Introduzione al *Visual Basic per applicazioni:* il caso Excel

di Gianni Giaccaglini

Significato del Visual Basic per applicazioni (VBA)

Fin dalla versione 5.0 di Excel la Microsoft ha introdotto il ***Visual Basic for Application*** – in sigla, **VBA** - inaugurando una strategia unificante, all'insegna del linguaggio di programmazione ***Visual Basic***. Il Visual Basic peculiare di Excel ha soppiantato il precedente sistema di macrofunzioni (rimaste tuttora in vita, per compatibilità) e, successivamente è stato esteso agli altri membri della famiglia **MS Office** (Word, Access ecc.).

Ma che cos'è un Visual Basic *per applicazioni*? Esiste una certa confusione al riguardo, persino fra i programmatori di professione, per cui vanno subito fatte due puntualizzioni:

a) non si tratta di un'estensione del linguaggio "padre", il Visual Basic standard, bensì di un'evoluzione del sistema di macro tradizionalmente associata ai fogli elettronici;

b) di conseguenza, pur aderendo da vicino alla sintassi Visual Basic stadard, una routine VBA si esegue soprattutto, anche se non esclusivamente, nell'ambiente del package Excel (dalla versione 5.0 in avanti);

c) grazie ad un avanzato "protocollo" Microsoft, l'**OLE Automation**, si danno però interessanti possibilità di cooperazione, in una logica *Client/Server* fra i vari membri del mondo Visual Basic.

Le opzioni avanzate offerte dall'OLE Automation implicano la possibilità di richiamare funzionalità di un pacchetto da parte di un programma scritto sia in Visual Basic standard sia in uno qualunque degli altri "dialetti" VBA. Per fare due esempi concreti, con OLE automation è possibile:

- in un programma Visual Basic 5.0 o 6.0 sfruttare la libreria di funzioni finanziarie di Excel sia direttamente che tramite fogli elettronici (creati anche estemporaneamente);
- da una macro Excel gestire documenti Word (o presentazioni PowerPoint).

In queste note introduttive al vasto mondo VBA non arriveremo a parlare di tali prospettive avanzate, tuttavia l'averne accennato è importante, se non altro per mettere in evidenza l'importanza del VBA:

Il Visual Basic per applicazioni è lo strumento principe per la creazione di veri e propri applicativi di OFFICE AUTOMATION, sulla base della più diffusa famiglia di pacchetti specifici, Microsoft Office.

Il discorso interessa in misura crescente i programmatori professionisti, mentre le difficoltà concettuali del VBA tagliano fuori molti fra gli utenti, salvo una élite di esperti.

NOTA - Queste dispense si occupano, per brevità, del VBA di Excel, se non il più importante, sicuramente il più ricco dialetto VBA.

Modelli Excel e ruolo delle macro

Una conoscenza non superficiale del foglio elettronico in generale e di Excel in particolare è un prerequisito per seguire questa trattazione, tuttavia non nuoce a nessuno la lettura di questo paragrafo, volto alla definizione concreta del significato di ***modello Excel***, inteso come una particolare applicazione sviluppata direttamente dall'utilizzatore di spreadsheet, fin dai tempi del progenitore VisiCalc, facendo a meno - per forza di cose, a quei tempi - di macro o altri mezzi strettamente informatici. Si trattò di una programmazione "senza accorgersene" grazie anche a funzioni speciali come quelle di ricerca tabellare ed a funzioni decisorie come la =SE().

Per far capire la cosa nel modo meno pedante possibile il ruolo - distinto ma *cooperante* – svolto dalle formule e dalle macro VBA facciamo un esempio semplice ma significativo. Si abbia una cartella di lavoro Excel il cui unico foglio di lavoro contiene diverse formule che danno adito a risultati relative alla "Analisi ABC" di uno stocki di magazzino. Con la manovra **Alt+F11** passiamo all'ambiente Editor Visual Basic di Excel, del tutto simile a quello del VB5 o VB6 (Visual Basic 5.0 o 6.0) e creaiamo un modulo Visual Basic *Modulo1*.

NOTA - I moduli speciali ***Foglio1 Foglio2*** ecc. e ***ThisWorksheet*** non verranno svolti in queste note introduttive, per brevità, ma anche perché esse hanno come scopo la trattazione degli aspetti più peculiari del VBA di Excel: soprattutto l'oggetto **Range** e relativi proprietà/ metodi.

Il modulo Visual Basic serve a contenere le procedure che automatizzano parzialmente il modello. Il foglio di lavoro in questione corrisponde a uno schema tabellare che riporta, a partire dalla colonna B, le voci seguenti:

PRODOTTO (nomi degli articoli di un magazzino)

GIACENZA (valore della)

VALORE%

VALORECOM%

CLASSE (A, B o C)

In altri termini, la situazione corrisponde al layout seguente:

	B	C	D	E	F
. . .	**……**				
8	**Analisi ABC di prodotti in magazzino**				
9	**PRODOTTO**	**GIACENZA**	**VALORE%**	**VALORECUM%**	**CLASSE**
10	viti	2.000	1,77%	1,77%	A
11	bulloni 900	0,80%	2,57%	A	
. . .	**……**				
100	dadi 1.000	0,89%	100%	C	
101	**Totale** 112.850				

Esaminiamo le formule principali inserite nelle varie celle. Tanto per cominciare in fondo alla serie di valori di giacenze di magazzino in colonna C, C10:C100, si abbia una funzione del tipo

=SOMMA(C10:C100)

Inserita nella cella C101 che supporremo battezzata *ValorInv*, la funzione detta totalizza le varie giacenze di magazzino (valore d'inventario). A tale sommatoria fanno riferimento, in colonna D, formule di percentuali del tipo:

D10=C10/ValorInv

D11=C11/ValorInv

D12= C12/ValorInv

. . . . . .

D100=C100/ValorInv

In estrema sintesi, la problematica dell'analisi ABC riferita a un magazzino consiste nell'assegnare alla classe A, alla classe B o C le merci i cui valori hanno incidenza alta, media o scarsa rispetto al totale. Le formule scritte nella cella E10 e sottostanti, valutano l'incidenza percentuale cumulativa:

E10=D10

E11=E10+ D11

E12= E11+ D12

. . . . . .

E100= E99+D100

Infine opportune funzioni condizionale (=SE(...) nidificate) inserite nella cella F10 e sottostanti provvedono alla classificazione ABC secondo criteri che qui non interessano. Ma non basta. Per motivi ben noti a chi s'intende di Analisi ABC e su cui sorvoliamo, affinché l'intero marchingegno dia il responso corretto è indispensabile che i prodotti in questione siano disposti in ordine decrescente per quanto riguarda l'incidenza percentuale (sempre la colonna VALORE%). Altrimenti il modello dà i numeri.

Nella sua estrema semplicità l'esempio è dunque interessante, per almeno due motivi:

a) si può pensare che la parte *statica* dell'applicativo sia stata creata da un normale utilizzatore, digiuno di informatica e VBA;

b) in casi come quello in esame l'esigenza di aggiungere routine per l'ulteriore automatizzazione dell'applicativo scaturisce non solo da esigenze di interfaccia, bensì anche, se non ancor più, dalla natura *dinamica* di modelli del genere.

Spieghiamoci meglio. Il foglio elettronico viene tuttora ritenuto adatto quasi solo per analisi di bilancio ed altre pur utili ma tristi e grigie applicazioni contabili. Nelle quali le voci in gioco e, di conseguenza, il layout risulta rigido. Nulla di scandaloso in sé, ma muovendo da tali casi più semplici e comuni anche gli esperti di spreadsheet hanno finito per confinare l'impiego delle macro nella creazione di menu personalizzati (con o senza corredo di finestre di dialogo) destinati a compiti di assoluta routine come la stampa o la visualizzazione di un grafico. Va da sé che simili cose, ad una fredda analisi talora non si rivelano neanche troppo pratiche, visto che chi usa un package evoluto come Excel se la cava benissimo da sé, coi comandi che l'ambiente gli mette a disposizione.

NOTA – Non ci si fraintenda: un'interfaccia gradevole è utile anche in un modello Excel, ma va da sé che i programmatori in Visual Basic se la cavano benissimo con i controlli offerti dal VBA di Excel, gli stessi (anche se in numero inferiore). Pertanto, in queste note non staremo a ripetere cose già note a chi mastica il VB standard.

Excel consente non solo di migliorare l'interfaccia ma offre anche la possibilità di rendere chiuso al massimo grado un modello (fra l'altro occultando formule del foglio e rendendo naccessibile il codice VBA), comunque l'automazione su cui focalizzeremo l'attenzione è di altra natura.

Tornando al precedente modellino, il suo carattere dinamico deriva da due fattori (riscontrabili anche altrove): la necessità di copiare formule (quelle contenute nelle celle delle colonne D, E ed F) ogni volta che si aggiunge una voce (il cui numero può non essere noto a priori) e la già citata esigenza di ordinare le voci. Ora entrambe queste operazioni possono essere compiute manualmente però anche l'utente più esperto perde tempo e rischia di imbrogliarsi. Ed ecco allora che anche un menu costituito magari da pochi pulsanti incollati sul foglio di lavoro, a ciascuno dei quali è associata una macro viene incontro alla bisogna: AGGIUNGI_VOCE e RIORDINA, ad esempio.

Le macro Visual Basic

Che cosa sono le macro? Come anche i programmatori dovrebbero sapere, le macro, altrove dette anche script, originariamente rispecchiavano le azioni dell'utente per lo più registrate automaticamente per essere poi rilanciate riproducendo così una sequenza ripetitiva di comandi (non a caso venivano chiamate macro "di tastiera"). L'evoluzione verso un vero e proprio linguaggio di alto livello come il VBA ha visto come tappe intermedie l'inserimento di istruzioni di controllo del flusso e l'abbandono, per lo più, del rispecchiamento delle manovre dell'utente. Si esamini ad esempio il seguente listato:

```
Sub Macro1()
  Range("B3:E6").Select
  Selection.Copy
  Range("D9").Select
  ActiveSheet.Paste
  Application.CutCopyMode = False
End Sub

Sub Macro2()
  With Selection.Font
    .Name = "Arial"
    .FontStyle = "Grassetto"
    .Size = 10
    .Strikethrough = False
    .Superscript = False
    .Subscript = False
    .OutlineFont = False
    .Shadow = False
    .Underline = xlUnderlineStyleNone
    .ColorIndex = xlAutomatic
  End With
End Sub
```

Macro1 e Macro2 sono due routine ottenute mediante il registratore di macro VBA. Il significato del codice è ragionevolmente leggibile, basta tener presente che la sintassi object oriented fa sì che l'oggetto *precede* la proprietà o il metodo. Di conseguenza *Range("B3:E6").Select* equivale a "seleziona l'intervallo (alias zona) B3:E6". Come è facile comprendere, la prima macro riproduce pedissequamente le azioni seguenti: 1) selezione dell'intervallo appena detto; 2) comando **Modifica Copia**; 3) selezione della cella D9 di destinazione; 4) comando **Modifica Incolla**; 5) chiusura delle operazioni Copia-taglia-incolla (tramite Esc o Invio).

La seconda macro è stata invece ottenuta, sempre per registrazione, applicando il grassetto ai caratteri della selezione corrente mediante il comando **Formato Celle...** (o la sua scorciatoia **Ctrl+1**). Stavolta il registratore si sforza di rivelarsi "intelligente", provvedendo a inserire le proprietà definite nella finestra di dialogo in questione entro il contrutto *With... End With*, che risparmia di ripetere *Selection.Font* per tutti gli attributi. Semmai l'eccesso di zelo del registratore produce ridondanze relative ad attributi default che non abbiamo inteso toccare, per cui ci converrà eliminarle come mostrato nella seguente *Macro3* (o ancora più drasticamente):

```
Sub Macro3()
'Semplificazione della Macro2
  With Selection.Character
    .Name = "Arial"
    .FontStyle = "Grassetto"
    .Size = 10
  End With
End Sub
```

L'importante è verificare che sia che si lanci la *Macro2* sia che si lanci la *Macro3* non si assiste minimamente alla comparsa delle finestre di dialogo dell'azione manuale.

NOTA - Una volta per tutte: I commenti, nei listati VB, sono preceduti da un apice (').

Sperimentando, forse più che studiando manuali cartacei o l'help in linea (straripante e completo ma poco sistematico) si scopre poi la possibilità di dare istruzioni "***a distanza***". Con tale espressione intendiamo riferirci ad una delle più importanti differenze concettuali fra le precedenti macro Excel 4.0 e il VBA: con le prime occorre sistematicamente (salvo poche eccezioni) selezionare o rendere attivo l'oggetto da trattare, sia esso un intervallo, un foglio di lavoro, un grafico o un controllo, mentre il Visual Basic per applicazioni permette di puntare a qualsiasi oggetto (presente sulla scrivania) e modificarne una proprietà o scatenarne un metodo senza dover necessariamente attivare il foglio di appartenenza o selezionare alcunché.

Unica condizione è che l'oggetto venga individuato in modo univoco. Ed ecco un caso tipico:

```
Sheets("Mio Foglio").Range("B5").Font.Bold = True
```

Chi ha familiarità con la sintassi con cui in Visual Basic si individuano gli oggetti dovrebbe comprendere come l'istruzione esemplificata punti al foglio di lavoro denominato "Mio Foglio" e, su questo, all'intervallo formato dalla sola cella B5, il cui carattere viene infine impostato in grassetto. Questo, si ripete, può avvenire per così dire nell'ombra, anche se attualmente è attivo un diverso foglio, poniamo il Foglio2.

Ancora più interessante è il caso della ***copia a distanza***, che qui ci preme anticipare:

```
Worksheets(2).Range("A1:C10").Copy Worksheets(1).Range("A1")
```

Come si vedrà meglio a suo tempo, la precedente istruzione copia l'intervallo A1:A10 del Foglio2 nella cella A1 del Foglio1, qualunque sia la cella e il foglio attivi correnti e, quel che è ancora più bello, senza passare per le fasi di copia & incolla (come avviene, fatalmente, nell'uso manuale).

Gli oggetti primari dello spreadsheet: un'anteprima

Ma è giunto il momento di puntualizzare il discorso focale di queste note. Il linguaggio VBA è orientato agli oggetti, ergo la sintassi relativa si esprime con una delle due forme seguenti:

oggetto.proprietà

oggetto.metodo

Ora una cartella di lavoro è infarcita di oggetti di varia natura, come gli oggetti grafici o i controlli (finestre di dialogo, pulsanti ecc.) incollati su un foglio. Ma non è questa la peculiarità di un foglio elettronico i cui mattoni costitutivi sono, partendo dal basso: le celle, gli intervalli, i fogli, le cartelle di lavoro. Analogamente, nel VBA di Word, si trovano caratteri, parole, paragrafi ecc. oppure campi, record, tabelle e database (Access VBA).

La piramide o albero genealogico di tali oggetti è facilmente reperibile nella Guida VBA. Al culmine si ha l'oggetto massimo, denominato *Application*, che costituisce, diciamo così, la madre di tutti gli Oggetti di Excel. Si tratta di Excel stesso e altrove Application corrisponderà a Word, Access e quant'altro. I figli, nipoti e pronipoti di tale capostipite sono:

- l'oggetto ***Workbook*** che è la cartella di lavoro (una di quelle attualmente aperte)
- l'oggetto ***Worksheet***, foglio di lavoro
- L'oggetto ***Range***, intervallo di calle.

Lo schema verrà ripreso e approfondito in seguito, per ora diamo alcune anticipazioni sintattiche:

1) ciascun oggetto fa parte di una famiglia o classe (di suoi simili) e l'accesso al singolo membro di ciascuna classe si effettua attraverso metodi, diciamo così, "pluralistici", ai quali corrispondono insiemi (collection) di oggetti omogenei: *Workbooks*, *Worksheet, Range* (un'apparente eccezione, vedremo perché), *Cells*;

2) esattamente come accade in altri lidi informatici (incluso il rude DOS) i membri più elevati di una catena gerarchica si possono omettere, nel qual caso è sottinteso il soggetto attualmente attivo.

3) l'istruzione ***Set*** serve a fissare un oggetto in una variabile.

Spieghiamoci un po' meglio. Sul primo punto, ci si accontenti di esempi:

Workbooks("BILAN.XLS") e *Worksheets("Foglio1")* individuano rispettivamente la cartella e il foglio di lavoro virgolettati entro parentesi, mentre *Workbooks(2)* e *Worksheets(3)* si riferiscono al secondo, rispettivamente al terzo fiore dei rispettivi... mazzi.

Quanto al secondo punto, la terrificante espressione genealogica (completa) seguente:

```
Application.Workbooks(1).Worksheets(2).Range("A1:B12")
```

può essere ridotta semplicemente a *Range("A1:B12")* se è attivo il secondo foglio della prima cartella. E il più delle volte si lavora con una data cartella, inoltre *Application* si può quasi sempre omettere, visto che si sta lavorando con Excel (ma vi sono casi particolari in cui, ciononostante, non se ne può fare a meno).

Esemplifichiamo infine la comoda istruzione *Set*:

```
Set MioInterv = Worksheets(2).Range("A1:B12")
```

Grazie ad essa, si potrà in qualunque momento successivo riferirsi a tale intervallo, poniamo il caso per grassettarne il contenuto, con:

```
MioInterv.Font.FontStyle = "grassetto"
```

NOTA - E c'è un altro grosso vantaggio: l'istruzione agisce anche se il caro *MioInterv* non è attualmente attivo!

Concludiamo questi antipasti offrendo nel listato che segue una macro VBA da sperimentare.

```
Sub AggiungiFormule()
  Set I = Range("inivoci")
  Set F = Range("iniform")
  NrVoci = Range(I, I.End(xlDown)).Count
  Range(F, F.Offset(NrVoci - 1, 2)).Select
  Selection.FillDown  'Ricopia in basso
EnD Sub
```

Del metodo *FillDown*, peraltro di significato intuitivo, si parlerà meglio in seguito, qui diciamo solo che la routine *AggiungiFormule* provvede con una tecnica un po' particolare alla copia dinamica delle formule del modellino seguente ogni volta che l'utente aggiunge una o più voci in fondo alle colonne B e C:

	B	C	D	E	F
. . .	**……**				
9	**Incremento vendite 1° trimestre 1999**				
10	**Prodotti**	**Dic. 98**	**Genn. 99**	**Febbr. 99**	**Mar. 99**
11	Cappelli	1.200	1.260	1.323	1.389
12	Berretti 900	945	992	1.042	
. . .	**……**				
20	Baschi 2.500	2.625	2.756	2.894	
21	Papaline	1.850	1.943	2.040	2.142

Le formule, facili da intuire, in colonna D e seguenti calcolano un incremento dello 0,5% rispetto al mese precedente.

Per ora basti sapere che i nomi "inivoci" (ossia "inizio voci") e "iniform" ("inizio formule") vanno assegnati alle celle, rispettivamente, B11 e D11 (non B10 ed E10, perché la prima riga contiene formule di inizializzazione).

Riepilogando...

Fin qui abbiamo delineato i principi generali, cercando di focalizzare la prospettiva aperta dal Visual Basic applicato al foglio elettronico, con il suo ruolo di avanguardia dei "fratelli" dialetti VBA relativi a Word, Access o, magari, PowerPoint.

Questa dispensa si propone di fornire note introduttive ma cruciali per la piena comprensione dei principali mattoni costitutivi di un foglio elettronico. Abbiamo già intravisto che cosa sono gli oggetti e come questi comprendano *proprietà* e *metodi*. Probabilmente anche un utente di Excel, e non solo un programmatore VB5 o VB6, considera già come oggetti (non certo a torto) gli oggetti grafici, e, ancor più spesso, le caselle di testo, i pulsanti, le finestre di dialogo e i controlli di varia natura e complessità. Ma l'obiettivo, qui, è quello di focalizzare gli oggetti che formano l'ossatura, il nucleo di uno spreadsheet.

NOTA - Una macro VBA può essere agevolmente "attaccata" a qualsiasi oggetto (disegno, grafico, immagine) incorporato sul foglio di lavoro. Le operazioni necessarie sono semplici: 1) selezionare l'oggetto; 2) clic destro, poi attivare la voce **Assegna macro...**; 3) scegliere la routine VBA da scatenare, supponiamo "MiaMacro". Da questo momento basta un clic sull'oggetto per lanciare *MiaMacro*.

Al centro dello spreadsheet, in definitiva, vi sono intervalli e celle con il loro contenuto di dati da elaborare o formule di legame. Il loro inquadramento nel mondo Visual Basic, fondamentale di per sé, aiuterà a capire meglio tutto il resto, ossia oggetti di contorno, aggiuntivi, speciali e di qualsivoglia natura.

Osiamo poi asserire che persino esperti di Visual Basic standard, che tendenziualmente identificano gli oggetti con i tipici controlli dell'interattività potrebbero trarre giovamenti concettuali dallo studio di Excel VBA, per approfondire la comprensione concettuale del mondo OOP.

Libreria degli oggetti tipici di Excel

Riprendiamo in esame la tassonomia relativa alla libreria di oggetti tipici di Excel. In testa a tale piramide si trova l'oggetto *Application*, nella fattispecie Excel. Limitandoci agli oggetti tipici abbiamo una struttura gerarchica o a scatole cinesi o, se si preferisce, una genealogica - sotto al capostipite - così suddivisa:

- ***WorkBooks*** (Cartelle di lavoro), genitrici e contenitrici di
- ***Sheets*** (fogli) di diversi tipi, come ***Charts*** (fogli per grafici) e, principalmente ***WorkSheets*** (Fogli di lavoro), questi ultimi comprendenti, a loro volta,
- insiemi ***Range*** (Intervalli), ovvero raggruppamenti di ***Cells*** (celle).

Si faccia attenzione. Nell'elenco appena fatto si è fatto direttamente ricorso alla proprietà VBA che danno accesso ai diversi oggetti, rispettandone la sintassi (che prevede maiuscole intermedie, peraltro facoltative). Si tratta di una sintassi di tipo, "pluralistico" (con l'eccezione - apparente, come si vedrà - della proprietà *Range*) che si riscontra, in VBA, in tutti i casi in cui si punta ad *insiemi* di oggetti, detti anche gruppi o *collection*. Altri esempi di collection Excel: ***Shapes*** (Oggetti grafici come quelli creati con la barra strumenti Disegno) e ***OLEObjects*** (oggetti OLE incorporati)..

Un particolare elemento di un dato insieme viene individuato tramite *indice*. Questo può essere, a scelta, il numero d'ordine o il nome fra virgolette. Così le proprietà *Workbooks(2)* e *Workbooks("MIACART.XLS")* puntano entrambe alla stessa MIACART.XLS sempreché questa sia la seconda fra quelle che al momento sono aperte sulla

scrivania di Excel. Dovrebbero fin d'ora essere chiari i pro e contro dei due sistemi d'indicizzazione: il primo è prezioso per spazzolare un insieme con cicli quali *For... Next*.

Con oggetti gerarchicamente strutturati - In perfetta analogia con il path di un file, in DOS - un elemento di basso livello può essere individuato in modo completo, indicandone l'intero percorso genealogico. L'esempio che segue, nella sua prolissità, lo considero autoeloquente:

```
Application.Workbooks("MIA.XLS") _
.Worksheets("Foglio1").Range("A2:B5")
```

Si noti che, come sa chi già mastica il Visual Basic, il separatore dei vari componenti è il punto (.), carattere perciò vietatissimo negli identificatori adottabili dal programmatore (contrariamente a quanto accade con funzioni e macrofunzioni Excel).

Proprio come nel DOS, se si omettono papà o nonni sono sottintesi quelli attualmente attivi. Ragion per cui, restando nel caso precedente, se è aperta la cartella MIA.XLS, sono consentite più stringate notazioni:

```
Worksheets("Foglio1").Range("A2:B5")
```

se poi è attivo il foglio Foglio1 di una data cartella possiamo scrivere solamente:

```
Range("A2:B5")
```

Per l'esattezza va anzi ribadito che le due ultime notazioni puntano, la prima, a qualunque intervallo A2:B5 del Foglio1 di qualsiasi cartella attuale o a qualunque Intervallo A2:B5 di qualsiasi cartella o foglio di lavoro attuali (d'altronde proprio come nei due banali esempi DOS seguenti: Dir MIADIR\PIPPO.MIO e Dir PIPPO.MIO).

Quanto ad *Application*, è quasi sempre facoltativo (visto che ci troviamo in Excel) salvo in casi speciali di omonimia col Visual Basic standard, che in parte avremo modo di vedere.

Osservazione sintattica sottile ma pratica. Con le proprietà "insiemistiche" nei primi tempi è facile scordarsi del plurale, scrivendo *Workbook("TALE:XLS")* anziché *Workbooks ("TALE:XLS")* oppure *Worksheet("Bilancio")* invece di *Worksheets ("Bilancio")*.

NOTA - Nella terminologia VBA i vocaboli (keyword, parole chiavi) *Workbook*, *Worksheet* ecc. esistono, però di fatto sono usati solo per definire o identificare il *tipo* di oggetto. Li si userà, di fatto, in una *Dim* (e da nessun'altra parte), esempio:
Dim MioFoglio As Worksheet
Quando si desidera una certa concreta cartella o un certo concreto foglio occorre invocare la proprietà giusta e questa, nel caso di collection, ha sintassi plurale.

Tutto chiaro? Di sicuro sì per chi già conosce il Visual Basic, comunque gli esempi e l'esercizio rafforzeranno questo basilare concetto della programmazione a oggetti.

Le Windows (finestre), oggetti secondari ma inevitabili

Prima di procedere è opportuno, ancorché tedioso, parlare di un oggetto particolare per semplicità non incluso nella tassonomia accennata nel paragrafo precedente. Si tratta dell'oggetto ***Window*** che, per così dire, costituisce un terzo incomodo rispetto alle cartelle e ai fogli. Ci conviene liquidarlo subito, così non se ne parla più e, al tempo stesso, ci servirà come oggetto propedeutico ad altri, più interessanti oggetti.

Per individuere una finestra occorre la proprietà "insiemistica" *Windows* appartenente sia all'oggetto Application che all'oggetto Wordkook. Il motivo di questa duplice appartenenza probabilmente è l'ereditarietà, ma più prosaicamente deriva dal fatto che una cartella può comprendere più finestre, a seguito del comando ***Nuova Finestra*** del menu **Finestra**. Per fissare le idee supponiamo che siano aperte le due cartelle GRAF.XLS e BILAN.XLS. Ebbene con successive applicazioni del comando appena detto possiamo pervenire a una situazione di sei finestre, di cui due appartenenti alla cartella GRAF.XLS e quattro a BILAN.XLS.

La proprietà *Windows* permette di impostare sia una particolare finestra che tutte le finestre, nel loro insieme, mediante l'una o l'altra delle forme sintattiche seguenti:

oggetto.*Windows(indice)*

oggetto.*Windows*

ove *indice* può essere o un numero d'ordine (1, 2, 3,...) oppure l'indicazione - da porre fra virgolette - che si legge nella cornice di una data finestra: la cosiddetta *Caption* (didascalia).

Con riferimento alla prima forma sintattica si ha il significato seguente:

Windows(1), *Windows(2)*, *Windows(3)* ecc. sottintendono l'anteposizione di "Application." e impostano ad una ad una tutte le finestre presenti sulla scrivania, ossia "puntano" a ciascuna delle sei finestre appena nominate. In alternativa, ad esempio Windows("BILAN.XLS") o Windows("GRAF.XLS:2") fanno riferimento, rispettivamente, alla finestra contenente la sola cartella BILAN.XLS o alla seconda delle due finestre di GRAF.XLS.

La seconda sintassi, *Windows*, si riferisce invece a tutte le finestre o, parlando in generale, a tutti gli elementi di un insieme: così sarà per *Wordkooks*, *Wordkooks* e simili.

Naturalmente non basta fissare un oggetto, quasi sempre occorre gestirlo opportunamente. E per usare un oggetto (o insieme di oggetti) occorre di norma applicargli una delle proprietà o metodi ad esso associato, secondo la sintassi già più volte citata: *Oggetto.proprietà* e *Oggetto.metodo,* con possibile corredo di argomenti.

Proprietà, metodi e puntualizzazioni varie

Ora ciascun oggetto, ai vari livelli, possiede a volte una caterva di proprietà e metodi, che il manuale cartaceo e gli stessi libri dedicati non possono che riportare parzialmente. Dall'interno di un Modulo VBA vi si può accedere sia tramite l'help in linea che tramite un semplice browser, il Visualizzatore degli Oggetti, attivabile col tasto **F2**, sulle cui modalità d'uso si rinvia alle esercitazioni pratiche. Occorre solo evidenziare la necessità di distinguere proprietà / metodi che hanno senso a) per una singola finestra; b) per l'intero insieme: nel caso a, nel browser (o nella Guida) occorre riferirsi alla voce "Windows", nel caso b alla voce "Windows" e lo stesso dicasi con le coppie Wordkook / Wordkooks, Wordkook / Wordkooks e via dicendo. Per essere ancora più chiari: proprietà e metodi di una singola finestra o cartella - individuata da *Windows("Tale")* o *Wordkooks(3)* - si attingono riferendosi all'*oggetto* (singolare!) *Window o Wordkook*.

Per l'oggetto singolare Window ci limitiamo a citare la proprietà *Caption* (didascalia), che restituisce l'etichetta che figura nella cornice di ciascuna finestra. Nel secondo caso il numero di proprietà e metodi è più specifico e ristretto (ad esempio, non ha certo senso la didascalia di tutte le finestre! ogni finestra ha la propria, per contro *Count* opera solo con Windows, non con una finestra singola). In particolare citiamo:

Count, proprietà che restituisce il numero di finestre dell'insieme;

Arrange, metodo che sistema le finestre sullo schermo secondo varie disposizioni (per default affiancandole, altrimenti *xlVertical, xlCascade* ecc. v. Guida).

ActiveWindow, proprietà che restituisce la finestra corrente, come oggetto.

Nell'ultimo come in altri casi, la differenza fra metodi e proprietà appare sottile, l'importante è comprendere pragmaticamente il ruolo di questa o di quello. Ad ogni buon conto non guasta precisare che:

- un metodo esegue una certa azione;
- una proprietà restituisce un valore ma, a volte, anche un oggetto (come nel caso di *ActiveWindow* o, analogamente, *ActiveWordkook*, *ActiveSheet*)

A volte poi l'assegnazione di una proprietà si confonde con un'azione. Esempio:

```
ActiveWindow.Height = 200
```

in cui l'assegnazione ha un preciso effetto: l'altezza della finestra cambia (e, naturalmente, si vede).

Chiudiamo queste tediose (ma inevitabili) note sintattiche richiamando un principio ignoto ai neofiti del mondo OOP: un metodo o una proprietà si aggancia, tramite il connettore "**.**", ad un trenino genealogico come un ulteriore vagoncino, magari generando a sua volta un oggetto oppure restituendo o modificando il valore di una proprietà (in senso stretto). Vediamo, al riguardo, un esempio concreto:

```
Wordkooks(2).ActiveSheet.Range("B2").Value = 1234
```

Seguiamo i passaggi esecutivi. La proprietà *Workbooks(2)* imposta la seconda cartella (poniamo MIACART.XLS) quindi la proprietà *ActiveSheet* restituisce il foglio attivo di MIACART.XLS (potrebbe essere il Foglio3), poi la palla passa alla proprietà *Range("B2")* che restituisce un ben preciso oggetto Range e, infine alla proprietà ***Value*** di tale oggetto Range. L'istruzione culmina con l'assegnazione del valore 1234 ad una cella ben precisa.

Ma adesso commentiamo le procedure del listato seguente, miranti fra l'altro a marcare certi distinguo. In particolare la seconda delle due routine in questione evidenzia l'esatta appartenenza delle varie finestre: all'oggetto *Application* oppure a singole cartelle di lavoro.

```
Sub EsploraFinestreInBlocco()
 'Esamina, indiscriminatamente, TUTTE le
 'finestre aperte sulla scrivania Excel

  For Each finest In Application.Windows
    MsgBox finest.Caption & " " & finest.Parent.Name
  Next
End Sub

Sub EsploraFinestre()
 'Esamina le finestre, definendone in dettaglio
 'la cartella di appartenenza
  Constant Messiniz = "La cartella n. "
  Dim i As Integer, j As Integer, Mess As Stringa
```

```
  Dim NumCart As Integer, NumFines As Integer
  NumFines = Windows.Count
  NumCart = Wordkooks.Count
  MsgBox "Finestre Totali: " & NumFines
  MsgBox "Cartelle di lavoro Totali: " & NumCart
  For i = 1 To NumCart
    With Wordkooks(i)
      Mess = Messiniz & i
      NumFines = .Windows.Count
      If NumFines = 1 Then
        MsgBox Mess & " ha una sola finestra"
      Else
        MsgBox Mess & " ha le seguenti " _
        & NumFines & " finestre..."
        For j = 1 To NumFines
          MsgBox .Windows(j).Caption
        Next
      End If
    End With
  Next
End Sub
```

Il ciclo *For Each... Next* e il costrutto *With... End With*

Dando per nota la funzione *MsgBox* dal corso VB5 parliamo del ciclo seguente:

```
For Each <NomeElem>  In <gruppo>
  istruzioni
Next
```

Dovrebbe essere già noto a chi ha pratica di VB5 o VB6, comunque ha una particolare importanza in Excel VBA (ove sono presenti molti tipi di oggetti). Si applica a qualunque gruppo di oggetti, come pure a vettori e matrici, restituendone ad uno ad uno gli elementi nella variabile *NomeElem*, senza obbligo di gestire un indice.

La prima semplice routinetta del listato precedente sfrutta tale comodo costrutto e le proprietà ***Caption*** e ***Parent*** per elencarci a video tutte, indistintamente, le finestre presenti sulla scrivania Excel nonché il padre, in senso oggettual-genealogico, di ciascuna. Si provino a questo punto, assieme o separatamente, le istruzione seguenti:

```
Babbo = ActiveWorkbook.Parent
MsgBox "Il papà della Cartella attiva è " & Babbo
Avo = ActiveSheet.Parent.Parent.Name
MsgBox "Il nonno del foglio attivo è " & Avo
```

(l'operatore **&** di concatenamento stringhe dovrebbe essere noto sia agli utenti Excel che a quelli di Visual Basic.)

Scopriamo così che il capostipite di tutti gli Oggetti Excel VBA si chiama "Microsoft Excel", come ci attendevamo. Però tornando alla routine *EsploraFinestreInBlocco*, c'è da scommettere che alcuni saranno sorpresi nell'apprendere che non tutte le finestre hanno come padri la cartella di appartenenza e non Excel.

Passiamo alla seconda routine del listato precedente, sorvolando sulla dichiarazione iniziale di variabili, che sovente si ometteranno (il tipo implicito o default è l'onnicomprensivo *Variant*). Per comprendere la routine *EsploraFinestre* occorre introdurre, per chi non la ricordi o la sfrutti poco, un'importante struttura del Visual Basic:

With <oggetto>... End With

grazie alla quale, una volta precisato <oggetto>, non occorre ripeterlo e se ne possono invocare proprietà e metodi - entro i "terminatori" *With* ed *End With* - preceduti dal carattere "." (punto). Esempio:

```
With Wordkooks(2).Range("A2:B10")
  .Valuee = 33
  .Font.Name = "Arial"
  .Font.FontStyle = "Corsivo"
  ... (omissis) ...
End With
```

Si evitano così tediose ripetizioni. Inoltre ai più esigenti non guasta sapere che questa sintassi rende anche più veloce il codice.

A questo punto l'analisi dettagliata della routine *EsploraFinestre*, lasciata come esercizio, svela che, dopo aver conteggiato mediante le rispettive proprietà *Count* il numero di cartelle e finestre presenti, la routine spazzola le finestre ad una ad una, segnalandone tutte le didascalie ma precisando altresì quante (oltre che quali) sono quelle che appartengono alle varie cartelle di lavoro.

Si consideri poi il semplice listato seguente e si notino le varianti rispetto alla precedente routine *EsploraFinestreInBlocco*.

```
Sub MostraTutteFinestre
'Rende via via attive le finestre
'attualmente presenti sulla scrivania
  For i = 1 To Windows.Count
    With ActiveWindow
      MsgBox .Caption
      .ActivateNext
    End With
  Next
End Sub
```

La più importante differenza sta nell'uso del metodo ***ActivateNext*** proprio dell'oggetto Window: equivale a **Ctrl+F6**, che nell'uso manuale rende attiva la finestra seguente. Analogamente, a **Maiusc+Ctrl+F6** corrisponde il metodo ***ActivatePrevious***. Il risultato, facile da comprendere e provare, è che stavolta alla segnalazione delle didascalie si accompagna la concreta attivazione (ed esibizione) delle finestre.

Anche da quest'ultimo esempio si può comprendere la differenza fra quella che abbiamo chiamato un'azione "a distanza", senza che nulla si muova, e il tradizionale modo di procedere delle vecchie macro.

NOTA - Non sempre lo stile seleziona-e-agisci è da scartare. Si danno infatti circostanze in cui è opportuno o "scenografico" far vedere all'utente quel che accade o, quantomeno, il risultato conclusivo.

Azioni sulle celle, un'anteprima

Il carattere straripante della materia ha fatto sì che si parlasse quasi solo di finestre. Abbiamo perso tempo? Non del tutto, perché molte proprietà di questo oggetto piuttosto secondario se non frivolo sono valide per altri. In seguito si potranno così liquidare altri oggetti insiemistici più rapidamente e giungere finalmente a quelli centrali nel mondo Excel: **intervalli** e **celle**.

Per lenire l'ansia dei più zelanti proponiamo il codice che segue:

```
For Each Sconto in Range("Sconti")
  Sconto.Value = Sconto.Value * 0,95
Next
```

L' esempio mostra come il ciclo *For Each... Next* si possa tranquillamente applicare a un intervallo (cosa di cui molti manuali non parlano. L'effetto, come è chiaro o perlomeno intuibile, è la riduzione del 5% di tutti gli sconti contenuti nell'intervallo denominato appunto "Sconti".

Infine il listato che chiude questo paragrafo oltre a riproporre la routine di copia dinamica intravista all'inizio ne aggiunge un'altra quasi altrettanto inesplicata (ma non inesplicabile) ed affidata momentaneamente all'apprendimento autonomo:

```
Sub AggiungiFormule()
  Set I = Range("inivoci")
  Set F = Range("iniform")
  NrVoci = Range(I, I.End(xlDown)).Count
  Range(F, F.Offset(NrVoci - 1, 2)).Select
  Selection.FillDown
End Sub

Sub CopiaRel()
  'Copia dinamica in basso di una riga di formule,
  'già battezzata "Rigaform", mediante la tecnica
  '(ingegnosa ma rudimentale) del nome temporaneo
  Application.Goto Reference:="Rigaform"
  ActiveCell.Offset(0, -1).Select
  Selection.End(xlDown).Select
  ActiveCell.Offset(0, 3).Select
  ActiveWorkbook.Names.Add Name:="ultimacella", _
        RefersToR1C1:=ActiveCell
  Range("Rigaform").Select
  Selection.Copy
  Range("Rigaform:ultimacella").Select
  ActiveSheet.Paste
  Application.CutCopyMode = False
  ActiveWordkook.Names("ultimacella").Delete
End Sub
```

Diciamo solo che la routine *CopiaRel* va applicata a un foglio in cui sia stato battezzato come "Rigaform" un intervallo di formule orizzontale (diciamo, ma solo per fissare le idee, E7:G7), sempre in modo dinamico ossia adeguandosi al numero (variabile) di voci nella prima colonna D di etichette.

Cartelle, fogli di lavoro, celle e intervalli

Continuando a dare per intuitiva la proprietà Valore [inglese *Value*], tipica di un oggetto Range, riproponiamo la routine vista poc'anzi:

```
For Each Sconto in Range("Sconti")
  Sconto.Value = Sconto.Value * 0,95
Next
```

Essa dimostra che *Range("Sconti")* altro non è che l'insieme delle celle che lo compongono. Forse pochi ci penserebbero, ma da qui deriva la possibilità di applicare anche a un oggetto Range la proprietà *Count* e i volgari indici numerici:

```
Set ZonaMia = Range("A1:C10")
NumCelle = ZonaMia.Count
  For i = 1 To NumCelle
    If ZonaMia(i).Font.Bold = True Then
       ZonaMia(i).Font.Italic = True
       'Aggiunge il corsivo solo alle celle grassettate
    End If
  Next
```

Si è già accennato all'istruzione ***Set***. Si contrappone a *Let*, storica istruzione BASIC facoltativa che nessuno adopera (tutti scrivono sempre *x* = ... anziché *Let x* =...). *Set*, che invece è obbligatorio, fissa un oggetto in una variabile di tipo Object (oggetto generico) o, quantomeno, di tipo Variant. Da quel momento tale variabile diventa, per così dire, un alter ego dell'oggetto. L'istruzione *Set* può essere, banalmente, usata per non dover ripetere la denominazione di oggetti, specie se lunga e verbosa. Così nel caso appena visto (in cui, si fa notare, non si poteva ricorrere a *With... End With*). Ma dal momento che una variabile "impostata" a un oggetto ne racchiude tutti i riferimenti identificativi, si intuiscono usi più raffinati che vedremo presto.

Per liberare memoria sarebbe opportuno, una volta che tale variabile ha assolto il suo scopo, il codice seguente:

```
Set ZonaMia = Nothing
```

NOTA - Ma sovente se ne può fare a meno; soprattutto istruzioni del genere, a parere di chi scrive, sono superflue se poste al termine di un programma che ritorna all'ambiente Excel, visto che in tal caso la liberazione di memoria avviene automaticamente.

Il secondo esempio che stiamo per dare introduce la proprietà ***Rows*** propria, come la sua parente stretta, la proprietà ***Columns***, di un oggetto Range come pure dell'oggetto Application (che può omettersi). Così *Application.Columns(3)* o, semplicemente *Columns(3)* ci dà l'intera terza colonna, ossia la C (del foglio corrente). Applicando *Rows* e *Columns* a un Range si ha l'insieme delle righe o colonne dell'intervallo, talché *Rows.Count* ce ne dà il numero mentre *Rows(3)* ci dà le celle della terza riga. Idem, mutatis mutandis, per *Columns*.

Ma ecco l'esempietto:

```
Sub Asterix()
  Set Z = Range("A1:J10")
  Nr = Z.Rows.Count
   For i = 1 To Nr
     Z(i, i).Value = "*"
     Z(i, i).HorizontalAlignement = xlFill
   Next
End Sub
```

La routine Asterix immette asterischi nelle celle diagonali della zona (quadrata) A1:J10 formattandole in modalità Riempi (per cui si ha il pieno di asterischi, indipendentemente dalla larghezza della colonna).

NOTA - A proposito delle proprietà *Rows* e *Columns*: i più svegli e audaci sostituiscano *Z(i, i)* con *Z.Rows(i).Columns(i)*. Vedranno che i due codici si equivalgono e... capiranno perché.

L'oggetto Application: cenni a proprietà e metodi

Dopo queste anticipazioni, facciamo un passo indietro per liquidare i principali metodi e proprietà della madre di tutti gli oggetti, Application. Possiamo farlo andando a spulciare la Guida.

NOTA - Si rammenta a chi non l'avesse già scoperto da sé, che basta premere **F1** dopo aver posto il punto di inserzione, in un modulo VBA, sul termine "Application". Stessa manovra per qualsiasi altro termine (es. "Range").

Nella finestra di aiuto troviamo una marea di metodi e una altrettanto nutrita caterva di proprietà. Impossibile trattarli tutti ma non possiamo neanche rinviarne l'esame sine die. Infatti, accanto a proprietà scontate come *Workbooks*, *Worksheets* e *Sheets* o a proprietà come *ActiveCell* (d'intuibile semantica) con cui "Application." si può omettere, troviamo altre proprietà o metodi peculiari, con i quali "Application." è d'obbligo: per motivi vari su cui non ci si dilunga ma che in genere mirano a evitare ambiguità.

Per tagliare la testa al toro proponiamo la seguente routinetta ibrida, con l'invito all'istruttiva sperimentazione (in proprio) cui essa si presta:

```
Sub ApplicationProprieta()
  With Application
    VecchiaDida = .Caption
   .Caption = "Excel mio bello"
    MsgBox "Osserva in alto..."
    .Caption = VecchiaDida
    .WindowState = xlNormale
    'fissa lo stato normale della finestra di Excel
    .Height = 200
  End With
End Sub
```

Si scopre così, tra l'altro, che anche Application gode come Windows della proprietà *Caption*, sia in lettura che in scrittura, sicché possiamo modificare a piacere il default "Microsoft Excel".

Attenzione però a mantenere separati il documento ed Excel: infatti se la dida è, poniamo, "Microsoft Excel - Cartel1" ai successivi lanci di ApplicationProprieta assisteremo a buffi ripristini: "Microsoft Excel - Cartel1 - Cartel1", poi "Microsoft Excel - Cartel1 - Cartel1 - Cartel1"!!!

La successiva istruzione dello stesso listato, ossia:

```
.WindowState = xlNormal
```

traduce in gergo VBA il comando **Ripristina** del menu di controllo di Excel. In sua assenza, se la finestra di Excel è a pieno schermo, il debugger dà errore di run-time e protesta con "Impossibile impostare la proprietà Height per la classe Application": infatti anche nell'uso manuale il comando **Ridimensiona** del menu di controllo è inoperante. Se adesso sveliamo che *WindowState* è una proprietà che pertiene altresì all'oggetto Window i più svegli capiscono al volo anche il rimedio ai comici ripristini detti poc'anzi:

```
ActiveWindow.WindowState = xlNormal
```

(da porre all'inizio, prima di registrare la dida di Excel in VecchiaDida).

I neofiti qui si chiederanno che cosa sia mai quel tal *xlNormal*. È un esempio della miriade di *costanti incorporate*, rappresentate in modo mnemonico anziché con il loro valore (in altri termini, nella fattispecie, *xlNormal* equivale per l'interprete VBA al valore –4143, davvero ostico da tenere a mente!).

Variabili incorporate ne abbiamo già introdotte, in modo surrettizio e affidandosi all'intuizione dell'allievo, nelle routine precedenti (es. *xlFillDown*, *xlAutomatic* ecc.). Se ne danno di due tipi:

a) precedute da "**vb**" - es. ***VbYes*** e ***vbNo*** – e in tal caso appartengono al Visual Basic standard

b) precedute da "**xl**", prefisso che le distingue come peculiari di Excel VBA.

Altre proprietà di *Application* corrispondono al fissaggio di opzioni relative all'intero ambiente. Ecco due esempi interessanti, che corrispondono ad attributi che si possono impostare nella finestra di dialogo **Opzioni**, scheda ***Modifica***:

```
Application.FixedDecimal = True
'imposta il formato numerico con DECIMALI FISSI
Application.FixedDecimalPlaces = 4
'imposta 4 decimali (dopo la virgola)
```

Accenniamo poi a un'istruzione abbastanza utile, di cui si parlerà in seguito: ***Application.Goto***. Qui l'anteposizione di "Application." è obbligatoria, per non confondere con ***GoTo***, istruzione di salto incondizionato. *Application.Goto* traduce il comando Excel **Modifica Vai a...**, alias tasto **F5**.

Parlimo ora di ***StatusBar***, proprietà di lettura/scrittura utile per depositare un messaggio, per l'appunto, nella barra di stato. Anche qui l'oggetto Application è d'obbligo, inoltre occorre che la barra di stato sia visibile: provvede alla bisogna la proprietà *VisuallzzaBarraDiStato*, che può essere impostata ai valori logici *True* e *False*. Ed ecco un esempio generico d'impiego:

```
With Application
  BarraStatoVisib = .DisplayStatusBar
  'registra True o False
  .DisplayStatusBar = True
  'Forza la visibilità della barra di stato
  .StatusBar = "Attendere, prego..."


  '... (omissis) ...

  .StatusBar = False
   .DisplayStatusBar = BarraStatoVisib
```

```
  'Ripristina la vecchia situazione
End With
```

Altre note su *Sheets*, *Workbooks* e *Windows*

A questo punto, prima di passare ai sospirati intervalli (oggetti Range) diciamo le cose essenziali sugli Sheets e Worksheets, aggiungendo alcuni distinguo, a mo' di flashback, sulle Workbooks (Cartelle di lavoro) e sulle Windows (Finestre). Il numero di metodi e proprietà specifici è talmente vasto da esigere uno sforzo di trattazione sistematica, fuori luogo su una dispensa introduttiva.

Per le proprietà che riguardano fogli ci limitiamo a dire che alla tipologia più usuale espressa dall'insieme *Worksheets* se ne affiancano altri come ***Charts*** (Grafici o, meglio, fogli per Grafici), inoltre ***Sheets*** punta a fogli di qualsiasi tipo. Il seguente listato comprende tre semplici procedure esplorative.

```
Sub EsploraTuttiFogli()
'Esamina indiscriminatamente i fogli d'ogni tipo
'della cartella corrente, indicandone via via il nome
  For Each fog In ActiveWorkbook.Sheets
    MsgBox fog.Name
  Next
End Sub


Sub MostraTuttiFogliLav()
'Rende via via attivi i fogli di lavoro
'della cartella corrente (qui sottintesa).
  For i = 1 To Worksheets.Count
    With Worksheets(i)
      .Activate
      MsgBox .Name
    End With
  Next
End Sub


Sub AttivaTuttiFogli()
  indAtt = ActiveSheet.Index
  nflav = Sheets.Count
  For i = indAtt To nflav
    With Sheets(i)
      .Activate
      MsgBox .Name
      End With
  Next
  For i = 1 To indAtt - 1
    With Sheets(i)
      .Activate
      MsgBox .Name
      End With
  Next
  Sheets(indAtt).Activate
End Sub
```

La prima delle tre precedenti routine, *EsploraTuttiFogli*, esamina indiscriminatamente tutti i fogli della cartella di lavoro attuale segnalandone il nome. La seconda routine, *MostraTuttiFogliLav()*, è simile alla *MostraTutteFinestre* vista nel paragrafo precedente, volta ad esibire all'utente le varie finestre presenti. Dal confronto emerge subito una differenza: non si ha più *ActivateNext*, che è un metodo esclusivo dell'oggetto Window. In sua vece si è fatto ricorso al metodo ***Activate***, di chiara semantica.

NOTA - Il metodo ***Activate*** appartiene alla maggioranza degli oggetti, incluso Window! Provere per credere una variante di *MostraTutteFinestre* che usa *Attivate* anziché *ActivateNext* (esperimento lasciato per esercizio).

Piuttosto i più accorti si saranno avveduti (o, che è lo stesso, i più avveduti si saranno accorti) che si perde, con *MostraTuttiFogliLav*, la ciclicità per cui si partiva da e tornava all'elemento corrente. Il rimedio c'è e si chiama ***Index***, proprietà che ci dà il numero d'ordine di un dato elemento di un insieme. Tipicamente:

```
ActiveSheet.Index
ActiveWindow.Index
```

e simili.

La routine *AttivaTuttiFogli* (v. sempre listato precedente) soddisfa i pignoli che hanno a cuore quella tale ciclicità. Di passaggio si fa poi notare che ***ActiveSheet***, proprietà degli oggetti Application, Window e Workbook, è unica per tutti i tipi di foglio.

NOTA - In altri termini: non esiste nessun *ActiveWorksheet*, come a volte capita di scrivere anche ai meno distratti.

Ci sarebbe molto ancora da dire per soddisfare una trattazione anche solo sommaria, ma si riteene che lo si possa rinviare al momento in cui i nodi verranno al pettine.

Concludiamo questo paragrafo elencando metodi e proprietà, in parte già visti, comuni a tutti gli insiemi (salvo eccezioni).

- ***Application***, proprietà addirittura di tutti gli oggetti (da non confondere con l'*oggetto* omonimo), in Excel VBA restituisce "Microsoft Excel" (in Word VB e Access VBA, si ha "Microsoft Word" e, rispettivamente, "Microsoft Access"). L'istruzione *MsgBox Application* ne offre la scontata verifica.
- ***Add***, metodo generatore di un nuovo elemento: ad esempio *Worksheets.Add*, equivale al comando **Inserisci Foglio di lavoro** nell'uso manuale, mentre a **File Nuovo** corrisponde *Workbooks.Add*.
- ***Item(indice)***, proprietà che punta a un dato elemento dell'insieme. Esempio: *Sheetsi.Item (3)* equivale al più semplice *Sheets(3)*. Se ne fa quasi sempre a meno, salvo che con gli argomenti di funzioni (v. Guida).
- ***Count***, ***Parent*** e ***Index***, parole chiave già incontrate.

Selection*, *ActiveCell*, *CurrentRegion

Per il trattamento di intervalli Excel VBA mette a disposizione una gran copia di mezzi. Centrale è la proprietà *Range*, che esamineremo a fondo, precisando fin d'ora che appartiene, oltre che a Worksheet anche ad Application, nonché a Range stesso. Di conseguenza è possibile definire in modo esaustivo una data zona anche mediante riferimenti *completi*. Ad esempio, sono del tutto leciti i codici seguenti:

```
x = Range("Foglio1!B12").Value
Range("Foglio1!A1:B10").Copy
Range("[SPESE.XLS]Foglio3!B2:C7").Copy
```

di cui gli ultimi due copiano negli Appunti l'intervallo specificato e *senza* che lo si debba preselezionare (come con le macro 4.0). L'ultimo caso può addirittura riferirsi ad una carlella esterna!

Peraltro è più elegante e funzionale la sintassi "a vagoncini", che esprime le prime due istruzioni precedenti in quest'altro modo:

```
x = Sheets("Foglio1").Range("B12").Value
Sheets("Foglio1").Range("A1:B10").Copy
```

Sarà a questa sintassi strutturalistica, OOP, che ci atterremo scrupolosamente.

Basilari, con zone e celle sono le proprietà ***Range*** e ***Cells*** e relativi, articolati intrecci. Prima di trattarli vogliamo parlare delle proprietà ***Selection*** e ***ActiveCell***, che richiedono opportuni distinguo. Si applicano entrambe agli oggetti Application e Window, ma mentre la prima si riferisce alla generica selezione corrente, che, si badi bene, potrebbe essere un oggetto o insieme di oggetti di *altra* natura (es. caselle di testo, frecce, fumetti, magari oggetti OLE) la seconda punta sempre ad un oggetto-cella.

NOTA - Per la precisione, *ActiveCell* restituisce la cella attiva della finestra attiva, quella attualmente in primo piano.

È nota, ma non guasta rammentarla, un'altra distinzione tra cella attiva e selezione. La prima è data da una singola, ben precisa, cella posta entro l'insieme di celle selezionate, mentre Selection può essere un intervallo di più celle, di cui una soltanto è la cella attiva (quella evidenziata in chiaro sullo sfondo in nero della selezione).

Le espressioni seguenti sono equivalenti e restituiscono tutte la cella attiva della *finestra* corrente:

```
ActiveCell
Application.ActiveCell
ActiveWindow.ActiveCell
Application.ActiveWindow.ActiveCell
```

NOTA - Chi ha in odio la prolissità può limitarsi a scrivere *ActiveCell* e buonanotte.

Piuttosto si faccia attenzione! I più spavaldi potrebbero pensare di poter puntare una cella attiva "a distanza", ad esempio con:

```
Worksheets(1).ActiveCell.Value = 123
```

Purtroppo se è attivo un foglio qualsiasi, diverso da Foglio1 il compilatore rigetta la (ingenua) pretesa, con un messaggio illuminante: "Worksheet non ha la proprietà ActiveCell". Questo dato di fatto (tanto sottile quanto sgradevole) non va dimenticato, pena continue frustrazioni.

NOTA - Detto di passaggio, l'oggetto Window, che pensavamo di aver buttato fuori dalla porta, ci è ritornato... dalla finestra (c'era da aspettarselo, con Microsoft Windows!).

Per procedere con qualche prova relativa a *Selection* e *ActiveCell* conviene senz'altro cominciare a trattare almeno le principali fra le innumerevoli proprietà dell'oggetto Range.

- ***Font***. Proprietà di lettura/scrittura che restituisce l'*oggetto* omonimo e, a sua volta, individua numerosi attributi del formato carattere, quali ***Color***, ***Bold*** (Grassetto), ***Italic*** Corsivo (i due ultimi possono essere *True* o *False*). Ed ecco due piccoli esempi:

```
ActiveCell.Font.Bold = True
```

```
Sub TogliCorsivo()
  For Each C In Selection
    With C.Font
      If .Bold And .Italic Then .Italic = False
    End With
  Next
End Sub
```

Il significato della prima istruzione dovrebbe essere ovvio, mentre la seconda routine toglie il corsive nelle sole celle con dati in neretto più corsivo.

- ***Value***. Proprietà di lettura/scrittura, che restituisce il valore presente nella cella, anche quando tale valore è il risultato di una formula o funzione Excel. Il termine Value si può omettere. In altri termini *ActiveCell*, da sola, viene equiparata dal compilatore ad ***ActiveCell.Value***. Esempio:

```
ActiveCell.Value = ActiveCell * 2
'Raddoppia il valore della cella attiva
```

NOTA - Anche con *Selection* l'omissione di ".Value" è lecita. Ad esempio *Selection* = *123* inserisce tutti valori 123 nelle celle dell'intervallo attualmente selezionato. Nella precedente versione di Excel VBA queste omissioni erano ammesse solo in lettura, con *ActiveCell* e *Selection*.

- ***CurrentRegion*** (Zona corrente). Interessantissima proprietà che punta all'intervallo costituito da tutte le celle, anche vuote, circondate da righe e colonne vuote (o bordi estremi del foglio). Gli utenti esperti di Excel conoscono la manovra canonica: **Modifica** / **Vai a...** seguito da ***Speciale***, poi clic sul pulsante di opzione **Zona corrente**, manovra che, a partire dalla cella attiva, seleziona in un sol colpo celle contigue (si ricordo che la comoda scorciatoia di tastiera è **Ctrl+***).

Orbene si tenga ben presente che CurrentRegion è una proprietà esclusiva dell'oggetto *Range*. È un fatto, se si riflette, di assoluto rigore logico: infatti è solo "attorno" ad una data cella (o intervallo) che ha senso individuare la cosiddetta zona corrente. Quindi non è affatto "cervellotico", come potrebbe apparire ai distratti, il fatto che *ActiveCell* non si applichi a un oggetto Worksheet, ma anzi è fonte di soddisfazioni. Infatti con *CurrentRegion* non soltanto non è obbligatorio selezionare alcunché, ma si può tranquillamente puntare a un intervallo remoto, a patto di non dimenticarsi di precisare sia il foglio sia un'opportuna cella "germinale".

Ed ecco un esempio semplice ma abbastanza significativo:

```
With Sheets(2).Cells(2, 2).CurrentRegion.Value = "ciao"
```

Anticipando che *Cells(2, 2)* equivale a *Range("B2")*, l'istruzione appena proposta riempie di amichevoli saluti tutte le celle della zona corrente del foglio di lavoro 2, di cui fa parte B2 (e questo, insistiamo fino alla noia, anche se al momento è attivo un foglio diverso e senza che occorra selezionare alcunché).

È anche il primo caso che incontriamo di *assegnazione simultanea* di dati ad un'intera zona specificata. In altri termini, istruzioni del tipo *Range("Tale").Value = ...* traduce molto semplicemente l'operazione che nell'uso manuale si effettua in tre mosse: 1) selezione dell'intervallo interessato; 2) digitazione del dato "comunitario"; e, 3) conclusione dell'immissione con **Ctrl+Invio** anziché con il semplice Invio.

Questa tecnica, palesemente, ha un raro interesse pratico con le costanti. Ma se il dato è una formula la faccenda cambia aspetto. Di questo parleremo più avanti, intanto a tutti si suggerisce un piccolo esperimento:

sostituire *.Value = "ciao"* con *.Formula = "=INT(CASUALE()*90 + 1)"* e vedere l'effetto risultante.

Due semplici esempi

Dopo tanta teoria certamente molti gradiscono qualcosa di più concreto, per cui interrompiamo la trattazione sistematica (ma pedante), con qualche anticipazione, propone e commentando due esempi semplici ma passabilmente istruttivi.

Il primo, che trova spazio nel prossimo listato, pone rimedio a un problema in cui prima o poi ci si imbatte: sovente una zona "corrente" comprende righe e/o colonne di contorno che vanno escluse dall'inserimento di dati. Ma ecco, intanto, il listato:

```
Sub PreparaCartelleTombola()
  For Each f in Worksheets
    Set Zc = f.Range("B2").CurrentRegion
    Nr = Zc.Rows.Count : Nc = Zc.Columns.Count
    Set Zc = Zc.Offset(1, 1).Resize(Nr – 1, Nc - 1)
    For Each c in Zc
      c.Value = Int(Rnd*90 + 1)
    Next
  Next
End Sub
```

Il caso più tipico del problema accennato sopra è quello di etichette di testa e laterali: Esaminiamo allora il campo su cui opera la routine *PreparaCartelleTombola*. Si tratta di una serie di fogli (in pratica, tutti quelli della cartella di lavoro: non ci si scandalizzi, è un esercizio...) ciascuno dei quali ha l'identico layout riportato qui sotto:

	A	B	C	D	E	F	G	H	I
1	**Cartella della Tombola di Mario**								
2		**RUOTA**	**N1**	**N2**	**N3**	**N4**	**N5**		
3		BARI	22	4	60	17	47		
4		NAPOLI	80	35	11	56	39		
...	**……**								
11		ROMA	9	24	43	69	23		
12									

La routine *PreparaCartelleTombola* ha come (frivolo) obiettivo l'inserimento di numeri casuali da 1 a 90 in tutti i fogli della cartella (di lavoro!), organizzati allo stesso modo, senza distruggere le intestazioni.

NOTA - Per generare numeri della Tombola (o, se si preferisce, del Lotto) si utilizzano le funzioni standard VB *Rnd* e *Int* che restituiscono, rispettivamente, un numero casuale da 0 (incluso) a 1 (escluso) e la parte intera del numero. Va da sé che *rnd * 90* ci dà un causale fra 0 e 89,999...9, di qui l'aggiunta di 1 prima dell'intervento della funzione *Int.*

Per il nostro scopo tornano assai comode le proprietà ***Resize*** e ***Offset*** (scarto), entrambe proprie dell'oggetto Range e dotate di argomenti. Gli argomenti di *Offset* esprimono di quante righe e colonne va spostato l'intervallo, mentre quelli di *Resize* lo ridimensionano, ridefininendone il numero di righe e colonne.

Si tengano sempre presenti il listato e la figura precedenti. Una volta "impostata" in *Zc* (ciascuna) zona corrente "attorno" a B2 (B2:G11 nella figura) e dopo aver calcolato in *Nr* e *Nc* il numero di righe e colonne, l'istruzione cruciale è la seguente (riportata nuovamente, per comodità):

```
Set Zc = Zc.Offset(1, 1).Resize(Nr – 1, Nc - 1)
```

Se concepiamo *Offset(1, 1)* e *Resize(Nr – 1, Nc - 1)* come due stadi di una lavorazione in cascata cui è sottoposto Zc, è facile capire che dal primo stadio fuoriesce l'intervallo C3:H12 e dal secondo stadio l'intervallo ridimensionato C3:G11.

Il gioco dei due cicli *For Each... Next* è lasciato come esercizio.

Il secondo esempio corrisponde al listato seguente:

```
Sub CopiaRel()
  'Copia dinamica in basso di una riga di formule,
  'già battezzata "Rigaform", mediante la tecnica
  '(ingegnosa ma rudimentale) del nome temporaneo

  Application.Goto Reference:="Rigaform"
  ActiveCell.Offset(0, -1).Select
  Selection.End(xlDown).Select
  ActiveCell.Offset(0, 3).Select
  ActiveWorkbook.Names.Add Name:="ultimacella", _ _
    ReferstoAR1C1:=ActiveCell
  Range("Rigaform").Select
  Selection.Copy
  Range("Rigaform:ultimacella").Select
  ActiveSheet.Paste
  Application.CutCopyMode = False
  ActiveWorkbook.Names("ultimacella").Delete
End Sub
```

La routine del listato precedente è già stata proposta senza commenti ai più preparati ed è giunta l'ora di spiegarle alla maggioranza, limitandosi a commenti stringati, istruzione per istruzione (e tutti quanti possono chiarirsi meglio le idee con la Guida in linea).

È facilmente immaginabile il campo delle operazioni. Queste consistono nella copia in basso di una riga di formule per tante righe quante sono le voci laterali, il cui numero non è noto a priori (in altri termini, si ha un layout del tipo già visto nel modello di analisi ABC).

La prima istruzione della macro equivale alla pressione di **F5** seguita dal nome Rigaform preassegnato alla riga di formule. È interessante sapere che, a patto che non vi siano sinonimi di zona assegnati in più fogli di una cartella l'individuazione e la conseguente *Selection* di quella tal zona è assicurata, qualunque sia il foglio di partenza. E lo stesso vale se si danno le coordinate dell'intervallo in gioco.

Seconda istruzione. Il metodo ***Select*** applicato ad una cella scartata di 0 righe e -1 colonna rispetto a quella attiva (E7) porta il cursore di cella in D7.

Terza istruzione. La proprietà ***End*** con argomento *xlDown* punta all'ultima cella non vuota di colonna D, ossia D11, che poi viene selezionata. Coi parametri predefiniti *xlDown*, *xlToRight* ecc. la proprietà *End* corrisponde alla manovra **Ctrl+*freccia*** o **End** seguito da freccia direzionale dell'uso manuale. Così la cella raggiunta varia col numero di voci contigue in colonna D. I parametri che corrispondono ai vari tasti freccia sono: *xlToLeft, xlToRight,*

La quarta istruzione seleziona una cella scartata di tre colonne a destra (G11, in figura). La quinta si affretta a denominare "ultimacella" tale cella (si noti come la proprietà ***Names*** si applichi alla *cartella*, non al foglio attivo). Va evidenziato poi che l'adozione di *ActiveCell* come valore del parametro *ReferstoAR1C1* assicura il carattere dinamico (oserei dire "adattativo") del battesimo.

Sesta e settima istruzione. Viene selezionata di nuovo la riga di formule e copiata: il metodo ***Copy*** applicato a un Range equivale, ovviamente, al comando manuale **Modifica / Copia**.

Ottava istruzione. Secondo una sintassi supportata da Excel, "Rigaform:ultimacella" individua l'intervallo compreso fra quelli così battezzati. E nel caso di figura 4 si tratta di E7:G11, ovviamente (per i neofiti di Excel: copiare E7:G7 in E7:G11 è perfettamente equivalente a copiare E7:G7 in E8:G11, d'accordo?).

Su quel che segue ci basti sottolineare che il metodo ***Paste***, equivalente a **Modifica / Incolla**, si applica a un oggette Worksheet (occorre ricordarsene!), dopodiché dovrebbe esser chiaro a tutti quel che accade: le sospirate formule vengono incollate dove volevamo, ulteriori (possibili) copie sono annullate (e le caselle ruotanti attorno alla zona origine spariscono).

Per capirci di più, oltre che il ricorso alla Guida in linea si consigliano due vie complementari: il macro registratore (ripercorrendo i passi qui descritti a guisa di commento) e il debugging passo passo.

Terminiamo con un commento che è anche un invito ai più bravi e volonterosi. Il programmino visto segue alquanto pedestremente le operazioni manuali (rispecchiate dal macro registratore) ed è un continuo seleziona-qua / seleziona-là. Queste azioni si possono scorciare e potare molto.

Completamento delle proprietà *Cells* e *Range*

Le proprietà ***Cells*** e ***Range*** si applicano a un oggetto che può essere Application, Worksheet e Range, con la differenza che il primo individua di norma una *singola* cella. Si danno due possibili sintassi con indici:

Cells(indiceRiga, indiceColonna) e

Cells(indiceRiga)

IndiceRiga e *IndiceColonna* costituiscono il numero della riga e il numero della colonna della cella così referenziata, o anche la lettera, fra virgolette, come in *Cells(3, "D")*, che equivale a *Cells(3, 4)*. Come tutti comprendono, *Cells(3, 4)* individua sul foglio attivo la cella D3.

Con la sintassi 2 l'intervallo o il foglio viene visto come se fosse a una sola dimensione e le celle sono numerate da sinistra verso destra e dall'alto in basso.

Excel VBA offre una terza sintassi, tipica degli insiemi, senza indici. Questa ha senso pratico (raramente, come si intuisce) solo con un foglio di lavoro, di cui restituisce l'insieme di tutte le celle. Liquidiamo questa perla con tre piccoli esempi.

```
With ActiveSheet.Cells
  MsgBox("Celle del foglio: " &.Count)
End With


With Cells.Font
  .Name = "Arial"
  .Size = 8
End With
'Fissa il font Arial, corpo 8 su tutto il foglio
```

```
Cells.Select
```

L'ultima istruzione equivale al comando **Seleziona tutto** (attivabile col pulsante omonimo, dato dall'incrocio delle intestazioni di righe e colonne).

Range ("A1") o semplicemente ***[A1]*** e *Cells(1, 1)* sono equivalenti: entrambi individuano la cella A1, tuttavia la notazione *Cells(indiceRiga, indiceColonna)* è di solito preferibile in programmazione, dato che gli indici numerici sono più facili da manipolare. Per esempio le seguenti istruzioni creano sul foglio le stringhe da "Trimestre 1" a "Trimestre 4" a partire dalla cella B1 fino alla cella E1.

```
For Ntrim = 1 To 4
 Cells(1, Ntrim + 1).Value = "Trim" & Ntrim
Next Ntrim
```

Con la proprietà *Range* potremmo arrangiarci con l'operatore &, cosa non ardua (ma un po' cervellotica) nel caso di righe:

```
For Nz = 1 To 4
  Range ("A" & Nz + 1).Value = "Zona vendita " & Nz
Next Nz
```

Ma la proprietà *Range* è tutt'altro che spregevole. Infatti essa è l'unica che permetta di impostare celle non singole. Finora ne abbiamo visto esclusivamente (tranne un caso) la sintassi con un solo argomento, ora è più che urgente esaminare la seconda, di ben maggiore importanza:

***Range*(*Cella1*, *Cella2*)**

che individua un intervallo in base alle celle diagonali. I due argomenti possono a loro volta essere ottenuti con proprietà (come *ActiveCell* o *CurrentRegion*) atto a individuare una cella, in particolare con la proprietà *Cells* o anche con la proprietà *Range*. Ecco un primo, banale esempio che abbina la sintassi 2 di *Range* con Cells (viene applicato il corsivo nell'intervallo A1:F10):

```
Range(Cells(1,1), Cells(10, 5)).Font.Italic = True
```

La casistica relativa a *Range(Cella1, Cella2)* è varia e non sempre semplice, specie quando si vuol puntare "a distanza" un intervallo. Ecco un esempio che impermalisce il debugger, con qualche sorpresa da parte nostra:

```
Range(Range("Miazona"), ActiveCell)
```

ove "Miazona" si suppone sia un nome di intervallo predefinito. La sopresa nasce nel caso che tale nome si riferisca, poniamo, a B10:B11, mentre la proprietà *Range* esige come argomenti *singole* celle.

NOTA - Invece, il codice *Range("Zona1:Zona2")*, già incontrato precedentemente, viene regolarmente accettato e funziona a dovere. Questo perché viene applicata la Sintassi 1 della proprietà *Range* (e "Zona1:Zona2" nella sintassi Excel individua un ben preciso intervallo)..

Applicazione relativa di *Cells* e *Range* a un intervallo

L'allievo è invitato ad esercitarsi con diverse combinazioni, come ad esempio *Range(ActiveCell, ActiveCell.Offset(...))*, che definisce una zona che parte dalla cella attiva e va a quella caratterizzata da un dato scarto rispetto alla prima.

Ma è tempo di completre la trattazione delle proprietà *Range* e *Cells* con quella che è la loro caratteristica più singolare (e valida): entrambe si applicano pure a un oggetto Range, nel qual caso l'intervallo individuato è *relativo* all'oggetto Range stesso.

Esempio:

Range("C5:H10").Range("B3")

individua la seconda colonna, terza riga di C5:H10, ossia D7

La definizione appena enunciata induce di primo acchito a pensare all'intervallo-oggetto come ad un foglio parziale (o sottofoglio) di quello di lavoro. E a questa stregua il significato dell'esempio appena dato è subito chiaro a tutti, dopo riflessione.

In realtà il principio è più generale. Infatti l'applicazione relativa di *Range* calcola i riferimenti di una cella o intervallo rispetto alla cella d'angolo (in alto a sinistra) dell'intervallo-oggetto (in altri termini ancora, tale cella d'angolo equivale alla cella A1 del foglio). È in gioco l'intero foglio e non solo il primo intervallo!

Supponiamo che sia selezionata una sola cella, la cella D4. Si avrà:

```
Selection.Range("B1")
Selection.Range("B1:D4")
```

Vengono restituite, nei due casi, la cella E4 e la zona E4:G7.

Altri esempi sparsi:

```
Range("C5:H10").Range("G9").Select
'Seleziona una cella distante 6 colonne e 8 righe
'dalla cella d'angolo C5, ossia I13 (esterna a C5:H10)

Range("A1:K20").Copy
'L'intervallo specificato è copiato negli Appunti
'N.B. - I trattini continuano a ruotare attorno ad esso
'perché l'operazione, in Excel, va completata

Selection.CurrentRegion.Range("A3:B4").Select
'Seleziona una sottozona di 2 x 2 celle relativa alla zona corrente

ActiveCell.Range("A1").Select
'Si resta nella cella attiva
```

L'ultimo caso mette in luce che *<oggetto-Range>.Range("A1")* in modo relativo non sposta un bel niente. Il registratore delle macro sovente inserisce un termine, pleonastico, *Range("A1")*, che si può tranquillamente eliminare.

Ripetiamo poi che invece il codice *ActiveSheet.Range("B1")* o semplicemente *Range("B1")* restituisce sempre la cella B1.

Con la proprietà *Cells* applicata in modo relativo le cose procedono del tutto analogamente, salvo che viene sempre individuata una singola cella. Pertanto bastino pochi esempi al riguardo:

```
Range("B4:D7").Cells(3, 2)
'Individua riga 3 colonna 2 rispetto a B4:D7, cioè C6

Range("B4:D7").Cells(7, 4)
'Individua la cella E10, esterna a B4:D7

Range("B4:D7").Cells(4)
'Individua la cella B5

Range("B4:D7").Cells(17)
'Individua la cella C9, esterna a B4:D4
```

Per sperimentare questi esempi (come altri consimili) può essere utile il ricorso alla finestra **Immediata**. Con riferimento al primo dei precedenti esempi (ma uno vale l'altro), si compiano le operazioni seguenti:

1) portarsi con **Alt+F11** nell'Editor Visual Basic;

2) attivare la finestra Immediata, con **Ctrl+G**;

3) nella finestra Immediata digitare:

```
?Range("B4:D7").Cells(3, 2).Address
```

4) premere **Invio**

Premesso che **?** è l'abbreviazione di *Print* e che la proprietà ***Address*** di un oggetto Range ne restituisce i riferimenti, nell'esempio in parola si otterrà la segnalazione C6, nella riga sottostante della finestra Immediata.

NOTA - Si faccia attenzione al fatto che *Cells(IndiceRiga)* applicato a una singola cella punta a celle poste sulla stessa colonna. Così *Range("B4").Cells(4)* restituisce B8 e non F4 come forse ci si potrebbe attendere.

Inserimento di dati e formule, congelamento di valori

L'inserimento simultaneo di dati o formule si ottiene con istruzioni di assegnazione applicate ad un oggetto Range in cui al secondo membro va posto il dato (costante) o la formula.

Le specifiche proprietà sono: ***Value***, ***Formula*** e ***FormulaR1C1***. Le formule, precedute dal segno =, vanno chiuse fra virgolette. Tutte e tre le proprietà inseriscono costanti se a secondo membro non è presente una formula virgolettata.

Tutte e tre le proprietà in esame sono poi fruibili anche in lettura. Di qui nasce l'idea di sfruttare la cosa per convertire una serie di formule in valori, illustrata nella routine seguente:

```
Sub Valorizza()
'Converte in valori le formule di Selection
  For Each MiaC In Selection
    MiaC.Value = MiaC
    'Equivale a MiaC.Value = MiaC.Value
  Next
End Sub
```

Si noti che *C.Value = C* equivale a *C.Value = C.Value* e che *C.Formula = C.Formula* lascerebbe invece le cose immutate. Interessante è poi la scoperta che tale routine si può sintetizzare, in quanto applicabile a un intero intervallo.

Per Esempio:

```
Range("A1:C20").Value = Range("A1:C20").Value
```

congela in valori tutte le formule dell'intervallo specificato.

Purtroppo Excel VBA in questi casi si rivela piuttosto lento: con un intervallo di 100 righe e 10 colonne pieno di normali formule occorrono circa 15 secondi mentre il codice che traduce il comando **Modifica / Copia** seguito da **Modifica / Incolla Speciale...** con opzione **Valori**. Si propone, per un confronto velocifero, questa seconda routine; il divario temporale, soprattutto rispetto alla routine Valorizza, risulterà evidente:

```
Sub ValorizzaExcel()
  Selection.Copy
  Selection.PasteSpecial Paste:=xlValues
  Application.CutCopyMode = False
End Sub
```

Lettura e scrittura di formule

Le proprietà viste non sono così specializzate come a prima vista si potrebbe credere: in realtà la proprietà **Formula**, se in lettura restituisce formule in stile **A1**, in scrittura accetta formule in entrambi gli stili **A1** ed **R1C1**. Per convircersene si provino i casi seguenti:

```
ActiveCell.Formula = "=A1+A2"
ActiveCell.FormulaR1C1 = "=R[-1]C"
ActiveCell.Formula = "=R[-1]C"
'Equivale alla precedente!
Selection.FormulaR1C1 = 123
```

Circa il significato della notazione R1C1 (ereditata dal defunto spreadsheet Multiplan, della Microsoft) non possiamo che rinviare al manuale utente di Excel, limitandoci qui a ricordare che è più potente del normale stile A1. Vediamo di spiegare la cosa con un esempio concreto.

Supponiamo che sia attualmente selezionato B7:B13, con B7 cella attiva. Volendo inserirvi la formula =B6+1, per creare una serie di progressivi successivi al valore iniziale (posto nella cella sovrastante), un ingenuo potrebbe pensare al codice seguente:

```
Selection.Formula = "=B6+1"
```

Con B7:B13 la cosa funziona: in B7 si leggerà =B6+1, in B8 =B7+1 e così via. Ma selezionando un intervallo differente, ad esempio E5:E11 non si può certo lanciare lo stesso codice, che darebbe B6+1 in E5, B7+1 in E6, e via di seguito, ossia una serie identica all'altra. D'altro canto la formula "=E4+1" sarebbe a sua volta dipendente dal nuovo intervallo e non applicabile a *qualunque* selezione.

Con lo stile R1C1 si avrebbe invece:

```
Selection.FormulaR1C1 = "=R[-1]C"
```

E stavolta tutto va a posto, perché i riferimenti R[-1]C puntano a una riga sopra, stessa colonna, pertanto si traducono in B6 nel primo caso e in E4 nel secondo caso. Insomma lo stile R1C1 è più flessibile.

Due esempi elementari: Fibonacci e conto corrente

Fibonacci è il matematico pisano del 1200, primo ad introdurre in Europa le cifre arabe, tuttora celebre per la serie numerica che simula il progredire di una popolazione di conigli immortali. Si immagini, da qualche parte del foglio di lavoro (corrente) una colonnina in cui creare la serie fibonaccesca, a partire da due valori costanti: solitamente 0 (popolazione iniziale) e 1 (capostipite). Nel listato che segue si ha la routine d'inserimento della serie, dotata di argomento e col seguito di una possibile routine di prova *ProvaFib*, che la richiama.

```
Sub Fibonacci(ZonaFib Come Oggetto)
  ZonaFib.FormulaR1C1 = "=R[-2]C+R[-1]C"
End Sub

Sub ProvaFib()
'Routine di prova della routine Fibonacci
  With Selection.Range("A1")
    .Offset(-1).Value = 1: .Offset(-2).Value = 0
  End With
  Fibonacci Selection
  'Applica alla selezione corrente la routine Fibonacci
End Sub
```

Si ribadisce che solo la notazione R1C1 consente di creare la nostra serie in qualsiasi selezione. Quanto alle istruzioni introduttive di ProvaFib, esse sfruttano la proprietà *Offset* a partire dalla prima cella della selezione (qualunque essa sia). In altri termini, *Range("A1")* punta in modo relativo a *Selection* (come tutti comprendono, andava bene anche *Cells(1, 1)* in luogo di *Range("A1")*.

Un secondo esempio si riferisce alla gestione dei successivi saldi in conseguenza di entrate e uscite. Queste sono riportate nelle colonne, rispettivamente, G e H, a partire dalla riga 6. Si parte da un saldo precedente, posto in I6, dopo di che la formula nella cella I7, da ricopiare nelle celle sottostanti, dovrebbe essere chiara a tutti. Comunque eccola: =I6+G7-H7 (saldo precedente più Avere meno Dare). Ed ecco la routine d'inserimento con relativa routine di prova:

```
Sub ContoCorr(ZonaCC Come Oggetto)
  ZonaCC.FormulaR1C1 = "=R[-1]C+RC[-2]-RC[-1]"
End Sub

Sub ProvaContoCC()
  ContoCorr Selection
End Sub
```

La procedurina *ProvaContoCC* funziona, naturalmente, a patto che l'utente preselezioni, prima di lanciarla, l'intervallo opportuno: ipotizzando che l'ultimo movimento, in entrata o uscita, si trovi a riga 50, la selezione preliminare sarà I7:H50.

Ma forse non tutti conoscono una particolarità di Excel, ossia la possibilità d'inserire in un sol colpo una formula in tutte le celle selezionate. Con riferimento al nostro caso, le mosse da compiere sono queste: 1) selezione di I7:H50; 2) digitazione della formula =I6+G7-H7 nella cella attiva I7; 3) conclusione con **Ctrl+Invio** anziché il normale Invio.

Ebbene la proprietà *FormulaR1C1* (come la più semplice *Formula*) applicata a un oggetto Range di più celle, riproduce in ambito programmatico tale (potente) peculiarità di Excel.

Esempi di copia (dinamica)

Per concludere, riprendiamo il discorso interrotto più volte, cercando di tirarne le fila. Riportiamo qui setto un modello di calcolo di importi lordi, sconti, importi netti, estremamente semplificato. Non ci si scandalizzi di ciò, l'importante è immaginare che le formule nelle colonne da E a G (ma, altrove, disposte lungo altre colonne e con un diverso numero di colonne) siano tali da potersi ricopiare in basso, in un numero di righe - ecco il punto - pari al numero, variabile e non noto a priori, di movimenti i cui dati sono introdotti dall'utente nelle colonne da B a D.

	B	C	D	E	F	G
. . .	**……**					
9	**Calcolo di importi, sconti, ecc...**					
10	**Prodotti**	**Quant**	**Prezzo**	**Importo**	**Sconto**	**Netto**
11	Mele	120	1.000	120.000	2.000	118.000
12	Pere	300	1.800			
. . .	**……**					
20	Pesche	500	2.000			
21	Banane	450	2.900			

(Nel precedente schema, com'è ovvio, l'intervallo E12:G21 è al momento vuoto: dovrà essere riempito delle formule in E11:G11, ricopiate in basso)

Per operare, tramite macro, la copia in basso di una riga di formule, aggiungiamo ora l'ipotesi che all'intervallo di formule in E11:G11 sia stato preassegnato il nome *Rigaform*. Nel listato che segue, la routine *CopiaInBasso* è una soluzione che fa uso delle proprietà *Offset* e *End*:

```
Sub CopiaInBasso()
  Application.Goto Reference:="Rigaform"
  NumRiga = Selection.Offset(0, -1).End(xlDown).Row
  NumCol = Selection.End(xlToRight).Column
  Set PrimaCella = Range("Rigaform").Cells(1, 1)
  Set UltimaCella = Cells(NumRiga, NumCol)
  Selection.Copy
  Range(PrimaCella, UltimaCella).Select
  ActiveSheet.Paste
  Application.CutCopyMode = False
End Sub
```

Per determinare l'intervallo di destinazione, ossia D11:F21 nel nostro caso, è sufficiente conoscere l'ultima riga della colonna C e l'ultima colonna della *Rigaform*. Entrambi i compiti sono affidate alla proprietà ***End***, che rispecchia la ben nota manovra manuale che consiste nella pressione del tasto **Fine** seguito da freccia o l'equivalente **Ctrl+freccia**.

Commenti. La seconda istruzione determina il numero di riga che si trova una cella a sinistra di *Rigaform* e tante celle in basso quante ne ha la "sponda", così chi scrive chiama (pur non giocando mai a biliardo) una serie di celle non vuote contigue poste su una colonna (o riga). Più facile il lavoro della terza istruzione, che determina l'ultima cella a destra della selezione corrente, ossia sempre *Rigaform*. Subito dopo vengono impostati nelle variabili PrimaCella e, rispettivamente, UltimaCella i riferimenti della cella di sinistra di Rigaform e della cella estrema di cui alla riga e colonna appena calcolate. Si noti, ancora una volta, l'uso relativo di *Cells(1,1)* in *Range.Cells(1, 1)*.

Il resto dovrebbe essere facilmente comprensibile.

Una variante, riportata qui sotto, fa ricorso al metodo ***FillDown***. Già anticipato in precedenza, tale metodo è il corrispettivo VBA del comando manuale **Modifica / Ricopia *in basso...***, (divenuto Modifica / Riempimento / In basso, in Excel 2000), che agisce direttamente sulla selezione completa, che va dalla riga in alto da copiare (inclusa) fino all'ultima riga interessata (analoghi risultati, mutatis mutandis, si hanno coi metodi ***FillRight***, ***FillLeft*** e ***FillUp***, che rispecchiano i comandi Ricopia a destra..., Ricopia a sinistra... e Ricopia in alto...). Ma ecco la routine in questione:

```
Sub RicopiaSotto()
  Range("Rigaform").Select
  'Equivalente (e più semplice) di Goto Reference:= "Rigaform"
  With Selection
    Nr = .Offset(0, -1).End(xlDown).Row - .Row
    .Resize(Nr + 1).Select
  End With
  Selection.FillDown
End Sub
```

Questa routine costituisce un'alternativa più compatta ed elegante della precedente, affidata all'analisi autonoma del lettore, nuovamente invitato a leggere la pagina dedicata al metodo *Resize* nella Guida in linea.

Un'osservazione non proprio banale però va fatta: nell'ultima istruzione "Selection." *deve* essere ripetuto! Infatti l'intervallo puntato da *With Selection*... *End With* rimane quello di partenza (ossia il solito intervallo Rigaform), perciò occorre riprendere *Selection* per puntare correttamente alla nuova selezione.

Copia con destinazione

I procedimenti fin qui visti rispecchiano più o meno da presso quelli compiuti dall'utente. Di conseguenza, oltre a produrre del codice abbondante, hanno il difetto di non operare "a distanza", basati come sono su operazioni di selezione.

Anche i più esperti e, ancor più, gli audaci e spericolati vanno incontro a seri rompicapi, a causa di scogli come i seguenti:

- ***Selection*** e ***ActiveCell*** sono proprietà di un oggetto-finestra e non di un oggetto foglio di lavoro;
- ***Paste***, contrariamente a *Copy,* è un metodo applicabile a un oggetto foglio di lavoro (è concettualmente ovvio, ma è facile dimenticarsene).

A onta di ciò, provando e riprovando, e con un po' di ingegno la "ricopiatura" in basso e *a distanza*, sempre nel problemino precedente, può essere compiuta con la variante che segue:

```
Sub RicopiaSottoADist()
  With Sheets("Foglio1").Range("Rigaform")
    Nr = .Offset(0, -1).End(xlDown).Row - .Row
    Set IntervRicop = .Resize(Nr + 1)
  End With
  IntervRicop.FillDown
End Sub
```

Ma Excel VBA ha ulteriori risorse. Per illustrarle si pensi a un modello simile a quello in parola, ma relativo alle solite previsioni di vendite che, per estrema semplificazione didattica, si suppongono crescere di una data percentuale a ogni periodo. La formula in gioco la si suppone scritta in una cella a parte e la macro, al solito modo dinamico e autoadattante, la deve copiare nell'intervallo opportuno. Quest'ultimo, per fissare le idee, sia D5:F11 ma sempre da determinare in base all'ampiezza (variabile e ignota a priori) dei valori di sinistra, mentre la cella con la formula si suppone pre-denominata "formulina". Sempre per meglio fissare le cose, la cella d'angolo in questione la si supponga predenominata "inizon".

Ebbene, entrambe le procedure CopiaInDest1 e CopiaInDest2, riportate qui sotto approfittano del fatto che il metodo **Copy** ammette anche un intervallo di destinazione:

```
Sub CopiaInDest1()
  With Range("Inizon").CurrentRegion
    .Select
    Nr = .Righe.Count: Nc = .Colonne.Count
  End With
  Range("Inizon").Offset(1, 2). _
  Resize(Nr - 1, Nc - 2).Select
  Range("formulina").Copy Destination:=Selection
End Sub


Sub CopiaInDest2()
  With Range("Inizon").CurrentRegion
    Nr = .Righe.Count: Nc = .Colonne.Count
  End With
  Set ZonaFormule = Range("Inizon").Offset(1, 2) _
  .Resize(Nr - 1, Nc - 2)
  Range("formulina").Copy ZonaFormule
End Sub
```

In altri termini la sintassi completa di tale metodo è la seguente:

Copy *Destination*

ove *Destination* è un argomento facoltativo, fin qui ignorato. Tornando al listato precedente, l'unica cosa meritevole di commento è l'ultima istruzione, facilmente interpretabile in questi termini: prendi l'intervallo di nome "formulina" e copialo nella destinazione Selection (individuata da precedenti manovre).

La routine *CopiaInDest2*, come è facile comprendere, è una variante, che opera a distanza e senza necessità di selezionare preliminarmente la destinazione. Se nelle precedenti istruzioni si sostituisce il termine *Range* con *Sheets("Foglio1").Range*, la routine *CopiaInDest2* agisce, per così dire, nell'ombra, ossia persino quando è attivo un foglio non di lavoro (per grafico o addirittura per macro). Provare per credere: il merito è anche del fatto che la proprietà *CurrentRegion* si applica a un foglio di lavoro non a un oggetto finestra, come *Selection*.

Vediamo ora una variante che fa uso della proprietà *Formula*. Secondo quanto già intravisto a suo tempo un'istruzione del tipo

```
TuoInterv.Formula = MiaCella.Formula
```

Equivale alla copia della formula contenuta in *MiaCella* nell'intervallo *TuoInterv*.

Dopo questo richiamo la maggioranza comprende da sé che la precedente routine poteva anche concludersi così:

```
ZonaFormule.Formula = Range("formulina").Formula
End Sub
```

Trascinamento di dati e formule

In Excel molto utile e popolare è il quadratino di trascinamento, mediante il quale si possono ottenere le più varie sequenze di dati o la copia di formule, a partire dal contenuto della o delle celle iniziali.

Com'è da attendersi, Excel VBA rispecchia tali procedimenti. E per chiudere questo piccolo corso sul trattamento di intervalli diamo un esempio, che crea, su indicazione del primo mese da parte dell'utente, una successione di mesi dell'anno: "Gennaio", "Febbraio",... oppure "Gen", "Feb".... Ovviamente si deve rispondere in modo preciso, in particolare il mese iniziale dev'essere esattamente uno di quelli previsti, pena il fallimento della macro:

```
Sub SerieRiempiMesi()
  Worksheets("Foglio2").Select
  'ActiveCell.Select è SUPERFLUO:
  'La macro NON fallisce se non è selezionata una sola cella
  Set C = ActiveCell 'Per abbreviare le righe seguenti
  C.Value = InputBox("Dammi un mese...")
  Set IntervDest = Range(C, C.Offset(11))
  mess = "In basso (SI) o verso destra (NO)?"
  DimmiTu = MsgBox(mess, vbSiNo)
  Se DimmiTu = vbNo Then Set _
  IntervDest = Range(C, C.Offset(0, 11))
  C.AutoFill Destination:=IntervDest, Type:=xlRicopiaMesi
End Sub
```

Tralasciamo ogni commento (vi sono già quelli intercalati nel listato) e rinvio a quanto recita la Guida circa il metodo ***Autofill***. Si viene così a sapere che l'argomento *Destination* ha lo stesso ruolo dell'omonimo del metodo *Copy*, mentre ***Type*** può assumere valori come *xlRicopiaSerie*, *xlRicopiaGiorni*, *xlRicopiaGiorniSettimana* (ossia "lunedì", "martedì" ecc.), *xlRicopiaMesi* ed altri speciali, più *xlRicopiaCopia*. Se ***Tipo*** è *xlRicopiaPredefinito* o omesso viene adottato il tipo più appropriato all'intervallo origine.

L'applicazione del codice VBA di riempimento automatico alla copia dinamica di formule viene lasciata come ultimo, simpatico e non arduo esercizio.

Congedo...

Il mondo di Excel VBA è molto vasto e in queste note introduttive abbiamo solo sfiorato le proprietà e i metodi principali, relativi agli oggetti tipici dello spreadsheet: fogli (e finestre), intervalli e celle. Anche gli esempi proposti, non esauriscono certo la materia. Sensa falsa modestia, ci sentiamo di dire che sono esempi semplici ma istruttivi, che sfruttano interessanti tecniche e artifici per la copia o l'inserimento dinamico e autoadattativo di dati e formule, tecniche alle quali, ci consta, pochi utilizzatori di Excel VBA sanno ricorrere.

Sulla scorta di tali suggerimenti, mutatis mutandis, il lettore riflessivo e non privo di fantasia creativa potrà escogitare per suo conto soluzioni altrettanto se non addirittura più valide. Ci contiamo.