di Sergio Polini

Metodi statici e metodi virtuali

Su MC-Link un abbonato ha chiesto se il Turbo Vision del nuovo Pascal 6.0 è «facile da usare». Sintetizzando al massimo argomentazioni che svolgo nell'ambito della prova (che potete trovare in altra parte della rivista), gli ho risposto: non facilissimo, ma meno difficile di quanto potremmo aspettarci dalle notevoli potenzialità del prodotto, soprattutto se si ha già familiarità con la OOP, in particolare con i meccanismi di derivazione di proprie classi da quelle comprese nella gerarchia implementata nelle unit precompilate. Un motivo di più per approfondire l'aspetto che forse meglio caratterizza la programmazione orientata all'oggetto

La volta scorsa abbiamo cominciato ad esaminare la «sintassi» della OOP, per osservare che tutto ruota attorno ad una piccola modifica del tradizionale tipo **record** e a poche nuove parole riservate. Abbiamo discusso due di queste, **object** e **private** (l'ultima introdotta con il Turbo Pascal 6.0), e... siamo già a buon punto! Le nuove parole riservate, quelle che consentono di trasformare il buon vecchio Pascal in un linguaggio orientato all'oggetto, sono infatti in tutto cinque: ci mancano **virtual, constructor** e **destructor**.

A rigore non sono tutte vere e proprie «parole riservate»; il manuale recita che **private** e **virtual** sono piuttosto «direttive standard» e potrebbero essere ridefinite, usate cioè come normali identificatori. Ai fini pratici, tuttavia, soprattutto quando si fa OOP, conviene «dimenticare» la possibilità di ridefinizioni che non porterebbero altro che confusione.

Classi astratte

Un mese fa vedevamo come, grazie ad un parametro «invisibile» (*Self*), le chiamate dei metodi possono facilmente essere ricondotte a normali chiamate di procedura con la sola aggiunta, appunto, di un parametro contenente l'indirizzo della variabile appartenente ad un tipo **object**. Si trattava però solo di una prima approssimazione. Per poter meglio apprezzare le possibilità della OOP, vi propongo quindi ora alcuni esempi un po' più articolati.

Cominciamo col dire che spesso, quando si disegna una gerarchia di classi, vengono previste anche *classi astratte*. Non si tratta di altro che di una trasposizione del nostro normale modo di atteggiarci nei confronti del mondo reale: possiamo dire che il cane «è un» animale, che anche il gatto «è un» animale, ma, mentre cani e gatti esistono realmente (accarezzo il cane, il gatto mi graffia, ecc.), «animale» è solo un concetto astratto. Quando pensiamo al concetto di «animale», pensiamo a quelle caratteristiche che accomunano cani, gatti, bisonti, elefanti, e così via, ma non individuano alcun animale reale. Analogamente, nel Turbo Vision vi sono classi astratte come *TMenuView* che definisce «in astratto» i menu, in quanto va intesa come classe con la quale si definiscono le caratteristiche comuni ai vari tipi di menu; non si prevede la definizione di variabili appartenenti al tipo (o meglio: di oggetti istanza della classe) *TMenuView*, ma solo di istanze di classi da questa derivate, come *TMenuBar* (menu a barra) e *TMenuBox* (menu pop-up). Così come i concetti astratti, quindi, le classi astratte aiutano a mettere ordine nei nostri ragionamenti.

In termini pratici, una classe astratta è una classe per la quale non viene definita l'implementazione di tutti i suoi metodi, in quanto si rimanda all'implementazione dei metodi con lo stesso nome dichiarati in classi derivate. Ogni linguaggio adotta i suoi accorgimenti. In C++, ad esempio, si usa un singolare artificio sintattico, ovvero una notazione che sembra quasi assegnare il valore zero al metodo la cui implementazione si desidera rimandare alle classi derivate; ciò consente di individuare le classi astratte fin dalla compilazione. In SmallTalk e in Objective-C si usano i «messaggi», rispettivamente, *implementedBySubClass* e *subclassResponsibility;* in Turbo Pascal si usa in genere una procedura *Abstract* (definita nella **unit** OBJECTS.PAS) che provoca un errore di esecuzione numero 211. In tutti e tre gli ultimi casi viene sì definita una implementazione per il metodo «astratto», ma questa si riduce all'invio di quei messaggi o alla chiamata di quella procedura.

So bene che l'importanza delle classi astratte può essere apprezzata appieno solo dopo aver provato a disegnare gerarchie non banali, ma proprio per questo, per... aiutarvi ad acquisire buone abitudini, gli esempi che vi propongo presuppongono una gerarchia composta da una classe astratta, *TAnimale,* da due classi da questa derivate, *TGatto* e *TCane,* e da una classe *TLupo* derivata da *TCane*. Sempre a proposito di buone abitudini, cercherò di uniformarmi alla notazione usata nel Turbo Vision: i nomi

delle classi iniziano quindi con una «T» (tipo), quelli dei tipi «puntatore alla classe X» con una «P» (puntatore).

Derivazioni imperfette

Cominciamo con la **unit** STATICI.PAS (figura 1) e il programma ANIMALI1.PAS (figura 2). Vengono dichiarate le classi *TAnimale, TGatto* e *TCane,* con gli associati tipi puntatore; la classe astratta comprende un solo dato comune a tutti gli animali, il loro verso (il gatto miagola, il cane abbaia), oltre ad un **constructor** *Init* e una procedura *ChiSei.* Il **constructor** è un particolare tipo di metodo che provvede a vari compiti di inizializzazione implicita delle istanze di una classe ed è quindi la sede più opportuna per provvedere anche alla esplicita inizializzazione dei campi-dati (per questo la Borland suggerisce di usare sempre il nome *Init*); la procedura implementa il metodo con cui ogni animale «risponde» al messaggio «Chi sei?». Nel nostro semplice caso, al campo-dato viene assegnata una stringa che descrive il verso di ogni animale, la risposta al messaggio consiste nella visualizzazione di una stringa con la quale ogni animale ci dice che tipo di animale è e quale è il suo verso.

L'esempio è terribilmente banale, ma consente di apprezzare già alcuni aspetti di non trascurabile importanza. Vediamo in primo luogo come va dichiarata una classe astratta: il metodo *ChiSei* della classe *TAnimale* si riduce alla chiamata di una procedura *ClasseAstratta,* che a sua volta non fa altro che provocare l'errore 211 mediante la procedura predefinita *RunError.* Vediamo poi che le classi derivate ereditano sia il campo-dato che i metodi della loro classe base, compreso il **constructor,** e si limitano a ridefinire il metodo *ChiSei,* come d'altra parte devono fare perché sia possibile usare in un programma variabili che siano loro istanza. ANIMALI1.PAS non fa altro che verificare che tutto funzioni: le variabili *Animale2 e Animale3,* alla domanda «Chi sei?», ci rispondono dicendoci che sono, rispettivamente, un gatto che miagola e un cane che abbaia; la variabile *Animale1* può essere dichiarata come istanza della classe astratta e può anche essere inizializzata, ma non può rispondere a quella domanda: si può dire che ogni animale ha un verso, ma il concetto «animale» non ne ha alcuno in particolare.

In ogni linguaggio di programmazione che si rispetti è possibile dichiarare non solo variabili «elementari», ma anche variabili «strutturate»; nel nostro caso, quindi, abbiamo ben diritto di aspettarci di poter usare array di animali. Ci proviamo nel programma ANIMALI2.PAS (figura 3).

Naturalmente desideriamo un array i cui elementi siano istanza di tutte le classi che abbiamo derivato da *TAnimale,* non un array di soli gatti o di soli cani. Ci imbattiamo così nel problema della compatibilità tra istanze di diverse

Figura 1
Una unit in cui viene dichiarata una piccola gerarchia di classi con uso di soli metodi statici.

```
unit Statici;

interface

type
  PAnimale = ^TAnimale;
  TAnimale = object      (* classe astratta *)
    Verso: string;
    constructor Init(v: string);
    procedure ChiSei;
  end;
  PGatto = ^TGatto;
  TGatto = object(TAnimale)
    procedure ChiSei;
  end;
  PCane = ^TCane;
  TCane = object(TAnimale)
    procedure ChiSei;
  end;

procedure ClasseAstratta;

implementation

constructor TAnimale.Init;
begin
  Verso := v;
end;

procedure TAnimale.ChiSei;
begin
  ClasseAstratta;
end;

procedure TGatto.ChiSei;
begin
  Writeln('Io sono un gatto e ', Verso);
end;

procedure TCane.ChiSei;
begin
  Writeln('Io sono un cane e ', Verso);
end;

procedure ClasseAstratta;
begin
  RunError(211);
end;

end.
```

Figura 2 - Il programma ANIMALI1.PAS, che propone un primo esempio di uso delle classi dichiarate nella unit STATICI.PAS.

```
program Animali1;

uses Statici;

var
  Animale1: TAnimale; (* NB: Variabile istanza di una classe astratta! *)
  Animale2: TGatto;
  Animale3: TCane;

begin
  Animale1.Init('');
  Animale2.Init('miagolo');
  Animale3.Init('abbaio');
  (* Animale1.ChiSei;    darebbe un errore di esecuzione num. 211. *)
  Animale2.ChiSei;          (* --> 'Io sono un gatto e miagolo' *)
  Animale3.ChiSei;          (* --> 'Io sono un cane e abbaio' *)
end.
```

```
program Animali2;

uses Statici;

var
  ArrayDiAnimali: array[1..2] of PAnimale;
  i             : integer;

begin
  ArrayDiAnimali[1] := New(PGatto, Init('miagolo'));
  ArrayDiAnimali[2] := New(PCane, Init('abbaio'));
  for i := 1 to 2 do
    ArrayDiAnimali[i]^.ChiSei;  (* --> Errore di esecuzione num. 211. *)
end.
```

Figura 3 - Il programma ANIMALI2.PAS usa la stessa unit STATICI.PAS per tentare di realizzare un array di animali. Ma non funziona in quanto la unit non dichiara come virtuale il metodo ChiSei.

classi. La regola è semplice: all'istanza di una classe si può assegnare l'istanza di una classe da quella derivata, un puntatore ad una classe derivata può essere assegnato al puntatore alla classe da cui quella deriva. Altre assegnazioni non sono ammesse. Nel primo caso, accade che ai campi-dati dell'istanza della classe base sono assegnati i valori degli stessi campi dell'istanza della classe derivata; è chiaro che l'inverso non sarebbe possibile, in quanto una classe derivata può aggiungere altri campi-dati a quelli ereditati e non si saprebbe cosa assegnare ai campi aggiunti. Va però anche precisato che *l'assegnazione non comporta alcuna inizializzazione,* in particolare non produce gli stessi effetti della chiamata di un **constructor** (vedremo tra breve alcuni di questi effetti); non è consigliabile, quindi, riempire un array di oggetti assegnando ai suoi elementi oggetti dichiarati e inizializzati altrove. Conviene usare array di puntatori ad oggetti. Possiamo in questo modo sfruttare la possibilità di assegnare il puntatore ad una classe da questa derivata, e quindi dichiarare un array di puntatori alla classe *TAnimale* da riempire poi con puntatori a gatti e cani. Il Turbo Pascal con oggetti ci offre, con una sintassi estesa della classica *New,* il modo più pulito per creare tali puntatori: *New* può infatti ora essere usata sia come procedura che come funzione, e soprattutto ammette come ulteriore parametro il **constructor** della classe di cui si vuole creare un'istanza. Otteniamo così una istanza perfettamente inizializzata; uno dei compiti affidati ai **constructor,** tra l'altro, consiste proprio nel fornire a *New* l'informazione della dimensione in byte delle istanze di una classe, in modo che possa essere allocata la quantità corretta di memoria nello heap.

Il problema è che il programma ANIMALI2 non funziona: scatta subito l'errore 211.

Polimorfismo

Potremmo pensare che tutto dipende dal fatto che, pur avendo creato un array di cani e gatti, lo avevamo dichiarato come array di puntatori ad istanze di una classe astratta. Proviamo quindi a derivare la classe *TLupo* da *TCane* (nel solo senso che il lupo «è un» cane che ulula invece di abbaiare: non si arrabbino gli zoologi!) e a dichiarare un array di puntatori a *TCane* (figure 4 e 5). Assegnati all'array il cane e il lupo allocati con *New,* proviamo a chiedere loro chi sono: ci rispondono il primo «Io sono un cane e abbaio», il secondo «Io sono un cane e ululo». C'è ancora qualcosa che non va, ma possiamo cominciare a capire perché ANIMALI2 non funzionava.

L'output del programma dimostra che il campo-dato *Verso* viene correttamente inizializzato, e correttamente ogni istanza accede al proprio campo-dato, ma l'istanza della classe *TLupo* non riesce ad accedere al metodo *ChiSei* che era stato definito per la sua classe. ANIMALI2, quindi, non funzionava perché sia il cane che il gatto, nel tentativo di rispondere ognuno con il proprio metodo, non riuscivano ad andare oltre il metodo definito per la loro classe base, usata nella dichiarazione dell'array. Sembrerebbe quasi che quella possibilità di assegnazione tra puntatori a classi diverse serva a ben poco. In realtà è sufficiente dichiarare **virtual** quei metodi per i quali si prevede che le classi derivate, invece che semplicemente ereditare, provvedano ad una ridefinizione. La **unit** VIRTUALI (figura 6) riproduce le dichiarazioni delle **unit** STATICI e STATICI2, ma rende virtuali i metodi *ChiSei.* Il programma ANIMALI4 (figura 7), usando la nuova unit, può finalmente dichiarare un array di puntatori alla classe base, assegnare agli elementi di tale array gli indirizzi di istanze di classi derivate, anche indirettamente, da quella, ed ottenere che tali elementi siano in grado di comportarsi secondo la dichiarazione della classe cui effettivamente appartengono gli oggetti «puntati», nonostante l'accesso avvenga mediante puntatori ad una classe base. Vi prego di notare

```
unit Statici2;

interface

uses Statici;

type
  PLupo = ^TLupo;
  TLupo = object(TCane)
    procedure ChiSei;
  end;

implementation

procedure TLupo.ChiSei;
begin
  Writeln('Io sono un lupo e ', Verso);
end;

end.
```

Figura 4 - La unit STATICI2 aggiunge alle classi dichiarate in STATICI una nuova classe TLupo, derivata da TCane.

Figura 5 - Il programma ANIMALI3 prova ad usare la unit STATICI2, ma gli animali allocati dinamicamente con New ci diranno il primo che è un cane che abbaia, il secondo che è un cane che ulula. C'è qualcosa che non va.

```
program Animali3;

uses Statici, Statici2;

var
  ArrayDiAnimali: array[1..2] of PCane;
  i             : integer;

begin
  ArrayDiAnimali[1] := New(PCane, Init('abbaio'));
  ArrayDiAnimali[2] := New(PLupo, Init('ululo'));
  for i := 1 to 2 do
    ArrayDiAnimali[i]^.ChiSei;
end.
```

```
unit Virtuali;

interface

type
  PAnimale = ^TAnimale;
  TAnimale = object      (* classe astratta *)
    Verso: string;
    constructor Init(v: string);
    procedure ChiSei; virtual;
  end;
  PGatto = ^TGatto;
  TGatto = object(TAnimale)
    procedure ChiSei; virtual;
  end;
  PCane = ^TCane;
  TCane = object(TAnimale)
    procedure ChiSei; virtual;
  end;
  PLupo = ^TLupo;
  TLupo = object(TCane)
    procedure ChiSei; virtual;
  end;

procedure ClasseAstratta;

implementation

constructor TAnimale.Init;
begin
  Verso := v;
end;

procedure TAnimale.ChiSei;
begin
  ClasseAstratta;
end;

procedure TGatto.ChiSei;
begin
  Writeln('Io sono un gatto e ', Verso);
end;

procedure TCane.ChiSei;
begin
  Writeln('Io sono un cane e ', Verso);
end;

procedure TLupo.ChiSei;
begin
  Writeln('Io sono un lupo e ', Verso);
end;

procedure ClasseAstratta;
begin
  RunError(211);
end;

end.
```

◄ *Figura 6 - La unit VIRTUALI ripete le dichiarazioni di classe delle unit STATICI e STATICI2, ma rende virtuali i metodi ChiSei.*

```
program Animali4;

uses Virtuali;

var
  ArrayDiAnimali: array[1..3] of PAnimale;
  i             : integer;

begin
  ArrayDiAnimali[1] := New(PGatto, Init('miagolo'));
  ArrayDiAnimali[2] := New(PCane, Init('abbaio'));
  ArrayDiAnimali[3] := New(PLupo, Init('ululo'));
  for i := 1 to 3 do
    ArrayDiAnimali[i]^.ChiSei;
end.
```

▲ *Figura 7 - Il programma ANIMALI4 usa la unit VIRTUALI: abbiamo così finalmente gatti che miagolano, cani che abbaiano e lupi che ululano. Abbiamo cioè finalmente, grazie ai metodi virtuali, un array «polimorfico».*

che, a differenza dei semplicissimi esempi che vi avevo proposto il mese scorso, quanto accade in ANIMALI4 non è così facilmente traducibile in normale Pascal (la traduzione non è impossibile, ma decisamente ardua). Siamo infatti di fronte ad un fenomeno del tutto estraneo ai meccanismi della programmazione tradizionale: pur se siamo dovuti ricorrere a puntatori, abbiamo realizzato un «array di elementi qualsiasi». Normalmente il tipo degli elementi di un array è già determinato al momento della sua dichiarazione (un array di interi non può essere altro che ... un array di interi); qui abbiamo un array di elementi appartenenti a tutte le classi che sono state o saranno derivate da una classe base, il cui tipo effettivo potrebbe anche essere determinato solo al momento dell'esecuzione del programma, ad esempio in funzione dell'andamento della interattività con l'utente.

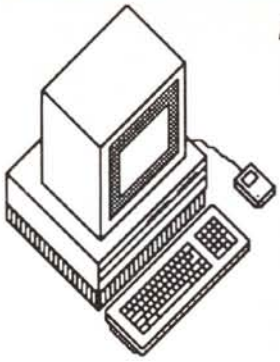
È quello che si chiama *polimorfismo.*

Dietro le quinte

Può essere interessante vedere come viene realizzato tutto ciò, provando a spulciare con il Turbo Debugger nel codice prodotto dal compilatore. Se si esamina il file ANIMALI1.EXE, si vede che la variabile *Animale1,* istanza di *TAnimale,* risiede all'indirizzo DS:$0050. Qui troviamo 256 byte riservati per il campo *Verso,* dichiarato come *string* (255 byte più uno contenente la lunghezza effettiva della stringa), seguiti da due byte contenenti $0002: questi vanno interpretati come l'offset nel segmento dati di una struttura denominata *Virtual Method Table* (VMT). Si tratta di una struttura creata in fase di compilazione per ogni classe che abbia un **constructor,** un **destructor** o metodi virtuali; nel caso di ANIMALI1 la VMT consta di soli quattro byte: i primi due denotano l'occupazione di memoria della classe (è questa l'informazione che il **constructor** prepara per le chiamate di *New* e *Dispose*), gli altri due lo stesso numero ma negativo, in modo da consentire il controllo durante l'esecuzione della validità dell'informazione, se si è attivata la direttiva $R. Nel nostro caso troviamo un $0102 (258 in decimale) seguito da un $FEFE.

In ANIMALI4.EXE vediamo che *ArrayDiAnimali[1]* contiene l'indirizzo $68AD:$0000, in cui si ripete la stessa struttura di prima (256 byte per *Verso* e due per l'offset della VMT). La VMT consta ora di otto byte: i primi quattro hanno gli stessi valori che avevano in ANIMALI1, gli altri due l'indirizzo $63B5:$0070, che altro non è che l'indirizzo di *Virtuali.TGatto.ChiSei.* Possiamo quindi indovinare che, quando viene «chiamato» un metodo virtuale, attraverso il parametro invisibile *Self* si accede all'indirizzo della VMT, e si provoca così l'esecuzione per ogni oggetto della versione del metodo definita per la classe di cui questo è istanza. Tutto ciò avviene, sottolineo, durante l'esecuzione: al momento della compilazione vengono sì create le VMT, ma nulla si sa su quelli che saranno i metodi da eseguire per gli elementi dell'array «polimorfico»; solo al momento dell'esecuzione, mediante lettura della VMT cui «punta» ogni elemento, si determinerà la rispettiva classe di appartenenza, e quindi il corretto metodo da eseguire.

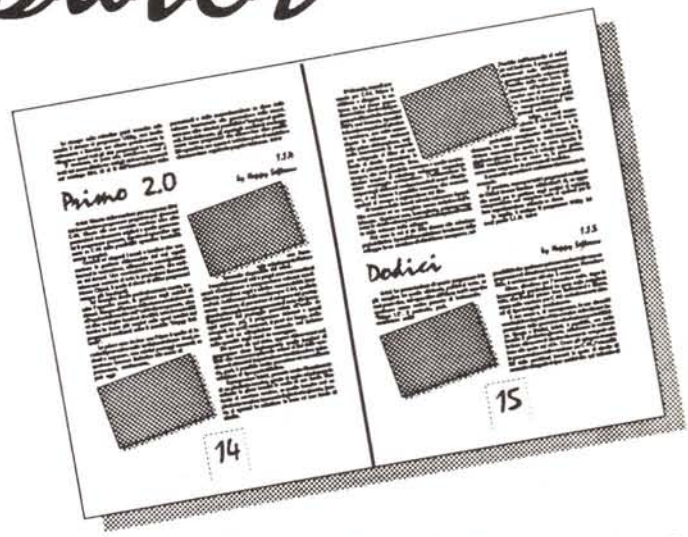
Si tratta di meccanismi su cui avremo modo di tornare, anche il mese prossimo quando parleremo dei **destructor;** dovrebbe comunque essere già chiara l'estrema importanza delle VMT, e quindi l'opportunità di seguire un consiglio della Borland: può capitare di dimenticare di chiamare il **constructor** prima di qualsiasi altra manipolazione di un oggetto; ne seguirebbe la mancata inizializzazione di quella struttura contenente l'indirizzo della VMT. Ad evitare questo, conviene attivare sempre (almeno fino al completo test di un programma) la direttiva $R, che consente appunto di verificare durante l'esecuzione che quel campo nascosto di due byte contenga effettivamente l'offset di una valida VMT. MC



Totocalcio & Computer

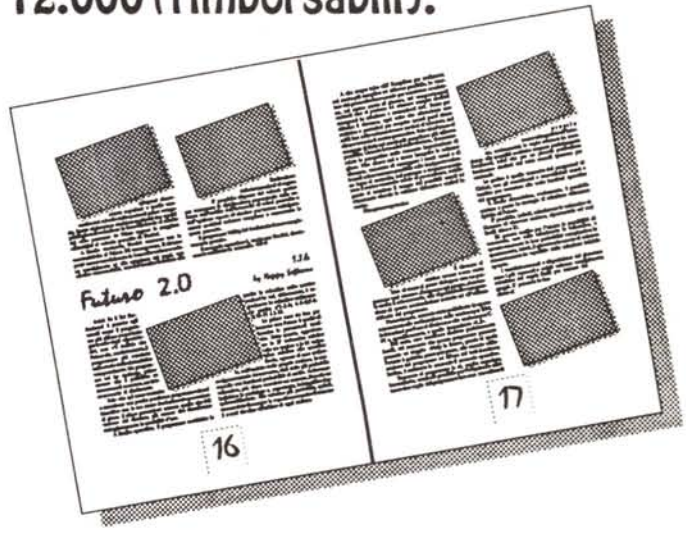
Totocalcio & Computer è la prima guida ragionata alla scelta ed all'uso del software professionale per giochi a pronostici.

Totocalcio & Computer è una rassegna dei programmi



più qualitativi e più economici, nonché un autentico manuale sistemistico integrato da un ampio glossario dei termini tecnici e da validi consigli per l'ottimizzazione delle giocate.

Totocalcio & Computer è presente in tutte le edicole d'Italia. Se è già esaurito puoi richiederlo col tagliando sottostante, allegando £ 12.000 (rimborsabili).



Gradirei ricevere presso il mio domicilio una copia di «Totocalcio & Computer». Allego versamento di L. 12.000 intestato a: La Schedina Srl - Viale Adriatico, 4 - 00141 Roma - Tel. 890481/2 - Fax 893476. **(non si effettuano spedizioni contrassegno)**

Nome e Cognome_____

Indirizzo _____

CAP_____ Città_____ Telefono_____