe** che non sono legati ad una singola istanza, ma che saranno condivisi fra tutte le istanze della classe.

- Le costanti sono considerate membri statici, anche senza esplicitare la parola chiave **static**
- I membri di istanza possono accedere ai membri statici (condivisi) della classe ma non viceversa.
- Quando si istanzia una classe, i membri statici non vengono allocati insieme all'istanza, ma vengono collocati nell'area "comune" insieme al codice dei metodi che compongono la classe.
- Tutti i membri statici di una classe vengono automaticamente istanziati in corrispondenza del primo fra i seguenti eventi:
 - primo accesso ad un membro Statico della classe
 - creazione di una istanza della classe

I membri statici di una classe possono essere invocati anche senza istanziare la classe, ma soltanto premettendo il nome della classe. Ad esempio tutte le funzioni della classe System.Math sono metodi Statici invocabili senza dover istanziare un oggetto Math.

```
ris = System.Math.Sqrt(x);
```

Esempio:

```
class Fattura {  
    private static int cntFatture = 0;  
    public readonly int nFattura;  
    public Fattura() {  
        cntFatture += 1;  
        nFattura = cntFatture;  
    }  
    public static void visualizza() {  
        MessageBox.Show("Numero fatture eseguite : " + cntFatture); // OK  
        MessageBox.Show("Numero fattura corrente : " + nFattura); // NOK  
    }  
}
```

Il campo **cntFatture** è statico, dunque visibile da tutte le varie istanze successivamente create. Questo campo viene incrementato in occasione di ogni nuova istanza e viene utilizzato per gestire in modo incrementativo il numero della prossima fattura.

Costruttori statici

All'interno di una normale classe possono coesistere membri statici e membri di istanza. Allo stesso modo possono coesistere **costruttori di istanza** (cioè quelli tradizionali fino ad ora utilizzati) e **costruttori statici**. I costruttori statici :

- devono essere preceduti dalla parola chiave **static**
- **Non** possono avere parametri
- Sono intrinsecamente **privati** (ma la parola chiave private **deve essere omessa**).

Il costruttore statico viene richiamato in corrispondenza in corrispondenza del primo fra i seguenti eventi:

- primo accesso ad un membro Statico della classe
- creazione di una istanza della classe (**prima del costruttore tradizionale**).

Può essere utilizzato per **inizializzare da codice i campi Statici della classe**. Il fatto però che non possa ricevere parametri rende di fatto impossibile l'esecuzione di inizializzazioni dinamiche parametrizzate. Va bene ad esempio per eseguire l'azzeramento di un vettore.

Classi Statiche

Classi che contengono esclusivamente membri statici. Sono definite mediante la parola chiave **static** e sono classi **non direttamente istanziabili** e **sealed** (però non è detto il viceversa). Sono dichiarate statiche tutte quelle classi che fungono da container per funzioni di libreria, come ad esempio System.Math.

```
static class classeStatica {  
    public static int n;
```

- All'interno di una classe statica **non** è possibile definire dei costruttori di istanza, né pubblici né privati. E' però possibile definire un costruttore statico (che per sua natura è privato e privo di parametri).
- La classe statica viene interamente istanziata in corrispondenza del primo accesso ad uno dei suoi membri
- Una classe statica è **non istanziabile** ed è anche **non ereditabile** (sealed).

Nota: I **moduli** sono equivalenti a classi Statiche. Un modulo è una classe dotata di un costruttore Private nascosto ed i cui membri sono contrassegnati implicitamente come Statici. Per accedere ai membri di un modulo occorre anteporre il nome del modulo. In VB c'è un import nascosto che consente l'utilizzo diretto dei membri del modulo

Classi sealed (non ereditabili)

La parola **sealed** è una parola chiave utilizzata in C++ che significa "sigillato". Le classi sealed sono classi che, per scelta del progettista, **non possono essere ereditate**. Per il resto sono classi "normali", dunque istanziabili. Un esempio di classi sealed è rappresentato dalle strutture. Il fatto che le classi **statiche** siano anche **sealed** non comporta affatto il viceversa, cioè che le classi sealed siano necessariamente statiche.

Le classi sealed sono definite mediante la parola chiave **sealed** (**NotInheritable** in VB).

```
sealed class classeSealed {  
    public static int n;  
    public classeSealed() {  
        MessageBox.Show("Costruttore di istanza di una classe sealed");  
    }  
}
```

Classi non direttamente istanziabili

Per rendere una classe **NON** – istanziabile è sufficiente definire solo **costruttori privati**. Il fatto che le classi **statiche** siano non istanziabili, non comporta il viceversa, cioè che una classe non istanziabile sia necessariamente statica.

Metodi Factory

Si definisce Factory un metodo che ha come scopo quello di **creare una nuova istanza di una classe** e restituire tale istanza al chiamante **senza che questo debba richiamare direttamente il costruttore**.

Rispetto all'istanza tradizionale tramite costruttore, i metodi Factory offrono il vantaggio di poter eseguire codice personalizzato prima di creare effettivamente l'istanza. I costruttori servono ad istanziare un oggetto, ma, una volta avviati, non possono "fermarsi". Qualora venissero riscontrati degli errori nei parametri di creazione dell'istanza, il costruttore creerebbe comunque un nuovo oggetto, ma molto probabilmente quest'ultimo conterrebbe dati errati. Un metodo Factory, invece, controlla che tutto sia a posto **prima** di creare il nuovo oggetto: in questo modo, se c'è qualcosa che non va, lo può comunicare al programmatore (o all'utente), ad esempio lanciando un'eccezione o visualizzando un messaggio di errore.

Normalmente i Metodi Factory sono **metodi statici** (altrimenti non si potrebbero richiamare prima della creazione dell'oggetto), definiti all'interno della stessa classe che corrisponde al suo tipo di output. In questo caso il costruttore può anche essere dichiarato **private** in modo da rendere la classe non direttamente istanziabile. Però potrebbe anche istanziare classi differenti dotate di costruttore esplicito (tipico caso di class over class).

Esempio:

```
public class Quadrato {  
    private int lato;  
    private Quadrato (int val) { this.lato = val; }  
    public static Quadrato creaQuadrato(int val) {  
        return new Quadrato(val);  
    }  
}
```

Il Metodo Factory creaQuadrato istanzia (richiamando lui il costruttore privato) e **restituisce** un nuovo oggetto Quadrato senza che il Client abbia possibilità di richiamare esplicitamente il costruttore. Esempio di Main :

```
Quadrato q = Quadrato.creaQuadrato(5);
```

Classi Singleton

Classi delle quali può esistere soltanto una singola istanza. Sono classi di questo tipo Console, System.GC (Garbage Collection), System.Environment (ambiente del SO), Application (Applicazione corrente).

L'implementazione più semplice di questo pattern può essere realizzata mediante un **costruttore privato**, che impedisce l'istanza diretta della classe, e un **metodo factory statico** che, in corrispondenza della richiesta di istanza, verifica se la classe è già istanziata o meno. Se non è ancora istanziata la istanza, altrimenti restituisce il puntatore all'istanza esistente. L'istanza può essere creata **preventivamente** oppure **alla prima chiamata del metodo factory** (*lazy initialization*), memorizzandone il riferimento in un attributo privato anch'esso statico.

```
public class Singleton{  
    private Singleton() { // costruttore privato  
        // eventuali inizializzazioni di istanza  
    }  
    private static Singleton _instance=null;  
    public static Singleton GetInstance() {  
        if(_instance==null) _instance=new Singleton();  
        return _instance;  
    }  
}  
  
int main() {  
    Singleton st = Singleton.GetInstance();  
    st.visualizza();  
}
```

Al momento dell'istanza, il chiamante può passare eventuali parametri al metodo factory in modo da inizializzare eventuali proprietà di istanza. Una seconda richiesta di istanza in pratica reinizializza l'oggetto.

Classi Indaxate (Indexed class)

In C# è possibile definire classi di tipo indicizzatore, cioè classi in grado di istanziare contemporaneamente una intera collezione di oggetti.

```
class myClass {
    private string[] myData;
    public myClass(int size) {                // costruttore
        myData = new string[size];
        for (int i = 0; i < size; i++)
            myData[i] = "empty";
    }
    public string this[int pos] {              // iteratore
        get { return myData[pos]; }
        set { myData[pos] = value; }
    }
}
```

L'iteratore diventa automaticamente la **Proprietà Predefinita** della classe.

```
int main() {
    myClass vect = new myClass(10);
    vect[1] = "pippo";
}
```

Il costruttore crea sostanzialmente 10 istanze di un certo oggetto (nel caso in questione una stringa).

La situazione è simile ma non identica alla seguente:

```
Obj [] vect = new Obj [10];
```

che però crea un vettore di **10 puntatori non inizializzati**.

Classi Annidate

DOT NET consente la **composizione diretta** delle classi, (rappresentata in UML mediante un rombo vuoto). Cioè consente l'annidamento delle classi una dentro l'altra (es si pensi agli oggetti della classe Font).

```
Class Outer {
    . . . . .
    Class Inner {
        . . . . .
    }
}
```

- Il codice all'interno della classe Outer può sempre istanziare oggetti della classe Inner, indipendentemente dal qualificatore di visibilità di quest'ultima (Private, Public, Friend).
- Le classi interne, **se dichiarate utilizzando un qualificatore diverso da Private**, possono essere istanziate anche dall'esterno di Outer (o dall'interno di classi sorelle di Outer) utilizzando la seguente sintassi:
Outer.Inner obj = new Outer.Inner();

Normalmente però le classi sono definite separatamente e, all'interno di una classe, si può definire un riferimento ad un'altra classe (**composizione indiretta**, rappresentata in UML mediante un rombo pieno), riferimento (o maniglia) che sarà poi istanziato dal costruttore.

Il rilascio di un oggetto : Il Garbage Collector

Per evitare i rischi di Memory Leak dovuti a oggetti rimasti allocati, Dot Net utilizza un approccio estremamente innovativo per il rilascio della memoria allocata.

In NET la **creazione** di un oggetto provoca sempre l'allocazione di un blocco di memoria all'interno del *managed heap*. Per assegnare un riferimento ad un oggetto, si memorizza all'interno del **riferimento** l'indirizzo a 32 bit dell'oggetto stesso posizionato all'interno della managed heap .

Per quanto riguarda il **rilascio** dell'oggetto, in DOT NET non esiste un metodo **delete**, ma per rilasciare un riferimento occorre semplicemente scrivere 0 (**null** in C#, Nothing in VB) all'interno del puntatore, senza preoccuparsi di deallocare l'istanza. La rimozione degli oggetti dalla managed heap si basa su un sofisticato oggetto del NET Framework denominato **GC Garbage Collection** (Raccolta della spazzatura) che:

- ❑ Viene lanciato automaticamente tutte le volte che si tenta di allocare memoria per un nuovo oggetto e la heap non ha sufficiente memoria libera. Per applicazioni di piccole dimensioni nelle quali non si raggiunge mai la saturazione della heap, la Garbage Collection non viene in pratica mai eseguita e tutti gli oggetti rimangono in vita fino al termine dell'applicazione.
- ❑ Può essere avviato manualmente da codice mediante l'apposito metodo **GC.Collect()**. Siccome GC è un processo piuttosto pesante è bene invocare il metodo Collect solo quando l'applicazione è inattiva (es in attesa di un input) e solo se le GC inattese sono fonte di rallentamento per l'applicazione.

La Garbage Collection analizza tutti gli oggetti presenti nella heap e individua quelli ancora in uso da parte dell'applicazione, deallocando quelli non più in uso. Il processo controlla anche i **riferimenti circolari** fra oggetti referenziati indirettamente. Quindi compatta l'heap e rende i blocchi di memoria disponibili per nuovi oggetti.

Il Distruttore

All'interno delle varie classi è possibile definire un **metodo distruttore** (in VB denominato **Finalize**), metodo che però non viene richiamato in corrispondenza del reale rilascio dell'istanza, ma viene invocato **soltanto** dal Garbage Collector nel momento in cui sta per deallocare l'oggetto dalla memoria.

```
~MiaClasse() {  
    Debug.WriteLine("L'obj sta x essere deallocato")  
}
```

Bisogna fare attenzione a che cosa si richiama dal distruttore, in quanto si rischia di richiamare oggetti che potrebbero già essere stati deallocati o dalla Garbage Collection oppure dalla chiusura in corso dell'applicazione. E' considerato sicuro l'accesso all'oggetto padre (**base**) in quanto l'oggetto figlio viene deallocato sempre prima del padre. L'oggetto **Debug** è sempre l'ultimo oggetto ad essere deallocato nel ciclo di vita di una applicazione.

Il metodo GC.Collect()

L'utilizzo diretto del metodo GC.Collect() è sconsigliato in quanto estremamente pesante.

Se però si decide di invocare manualmente il metodo GC.Collect(), occorre subito dopo invocare anche il metodo **GC.WaitForPendingFinalizers()** che arresta il thread corrente fino a che non è stato eseguito il distruttore di tutti gli oggetti da deallocare.

Il metodo Dispose()

Se l'oggetto rilasciato utilizzava risorse condivise anche da altre applicazioni, come file, connessioni a database, porte seriali e parallele, è bene che queste vengano liberate il prima possibile, senza attendere il GC, in modo da risultare disponibili anche per le altre applicazioni.

Il metodo **Dispose** è un metodo standard, realizzato mediante l'implementazione di una interfaccia, (quindi comune a tutti gli oggetti) che dovrebbe essere inserito manualmente all'interno di ogni classe allo scopo di rilasciare le risorse condivise **nel momento stesso in cui l'ultima istanza della classe sta per essere deallocata**.

A differenza del distruttore, invocato automaticamente, in questo caso **il Client dovrà essere lui a preoccuparsi di richiamare il metodo Dispose()** della classe **prima di rilasciare l'ultimo riferimento alla classe**

Il metodo Dispose di un oggetto deve :

- invocare il metodo Dispose di tutti gli oggetti istanziati in modo che possano rilasciare eventuali risorse
- rilasciare l'ultimo puntatore alle varie istanze.

Ad esempio se l'oggetto Person ha istanziato al suo interno un oggetto Timer, il metodo Dispose della classe Person dovrà invocare prima il metodo Dispose dell'oggetto Timer e poi rilasciare il puntatore.

Se il metodo Dispose è un **override** al termine occorre richiamare *base.Dispose()*.

La particolarità del metodo Dispose() è che viene derivato dall'**Interfaccia IDisposable**, che espone un unico metodo Dispose(), e che dunque sarà comune a tutti gli oggetti che implementano quell'interfaccia.

```
Class MiaClasse : IDisposable
    public void Dispose ( )    {    }
```

La Generazione degli eventi nelle classi

I Delegate

Tutti gli eventi NET sono implementati internamente attraverso l'utilizzo dei Delegate, l'analogo delle funzioni di Callback del C++. **Il Delegate definisce la firma (signature) di un metodo richiamabile da un altro metodo.**

Tramite i delegate è possibile **usare i metodi come oggetti** e passarli ad un'altro metodo (il delegate) che li utilizza mascherandone il comportamento. Si tratta di un concetto simile ai puntatori a funzione del'ANSI C, nel senso che consente di invocare una procedura attraverso un puntatore alla procedura stessa.

Rispetto però ai puntatori a funzione del'ANS C, i Delegate sono più sicuri in quanto sono

- **type-safe** cioè sono tipizzati e quindi il metodo invocato dal delegate ne deve rispettare la signature (firma) ovvero deve avere lo stesso numero e tipo di parametri e deve tornare lo stesso tipo definito nella dichiarazione del delegate
- **object oriented**, cioè i delegate sono classi e come tali devono essere definiti e poi istanziati.
- Un Delegate può richiamare **sia** metodi statici **sia** metodi di istanza.

Per utilizzare i Delegate occorre fare tre cose:

- Dichiararli
- Instanziarli
- Richiamarli

Dichiarazione del delegate

```
public delegate void writeHandler(string msg);
```

Il tipo restituito può anche essere diverso da void. Questa riga definisce in realtà una vera e propria *classe* (creata dietro le quinte dal compilatore), classe denominata `_ writeHandler` che eredita dalla classe `System.Delegate`. Questa dichiarazione è del tutto equivalente alla definizione di una classe mediante l'istruzione **class** e pertanto può essere scritta :

- all'inizio del file, prima delle classi, (**preferibile**) con un qualificatore **public** o **private**
- all'interno di una classe, sempre con visibilità **public** o **private** (cioè visibile solo nella classe)

Istanza del delegate

Una volta definita la classe di tipo delegate denominata `_ writeHandler`, è possibile, all'interno di una qualsiasi procedura, dichiarare un riferimento alla nuova classe, e poi istanziarla.

```
writeHandler write = new writeHandler (screenWrite);
```

dove `screenWrite` è una generica procedura utente che deve avere la *stessa firma* del delegate e che si occupa di stampare a video il messaggio ricevuto.

write è una istanza del delegate; in sostanza è un riferimento che sta puntando alla funzione `screenWrite`.

Richiamo della procedura tramite il delegate

Una volta istanziato il delegate, si può invocare la procedura puntata dal delegate semplicemente utilizzando il nome dell'istanza seguito dai parametri da passare alla funzione.

```
write("Salve a tutti");
```

Se anziché stampare a video si decide di utilizzare una altra funzionalità di output, ad esempio stampare su file oppure su carta, è sufficiente modificare l'istanza del delegate e farlo puntare ad una diversa procedura che vada a stampare su file oppure su carta. Alta portabilità

```
write = new writeHandler (fileWrite);
```

I Delegate multicast

I delegate multi cast derivano da una classe apposita, la classe **System.MulticastDelegate** e consentono di richiamare sequenzialmente più procedure, aventi però sempre la stessa firma. Per aggiungere ulteriori metodi alla lista d'invocazione, si usano l'operatore +=.

Ad es. per far sì che l'oggetto **write** scriva sia sul video sia sulla stampante, è sufficiente aggiungiamo alla sua lista d'invocazione entrambe le procedure precedenti:

```
writeHandler write;  
write += new writeHandler (screenWrite);  
write += new writeHandler (printerWrite);
```

I delegate come strumento di gestione degli eventi

Esempio 1

Si vuole fare in modo che, al termine del salvataggio di un record su file, una certa classe generi un evento *onSalvataggioEseguito* a cui verrà passato come parametro il numero d'ordine (ID) di quel record. Il main potrà utilizzare questo evento per dare un messaggio di conferma all'utente. I passi da eseguire sono i seguenti:

Dichiarazione del delegate

```
public delegate void salvataggioEseguitoEventHandler(int id);
```

Dichiarazione dell'evento all'interno della classe

```
public event salvataggioEseguitoEventHandler salvataggioEseguito;
```

Questa riga dichiara sostanzialmente un **riferimento** al delegate **salvataggioEseguitoEventHandler**. Questo riferimento al momento non è inizializzato, cioè non punta a nessuna procedura effettiva.

Generazione dell'evento

Alla fine del metodo **SalvaSuFile**() di salvataggio dei dati, occorre aggiungere la riga di generazione dell'evento:

```
salvataggioEseguito(codice) ;
```

Intercettazione degli eventi dal programma principale

Affinché il client possa gestire gli eventi generati dalla classe, dopo aver istanziato la classe occorre collegare dinamicamente la procedura di gestione dell'evento posizionata nel main con il relativo evento della classe. Cioè in termini più tecnici **occorre fare in modo che il delegate della classe punti alla procedura del main dedicata alla gestione dell'evento**. Come sottolineato più volte tale procedura dovrà avere la stessa firma del delegate definito all'interno della classe. In C# NON è ammesso il costrutto **WithEvents** che in VB consentiva di assegnare gli eventi al riferimento invece che all'istanza.

```
private void button_Click(object sender, EventArgs e)
{
    Fattura f = new Fattura();
    f.salvataggioEseguito += new salvataggioEseguitoEventHandler(miaProcedura);
    .....
    f.salvaSuFile();
}

void miaProcedura(int n) {
    MessageBox.Show("fattura " + n + " salvata correttamente");
}
```

In pratica un evento è una procedura definita all'interno della classe, ma con il codice di gestione scritto non dentro la classe, ma dentro il client, che lo può gestire a suo piacimento.

Esempio 2

Questo secondo esempio fornisce **due** ulteriori miglioramenti rispetto al semplice esempio precedente:

- Alla procedura di evento viene assegnata la **firma tipica** (Object sender, EventArgs e)
- All'interno del metodo che deve generare l'evento (nell'esempio precedente il metodo SalvaSuFile) anziché richiamare direttamente il delegate di gestione dell'evento, viene richiamata una funzione intermedia che una eventuale classe derivata potrà eventualmente modificare aggiungendo dei contenuti **prima** del verificarsi dell'evento, e potrà anche richiamare direttamente per sollevare l'evento.

La seguente classe genera un evento di trigger quando il campo count raggiunge un valore max prefissato.

```
public class OverMaxEventArgs : EventArgs {
    private int _valoreSoglia;
    public int valoreSoglia { // Property visibile all'interno del parametro e
        get { return _valoreSoglia; }
    }
    public OverMaxEventArgs(int num) { // costruttore
        this._valoreSoglia = num;
    }
}

public delegate void OverMaxEventHandler(object sender, OverMaxEventArgs e);
public class Counter {
    public event OverMaxEventHandler OverMax;
    public int count=0;
    private int maxVal;
    public Counter (int n) { // costruttore a cui passo il valore di trigger
        if (n>1000) // 1000 è la soglia massima
            throw new ArgumentException("max Value = 1000");
        else maxVal=n;
    }
    public void increment() {
        count++;
        if (count >= maxVal) {
            OverMaxEventArgs e = new OverMaxEventArgs(count);
            OnOverMax(this, e);
        }
    }
    protected virtual void OnOverMax(object sender, OverMaxEventArgs e) {
        if (OverMax != null) { // se OverMax punta ad una funzione del main
            OverMax(this, e);
        }
    }
}

L'ultimo metodo è protected, dunque ereditabile ma non visibile fuori della classe

// main
Counter cnt = new Counter(30);
cnt.OverMax += new OverMaxEventHandler(miaProcedura);
private void miaProcedura(object sender, OverMaxEventArgs e) {
    MessageBox.Show("Reached: " + e.valoreSoglia.ToString());
}
```

Ereditarietà (Inheritance)

Possibilità di ereditare una nuova classe (classe derivata o **sottoclasse**) da una classe di base più semplice (**superclasse**). La classe derivata eredita tutte le proprietà e metodi della classe di base, sia quelli di istanza che quelli statici.

All'interno della classe ereditata è possibile :

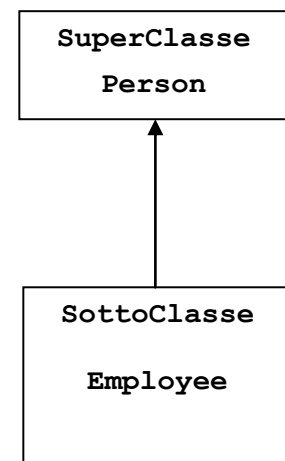
- **Definire nuovi membri (Proprietà / Metodi)**
- **Ridefinire Proprietà e Metodi ereditati (utilizzando sempre la stessa firma).**

E' possibile ereditare da qualsiasi oggetto, inclusi quelli per i quali non si possiede il codice sorgente. DOT NET gestisce però soltanto l'eredità singola, cioè non è possibile ereditare da più classi di base.

Per derivare una classe da un'altra si utilizzano i **DUE PUNTI**:

```
class Person {  
    public string nome;  
    public string cognome;  
    private DateTime dataNascita;  
    public string nomeCompleto( ) {  
        return nome + " " + cognome;  
    }  
}
```

```
class Employee : Person {  
    public double stipendioBase;  
    public int oreSttimanali;  
    private double _incentivoOrario;  
    public double incentivoOrario {  
        get {  
            return _incentivoOrario;  
        }  
        set {  
            _incentivoOrario = value;  
        }  
    }  
    public double stipendioTotale( ) {  
        return stipendioBase + incentivoOrario * oreSttimanali;  
    }  
}
```



La sottoclasse eredita tutti i membri della classe base (campi, proprietà, metodi ed eventi) tranne:

- i membri Private
- i Costruttori

La sottoclasse deve avere un qualificatore di visibilità uguale o inferiore rispetto alla superclasse

Nota sui membri privati

Quando una nuova classe eredita da una superclasse, è come se la superclasse venisse 'inglobata' all'interno della sottoclasse. Il fatto che i membri privati non vengano ereditati, non significa che essi non esistano all'interno della sottoclasse, ma significa che i nuovi metodi aggiunti o ridefiniti all'interno della sottoclasse non vi possono accedere (nemmeno attraverso gli overrides).

Essi infatti sono a tutti gli effetti esistenti all'interno della sottoclasse, ma solo i metodi della superclasse possono accedere a Proprietà e Metodi Privati della superclasse stessa.

Up Casting

E' sempre possibile **copiare un riferimento a una sottoclasse -> in un riferimento a superclasse** (anche nel caso di eredità indiretta, cioè di classi che a loro volta ereditano da Employee). Proprietà e Metodi della superclasse sono infatti sicuramente tutti disponibili all'interno della sottoclasse.

Esempio:

```
Employee e = new Employee();  
e.nome = "Mario" ; e.cognome = "Rossi"; e.stipendioBase = 100;  
Person p = (Person) e; // non obbligatorio  
MessageBox.Show(p.nomeCompleto()); //Mario Rossi
```

Nell'esempio p è un puntatore a **Person** che in realtà punta all'impiegato Mario Rossi. L'up casting è sempre possibile, in quanto qualunque impiegato è sicuramente una persona. In altre parole **tutte le classi che derivano da una classe di base, possono essere utilizzate come se fossero la classe di base.**

Ovviamente nel momento in cui viene utilizzato un riferimento alla classe di base, proprietà e metodi aggiuntivi non risultano visibili, a meno di eseguire un nuovo down casting.

- Nel caso dell'upCasting, il cast esplicito non è obbligatorio.

Ad esempio quando ad una procedura non si sa quale oggetto verrà passato (ad esempio ipotizziamo che verrà passato un controllo generico), allora si può prevedere per quella procedura un parametro di tipo **Object** (oggetto generico) oppure **Control** (Controllo, derivato da Object), dopo di che quando il chiamante eseguirà la chiamata alla procedura eseguirà un **up casting** (da Button a Object) del controllo da passare alla procedura.

Se una procedura si aspetta un parametro di tipo Object, a quella procedura è possibile passare qualunque cosa!

Down Casting

L'operazione inversa (**down Casting**) non sempre è possibile, in quanto non è detto che una persona sia necessariamente un impiegato. Il down casting va a buon fine solo se il riferimento alla superClasse punta effettivamente ad un oggetto di tipo sottoClasse, cioè se la persona è effettivamente un impiegato.

Dato il riferimento p dell'esempio precedente, si potrebbe rieseguire un down casting nel modo seguente:

```
Employee e2 = (Employee) p;
```

dove il riferimento p in realtà stava puntando ad un Employee. Questo è esattamente ciò che succede all'interno di un evento quando si fa il cast del **sender** da Object generico a Button o altro controllo. In tal caso prima del cast, per evitare di andare in errore, occorre necessariamente eseguire un controllo per vedere se il down casting è possibile

Concetto di Late Binding

Object è la classe più generica di DOT NET, dalla quale derivano tutte le altre classi. Talvolta, invece di dichiarare una variabile come tipizzata, può essere comodo dichiarare la variabile come Object e poi tipizzarla soltanto più tardi (**late binding**) al momento della necessità. Questo va bene ad esempio nell'interfacciamento con Office e non si sa bene quale versione di Office si andrà ad utilizzare. Esempio :

```
Object o;  
o = "Salve mondo"; // late binding = Tipizzazione Ritardata  
MessageBox.Show("The variable data type is: " + o.GetType().ToString());  
o = Convert.ToDateTime("5/11/2011"); // string  
MessageBox.Show("The variable data type is: " + o.GetType().ToString());  
// dateTime
```

Ridefinizione di Proprietà e Metodi : OVERRIDE

All'interno della sottoclasse è possibile ridefinire Proprietà / Metodo ereditati (sempre con la stessa firma; in caso di firma diversa si tratta sostanzialmente di un nuovo metodo, per cui non ha nemmeno senso parlare di override).

Per poter ridefinire una Proprietà / Metodo :

- nella classe base il metodo deve essere dichiarato **virtual** (Overridable in VB)
- nella classe derivata, davanti al nome del metodo, occorre scrivere **override**

Classe base

```
public virtual string nomeCompleto( )  
{ return nome + " " + cognome; }
```

Classe derivata

```
public override string nomeCompleto()  
{ return nome + " " + cognome + " " + stipendioBase; }
```

- Un metodo **non** dichiarato **virtual** non è sovrascrivibile.
- Un metodo contrassegnato come **override**, è esso stesso ridefinibile, **senza le necessità di aggiungere anche virtual**, che anzi deve essere omissso.
- Un metodo contrassegnato come **override** che non si vuole che sia ulteriormente overridable, deve essere contrassegnato come **sealed override**. (sovrascrive ma NON è ulteriormente sovrascrivibile).
- La classe derivata che ha sovrascritto un metodo, può comunque accedere al metodo originale della classe base facendo uso della parola chiave **base.nomeMetodo()**. **Verso l'esterno però il nuovo metodo sostituisce completamente il metodo originale, che non è più in nessun modo accessibile**, nemmeno quando si fa un up cast a superclasse (vedasi esempio successivo)
- *Al posto di **override**, all'interno della classe derivata, si può usare la parola chiave **new** (facoltativa ma consigliata) che esegue lo **shadowing** del metodo originale, nel senso che si affianca al metodo originale e, dall'esterno, saranno ora entrambi accessibili. Il metodo originale sarà al solito accessibile dall'interno della classe derivata tramite **base.nomeMetodo()**, ma rimarrà accessibile anche dall'esterno come indicato nell'esempio. A differenza di **override**, lo **shadowing** è possibile anche se il metodo base non è stato definito **VIRTUAL** nella superclasse. Un metodo dichiarato **new virtual** nella sottoclasse potrà subire override.*
- Un metodo **static** non può essere definito anche **virtual**, dunque non è sovrascrivibile. Però è possibile eseguire lo shadowing tramite **new** ed affiancargli un nuovo metodo statico.

Esempio di override/ shadowing

```
Employee e = new Employee();  
e.nome = "Mario" ; e.cognome = "Rossi"; e.stipendioBase = 100;  
Person p = (Person) e;  
MessageBox.Show(e.nomeCompleto()); //Metodo della classe derivata : Mario Rossi 100  
MessageBox.Show(p.nomeCompleto()); //Metodo della classe base : Mario Rossi
```

Se nella classe derivata (Employee) il metodo nomeCompleto() è stato definito come **new**, significa che il nuovo metodo si affianca a quello originale ma **solo nella classe derivata** per cui **p.NomeCompleto()** utilizzerà il metodo della classe base visualizzando Mario Rossi **senza** il 100.

Se nella classe derivata (Employee) il metodo nomeCompleto() è stato definito come **override**, significa che **sovrascrive completamente quello originale**, dunque **p.NomeCompleto** visualizza anche il 100

Inoltre, all'interno della classe derivata:

- **Non è consentito modificare l'ambito di visibilità** di un metodo ridefinito. Se ad esempio una classe di base contiene un metodo Protected non è consentito ridefinirlo come Private o Public. Questo ha lo scopo di assicurare che se un riferimento a superclasse (Person p) punta ad una sottoclasse (Person p = e), essa possa invocare senza problemi tutti metodi eventualmente ridefiniti all'interno della classe derivata.
- **Non è possibile modificare gli eventuali attributi Read Only e Write Only** (cioè se una proprietà era Read Only, deve rimanere Read Only anche nella classe derivata)
- Se all'interno della classe derivata si aggiunge un nuovo metodo con lo stesso nome ma firma diversa, non è necessario (anzi è vietato) utilizzare **override** in quanto il metodo rappresenta un nuovo membro della classe
- Non abusare della ridefinizione. Il compilatore genera codice più efficiente quando invoca metodi non ridefinibili, per i quali spesso viene effettuato l'**inlining**, lo spostamento del codice dal chiamato al chiamante

La parola chiave base per l'accesso ai membri originali overridden

Quando una classe derivata espone un membro ridefinito, la parola chiave **base** consente di accedere al membro originario della classe base. **base** rappresenta la componente dell'istanza corrente ereditata dalla classe base.

Si supponga che la classe Employee contenga un campo aggiuntivo **title** (Sir, Doct,...).

Essa può ridefinire il metodo NomeCompleto() nel modo seguente:

```
public override string nomeCompleto( ) {  
    return title + " " + base.nomeCompleto();  
}
```

E' un'ottima tecnica per evitare di riscrivere l'intero codice col rischio di introdurre errori.

I costruttori nelle classi derivate

A differenza di proprietà e metodi, **i costruttori non vengono ereditati**. Questo significa che il costruttore base viene conglobato nella sottoclasse, ma è visibile soltanto da eventuali nuovi costruttori.

- Se nella classe derivata non si specifica nessun costruttore esplicito, viene creato automaticamente un costruttore implicito nascosto che **automaticamente richiama il costruttore della classe base**.
- Se invece nella classe derivata viene scritto un costruttore esplicito questo, prima di iniziare le proprie istruzioni, **implicitamente richiama il costruttore della classe base**, e solo dopo, esegue le proprie istruzioni.
- Se la classe base espone un **costruttore con parametri**, la classe derivata, in corrispondenza del proprio costruttore, **deve esplicitamente richiamare il costruttore della classe base**. Questo assicura venga prima eseguito il costruttore della classe base e, dopo, quello della classe derivata.
- Il costruttore della classe derivata può anche avere una firma differente rispetto al costruttore della classe base.

```
class Person {  
    private string _nome;  
    public Person (string nome) {  
        this._nome = nome;  
    }  
}  
  
class Employee : Person {  
    private string _title;  
    public Employee(string nome, string title) : base (nome) {  
        this._title = title;  
    }  
}
```

Classi Astratte (opposte alle normali classe istanziabili dette "concrete")

Classi che non possono essere istanziate direttamente così come sono, ma che possono essere utilizzate soltanto per derivare nuove classi che ereditano da esse. Una classe astratta è introdotta dalla parola chiave **abstract** e deve presentare **almeno un metodo astratto**, cioè un metodo la cui implementazione viene demandata alle classi derivate. Nella definizione dei metodi astratti occorre utilizzare ancora la parola chiave **abstract** e specificare **solo la Firma del Metodo** senza scrivere il corpo della funzione. Un metodo astratto è ovviamente virtuale, al punto che la keyword virtual deve essere omessa.

Se una classe possiede uno o più metodi astratti, diventa astratta anch'essa anche senza essere contrassegnata con la parola chiave abstract (che però rende più chiara l'astrazione).

Le classi astratte hanno largo impiego quando l'obiettivo è quello di definire dei comportamenti comuni che dovranno appartenere a tutte le classi derivate.

Esempio

Si consideri una classe Report che suddivide la stampa in 3 sezioni Header, Body e Footer. Si possono definire 3 metodi **abstract** in modo da permettere a chi eredita di personalizzare il comportamento della stampa :

```
abstract class Report {  
    protected abstract void printHeader( );  
    protected abstract void printBody( );  
    protected abstract void printFooter( );  
    public void print( ) {  
        printHeader( );  
        printBody( );  
        printFooter( );  
    }  
}
```

I precedenti metodi virtuali saranno definiti all'interno della classe derivata:

```
class elencoStudenti : Report {  
    public override void printBody( ) {  
        for(int i = 0; i < studenti.count-1; i++)  
            sw.WriteLine(studenti[i].cognome + " " + studenti[i].nome)  
    }  
}
```

La classe base System.Windows.Forms.Form è una **classe Astratta** con molti metodi astratti in modo che le **User Form derivate** possano controllare ogni minimo dettaglio riguardo all'aspetto e al comportamento.

Eventi nella Classe Derivata

Gli eventi NON vengono ereditati dalla classe derivata, nel solito senso che l'evento viene comunque incapsulato all'interno della classe derivata (per cui quando i metodi della classe base richiamano l'evento, **questo viene notificato anche alla classe derivata**), ma i nuovi metodi della classe derivata non possono richiamare direttamente l'evento in quanto questo non risulta visibile.

In pratica gli eventi possono essere richiamati solo dall'interno della classe che li ha dichiarati

Le classi derivate non possono richiamare direttamente gli eventi dichiarati all'interno della classe base.

*Con riferimento all'Esempio 1 di pag 45, l'evento **salvataggioEseguito** viene generato ogni volta che la nuova classe derivata esegue il metodo **salvaSuFile** ereditato dalla classe base.*

Se però la nuova classe volesse generare l'evento anche in corrispondenza di altri metodi, non può.

A tal fine, come già accennato **nell'Esempio 2 di pag 46**, quando si crea una classe che può essere utilizzata come classe base per altre classi, affinché la classe derivata possa richiamare gli eventi della classe base **occorre creare un metodo di chiamata protetto nella classe base che esegue il wrapping dell'evento.**

Le classi derivate possono richiamare indirettamente l'evento richiamando semplicemente il wrapper.

```
protected virtual void OnOverMax(OverMaxEventArgs e) {  
    if (OverMax != null) {        // se OverMax non punta a nessuna funzione  
        OverMax(this, e);        // allora è inutile richiamarlo  
    }  
}
```

La classe derivata può:

- Ereditare e sovrascrivere il metodo **onOverMax** aggiungendo ad esempio ulteriori condizioni che fanno scatenare l'evento soltanto se risultano tutte vere.
- Richiamare il metodo **onOverMax** dall'interno di qualsiasi altro metodo.

Esempio Completo:

Ripartendo dalla classe Cunter già descritta a pagina 46,

```
public class Counter {
    public event OverMaxEventHandler OverMax;
    public int count=0;
    private int maxVal;
    public Counter (int n) { //costruttore a cui passo il valore di trigger
        if (n>1000) // soglia massima
            throw new ArgumentException("max Value = 1000");
        else maxVal=n;
    }
    public void increment() {
        count++;
        if (count >= maxVal) {
            OverMaxEventArgs e = new OverMaxEventArgs(count);
            OnOverMax(this, e);
        }
    }
    protected virtual void OnOverMax(object sender, OverMaxEventArgs e) {
        if (OverMax != null) { // se OverMax punta ad una funzione del main
            OverMax(this, e);
        }
    }
}
```

si può scrivere la classe derivata nel modo seguente:

```
Class Derivata : Counter {
    protected override void OnOverMax(object sender, OverMaxEventArgs e) {
        // genero l'evento solo se la soglia impostata è > 30
        if (e.valoreSoglia > 30 && myBool == true)
            base.onOverMax(sender, e);
    }
}
```

Quando qualunque metodo della nuova classe richiama la procedura onOverMax, questa controlla se il valore che ha fatto scattare il trigger è maggiore di 30.

- Se è > 30 fa scattare l'evento richiamando `base.onOverMax(e);`,
- altrimenti l'evento non viene generato. Cioè in sostanza se la classe utilizza una soglia di trigger inferiore a 30, gli eventi NON vengono generati

La classe derivata può anche ridefinire la funzione **increment()** che fa scaturire l'evento.

In questo caso la nuova funzione **increment()** ha diverse possibilità:

- Fare eventuali controlli aggiuntivi e poi richiamare **base.increment()** in modo da eseguire il metodo nativo che esegue l'incremento e scatena l'evento,
- Riscrivere completamente il metodo di incremento e poi richiamare esplicitamente la procedura **onOverMax** della classe derivata, la quale a sua volta richiamerà **base.onOverMax** che farà scaturire l'evento.
- Riscrivere completamente il metodo di incremento e poi richiamare esplicitamente la procedura **base.onOverMax** della classe base, la quale richiamerà direttamente l'evento.

Ereditarietà applicata ai controlli

In NET è molto semplice derivare un nuovo controllo da un controllo già esistente, aggiungendo nuove Proprietà e/o nuovi Metodi. L'esempio seguente mostra un nuovo controllo derivato da un Text Box a cui vengono aggiunti i metodi `ToUpper()` e `ToLower()` per convertire rispettivamente in maiuscolo e minuscolo il testo contenuto.

```
class myTextBox : TextBox {  
    public void toLowerCase() {  
        this.Text = this.Text.ToLower();  
    }  
    public void toUpperCase() {  
        this.Text = this.Text.ToUpper();  
    }  
}
```

Creando un apposito progetto di Classe e compilando la classe, viene creata una DLL contenente il nuovo controllo che potrà poi essere importato all'interno della ToolBox insieme a tutti gli altri controlli. Per importare il nuovo Controllo sulla ToolBox, andare sull'intestazione di un qualsiasi gruppo (tipicamente **Generale** che è adibito a questo compito) ed eseguire i seguenti passi:

- => **Tasto Destro**
- => **Scegli Elemento**
- => **Sfoglia**
- => ricercare la DLL contenente il controllo
- => **OK**

Esempio di main

```
private void Form1_Load(object sender, EventArgs e)  
{  
    myTextBox txt = new myTextBox();  
    txt.Name = "myTxt";  
    txt.Location = new Point(50, 50);  
    this.Controls.Add(txt);  
}  
private void button1_Click(object sender, EventArgs e)  
{  
    myTextBox txt = (myTextBox)this.Controls["myTxt"];  
    txt.toUpperCase();  
}
```

Nota sul qualificatore Friend

Friend (**internal in C#**) è molto utilizzato nell'ereditarietà applicata alle form.

I controlli dichiarati FRIEND all'interno di una form (tramite il qualificatore di visibilità **Modifiers**), vengono ereditati insieme alla form., ma le loro proprietà **NON** sono modificabili a Design Time ed i loro eventi **NON** possono essere intercettati.

Interfacce

Estensione del concetto di classe astratta. **Una interfaccia è una classe astratta che non contiene codice, ma si limita a definire proprietà e metodi di un oggetto, cioè stabilisce quale aspetto dovrà avere l'oggetto.**

Le interfacce:

- costituiscono la base del **Polimorfismo** della OOP. Definiscono proprietà e metodi comuni che dunque risulteranno identici per tutti quegli oggetti che implementano quell'Interfaccia.
- L'interfaccia è un modo per realizzare **l'eredità multipla**. Una classe può ereditare da una classe base ma **può implementare più interfacce, separate da virgola**.
 - ✓ Il nome di una Interfaccia deve sempre iniziare con la lettera **I maiuscola**
 - ✓ Un'interfaccia **definisce soltanto la firma** (Nome, Parametri, Tipologia del valore restituito) di proprietà e metodi che dovranno essere esposti dalla classe che implementerà l'interfaccia..
 - ✓ Le interfacce **non** possono contenere **Qualificatori di visibilità** (public, private, etc). Ogni membro dell'interfaccia è implicitamente public, ma la parola public deve essere omessa
 - ✓ Le interfacce **non** possono contenere **Campi**, ma solo Proprietà e Metodi.
 - ✓ Una interfaccia può contenere anche la definizione di eventi, anche se normalmente sono sconsigliati

Per definire una Interfaccia si utilizza la seguente sintassi:

```
public interface IDispAggiuntivo {  
    int id {  
        get;  
        set;  
    }  
    Boolean status { // readonly Property  
        get;  
    }  
    void connetti(string s);  
    void disconnetti(string s);  
}
```

Implementazione di una interfaccia

Nel caso delle Interacce, invece di dire che una classe eredita da una classe astratta, **si dice che una classe implementa una Interfaccia**, ma la sostanza è la stessa. La classe che implementa una certa interfaccia deve definire il codice vero e proprio relativo a TUTTE le proprietà e metodi definiti all'interno dell'interfaccia. Classi diverse che implementano la stessa interfaccia **possono così esporre metodi identici** internamente realizzati in modo completamente diverso. Principio base del Polimorfismo.

Esempio di implementazione dell'interfaccia precedente :

```
public class MyComponent : IDispAggiuntivo{  
    private int _id;  
    public int id {  
        get { return _id; }  
        set { _id = value; }  
    }  
    void IDispAggiuntivo.connetti (string s) { ..... }  
    public void disconnetti (string s) { ..... }  
}
```

- **Proprietà e Metodi che implementano i membri dell'Interfaccia devono avere lo stesso nome**
- Proprietà e Metodi che implementano I membri dell'Interfaccia **devono essere pubblici**
- Per rendere più chiaro il codice, si può aggiungere il **nome dell'interfaccia** davanti al metodo, però:
 - O si antepone il nome dell'interfaccia (come nel caso di connetti)
 - O si scrive public (come nel caso di disconnetti)

Riferimento a Interfaccia

L'interfaccia può essere utilizzata come tipo per dichiarare dei riferimenti esattamente come si potrebbero dichiarare riferimenti ad una normale classe. L'unica particolarità è che non può essere istanziata. Esempio:

```
MyComponent comp = new MyComponent();  
IDispAggiuntivo icoomp = (IDispAggiuntivo) comp; // up casting
```

E' anche possibile utilizzare il nome di una interfaccia come tipo di parametro da passare ad una funzione.

```
public void aggiungiComponente (IDispAggiuntivo obj)
```

Questa funzione accetterà come parametro soltanto riferimenti ad oggetti che implementano quella interfaccia.

Implementazione di più interfacce

Come detto una classe può implementare due o più interfacce contemporaneamente (ereditarietà multipla).

Un Metodo (o una Proprietà) può anche implementare contemporaneamente due metodi distinti di due Interfacce distinte (ammesso che abbiano lo stesso nome e la stessa firma). In caso di nome uguale e firma diversa, occorrerà fare due metodi distinti, uno relativamente ad una interfaccia e l'altro relativamente all'altra interfaccia.

Note su Interfacce ed Ereditarietà

- 1) Una Interfaccia può ereditare da una altra interfaccia, secondo le stesse regole valide per le classi. Utile per definire interfacce più estese. Un'interfaccia derivata però **non può ridefinire i membri di quella di base**. Dunque non è consentita la parola chiave **override**. Se un membro dell'interfaccia derivata ha lo stesso nome di uno dell'interfaccia di base, esso nasconde quest'ultimo (shadowing, con parola chiave **new**).
- 2) Una classe derivata eredita automaticamente tutte le interfacce implementate nella classe base:
La classe derivata può normalmente ridefinire i metodi di interfaccia definiti **virtual** nella classe di base.

L'Interfaccia DotNet IComparable

La classe Array espone il metodo statico **Sort** che consente di ordinare un vettore qualora questo contenga dei dati semplici, come numero o stringhe. Nel caso però di un vettore di record, il metodo Sort non è più utilizzabile perché non sa come confrontare due record (su quale campo eseguire il confronto).

L'implementazione dell'interfaccia IComparable all'interno di una certa classe, consente di ordinare vettori di oggetti appartenenti a quella classe mediante il metodo Sort del vettore. Comodissimo.

Questa interfaccia espone un solo metodo **CompareTo** che riceve un oggetto e restituisce -1 0 +1 a seconda che l'oggetto corrente (quello da cui viene invocato CompareTo) sia minore, uguale o maggiore di quello ricevuto come argomento. Es

```
class Person : IComparable {  
    public string nome;  
    public string Citta;  
    public Person (string nome, string citta){  
        this.nome = nome;  
        this.citta = citta;  
    }  
    int IComparable.CompareTo(Object obj) {  
        // Ogni oggetto non null viene considerato > di null  
        if (obj == null) return 1; // io sono maggiore  
        Person p = (Person)obj;  
        return String.Compare(this.nome, p.nome, true); case unsensitive  
    }  
}
```

Il main agirà nel modo seguente:

```
Person[] vect = new Person[] { new Person("Mario Rossi", "Cuneo"),
                                new Person("Enrico Bonavia", "Fossano") };
Array.Sort(vect);
```

- Array.Sort utilizza un certo algoritmo di ordinamento (nascosto) che scandisce tutti i record, li confronta sulla base della funzione CompareTo e, dopo ogni scansione, ad esempio porta in alto quello più leggero (principio del Bubble Sort).
- Se la classe non implementa l'interfaccia IComparable si avrà un errore in fase di compilazione.
- Notare come il parametro passato a CompareTo debba *necessariamente* essere di tipo Object come definito nella **firma** dell'Interfaccia, ed occorra pertanto eseguire un Cast per convertirlo in un riferimento a Person.

L'Interfaccia IComparer

Se la classe finale ha la necessità di dover gestire più procedure di ordinamento (una per il campo **nome**, una per il campo **cognome**, una per il campo **città**, etc.), è possibile passare al metodo statico **Array.Sort** un secondo parametro opzionale (oltre al vettore da ordinare) che consente al chiamante di specificare il **nome** della procedura di ordinamento da utilizzare (fra quelle scritte all'interno della classe).

Questo secondo parametro deve essere un oggetto che implementa l'Interfaccia **IComparer**, IComparer è una interfaccia molto simile alla precedente ma non identica. Espone anch'essa un unico metodo **Compare** simile al metodo CompareTo precedente, che però esegue il confronto fra due oggetti indipendenti entrambi passati come parametro. Al solito Restituisce -1 0 +1 a seconda che il primo oggetto sia minore, uguale o maggiore del secondo.

Operativamente occorre dichiarare, all'interno della classe principale, due o più classi annidate ciascuna delle quali implementi l'interfaccia **IComparer** realizzando di fatto due o più procedure di ordinamento differenti.

Nota: Questo ovviamente si sarebbe potuto fare anche con l'interfaccia precedente, ma non esiste una firma di Array.Sort che accetto un oggetto IComparable come parametro di ordinamento.

```
class Person {
    public string nome;
    public string citta;
    public Person (string nome, string citta) {
        this.nome = nome;
        this.citta = citta;
    }
    public class confrontaNome : IComparer<Object> {
        int IComparer<Object>.Compare(Object o1, Object o2) {
            if (o1 == null && o2 == null) return 0;
            else if (o1 == null) return 1;
            else if (o2 == null) return -1;
            else {
                Person p1 = (Person)o1;
                Person p2 = (Person)o2;
                return String.Compare(p1.nome, p2.nome, true);
            }
        }
    }
}
```

Il main agirà nel modo seguente:

```
Person[] vect = new Person[] { new Person("Mario Rossi", "Cuneo"),
                                new Person("Enrico Bonavia", "Fossano") };
Array.Sort(vect, new Person.confrontaNome());
```

Il 2° parametro da passare ad Array.Sort deve essere un oggetto istanziato che implementi l'interfaccia IComparer. Per rendere ancora migliore il codice, si potrebbe dotare la classe Person di 2 metodi Shared cmpNome e cmpCognome che rispettivamente istanzino le classi ConfrontaNome e ConfrontaCognome restituendo la corrispondente interfaccia (questo accorgimento non può essere utilizzato all'interno dell'interfaccia in quanto l'interfaccia non ammette membri Static).

```
static IComparer cmpNome {
    return new ConfrontaNome;
}

static IComparer cmpCognome {
    return new ConfrontaCognome;
}
```

In tal modo il codice Client può essere scritto nel modo seguente, senza il New:

```
Array.Sort(vectPers, Person.cmpNome) // ordinamento sul nome
Array.Sort(vectPers, Person.cmpCognome) // ordinamento sul cognome
```

L'Interfaccia ICloneable

Quando si copia una variabile riferimento su un'altra variabile riferimento, ciò che in realtà viene copiato è il puntatore, per cui il risultato sono due riferimenti che puntano allo stesso oggetto.

Per ottenere invece una copia del dato, occorre dotare la classe di un opportuno metodo che esegua la copia del dato. Nel rispetto del polimorfismo, ogni classe dovrebbe implementare l'Interfaccia **ICloneable** che espone un unico metodo **Clone()** As Object che restituisce la copia fisica dell'istanza a cui viene applicato.

Quasi tutti gli oggetti Net implementano questa Interfaccia: Array, ArrayList, BitArray, Font, Icon, Queue, Stack

```
class Person : ICloneable {
    public string nome;
    public string citta;
    Automobile automobile;
    public Person (string nome, string citta) {
        this.nome = nome;
        this.citta = citta;
    }
    public Object ICloneable.Clone() {
        Person p = new Person(this.nome, this.citta);
        p.automobile = this.automobile;
        // p.automobile punta però ora all'oggetto Automobile interno
        // all'oggetto corrente. Non viene creato un nuovo obj Automobile
        return p;
    }
}
```

Per evitare di copiare a mano tutti i membri della classe, si può utilizzare il metodo **MemberwiseClone** che tutte le classi ereditano da System.Object, che però esegue la Shallow Copy e non risolve il problema precedente.

```
public Object ICloneable.Clone() {
    return this.MemberwiseClone();
}
```

Main: Person p2 = (Person) p1.Clone();

Shallow Copy e Deep Copy

Come ampiamente sottolineato, il Clone precedente (sia quello realizzato manualmente sia quello realizzato mediante `MemberwiseClone()`), non crea una nuova istanza dell'oggetto Automobile, ma, all'interno di p2, automobile sarà semplicemente un puntatore che punta allo stesso oggetto Automobile contenuto in p1.

Provando a scrivere

```
p2.Automobile.Nome = "Nuovo Automobile"
MsgBox(p2.Automobile.Nome)
MsgBox(p1.Automobile.Nome)
```

I 2 MsgBox producono ovviamente lo stesso risultato, visto che p1 e p2 puntano allo stesso oggetto Automobile

Si parla in questo caso di **Shallow Copy** (Copia Superficiale) che crea semplicemente una copia dell'oggetto, senza creare una copia degli oggetti secondari eventualmente contenuti all'interno dell'oggetto clonato.

Il che, in alcuni casi potrebbe anche andare. Immaginiamo un campo **Person Capo**. Se il Capo dovesse cambiare tutti i riferimenti punterebbero automaticamente al nuovo capo.

Più frequentemente però occorre clonare l'oggetto completo, comprese le eventuali sottoclassi.

Si parla in questo di **Deep Copy** (Copia Profonda) che significa clonare **l'intero grafo** dell'oggetto.

Il Deep Copy può essere eseguito manualmente nel modo seguente:

```
public Object ICloneable.Clone() {
    // si inizia con una shallow copy globale
    Person p = (Person) this.MemberwiseClone();
    if (this.automobile != null)
        p.automobile = (Person) this.automobile.Clone();
    return p;
}
```

Questa funzione utilizza il metodo `MemberwiseClone` per copiare tutti i campi che *non sono* oggetti, e poi utilizza il metodo `clone` per istanziare, uno alla volta, tutti gli oggetti secondari.

Per annidamenti più complessi, questa tecnica è però abbastanza gravosa.

Esiste una soluzione alternativa basata sul concetto di **serializzazione degli oggetti**.

Attributi

Concetto introdotto in Net per gestire in modo unificato le **direttive di compilazione** (quelle che erano le `#pragma` del C oppure i blocchi `#If` di VB6). Gli attributi possono essere considerati come annotazioni inserite qua e là nel codice e possono essere applicate a qualsiasi entità NET (classi, moduli, strutture, proprietà).

La sintassi per la scrittura degli attributi cambia leggermente tra i vari linguaggi, ma il meccanismo di gestione è sempre lo stesso.

Sintatticamente gli attributi sono **classi particolari** che :

- Terminano sempre con il nome **Attribute**, che può essere omesso ma la cui presenza migliora la leggibilità
- devono essere istanziate mediante **parentesi quadre** `[]` (C#) oppure acute `<>` (VB)
- **non** possono essere referenziate da codice. Non essendo referenziabili da codice, in genere le classi **Attribute** espongono **il solo metodo costruttore** più eventuali proprietà **Private**.
- rimangono istanziate per l'intero ciclo di vita dell'applicazione
- All'interno delle parentesi quadre di istanza possono essere definiti più **Attribute**, separati da virgola.
- Per le classi **attribute** è possibile utilizzare la stessa sintassi abbreviata degli imports. Ad esempio l'attributo **DescriptionAttribute** è definito all'interno del namespace **System.ComponentModel**.

ObsoleteAttribute

Posizionato davanti ad una intera classe o ad un singolo metodo, consente di contrassegnare la classe / il singolo metodo come **Obsoleto**, generando un warning di compilazione nel momento in cui il client, a Design Time, richiamerà quel metodo o istanzierà quella classe. Il costruttore ha due parametri, entrambi opzionali:

- **string msg**, cioè messaggio da visualizzare in corrispondenza del warnig
- **bool mode** che normalmente vale FALSE. Se impostato a TRUE, invece di un warning genererà un errore in fase di compilazione.

```
[ObsoleteAttribute ("Non usare questo metodo ma il nuovo metodo .....")]  
public void confrontaNome () { }
```

Le parentesi quadre sostanzialmente istanziano la classe **ObsoleteAttribute** richiamando il costruttore a cui viene passato come unico parametro **string msg**. Questa classe sarà associata al metodo **confrontaNome**.

L'attributo Conditional

Posizionato davanti ad una intera classe o ad un singolo metodo, condiziona il richiamo del metodo / l'istanza della classe:

```
[ConditionalAttribute ("LOG")]  
public void LogMsg(string s) {  
    Console.WriteLine(s)  
}
```

La procedura controllata dal **ConditionalAttribute** viene sempre inclusa nell'applicazione, ma viene eseguita, quando richiamata, soltanto se in testa al file è stata definita la relativa **Costante di Compilazione** "LOG". Le **Costanti di Compilazione** possono essere definite all'inizio del file (prima ancora degli using) nel modo seguente:

```
#define LOG
```

E' possibile specificare le Costanti di Compilazione anche all'interno della Project Property Page.

- ❑ L'attributo Conditional funziona solo con le procedure che restituiscono void e **non** con le funzioni che restituiscono un valore, nel qual caso viene ignorato.
- ❑ L'attributo Conditional consente definizioni multiple all'interno delle stesse parentesi quadre. La chiamata sarà eseguita solo se **tutte** le Costanti di Compilazione sono state definite.

```
[Conditional ("LOG1"), Conditional ("LOG2")]  
public void LogMsg(string s) { }
```

L'attributo DebuggerStepThrough

Consente di contrassegnare una routine in modo che venga ignorata dal Debugger. Non ha parametri.

L'attributo DescriptionAttribute

Consente semplicemente di associare una descrizione ad un metodo / classe, Questa descrizione può poi essere letta mediante una scansione della **AttributeCollection**

```
Imports System.ComponentModel
```

```
[DescriptionAttribute ("Classe descrittiva di una persona")]  
Class Person { }
```

Regular Expression

```
using System.Text.RegularExpressions;
Regex reg = new Regex("^[a-zA-Z]+$", RegexOptions.IgnoreCase);
return reg.IsMatch(str); oppure
return Regex.IsMatch(str, "[a-zA-Z]+$");
```

regexpal.com ottimo sito di prova. Vengono mostrate in colore le parti di stringa che soddisfano la RegEx.

Caratteri Speciali

\ Individua un carattere speciale.
 ^ Individua l'inizio della stringa
 \$ Individua la fine della stringa

Caratteri relativi al Tipo

. Puntino. Qualsiasi carattere singolo (escluso il \n)
 \d cifra numerica
 \w cifra alfanumerica (minuscole, maiuscole, numeri e underscore)
 \s cifra di tipo WhiteSpace (spazio, tab, a Capo, etc)
 \b "word boundary", cioè delimita inizio / fine di una parola
 \n a capo
 \D cifra NON numerica
 \W cifra NON alfanumerica
 \S cifra NON di tipo WhiteSpace
 \B "word NON boundary", cioè che non delimita inizio / fine di una parola
 [] Elenco esplicito dei caratteri ammessi. Possono essere scritti uno dopo l'altro senza separatori. Eseguie in pratica una OR fra i caratteri indicati
 ^ Usato subito dopo l'aperta quadra indica la **non** presenze dei caratteri indicati
 \ Usato dentro le quadre consente il match di caratteri speciali che avrebbero un significato speciale. Ad esempio \^ \\$ \. * \+ \\ \(\ \)

Caratteri di ripetizione

? L'ultimo termine può ripetersi **Uno o Zero** volte
 {n} L'ultimo termine deve ripetersi **esattamente** n volte
 {n,m} L'ultimo termine deve ripetersi da un minimo di n volte ad un massimo di m volte
 * L'ultimo termine può ripetersi un numero imprecisato di volte (anche 0 volte)
 + L'ultimo termine può ripetersi un numero imprecisato di volte, ma almeno 1 volta

I Caratteri di ripetizione non possono essere consecutivi, ma devono sempre essere intervallati da almeno un carattere di Tipo.

Altro

() Raggruppamento. ES ([\.\-]? \w+) *
 Punto o Trattino (facoltativi) seguiti da un elenco di caratteri (almeno 1). La sequenza racchiusa tra parentesi tonde può ripetersi un numero imprecisato di volte (anche 0 volte).
 | OR – Consente di inserire più espressioni regolari (alternative) sulla stessa riga.

Esempi

"pippo"	Verifica se è presente la parola pippo, eventualmente anche come sottostringa
"^pippo\$"	Verifica se il contenuto COINCIDE con la parola pippo
"[aeiou]"	Verifica se nel testo è presente una qualsiasi vocale
"^[aeiou]\$"	Verifica se il testo è costituito da un solo carattere e questo carattere è una vocale
"\bpippo"	Verifica se ci sono parole che iniziano per pippo,
"pippo\b"	Verifica se ci sono parole che terminano per pippo,
"\bpippo\b"	Verifica se è presente la parola pippo come parola intera
"\bpippo\b.*\bpluto\b"	Cerca la parola pippo seguita dalla parola pluto sulla stessa riga
"\b\w{5,6}\b"	Individua tutte le parole costituite da 5 o 6 caratteri
"* [.?!] \$"	Verifica se la frase termina con un punto semplice, interrogativo o esclamativo Notare che dentro le [] . e ? perdono il loro significato speciale
"^[^x] \$"	Qualunque carattere diverso da x
"\b\d\d\d-\d\d\d\b"	Verifica un numero di telefono nel formato XXX-XXXX
"\b\d\d\d-\d{4}\b"	stessa cosa
"\b\d{3}-\d{4}\b"	stessa cosa
"^\d{3}-\d{4} \$"	Verifica se la stringa è costituita interamente da un num tel XXX-XXXX
"\b\d{3}\s\d{3}-\d{4}"	Num Tel in format o XXX XXX-XXXX
"(?:\d{3}())\s?\d{3}[-]\d{4}"	(xxx) xxx xxxx o xxx xxx-xxxx o (xxx)xxx xxxx

Esempio: Password costituita da lettere e numeri e che inizia con una lettera.

"[A-Za-z]+\d+" Lettere seguite da numeri, il tutto eventualmente ripetuto
 "^[A-Za-z]+\d+)\$" equivalente a sopra
 "^[A-Za-z]+\d+\$" Questo invece ha un altro significato. Accetta SOLO sequenze letterali seguite da sequenze numeriche e con subito dopo il Fine Stringa.
 aa3a non è accettato.

In dot.net invece di [A-Za-z] si può anche scrivere [A-Z][a-z]

Esempio: Ricercare I nomi che NON contengono le lettere **a e u** fra i primi 4 caratteri.

"^[^aeuAEU]{4}.+\$"

All'inizio ci devono essere 4 lettere diverse da **A U E**,
 poi ci può essere un qualunque carattere (puntino) il quale può ripetersi un numero imprecisato di volte ma almeno una. In sostanza la stringa ricevuta deve essere lunga almeno 5 caratteri e non deve contenere **A U E** fra i primi 4.

Scrivendo solo

"^[^aeuAEU]{4}\$"

Vengono individuate SOLO stringhe lunghe 4 (sempre prive dei caratteri **A U E** fra i quattro caratteri).

Utilizzo embedded di msWORD

Nel modello COM (ma anche poi in NET) ogni applicazione è vista come un oggetto che espone proprietà e metodi che consentono all'applicazione stessa di essere utilizzata da altre applicazioni.

Progetto / Aggiungi Riferimento / COM / **Microsoft Word 12.0 Object Library**

Versioni:

11.0 -> Office 2003 - XP
12.0 -> Office 2007 - Vista
13.0 -> Office 2010 - Seven- VisualStudio 2010 Framework 4

using Microsoft.Office.Interop.Word;

Campi a visibilità globale

```
public Microsoft.Office.Interop.Word._Application myWord;  
public Microsoft.Office.Interop.Word._Document myDoc;
```

Istanza dell'applicazione WINWORD.EXE

```
myWord = new Microsoft.Office.Interop.Word.Application();  
myWord.Visible = true / false;  
.....  
myWord.Quit();
```

Creazione di un nuovo Documento

```
myDoc = myWord.Documents.Add();  
.....  
if (!myDoc.Saved) {  
    myDoc.SaveAs(Application.StartupPath + "\\prova");    // .docx  
    myDoc.Saved = true;    // inutile, fatto in automatico da Word.  
}  
// per chiudere senza salvare  
object saveOption = WdSaveOptions.wdDoNotSaveChanges;  
myDoc.Close(ref saveOption);
```

Gestione dei Paragrafi

myDoc.Sentences è una collection di paragrafi che **parte da 1**. Nel caso di Documento nuovo appena creato, c'è sempre almeno un paragrafo (chiuso dal segno di fine paragrafo) creato automaticamente insieme al documento.

myDoc.Sentences.Count Numero di paragrafi contenuti nella collection.

Object **start = myDoc.Sentences[i].Start**; Punto di partenza del i-esimo paragrafo. Pur essendo un Object rappresenta sostanzialmente la posizione assoluta del carattere all'interno dell'intero testo, esattamente come in NotePad. Il primo carattere ha indice 0.

Object **end = myDoc.Sentences[i].End - 1**; Punto di fine del i-esimo paragrafo. Facendo - 1 si esclude il segno di paragrafo posizionando il cursore dopo l'ultimo carattere effettivo.

myDoc.Words è una collection di parole.

La formattazione del testo: l'oggetto RANGE

Una volta individuato mediante gli oggetti **start** e **end** il testo da elaborare, si può accedere al testo mediante il seguente codice. **Start** e **end** sono degli object, ma è possibile assegnare loro direttamente dei numeri interi.

```
Range myRange = myDoc.Range(ref start, ref end);
```

```
myRange.Font.Name = "Arial";  
myRange.Font.Size = 12;  
myRange.Font.Color = WdColor.wdColorBlue;  
myRange.Font.Bold = Convert.ToInt16(true); // non accetta true / false!!  
myRange.Font.Italic = Convert.ToInt16(false);  
myRange.Font.Underline = WdUnderline.wdUnderlineDash;  
myRange.ParagraphFormat.Alignment = WdParagraphAlignment.wdAlignLeft;  
// Sostituisce il testo contenuto nel RANGE attuale con il contenuto del Text Box  
// Il segno di fine paragrafo, anche se contenuto all'interno del range, non viene comunque MAI sovrascritto  
myRange.Text = txtTesto.Text;  
myRange.Select(); // Evidenzia il Range attuale
```

Per andare a capo all'interno del testo si può aggiungere \n oppure usare Text Box multiline

L'oggetto myDoc.Content

Rappresenta il **testo contenuto nell'intero documento**, ed è indipendente dalla eventuale suddivisione in paragrafi.

myDoc.Content.Start Posizione assoluta a sinistra del primo carattere (0)

myDoc.Content.End Posizione assoluta a destra dell'ultimo carattere (Length)

L'oggetto myWord.Selection

L'oggetto myWord.Selection rappresenta il testo attualmente selezionato.

Selection.Start imposta / restituisce la posizione assoluta del cursore, (esattamente come avviene nel Blocco Note). Ogni segno di fine paragrafo conta esattamente come **UN** carattere.

Selection.End rappresenta la posizione del primo carattere successivo rispetto alla selezione.

Se non c'è testo selezionato, **selection.end** coincide con **selection.start**.

Il metodo find

```
object o = "ciao";  
myWord.Selection.Start = myDoc.Content.Start;  
myWord.Selection.Find.ClearFormatting();  
myWord.Selection.Find.Execute(ref o); Occorre probabilmente installare una patch del framework  
Range myRange = myDoc.Range(ref start, ref end);  
myRange.Text = newString;
```

La ricerca parte dalla posizione attuale del cursore (**selection.start**). Nel caso invece di testo selezionato (**selection.end** diverso da **selection.start**), la ricerca parte automaticamente dal fondo della selezione.

In caso di occorrenza, sposta **selection.start** all'inizio della parola trovata, SELEZIONA il testo ricercato e restituisce True (altrimenti False). Arrivati alla fine del documento, la ricerca non ricomincia automaticamente dall'inizio ma occorre spostare manualmente il cursore all'inizio.

Per default la ricerca è **case sensitive**. Dopo il primo parametro ci sono molti altri parametri che possono essere impostati, come ad esempio la ricerca case un sensitive e il valore di Replace. Se si vuole trascurare qualche parametro intermedio, non è possibile utilizzare la sintassi *virgola virgola*, ma occorre impostare per questi parametri il valore **Type.Missing**.

Parametrizzazione delle formattazioni**Font (collection)**

```
foreach (FontFamily family in FontFamily.Families)
    cmbFont.Items.Add(family.Name);
cmbFont.SelectedIndex = 0;

myRange.Font.Name = cmbFont.Text;
```

Color (enum)

```
WdColor[] vectColor = (WdColor[])Enum.GetValues(typeof(WdColor));
foreach (WdColor w in vectColor)
    cmbColor.Items.Add(w.ToString().Substring(7));
);
cmbColor.SelectedIndex = 0;

myRange.Font.Color = (WdColor)Enum.Parse(typeof(WdColor), cmbColor.Text);
```

Bold, Italic, Underline (enum)

```
WdUnderline[] vectUnder = (WdUnderline[])Enum.GetValues(typeof(WdUnderline));
foreach (WdUnderline w in vectUnder)
    cmbSottolineato.Items.Add(w.ToString().Substring(11));
cmbSottolineato.SelectedIndex = 0;

myRange.Font.Bold = Convert.ToInt16(chkBold.Checked);
myRange.Font.Italic = Convert.ToInt16(chkItalic.Checked);
myRange.Font.Underline = (WdUnderline)Enum.Parse(typeof(WdUnderline),
                                                    cmbSottolineato.Text);
```

Creazione di una tabella

```
int nRighe = 3;
int nColonne = 4;

// spostato il cursore sull'ultimo segno di paragrafo
object start = myDoc.Sentences[myDoc.Sentences.Count].End - 1;
object end = myDoc.Sentences[myDoc.Sentences.Count].End;
Range myRange = myDoc.Range(ref start, ref end);

Table myTable = myDoc.Tables.Add(myRange, nRighe, nColonne);
// per default i borders sono trasparenti
myTable.Borders.Enable = 1;
for (int i=1; i<=nRighe; i++)
    for (int j=1; j<=nColonne; j++){
        myTable.Cell(i, j).Range.Text = "r" + i.ToString() + ",c" + j;
        myTable.Cell(i, j).Range.Bold = 1;
        myTable.Cell(i, j).Range.Font.Size = 15;
        myTable.Cell(i, j).Range.Paragraphs.Alignment = ParagraphCenter;
    }
```

```
// se inserisco due tabelle in sequenza vengono attaccate una sotto l'altra
start = myDoc.Sentences[myDoc.Sentences.Count].End - 1;
end = myDoc.Sentences[myDoc.Sentences.Count].End;
myRange = myDoc.Range(ref start, ref end);
myRange.Text = "\n";
```

Creazione di un PDF

```
string p =Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory);
myDoc.ExportAsFixedFormat(p + "\\nome.pdf", WdExportFormat.wdExportFormatPDF, false)
```

La gestione dei Moduli

Creazione di un documento Word contenente i moduli da compilare {

```
myDoc = myWord.Documents.Add();
Range myRange;
ContentControl cc;

// campo NOME
myRange=assegnaRange();
myRange.Text = "Nome: ";
myRange.Font.Bold = 1;

myRange = assegnaRange();
cc=myRange.ContentControls.Add(WdContentControlType.wdContentCtrlText)
cc.Title = "txtNome";
cc.Range.Text = "Inserire il nome"; // contenuto
myRange.Font.Bold = 0; // solo DOPO aver assegnato un valore

// campo SESSO
myRange = assegnaRange();
myRange.Text = "\nSesso: ";
myRange.Font.Bold = 1;

myRange = assegnaRange();
cc=myRange.ContentControls.Add(WdContentCtrlType.wdContentCtrlComboBox)
cc.Title = "cmbSesso";
cc.DropDownListEntries.Add("Maschio", "m", 0);
cc.DropDownListEntries.Add("Femmina", "f", 1); // l'index non è automatico
myRange.Font.Bold = 0;

// campo DATA DI NASCITA
myRange = assegnaRange();
myRange.Text = "\nData di Nascita: ";
myRange.Font.Bold = 1;

myRange = assegnaRange();
cc=myRange.ContentControls.Add(WdContentControlType.wdContentCtrlDate);
cc.Title = "ctrlDataNascita"; // name
cc.DateStorageFormat=WdContentCtrlDateStorageFormat.wdContentCtrlDateStorageDate
// oppure cc.DateDisplayFormat = "dd/MM/yyyy";
myRange.Font.Bold = 0;

// PROTEZIONE DEL DOCUMENTO
myDoc.Protect(WdProtectionType.wdAllowOnlyFormFields, Type.Missing, "pippo");
myDoc.Unprotect("pippo");
```



```
Lettura dei valori inseriti dall'utente {  
    if (myDoc.ContentControls.Count > 0) {  
        foreach (ContentControl cc in myDoc.ContentControls)  
            switch (cc.Title) {  
                case "txtNome": MessageBox.Show(cc.Range.Text); break;  
                case "cmbSesso": MessageBox.Show(cc.Range.Text ); break;  
                case "ctrlDataNascita": MessageBox.Show(cc.Range.Text); break;  
            }  
    }  
}
```

Utilizzo embedded di msExcel

```
Microsoft.Office.Interop.Excel.Application myExcel;  
Microsoft.Office.Interop.Excel.Workbook myWorkBook;  
Microsoft.Office.Interop.Excel.Worksheet myWorkSheet;  
  
private void button Click() {  
    myExcel = new Microsoft.Office.Interop.Excel.Application();  
    myExcel.Visible = true;  
    myWorkBook = myExcel.Workbooks.Add();  
  
    myWorkBook.Worksheets.Add();  
    // (*) cast a Worksheet obbligatorio con Framework 3.5  
    myWorkSheet = (Worksheet) myWorkBook.Worksheets.Item[2];  
    myWorkSheet.Name = "Foglio 2222 ";  
  
    myWorkBook.Worksheets.Add();  
    myWorkSheet = (Worksheet) myWorkBook.Worksheets.Item[3];  
    myWorkSheet.Delete();  
  
    myWorkSheet = (Worksheet) myWorkBook.ActiveSheet;  
    // Le celle "utente" partono da 1, 1  
    // 0 rappresenta la prima riga e prima colonna (cioè le intestazioni)  
    myWorkSheet.Cells[1, 1] = "asse x";  
    myWorkSheet.Cells[1, 2] = "asse y";  
    myWorkSheet.Range["A5"].Value = "Accesso in base al nome della cella";  
  
    // oppure  
    Microsoft.Office.Interop.Excel.Range myRange;  
    myRange = myWorkSheet.get_Range("A1", "A1");  
    myRange.Value = "ASSE X";  
    myRange.Font.Size = 20;  
    myRange.Font.Bold = 1;  
    // myRange.Cells.WrapText = true; // va a capo nella cella  
    myRange.EntireColumn.AutoFit(); // allargamento 'automatico' colonna  
  
    // 2° colonna  
    myRange = myWorkSheet.get_Range("B1", "B1");  
    myRange.Value = "ASSE Y";  
    myRange.Font.Size = 20;  
    myRange.Font.Bold = 1;  
    myRange.EntireColumn.AutoFit(); // allargamento 'automatico' colonna  
  
    // bordo celle  
    myRange = myWorkSheet.get_Range("A1", "B1");  
    myRange.BorderAround(XlLineStyle.xlDouble, XlBorderWeight.xlMedium,  
                        XlColorIndex.xlColorIndexAutomatic);  
}
```

```

//creo i dati x e f(x)
for (int i = -10; i <= 10; i++) {
    // prima riga, poi colonna
    myWorkSheet.Cells[13 + i, 1] = i.ToString();
    // = Potenza(A2;2)
    myWorkSheet.Cells[13 + i, 2] = "=Potenza(A" + (i + 13).ToString() + ";2)";
    // Viene però fuori un errore RunTime sulla lingua.
    // La parola 'Potenza' non viene trovata. Si potrebbe scrivere pow() oppure cambiare Framework
    // impostando il 3.5. Così facendo viene fuori un errore (*) di cast obbligatorio facilmente risolvibile
}

// Creazione di un Grafico
ChartObjects lstCharts = (ChartObjects)myWorkSheet.ChartObjects();
// Coordinate dei 4 punti della finestra in cui visualizzare il grafico
ChartObject myChart = (ChartObject)lstCharts.Add(200,80,400,230);
// colonna contenente i dati da visualizzare
myRange = myWorkSheet.get_Range("A3", "B23");
// chart myPage = myChart.Chart;
// ATTENZIONE: Occorre impostare PRIMA il ChartType e DOPO il DataSource.
myChart.Chart.ChartType = XlChartType.xlXYScatter;
myChart.Chart.SetSourceData(myRange);
myChart.Chart.HasTitle = true;
myChart.Chart.ChartTitle.Text = "Grafico funzione y=x^2";
myChart.Chart.ChartTitle.Font.Color = ColorTranslator.ToOle(Color.Green);
myChart.Chart.ChartTitle.Border.Color = ColorTranslator.ToOle(Color.Green);
myChart.Chart.HasLegend = true;
myChart.Chart.Legend.Border.Color = ColorTranslator.ToOle(Color.Red);
// La collection delle serie parte da 1
Series mySerie = (Series) myChart.Chart.SeriesCollection(1);
mySerie.Name = "f(x)";
mySerie.HasDataLabels = false; //true per inserire le etichette sul grafico
}

private void btnSalva_Click(object sender, EventArgs e) {
    myWorkbook.SaveAs(Forms.Application.StartupPath + "\\prova.xlsx");
    myWorkbook.ExportAsFixedFormat(XlFixedFormatType.xlTypePDF,
        Forms.Application.StartupPath + "\\prova.pdf");
    myChart.Chart.Export(Forms.Application.StartupPath + "\\grafico.bmp", "BMP")
    myWorkbook.Close();
    myExcel.Quit();
}

private void btnApri_Click(object sender, EventArgs e) {
    myExcel = new Microsoft.Office.Interop.Excel.Application();
    myExcel.Visible = true;
    myWorkbook = myExcel.Workbooks.Open(StartupPath + "\\prova.xlsx");
    myWorkSheet = (Worksheet)myWorkbook.Worksheets.getItem(1); // seleziono foglio 1
    Microsoft.Office.Interop.Excel.Range myRange;
    string s="", sx="", sy="";
    for (int i = 3; i <= 23; i++) {
        myRange = myWorkSheet.get_Range("A"+i.ToString());
        sx = myRange.Value.ToString();
        myRange = myWorkSheet.get_Range("B" + i.ToString());
        sy = myRange.Value.ToString();
        s += "(" + sx + "," + sy + ")\n";
    }
    MessageBox.Show(s);
}

```

Utilizzo del controllo msChart

```
using System.Windows.Forms.DataVisualization.Charting;
```

Collections

L'oggetto msChart è costituito da 4 collection principali che sono:

- **ChartAreas** Area su cui verrà visualizzato il grafico
- **Series** Serie dei dati da visualizzare (X e Y)
- **Titles** Titolo del grafico
- **Legends** Eventuale Legenda

Ogni collection può contenere più oggetti del relativo tipo (ad esempio Titles può contenere più Titoli)
Prima di iniziare qualunque attività è raccomandato di ripulire tutte le quattro collections:

```
private void clearChart() {  
    myChart.ChartAreas.Clear();  
    myChart.Series.Clear();  
    myChart.Titles.Clear();  
    myChart.Legends.Clear();  
}
```

dove **myChart** è il nome assegnato al controllo msChart sulla Form.

Impostazione della Chart Area

```
myChart.ChartAreas.Add("myChartArea");  
// oppure  
ChartArea myChartArea = new ChartArea();  
myChart.ChartAreas.Add(myChartArea);  
myChartArea.Position.Auto = true;  
//  
myChart.ChartAreas["myChartArea"].Position.Auto = true;  
myChart.ChartAreas["myChartArea"].Area3DStyle.Enable3D = false;  
myChart.ChartAreas["myChartArea"].AxisX.Title = "1=LST 2=INF 3=ELT 4=MEC 5=RAG";  
myChart.ChartAreas["myChartArea"].AxisY.Title = "% Promossi";
```

Impostazione della Serie di dati

```
Series srPromossi2011 = new Series();  
srPromossi2011.Name = "Promossi 2011";  
Series srPromossi2012 = new Series();  
srPromossi2012.Name = "Promossi 2012";  
  
Random rnd = new Random();  
for (int i = 1; i < 6; i++) {  
    srPromossi2011.Points.Add(rnd.Next(30, 50));  
    srPromossi2012.Points.AddXY("x"+i, rnd.Next(30, 50));  
} // AddXY aggiunge anche una etichetta alle singole ascisse.  
  
myChart.Series.Add(srPromossi2011);  
myChart.Series.Add(srPromossi2012);  
  
myChart.Series["Promossi 2011"].Color = Color.Aqua;  
srPromossi2012.Color = Color.Magenta;  
myChart.Series["Promossi 2011"].IsValueShownAsLabel = true;  
srPromossi2012.IsValueShownAsLabel = true;
```

```
// Nel caso di chartAera unica le righe seguenti non servono
srPromossi2011.ChartArea = "myChartArea";
srPromossi2012.ChartArea = "myChartArea";
```

Impostazione di Titolo e Legenda

```
myChart.Titles.Add("Grafico Promossi a.s. 2011-2012");
myChart.Titles[0].BackColor = Color.Yellow;
myChart.Titles[0].BorderColor = Color.Red;
myChart.Titles[0].Font = new Font("Times New Roman", 12, FontStyle.Bold);
myChart.Legends.Add("Legenda");
```

Legenda è soltanto il nome assegnato all'oggetto Legend . All'interno della Legenda verranno automaticamente visualizzati i nomi delle 2 Serie.

Impostazione del tipo di Grafico

```
myChart.Series["Promossi 2011"].ChartType = SeriesChartType.Column;
myChart.Series["Promossi 2012"].ChartType = SeriesChartType.Column;
```

I tipi di grafici a linee sono :

- **Column** Barre Verticali
- **Bar** Barre Orizzontali
- **Line**
- **Point**

I tipi di grafici ad area sono :

- **Pie** Torta, **Pyramid** Piramide, **Funnel** Imbuto

I grafici a **linee** consentono di rappresentare più serie contemporaneamente.
Viceversa nei grafici ad **area** si può visualizzare una UNICA serie per volta,

In entrambi i tipi di grafico si imposta di solito **mySerie.IsValueShownAsLabel = true** in modo che sul diagramma vengano visualizzati i valori Y.

Nel caso dei grafici ad **area**, al posto dei valori Y, è comodo impostare una label alfanumerica a piacimento:

mySerie.Points[0].Label="Respinti" che verrà visualizzata sia sul grafico al posto del valore Y sia sulla Legenda. Volendo invece impostare il valore solo nella Legenda (con il valore numerico Y sul grafico) si può utilizzare la proprietà : **Serie.Points[0].LegendText = "NuovoNome"**

Acquisizione Real Time

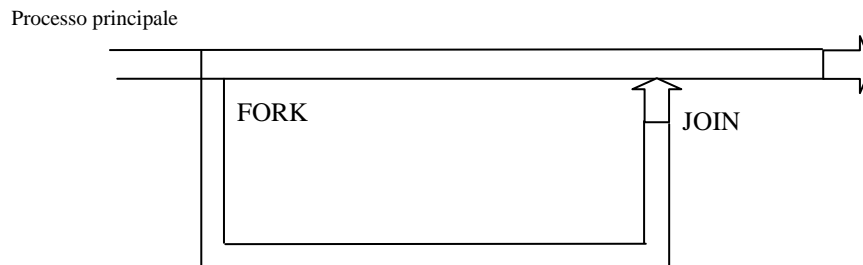
Se la ADD di un punto alla serie viene fatta sotto timer, il nuovo punto viene aggiunto al grafico in tempo reale. L'aggiornamento del grafico è automatico. Ogni volta il grafico viene però rimpicciolito in modo da farci stare anche il nuovo punto. Il problema è che dopo un pò il grafico diventa illeggibile, dunque occorre farlo scorrere verso sinistra eliminando i punti più vecchi.

```
int ora=0;
private void timer1_Tick(object sender, EventArgs e) {
    double valore = rnd.Next(25, 86);
    ora++;
    myChart.Series[0].Points.AddXY(ora, valore);

    myChart.ResetAutoValues(); // Reimposta la scala XY
    while (myChart.Series[0].Points.Count > Convert.ToInt(txtNchar.Text))
        myChart.Series[0].Points.RemoveAt(0);
    myChart.Invalidate(); // Forza il refresh del grafico }
}
```

I THREAD

NET consente di creare nuovi Thread, cioè, dato un processo in esecuzione sulla CPU, al fine di alleggerire la CPU stessa, è possibile sdoppiare il processo (mediante un tipico FORK) in più Thread concorrenti, **tutti però mappati nella stessa area di memoria**, dunque con la possibilità di condividere dei dati (ma dotati al contempo di dati propri).



Tipicamente il processo principale è costituito dall'interfaccia utente (form), mentre i processi figli in esecuzione parallela possono occuparsi della gestione di risorse periferiche, come la bufferizzazione dei dati provenienti da una seriale o da un socket, oppure possono gestire in background elaborazioni particolarmente pesanti che diversamente sarebbero bloccanti per la CPU. Occorre comunque non abusare dei thread in quanto, in sistemi monoprocesso, creano notevole over head.

Comunicazione fra Thread in NET

In NET gli eventi di ascolto della **SerialPort** vengono automaticamente gestiti all'interno di un Thread separato rispetto al Thread principale dell'interfaccia grafica. I due Thread possono condividere le variabili globali dell'applicazione (create prima di istanziare il thread) ma non possono condividere ad esempio i controlli della Form, visibili soltanto dalla Form e non dai Thread figli. Per poter "copiare" un valore restituito da un thread figlio (ad esempio un dato proveniente dalla seriale) all'interno di un Text Box della form principale, il thread figlio deve salvare il valore all'interno di una variabile globale, e poi **invocare** una procedura del form genitore che provveda a copiare il contenuto della variabile all'interno del Text Box.

Il metodo **Me.Invoke** consente di invocare dall'interno di un processo figlio un **Delegate**, cioè un **puntatore che andrà a puntare ad una procedura appartenente al Thread principale**. Il Me è riferito alla form. Nell'esempio viene creato un nuovo EventHandler (cioè un Delegate) e si associa a questo delegate la procedura DisplayData del Thread principale. Questa procedura, essendo una procedura di evento, avrà necessariamente i due soliti parametri relativi alle procedure di evento.

Dim dato As String

' Questo evento si attiva ogni volta che arrivano dati sulla porta seriale.

Private Sub Sp_DataReceived(sender As Object, e As IO.Ports.SerialDataReceivedEventArgs) Handles Sp.DataReceived

 If (e.EventType = IO.Ports.SerialData.Chars) Then

 Try

 dato = Sp.ReadLine()

 Catch ex As Exception

 MessageBox.Show(ex.Message)

 End Try

 End If

Me.Invoke(New EventHandler(AddressOf DisplayData))

End Sub

Private Sub DisplayData(ByVal sender As Object, ByVal e As EventArgs)

 txtDatiRicevuti.Text = dato

End Sub

I Thread in C#

```
Thread[] thLadro = new Thread[nLADRI];

for (int i = 0; i < nLADRI; i++)    {
    thLadro[i] = new Thread(new ParameterizedThreadStart(fThMoveLadro));
    thLadro[i].Start((object)i)
}

for (int i = 0; i < nLADRI; i++)    {
    if (posGuardia.X == posLadro[i].X &&
        posGuardia.Y == posLadro[i].Y &&
        thLadro[i].IsAlive)
    {
        nLadriCatturati++;
        thLadro[i].Abort();
    }
}
```

Una procedura interna ad un thread può invocare una procedura appartenente al thread principale utilizzando la seguente sintassi:

```
this.Invoke(New EventHandler(AddressOf DisplayData))
```

dove display data è una normalissima procedura (con la firma tipica delle procedure di evento) appartenente al thread principale:

```
Private Sub DisplayData(ByVal sender As Object, ByVal e As EventArgs)
    txtDatiRicevuti.Text = dato
End Sub
```

E' anche possibile da un thread richiamare una procedura appartenente ad un altro thread, ma occorre approfondire la sintassi.

ADO Net

Il Namespace **System.Data** contiene 4 differenti **provider di dati**, aventi tutti la stessa interfaccia (polimorfismo) **System.Data.OleDb** basato sulla tecnologia COM per l'accesso diretto ai database che disponga di un **provider** OleDb (ad esempio Access, SQL Server fino alla versione 6.5, Oracle antecedente a 8.1.7) **System.Data.SqlClient** Provider di accesso diretto ad SQL Server, basato sul nuovo protocollo TDS (Tabular Data Stream) utilizzato da SQL Server 7.0 in avanti (vale a dire da SQL Server 2000 in avanti). **System.Data.OracleClient** Provider di accesso diretto a origini dati Oracle dalla versione 8.1.7 in avanti **System.Data.Odbc** provider di accesso ODBC per qualunque database che disponga di un **driver** conforme allo standard ODBC (ad esempio MySql) (OleDb parla di provider invece che di driver).

Modello Oggetti di ADO NET (ActiveX Data Object .NET)

ADO NET è costituito da un insieme di oggetti indipendenti, ciascuno preposto a svolgere una certa funzionalità. I vari oggetti interagiscono secondo il seguente modello gerarchico a **4 livelli** (partendo dal basso):

- 1) **OleDbConnection** – Gestisce la comunicazione fisica con il database
- 2) **OleDbCommand** – Consente di impostare i comandi da “passare” al database (ad esempio il nome della tabella da leggere)
- 3a) **OleDbDataAdapter** – Lancia in esecuzione il comando memorizzato all'interno dell'oggetto Command. Eseguendo un accesso in *modalità asincrona* nel senso che, mediante il **metodo Fill**, esegue una lettura dei dati del database, **copiandoli** all'interno di un oggetto disconnesso detto dataTable o dataSet. Il **metodo .Update** consente invece di salvare i dati dal dataSet/dataTable al database.
- 3b) **OleDbDataReader** – consente di accedere al database in *modalità connessa*, nel senso che mostra direttamente i record del database, senza creare copie. Legge però soltanto un record per volta ed in modo Read Only.
- 4) **DataTable** e **DataSet** sono oggetti disconnessi dal database (recordset statici) all'interno dei quali l'oggetto Adapter va a copiare i dati letti. La differenza tra i due oggetti è che il **dataTable consiste in una unica tabella**, mentre il **dataSet può contenere un insieme di più tabelle**.

1° Livello : Oggetto CONNECTION -> Creazione della connessione

- 1) Istanza dell'oggetto: `OleDbConnection cn = new OleDbConnection();`
- 2) Impostazione della proprietà **ConnectionString**, cioè la stringa di connessione al database. Dipende dal tipo di database a cui ci si vuole connettere. Nel caso di ACCESS la stringa di connes è costituita da una serie di coppie Chiave=Valore delimitate da punto e virgola (come ADO COM). Può essere modificata solo a connessione chiusa.
`s = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=Libri.mdb;Jet OLEDB:Database Password=pippo";`
`cn.ConnectionString = s;`
- 3) Apertura della connessione mediante il metodo **Open()** *Es* `cn.Open()` L'oggetto adapter non necessita dell'apertura della connessione, in quanto si apre autonomamente la connessione su cui è appoggiato. La proprietà **State** indica lo stato della connessione (Open, Closed, Connecting)
Il metodo **CreateCommand()** consente di creare un oggetto Command appoggiato sulla connessione corrente.

2° Livello : Oggetto COMMAND -> Impostazione del comando da eseguire

Consente di impostare il **comando SQL** da inviare al database. **Come comando SQL si può intendere semplicemente il nome della tabella del database a cui ci si vuole connettere.**

La proprietà **Connection** consente di associare l'oggetto Command ad una Connessione sottostante

```
OleDbCommand cmd = new OleDbCommand();  
cmd.Connection = cn;
```

In alternativa si può utilizzare il metodo CreateCommand dell'oggetto Connection:

```
OleDbCommand cmd = cn.CreateCommand();
```

La proprietà **CommandType** indica il tipo di comando che si sta inviando. Può assumere i seguenti valori:

CommandType.**TableDirect** = nome della tabella da leggere

CommandType.**Text** = comando SQL vero e proprio di tipo Select, Insert, Delete o Update

CommandType.**StoredProcedure** = Procedura SQL memorizzato all'interno del database.

Se non si specifica il CommandType, il comando può essere lanciato comunque, ma più lentamente perché ADO deve individuare da se, per tentativi, di quale tipo di comando si tratta.

La proprietà **CommandText** di tipo stringa consente di specificare **il comando vero e proprio**, ad esempio il nome della tabella a cui connettersi.

Il metodo **.Prepare()** consente di precompilare il comando SQL da inviare al DBMS. La prima esecuzione sarà più lenta, ma tutte le esecuzione successive risulteranno molto velocizzate.

Il metodo **.ExecuteScalar()** dell'oggetto Command consente l'esecuzione di query scalari.

Il metodo **.ExecuteNonQuery()** dell'oggetto Command consente l'esecuzione di comandi di tipo DML e DDL.

3° Livello Interrogazioni Tabellari

Per eseguire interrogazioni tabellari, cioè interrogazioni che restituiscono un insieme di record (detto **recordset**), sono disponibili due oggetti: **DataAdapter** (modalità disconnessa) e **DataReader** (modalità connessa).

Oggetto DataReader

Oggetto basato su un cursore **server** di tipo **ForwardOnly** con locktype = **ReadOnly** (cioè senza nessun blocco sui record). **E' molto più veloce rispetto al DataAdapter, ma consente soltanto la lettura dei dati e non la scrittura.** Eventuali aggiornamenti devono essere eseguiti *manualmente* mediante Command asincroni. Non sono ammessi altri tipi di cursore come ad esempio il cursore **Dynamic** molto utilizzato in ADO COM ma molto oneroso da gestire (dunque poco affidabile). L'aggiornamento manuale mediante Command comporta sicuramente la scrittura di più codice client ma è indubbiamente molto più affidabile.

Il metodo **.ExecuteReader()** dell'oggetto Command, quando invocato, restituisce un oggetto DataReader, cioè un insieme di record connessi al database ma Read Only / Forward Only. Questo recordset può essere utilizzato ad esempio per caricare manualmente i dati dentro una lista.

A livello di esecuzione **questa tecnica è molto più rapida rispetto a qualunque data binding.**

Il metodo **.Read()** consente di leggere un record alla volta spostando automaticamente il cursore sul prossimo record.

```
OleDbDataReader rdr = cmd.ExecuteReader();  
while (rdr.Read()) // restituisce True se ha letto qualcosa  
    ListBox1.Items.Add(rdr["nomeCampo"].ToString());
```

Oggetto DataAdapter

L'oggetto DataAdapter stabilisce una comunicazione asincrona con il database, che può avvenire anche a connessione chiusa. Quando si istanzia l'oggetto DataAdapter si passa in genere come parametro l'oggetto Command che a sua volta utilizzerà la Connection sottostante. In alternativa, esiste un costruttore di overload che, *senza passare attraverso l'oggetto Command*, accetta come primo parametro direttamente il comando SQL (nome della tabella a cui collegarsi) e come secondo parametro l'oggetto Connection a cui fare riferimento.

```
OleDbDataAdapter adp = new OleDbDataAdapter(cmd);  
OleDbDataAdapter adp = new OleDbDataAdapter(sql, cn);
```

oppure

Il metodo **fill** consente di caricare un insieme di dati (**recordset**) da database a dataTable

Il metodo **update** consente di salvare un insieme di dati (**recordset**) da dataTable a database

Prima di richiamare il metodo **fill** occorre però istanziare il dataTable o il dataSet

4° Livello Oggetti disconnessi DataSet e DataTable

```
DataTable daTable = new DataTable();           oppure  
DataSet daSet = new DataSet();
```

L'oggetto **DataTable** consente di caricare al suo interno un unico recordset (in pratica una unica tabella)

L'oggetto **DataSet** consente di gestire al suo interno più tabelle parallele. All'interno del DataSet è possibile impostare indici e relazioni fra le varie tabelle, costruendo di fatto un vero e proprio database in memoria. Su tutte le relazioni viene implementato il vincolo di integrità referenziale, con aggiornamenti ed eliminazioni a catena.

Il metodo Adapter.Fill()

Esegue fisicamente la lettura dei dati dal database, caricandoli all'interno di un oggetto dataTable / data Set disconnesso dal database. **Sostanzialmente esegue una copia dei dati dal database al dataTable.**

```
adp.Fill(daTable);                               oppure  
adp.Fill(daSet, "Tabella1");
```

Nel caso del DataSet, occorre specificare anche un **secondo parametro** che indica il nome da assegnare a tale recordset (in pratica viene creata, nel dataSet in memoria, una nuova tabella avente il nome indicato, all'interno della quale vengono caricati i dati). Ogni tabella del dataSet è in realtà un dataTable.

Il metodo .Fill() presenta un overload che consente di caricare nel DataTable soltanto un certo numero di record:
`adp.Fill(daTable, startIndex, qta);` dove
startIndex è l'indice del primo record da caricare (fra quelli restituiti) e *qta* rappresenta il n° di record da caricare.

La proprietà `adp.MissingSchemaAction = MissingSchemaAction.AddWithKey`

da impostare prima del Fill, indica all'adapter che, al momento del Fill, dovrà caricare nella dataTable anche le informazioni relative alla chiave primaria. La presenza di una chiave primaria all'interno di una dataTable sarà condizione *indispensabile* per l'esecuzione del metodo Find sul DataTable e anche per l'update automatico del dataAdapter (in modo che venga rifiutata l'aggiunta di nuovi record con chiavi primarie già esistenti).

Il metodo Adapter.Update()

Consente di salvare i dati dal dataTable al database, **sovrascrivendo** i record modificati (anche in caso di modifica di modifica della chiave primaria). Si aspetta come parametro il nome del dataTable da salvare.

```
myAdapter.update(daTable)  
myAdapter.update(daSet, "Tabella1")
```

Il metodo update esegue una scansione di tutti i record presenti nel dataTable ed analizza la proprietà RowState. Se lo stato è *Modified*, il record verrà salvato nel database. Se è *Added*, verrà aggiunto in coda alla tabella.

*Prima di richiamare il metodo update, **occorrerebbe** impostare all'interno dell'oggetto Command una apposita **CommandString** che indichi le modalità di salvataggio. A tal fine l'obj Adapter dispone di tre proprietà di tipo String (**InsertCommand, UpdateCommand, DeleteCommand**) in cui occorre scrivere in modo parametrizzato i comandi SQL da "passare" all'oggetto COMMAND. Durante la scansione dei record, il metodo Update, prima di passare la CommandString all'oggetto Command provvederà a completarla con i valori relativi al record corrente.*

<i>Se RowState = Added,</i>	<i>verrà eseguita la InsertCommand</i>
<i>Se RowState = Modified,</i>	<i>verrà eseguito la UpdateCommand</i>
<i>Se RowState = Deleted,</i>	<i>verrà eseguito la DeleteCommand</i>

La costruzione manuale di questi comandi SQL è però tutt'altro che semplice.

5° Livello : Collegamento di una Griglia ad un DataTable

```
grid.DataSource = daTable;           // La griglia punta ai dati del dataTable
grid.DataSource = dataSet.Tables("nomeTabella");
```

DataGrid presenta una sintassi alternativa poco utilizzata basata sul DataMember (non utilizzabile su ListBox e ComboBox).

```
grid.DataSource = dataSet;           grid.DataMember = "TableName";
```

Collegamento di una Lista o un Combo ad un DataTable

Il collegamento può essere eseguito in automatico, legando il ListBox ad una colonna del DataTable, oppure in manuale, scorrendo tutti i record del DataTable e caricando nella Lista soltanto i valori desiderati.

Binding automatico:

```
lstNomi.DataSource = daTable;
lstNomi.DisplayMember = "Nome Colonna da Visualizzare";
lstNomi.ValueMember = "Nome Colonna Nascosta";

Text consente di accedere al valore selezionato
SelectedValue consente di accedere al corrispondente valore nascosto
SelectedIndex consente di accedere all'indice della voce selezionata
```

Caricamento manuale:

```
for(int i = 0; i < daTable.Rows.Count; i++)
    ListBox1.Items.Add(daTable.Rows[i]["NomeColonna"].ToString());
```

NB 1 L'evento SelectedIndexChanged viene richiamato anche all'inizio quando si fa il binding del controllo al database. In questa fase i dati non sono ancora pronti e dunque può verificarsi un errore. Tre possibili soluzioni:

- avvolgere le istruzioni dell'evento selectedIndexChanged all'interno di un **TRY CATH**
- all'inizio di selectedIndexChanged eseguire il seguente test
if (! cbo.selectedValue.GetType().IsValueType) return.;
- Impostare al termine del Form Load **cmbBox.Tag="OK"** e all'inizio di selectedIndexChanged fare:
If (cmbBox.Tag != null && cmbBox.Tag.ToString() == "OK")

NB 2: Quando un ListBox ComboBox è linkato ad un DataSource, nel momento in cui si vuole accedere alle singole voci mediante `Items.Item[i]`, occorre necessariamente specificare anche a quale campo si vuole accedere fra il `DisplayMember` ed il `ValueMember`. Occorrerà cioè scrivere `Items.Item[i]["NomeCampo"]`. In alternativa il metodo **myList.GetItemText(myList.Items[i])** restituisce il `DisplayMember` selezionato.

Binding semplice di un Text Box ad un campo di un DataTable

A differenza di DataGrid, ListBox e Combo Box che supportano il cosiddetto **Binding Complesso**, (collegamento del controllo ad una intera sorgente di dati), molti controlli VB supportano il cosiddetto **Binding Semplice**, che consiste nel linkare una **proprietà** del controllo ad un singolo campo di un DataSet / dataTable:

```
lblCodice.DataBindings.Add("text", daTable, "Codice");
lblCantante.DataBindings.Add("text", daTable, "Cantante");
lblTitolo.DataBindings.Add("text", daTable, "Titolo");
```

- Il primo parametro indica la proprietà della Label (o altro controllo) che deve essere linkata al dataTable.
- Il terzo parametro indica il campo. Nell'esempio la proprietà **Text** delle varie Label viene linkata ai diversi campi dell'oggetto daTable. **In questo modo all'interno delle label in ogni momento verranno mostrati i valori relativi al record corrente.** Mentre la proprietà Text viene linkata al campo Cantante, un'altra proprietà della stessa Label (es BackColor) potrebbe essere linkata ad un altro campo della tabella (es il campo "Colore"). Al posto del dataTable si può utilizzare anche un Binding Source.

In alternativa è possibile utilizzare la seguente sintassi basata sull'oggetto **BINDING**:

```
Binding sr = new Binding("text", srcDischi, "titolo");
lblTitolo.DataBindings.Add((sr);
```

Proprietà e Metodi di un DataSet / DataTable

Il DataSet è visto in NET come una generica sorgente dati (al pari ad esempio di un XmlDocument), che può essere scandita mediante dei cicli, ma che in pratica non supporta il concetto di record “attualmente in uso” da parte dell’applicazione. Dunque non dispone di alcun metodo di navigazione.

D’altro lato, invece, il collegamento tra controlli rinati e DataSet è bidirezionale. Se l’utente esegue delle modifiche su un controllo linkato, queste modifiche vengono immediatamente riportate all’interno del DataSet che è un tutt’uno con i controlli.

Le seguenti Proprietà / Metodi valgono per un oggetto DataTable o per una singola tabella di un DataSet.

.Clear() Cancella tutti i record della tabella, senza toccare la struttura delle colonne

Il metodo Clear() è disponibile anche sull’oggetto DataSet e cancella il contenuto dell’intero DataSet.

.Reset() Cancella sia i dati sia la struttura dell’intera tabella (colonne).

.PrimaryKey Consente di definire una chiave primaria all’interno della tabella. Ad esempio:

dataTable.PrimaryKey = New DataColumn(dataTable.Columns("ID")) ‘Associa la chiave primaria al campo ID

Oggetto Rows Insieme delle righe del dataSet

Collezione di oggetti DataRow che rappresentano le singole righe della tabella.

.Count Numero complessivo di righe

.Add(DataRow riga) Consente di aggiungere una nuova riga in coda al dataTable. Esempio

```
DataRow riga = dataTable.NewRow();  
riga["Nome"] = "xxxxxxx";  
riga["Cognome"] = "yyyyyyyy";  
dataTable.Rows.Add(riga);
```

.Find(primaryKeyValue) Consente di ricercare rapidamente il record avente la chiave primaria indicata.

La ricerca riparte sempre dall’inizio della tabella. Affinché il metodo Find possa funzionare, è necessario, al momento del popolamento del dataSet, caricare anche l’informazione relativa alla chiave primaria.

Find restituisce l’oggetto DataRow relativo al record ricercato, Nothing in caso di insuccesso.

Oggetto Rows[i] Riga i-esima all’interno del dataSet

E’ possibile accedere alla singola cella con “**indicizzazione diretta**” tramite nome o indice:

```
dataTable.Rows[i]["Titolo"];  
dataTable.Rows[i][3];
```

oppure, in alternativa, è possibile utilizzare le seguenti proprietà:

.ItemArray[index] Restituisca **in sola lettura** il contenuto del campo indicato.

Come parametro si può utilizzare soltanto l’indice; non è ammesso il nome del campo.

.SetField(columnName, value) **Imposta** il contenuto del campo indicato al valore indicato.

Come parametro si può utilizzare indifferentemente l’indice oppure il nome del campo.

Esempio di scansione dei record di un dataSet

```
for (int i = 0; i < dataTable.Rows.Count; i++)  
    MsgBox(dataTable.Rows[i].ItemArray[2]);
```

Le celle sono comunque oggetti, per cui, almeno per il confronto, occorre SEMPRE fare il .toString()

.BeginEdit() mette la riga in modalità di modifica. Non è necessario, in quando lo stato di modifica subentra automaticamente in corrispondenza della prima variazione

.EndEdit() conferma la modifica. Come in ADO COM, le modifiche vengono automaticamente confermate (e quindi copiate nel dataSet) nel momento in cui si sposta il cursore su un record differente. Se però l’utente modifica un record e poi, senza spostarsi di posizione, lancia la funzione “Salva”, le modifiche sul record andranno perse. Per evitare questa situazione, “Salva” dovrebbe, per prima cosa, terminare l’eventuale editazione del record in corso richiamando appunto il metodo EndEdit().

.CancelEdit() annulla tutte le modifiche fatte sul record in editazione, riportando tutti i campi al valore che avevano in precedenza.

- .Remove()** elimina il record corrente dal dataSet, marchiandolo come “Detached”, cioè scollegato dall’insieme delle righe. Questo record non viene più conteggiato all’interno di Count e risulta escluso da qualunque scansione. Questo metodo ha senso quando si fa uso di un dataSet senza un database di Back – End (infatti se il record viene rimosso dal dataSet, non ha più alcuna possibilità di essere rimosso dal database!).
- .Delete()** cancella il record corrente, che viene marchiato come “Deleted” all’interno del dataSet.
- .GetChildRows(“RelationName”)** In caso relazione 1: N fra tabelle, restituisce in un vettore tutti i record correlati al record corrente (metodo applicabile soltanto ai record della tabella 1).
- .RowState** stato della riga corrente. Può essere Unchanged oppure **Modified**, **Added**, **Deleted**, oppure **Detached** che significa che si tratta di un nuovo record appena creato e non ancora linkato all’insieme delle Rows, oppure di un record rimosso mediante il metodo Remove.

Oggetto Columns Insieme delle colonne

Collezione di oggetti **DataColumn** che descrivono i vari campi del record corrente.

Partendo da un dataTable vuoto occorre creare prima le colonne, dopo le righe.

```
DataTable dataTable = new DataTable("Immagini");
DataColumn colonna1 = dt.Columns.Add("titolo", typeof(String));
    colonna1.AllowDBNull = false;
    colonna1.Unique = true;
dataTable.Columns.Add("link", typeof(String));
.Count Numero complessivo di colonne
[“NomeColonna”] Consente di accedere alla colonna avente il nome o l’indice passato come parametro.
```

Proprietà dell’oggetto DataColumn o Columns[i]

- .ColumnName** Nome della colonna
- .DataType** Tipo di dati della colonna
- .Caption** Intestazione della colonna
- .ReadOnly** Consente di impostare la modalità Read Only sulla colonna attuale
- .AllowDBNull** = True /False Accetta Null come valore valido all’interno della colonna
- .Unique** = True /False Fa sì all’interno della colonna siano accettati soltanto valori univoci

Esempio di scansione dei record di un dataSet

```
for (int j = 0; j< dataTable.Columns.Count; j++)
    s += dataTable.Columns[j].ColumnName + "\t";
s += "\n";
for (int i = 0; i< dataTable.Rows.Count; i++) {
    for (int j = 0; j< dataTable.Columns.Count; j++)
        s += dataTable.Rows[i].ItemArray[j] + "\t";
    s += "\n";
}
```

oppure

```
foreach (DataColumn col in dataTable.Columns)
```

Test su un Campo Nullo

Per verificare se un certo campo di un certo record contiene il valore speciale NULL, occorre utilizzare la seguente sintassi:

```
if (ado.dataTable.Rows[i].ItemArray[j] == DBNull.Value) { ..... }
```

Creazione di una DLL Compilata

- 1) Creare un nuovo progetto di tipo **Class Library** e assegnargli il nome adoNet
- 2) Assegnare alla classe il nome **adoNet** e scrivere il codice
- 3) Assegnare al root Namespace il nome **adoDLL**
- 4) Compilare mediante Build

Utilizzo della DLL da parte di una Window Application

- 1) Copiare la dll all'interno della cartella BIN/DEBUG dell'applicazione
- 2) Da MyProject / References fare **Add Reference** / Browse, selezionando la dll all'interno della cartella BIN/DEBUG dell'applicazione
- 3) All'inizio del codice eseguire : **using adoDLL;**

La dll risulta così legata all'applicazione e quindi utilizzabile.

L'intellisense mostrerà proprietà e metodi della classe come per qualsiasi altro oggetto.

Note sull'utilizzo della DLL

- 1) Anche senza copiare la dll all'interno della cartella dell'applicazione, nel momento in cui si fa **Add Reference** si può andare a cercare la dll all'interno della sua cartella originale di creazione.
In fase di compilazione del Window Project la ddl verrà automaticamente copiata all'interno della cartella BIN/DEBUG del progetto
- 2) All'interno della dll possono anche essere aggiunte delle Windows Forms (ad esempio una About Form contenente alcune informazioni sulla classe), Forms che dovranno essere gestite e visualizzate dall'interno della classe adoNet. Ad esempio si potrebbe dotare la classe di un metodo **void About()** così strutturato:

```
frmAbout frm = new frmAbout();  
frm.ShowDialog();  
frm = null;
```

La form avrà un unico pulsante Chiudi. Lo showDialog blocca fino alla chiusura della form About..

Stringhe di Connessione

'Access 2003 : dbName = Percorso del file .mdb
cnString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & dbName & ";"
cnString += "Jet OLEDB:Database Password=pippo;"

'Access 2007 : dbName = Percorso del file .mdb
cnString = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" & dbName & ";"
cnString += "Jet OLEDB:Database Password=pippo;" Persist Security Info=False;

'SQL Server 2005 Express : dbName = Percorso del file .mdf
cnString = "Data Source=.\SQLEXPRESS;AttachDbFilename=" & dbName & ";"
cnString &= "Integrated Security=True;Connect Timeout=30;User Instance=True;"

'SQL Server 2000 / 2005 : dbName = Nome del database (es. dbLibri)
cnString = "Data Source=localhost;Integrated security=SSPI;Initial Catalog=" & dbName & ";"
cnString &= "user id=sa;password=";

MySQL

cnString = "server=localhost;user id=" & user & ";password=" & psw & ";port=" & port & ";"
database=" & dbName

Il controllo BindingSource [System.Windows.Forms]

Il dataSet, essendo visto come una genetica sorgente dati, non consente la navigazione visuale dei record. In caso di necessità di navigazione si può utilizzare un ulteriore oggetto, il BindingSource, a sua volta appoggiato su una tabella di un DataSet / dataTable. Il BindingSource, rispetto al DataSet gestisce

1. la **navigazione** visuale fra i record mediante il concetto di **record corrente**: proprietà Position e CurrentItem, metodi Move, possibilità di cancellare il record visualmente selezionato mediante il metodo Delete(), più un interessante evento, Position_Changed, richiamato quando l'utente seleziona un nuovo record.
2. l'impostazione di **viste personalizzate** mediante **Filter** e **Sort** (senza dover ricorrere all'oggetto View)
3. l'esecuzione di **ricerche dirette** mediante il metodo **Find**

Il BindingSource viene fisicamente interposto fra il DataSet e la DataGridView nel modo seguente:

```
public BindingSource srcDischi = new BindingSource();
srcDischi.DataSource = adoDischi.dataTable;
grid.DataSource = srcDischi;
```

Il Binding Source realizza una **Vista** al di sopra del Data Table. Per cui, quando si imposta un filtro :

- **il BindingSource vede solo i record filtrati,**
- **il DataTable continua a vedere TUTTI i record.**

In pratica la Vista non viene applicata all'intero DataTable, ma solo alla sua proprietà DataSource.

Viceversa le variazioni fatte sul BindingSource vengono intercettate anche sul Data Table

Quando si vuole inserire un nuovo record e controllare che il codice non sia già esistente, la ricerca **deve** essere fatta sul DataTable, in quanto il BindingSource vede solo i record filtrati. Viceversa l'eliminazione del record corrente deve essere fatta sul Binding Source in quanto, se si è impostato un filtro, i dati non sono allineati.

Proprietà / Metodi del Binding Source

.Count Numero di record visti dal BindingSource (chiaramente lo stesso del dataSet)

[i] ["NomeCampo"] Consente di accedere al singolo campo del record i-esimo. Il record è però visto come un **Object**, mentre ["NomeCampo"] può essere applicato **solo** a controlli di tipo DataRowView

Per cui occorre necessariamente eseguire il seguente cast:

```
DataRowView rec = (DataRowView) src[i];
rec["titolo"]="Il fu Mattia Pascal"
```

.Position Posizione attuale del cursore all'interno del recordset (compresa fra 0 e count - 1)

.Current ["NomeCampo"] Consente di accedere ai campi del record corrente.

.MoveFirst() Sposta il cursore sul primo record

.MoveNext() Sposta il cursore sul record successivo

.MovePrevious() Sposta il cursore sul record precedente

.MoveLast() Sposta il cursore sull'ultimo record

.Filter = "NomeCampo = " + valVariabile + " " Visualizza soltanto i record aventi quel valore su quel campo. E' consentito impostare più condizioni utilizzando gli operatori AND e OR all'interno della stringa di filtro. Occorre però ripetere il nome del campo: **.Filter** = "Autore = 'Pirandello' AND prezzo < '10' "

.Sort = "NomeCampo1 ASC/DESC, NomeCampo2" 'Ordinamento sulla base dei campi indicati. ASC è il default

.Find (campoRicerca As String, valoreDaRicerca As String) restituisce la Posizione della prima occorrenza incontrata (da scriversi nella proprietà Position, in modo da far diventare la ricorrenza record corrente). La ricerca parte sempre dal **primo** record. Il confronto è case insensitive Se la ricerca non produce risultati, Find restituisce -1

.RemoveSort() Rimuove l'eventuale ordinamento

.RemoveFilter() Rimuove l'eventuale filtro

.List() Quando si imposta un filtro, il Binding Source conterrà solamente i record filtrati, ma il dataSet sottostante continua a mostrare l'elenco complessivo dei record. La proprietà LIST del Binding Source contiene in tal caso l'elenco dei record filtrati. **Alcuni controlli (come ad esempio Crystal Report) non possono essere legati direttamente al BindingSource, ma devono essere legati a BindingSource.List**

.AddNew() Aggiunge un nuovo record vuoto in fondo al dataSet. Dopo l'AddNew occorre fare un MoveLast() per far diventare il nuovo record *Record Corrente* e quindi andare a scrivere dentro il record. Attenzione che, terminata l'editazione, il dataSet vede il nuovo record solo dopo un ulteriore metodo Move oppure un **EndEdit**.

.Add(obj) Aggiunge un oggetto già creato in coda al binding source

.RemoveCurrent() Rimuove il record corrente

.RemoveAt(i) Rimuove il record i-esimo. Quando si fa una ricerca sequenziale e si individua il record da eliminare, tale record **non è** il record corrente, quindi o si usa RemoveAt(i), oppure lo si fa diventare record corrente scrivendo i dentro Position.

.EndEdit() Termina l'editazione in corso, salvando le modifiche nel DataTable. Da farsi dopo ogni modifica, altrimenti quando si va a salvare il DataTable il record viene visto come "in editazione" e **non viene salvato**.

L'evento **Position_Changed()** viene richiamato ogni volta che la proprietà Position subisce una modifica. Utile per gestire una label numerica relativa alla posizione del navigatore.

Nota 1: Se il campo codice è di tipo contatore, nel momento in cui si fa AddNew, avendo impostato la chiave primaria, il .count viene subito automaticamente incrementato.

Nota 2: se dopo aver fatto AddNew si va a leggere l'informazione selezionata in un ListBox / Combo Box, in seguito all'ultimo MoveLast il cursore si trova sul record finale vuoto, dunque non più sul record selezionato dall'utente. A tal fine occorre leggere i valori selezionati nelle Liste *prima* di fare l'AddNew.

Nota 3: Due Binding Source differenti non possono agire sullo stesso dataTable. Perlomeno, in tal caso, quando si applicano contemporaneamente dei filtri sui due Binding Source, questi agiscono contemporaneamente per cui, alla fine, entrambi i BindingSource mostreranno come risultato l'intersezione dei due Filtri. In pratica ogni BindingSource deve *sempre* far riferimento ad oggetti dataTable differenti (cioè istanze indipendenti di adoNet)

Le Viste - View

Su ogni tabella del dataSet è possibile definire una vista, cioè un modo diverso di vedere i dati della tabella :

- Ordinati sulla base di un certo campo **.Sort**
- Filtrati sulla base di una certa condizione **.RowFilter**
- Filtrati sulla base del RowState (Unchanged, Modified, Added, Deleted) **.RowStateFilter**

```
Private void btnView_Click( ) {  
    String s = "";  
    DataView daView = new DataView(daSet.Tables("tabPiloti"))  
    daView.Sort = "CodiceSquadra";  
    daView.RowFilter = "CodicePilota < 20 and CodiceNazione > 9";  
    DataGridView1.DataSource = daView;  
    'oppure, lavorando da codice:  
    foreach (DataRowView rec in daView)  
        s += rec("CodicePilota") + " " + rec("Nome") + vbCrLf  
    MsgBox(s);  
}
```

Salvataggio di un DataSet in formato XML

Sia l'oggetto dataSet sia le singole Table dispongono del metodo **.WriteXml("NomeFile.xml")** che consente di salvare l'intero dataSet oppure una singola tabella su un file in formato XML. Se il file non esiste viene automaticamente creato. Se già esiste viene sovrascritto. Il metodo analogo **.ReadXml("NomeFile.xml")** consente di leggere le informazioni all'interno di un file XML e **accodarle** nel dataTable (preventivamente **azzeratio**) .

Crystal Report

Come creare un Report di stampa dei dati contenuti all'interno di un dataSet o di un dataTable. Tre passi:

- Definizione della struttura dei dati da visualizzare
- Creazione grafica del report
- Link del report al dataSet vero e proprio

1) Definizione della struttura dei dati da visualizzare sul Report. Fare tasto destro / Add New Item e aggiungere un componente **dataSet**. Questo passo consente di definire (e salvare su un file .xsd) soltanto la struttura del dataSet da utilizzare per visualizzare i dati sul report, cioè lo schema XML del dataSet stesso (che in realtà è un albero XML). Salvare questo file come **schemaReport1.xsd**.

Lo schema XML può essere costruito in modo visuale visualizzando la Toolbox del dataSet e **trascinando un oggetto dataTable**. Assegnare a questa tabella il nome **schemaReport1**. Sulla dataTable fare **tasto destro / Add Column** e aggiungere il nome del primo campo che si intende visualizzare sul Report. Questo nome deve coincidere col nome di un campo della tabella oppure con il nome di un campo restituito da una query SQL. Aggiungere via via i nomi di tutti i campi da visualizzare. Aggiungere eventualmente l'informazione relativa alla chiave primaria (che consentirà di fare ricerche direttamente sul Report). Per ogni campo è molto importante aggiungere il **Data Type** del campo stesso. Eventuali elaborazioni numeriche o relative alle date saranno rispettivamente possibili soltanto su campi Numerici o di tipo Date Time.

2) Creazione del Report vero e proprio. Fare tasto destro / Add New Item e aggiungere un componente **CrystalReport**. Salvare questo file come **Report1.rpt**. CrystalReport propone diverse tipologie grafiche di Report. Selezionare Report Wizard, Tipologia Standard (default).

Dalla Cartella Project Data / ADO NET DATASETS selezionare lo schema XML **schemaReport1** precedentemente creato. Trascinarlo nell'area di destra delle *Selected Tables* e fare Avanti.

La finestra successiva consente di selezionare quali campi dello schema si intende visualizzare sul Report (probabilmente tutti, ma non necessariamente. Qualcuno potrebbe essere utilizzato soltanto per calcoli).

La finestra successiva consente di definire dei gruppi sulla base di un certo campo (ad esempio visualizzare i record raggruppati in base al nome del cantante).

La finestra successiva consente di definire una operazione di **Summary** rispetto ai vari gruppi, ad esempio il numero totale di dischi presenti per quel cantante, oppure il valore complessivo di quei dischi (somma dei prezzi). Il combo inferiore consente di scegliere fra un numero elevatissimo di funzioni matematiche. Occorre inoltre indicare su quale campo numerico devono agire.

La finestra successiva consente di impostare un ordinamento sulla base del valore numerico di Summary (altrimenti l'ordinamento viene automaticamente fatto sulla base del campo utilizzato per creare i gruppi).

Le ultime due finestre consentono una ulteriore personalizzazione dello stile grafico del report.

Al termine viene creato un Report suddiviso nelle seguenti sezioni:

Report Header	Intestazione del documento (per default SUPPRESS=TRUE, dunque non visibile)
Page Header	Intestazione della pagina
Group Header	Intestazione dei Gruppi
Details	Dati
Group Footer	Fine dei Gruppi
Page Footer	Fine pagina
Report Footer	Fine documento

Per vedere le proprietà delle varie sezioni, clickare sulla barra della sezione

Per creare eventuali gruppi in un secondo tempo, fare *Tasto Destro / Insert Group* e selezionare il campo da utilizzare per fare i gruppi. Per creare i Summary andare col mouse nella sezione GROUP FOOTER e fare *Tasto Destro / Insert Summary* - **SummaryLocation = Group1**

Oggetti della Toolbox

Quando è visualizzato il Report, la Toolbox contiene soltanto pochi oggetti utilizzabili sul report stesso:

ITextBox cioè TextBox non linkati a database, in pratica delle Label, utilizzabili per impostare dei titoli come ad esempio, nel Page Header, l'intestazione delle varie colonne.

ILineObject che consente di tracciare linee sul Report

IBoxObject che può fungere da contenitore per immagini.

Field Explorer

La finestra Field Explorer (visualizzabile mediante **View / Other Windows / Document Outline**) consente di aggiungere al Report i seguenti interessanti oggetti:

IFieldObject (Database Fields), cioè TextBox linkati ai campi impostati nello schema XML. Questi TextBox vengono in genere utilizzati nella sezione Details e verranno ripetuti tante volte quanti sono i record.

SpecialFields campi speciali da utilizzare nelle intestazioni / piè di pagina, come ad esempio Page Number, Record Number, Date Time, etc.

RunningTotalFields consentono di impostare campi di Summery (es totale dei prezzi) tipo quelli creati dal wizard. Fare tasto destro / new, selezionare il campo da sommare, impostare **Evaluate** = For Each Record, impostare **Reset** = On Change Group (cioè il contatore viene resettato ad ogni cambio di gruppo).

Per modificare il contenuto di questi campi fare tasto destro / Edit Running Total.

Notare che se questo campo viene utilizzato all'interno di un Group Footer occorrerà impostare Reset = On Change Group; se invece lo si utilizza in un Report Footer occorrerà impostare Reset = Never

FormulaFields consentono di impostare colonne calcolate automaticamente sulla base del valore di certi campi

Se si modifica un campo all'interno dell'XML Schema, occorre poi, su Crystal Report, andare sul FIELD EXPLORER, fare tasto destro / VERIFY DATABASE, in modo da forzare un Refresh dei campi, e poi trascinare il nuovo campo sul layout grafico sostituendolo al campo con il vecchio nome.

- 3) Aggiungere infine all'applicazione una nuova form vuota che fungerà da anteprima di stampa del Report. Sulla form trascinare un controllo **Crystal Report Viewer** che consente la visualizzazione del Report impostato. Assegnare a questo oggetto il nome **rptViewer**.

Nel form load di questa form aggiungere il codice seguente:

```
Report1 rpt = new Report1();    // nome del report creato al punto 2
rpt.SetDataSource(Form1.adodischi.daTable);
rptViewer.ReportSource = rpt
```

dove adoDischi è un oggetto adoNet istanziato sulla form principale.

Dall'anteprima è possibile:

- Fare ricerche (selezionando i gruppi nella apposita area di sinistra oppure utilizzando il comando Find)
- Spostarsi fra le pagine
- Lanciare la stampa / Esportare il file.

Record Singoli

Tutti i controlli posizionati sul Crystal Report sono Read Only e non è possibile aggiungere dinamicamente da codice nuovi controlli, per cui in pratica non si può visualizzare un record singolo come in VB6. L'unica possibilità è di filtrare un unico record sul BindingSource e poi legare il Crystal Report al BindingSource.List

```
rpt.SetDataSource(Form1.srdischi.List)
```

In alternativa si può creare manualmente un nuovo daTable in cui copiare l'unico record del Binding Source:

```
DataTable daTable = new DataTable();
daTable.Rows.Add();
for (int i = 0; i < form1.adodischi.daTable.Columns.Count; i++) {
    daTable.Columns.Add(form1.adodischi.daTable.Columns.Item(i).ColumnName);
    daTable.Rows[0][i] = Form1.srdischi.Item[0][i];
}
```

Classe per l'accesso ad un DB Access tramite ADO Net**public class AdoNet : IDisposable {**

```
private static string cnString;  
private OleDbConnection cn;  
private OleDbDataReader reader
```

```
public OleDbCommand cmd;  
public OleDbDataAdapter adp;
```

public static void impostaConnessione(string dbName){

```
if (File.Exists(dbName)) {  
    cnString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" + dbName; //2003  
    cnString = "Provider=Microsoft.Ace.OLEDB.12.0;Data Source=" + dbName; //2007  
}
```

public AdoNet(){

```
    init();  
}
```

public AdoNet (string dbName){

```
    impostaConnessione(dbName);  
    init();  
}
```

private void init() {

```
if (cnString != "") {  
    cn = new OleDbConnection();  
    cn.ConnectionString = cnString;  
    cmd = new OleDbCommand();  
    cmd.Connection = cn;  
    adp = new OleDbDataAdapter(cmd);  
    adp.MissingSchemaAction = MissingSchemaAction.AddWithKey;  
}  
else  
    throw new Exception("Connection String non inizializzata!");  
}
```

public void Dispose() {

```
    if (adp != null) adp.Dispose();  
    if (cmd != null) cmd.Dispose();  
    if (cn != null) cn.Dispose();  
}
```

public void apriConnessione(){

```
    if (cn != null && cn.ConnectionString != "" && cn.State == ConnectionState.Closed)  
        cn.Open();    }
```

public void chiudiConnessione(){

```
    if (cn != null && cn.ConnectionString != "" && cn.State == ConnectionState.Open)  
        cn.Close();    }
```

public DataTable eseguiQuery (string sql, CommandType tipo){

```

    DataTable daTable = new DataTable();
    cmd.CommandText = sql;
    cmd.CommandType = tipo;
    if(daTable.IsInitialized) daTable.Clear();
    // apriConnessione();
    adp.Fill(daTable);
    // chiudiConnessione();
    return daTable;

```

}

public int eseguiNonQuery (string sql , CommandType tipo) {

```

    int ris = -1;
    cmd.CommandText = sql;
    cmd.CommandType = tipo;
    apriConnessione();
    ris = cmd.ExecuteNonQuery();
    chiudiConnessione();
    return ris;

```

}

public string eseguiScalar (string sql , CommandType tipo) {

```

    string ris = string.Empty;
    cmd.CommandText = sql;
    cmd.CommandType = tipo;
    apriConnessione();
    Object obj = cmd.ExecuteScalar();
    if (obj != null)
        ris = obj.ToString();
    chiudiConnessione();
    return ris;

```

}

public OleDbDataReader creaLettore (string sql, CommandType tipo) {

```

    cmd.CommandText = sql;
    cmd.CommandType = tipo;
    apriConnessione();
    reader = cmd.ExecuteReader();
    return reader;

```

}

public void chiudiLettore() {

```

    chiudiConnessione();
    reader.Dispose();

```

}

```

while (reader.Read())
    msgBox(reader["nomeCampo"].
    ToString() );

```

Utilizzo

```

void Form_Load() { adoNet.impostaConnessione("database.mdb"); }

```

```

void btnInvia_Click( ) {
    adoNet ado=new adoNet();
    string sql = "select * from Piloti ";
    grid.DataSource = ado.eseguiQuery(sql, CommandType.Text);

```

Guida in linea di tipo HTML Help

Inizialmente le Guide in Linea per programmi windows erano pagine RTF scritte con Word, compilate dal Microsoft Windows Help Compiler (HCW) come file .HLP (caratterizzati dalla tipica icona del libro) e visualizzate mediante il programma **WinHelp.exe** installato insieme a Windows. I file .HLP, se lanciati da una finestra di Gestione Risorse, richiamano automaticamente WinHelp.exe. Con opportuni link possono però essere richiamati anche da una pagina VB.

Le Guide in Linea più recenti sono invece costituite da pagine HTML ipertestuali realizzabili mediante un qualsiasi web editor, compilate dal HTML HELP WORKSHOP (hhw.exe presente sui CD di DOT NET) che produce un file .CHM interpretato e visualizzato mediante il programma **hh.exe (Html Help.exe)** appartenente al sistema operativo da windows 2000 in avanti ed eventualmente installato insieme alle versioni più recenti di IE. Questi file possono essere lanciati direttamente da Gestione Risorse oppure richiamati da una pagina VB.

La guida deve essere inizialmente progettata su carta con argomenti e sottoargomenti, per ognuno dei quali occorre produrre una corrispondente pagina .html (per ognuna non tralasciare il tag Title). Esempio:

Presentazione del programma [Default.htm]

Struttura della tabella Dischi (codice, cantante, titolo,) [structDischi.htm]

Struttura della tabella Generi [structGeneri.htm]

I menù del programma [mnuElencoMenù.htm]

Inserimento di un nuovo disco [mnuInserisci.htm]

Eliminazione di un disco [mnuElimina.htm]

Ricerca di un disco [mnuRicerca.htm]

All'interno delle pagine html si possono anche inserire dei link di collegamento tra i diversi documenti. Ad esempio all'interno della pagina Inserisci.htm, alle voci Codice si può associare un link alla pagina structDischi.htm. E' possibile anche inserire link a siti Internet esterni.

In fase di progettazione è bene costruire un elenco di PAROLE CHIAVE che si vogliono gestire nella guida (es Disco, Come Leggere la Guida), riportando manualmente per ognuna l'elenco delle pag che trattano l'argomento.

HTML HELP WORKSHOP hhw.exe [disponibile gratuitamente sul sito Microsoft la release 4.74]

Occorre innanzitutto creare un progetto mediante FILE / NEW / **PROJECT**. Parte un wizard che presenta le seguenti finestre:

- 1) Richiesta di eventuale conversione di una guida esistente di tipo WinHelp in guida Html (non checkare)
- 2) Nome del progetto (miaGuida) a cui verrà automaticamente aggiunta l'estensione **.hhp**.
Occorre necessariamente specificare il percorso assoluto della cartella in cui memorizzare il progetto, percorso impostabile mediante l'apposito pulsante Browse (scrivere soltanto il nome del file)
- 3) Ricerca di tutte le pagine html create (Tabella dei contenuti e Indice verranno creati dopo)

Comandi del menù File:

NEW - Crea un nuovo Progetto, oppure una nuova Tabella dei Contenuti, oppure un nuovo Indice, oppure semplicemente una nuova Pagina html (html help workshop dispone di un mini editor html integrato).

OPEN - Apre un progetto esistente

COMPILE HTML FILE - Compila il progetto .hhp corrente in un file .chm

DECOMPILE HTML FILE - Consente di decompilare un file .chm ricreando tutti i file sorgenti

CHM INFORMATION - Apre la guida in linea relativamente al progetto in fase di sviluppo.

Una volta creato il progetto si rendono disponibili alcuni pulsanti che consentono rispettivamente di:

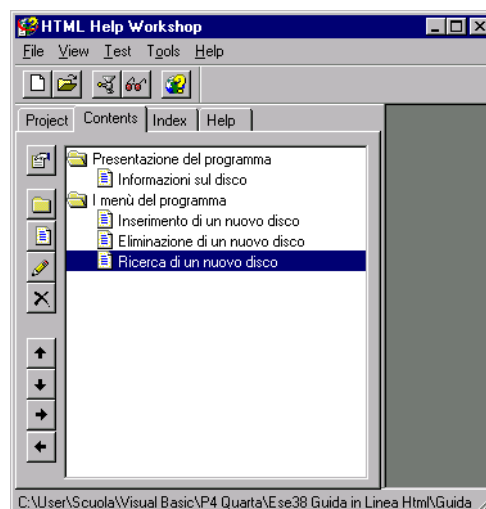
- 1) Impostare le opzioni del progetto, fra cui:
 - il **Title** che comparirà nella barra del titolo della Guida,
 - il **Default File** cioè il nome del file che dovrà essere visualizzato all'apertura (default.htm / index.htm)
 - la **DefaultWindow**, che consente di associare al progetto una eventuale barra di navigazione superiore
 - il **CompiledFile** cioè il nome del file finale .chm che verrà creato (nome del progetto con estensione .chm)
 - il **Log File**, in cui verranno salvati i vari messaggi di compilazione (utile per memorizzare msg di errore)

- 2) Aggiungere / Rimuovere files HTML al progetto [Topic Files]
- 3) Aggiungere / Modificare delle **Windows Definitions**, cioè impostazioni da associare ad una eventuale barra di navigazione superiore, che dovrà essere associata al progetto come **Default Window**
- 3) Visualizzare / Modificare il codice html della pagina selezionata
- 4) Salvare il progetto
- 5) Salvare il progetto e lanciare la compilazione

Creazione della tabella dei contenuti hhc (html help contents)

Una volta definito il progetto occorre fare click sulla scheda **Contents** per impostare manualmente come dovranno essere organizzati i Contenuti della Guida. I Contents sono in pratica gli argomenti che compariranno nel Sommario della Guida. I Contents dovranno essere salvati su un file .hhc (ad esempio Sommario.hhc).

- Facendo click sul comando “**Insert a Heading**” (quello a forma di cartella) viene aperto un editor in cui si può inserire il primo degli argomenti principali in cui era stato suddiviso il progetto iniziale.
- Nella caselle **Entry Title** inserire il titolo da assegnare alla prima voce del sommario (ad esempio *Presentazione del programma*). Il pulsante **Add** consente di associare una pagina html alla voce attuale (es il file default.htm). I file html vengono presentati sulla base del tag Title. Ad ogni voce possono essere associate più file che verranno automaticamente mostrati come elenco. Chiudere la finestra
- Dopo aver impostato l’heading, fare click sul comando “**Insert a Page**”(terzo pulsantino) e inserire, il primo sottoargomento. Inserire quindi i sottoargomenti successivi.
- Inserire quindi un nuovo argomento principale (**Insert Heading**) e poi i suoi sottoargomenti (**Insert Page**).
- Il comando “**Edit Selection**” (la matita) consente di modificare la voce attualmente selezionata
- Il comando “**Delete**” cancella la voce selezionata
- Le frecce in basso a sinistra consentono di spostare la varie voci fra di loro (prima / dopo, etc.)



Creazione dell’indice delle parole chiave hhk (html help keys)

E’ sufficiente fare click sulla scheda **Index** ed assegnare il nome al file (ad esempio Indice.hhk). Le keywords sono le parole chiave predefinite che compariranno nell’Indice della Guida. Facendo click sulla chiave gialla (“Insert a keyword”) si apre una finestra praticamente identica alla precedente in cui occorre inserire, una alla volta, le varie keywords associando a ciascuna uno o più files html. Anche sugli Indici si può impostare una gerarchia come per il Sommario, agendo sempre sulle frecce.

Compilazione del progetto

Per compilare il progetto aperto lanciare il comando **Compile** che, in assenza di errori, produce il file compilato “Guida.chm” applicando l’estensione .CHM al nome del progetto.

Opzioni

Prima di lanciare la compilazione, dalla finestra **OPTION** del progetto, (prima icona), si possono impostare alcune opzioni di compilazione, tipo “**Compiler / Full-Text search Information**” che consentirà a run time la ricerca libera di una stringa sull’intero testo della guida. Con questa opzione, all’interno dell’help in linea comparirà anche la scheda **Cerca** in cui l’utente potrà inserire una stringa da ricercare. La guida mostrerà come risultato tutte le pagine che contengono la stringa ricercata, evidenziando in ciascuna le corrispondenze trovate. Le altre opzioni di **Compiler** sono da provare.

Il contenuto di tutti i files HTML utilizzati per creare la Guida viene caricato dentro il file .CHM. I files sorgente HTML possono dunque essere archiviati a parte.