Delphi 2005 del 2 - programspråken

2004-11-13
Lars Gustafsson
DATABITEN AB

Som rubrik för denna artikel har jag medvetet valt "programspråken", inte "programspråket". Skälet är att Delphi 2005, inte bara är en uppgradering från tidigare Delphi-versioner och Object Pascal*. Delphi 2005 är också en uppgradering från Borland C#Builder 1.0 och inkluderar därför **C# .NET** som ett av sina två huvudspråk vid .NET-utveckling. Vid Win32-utveckling är dock Object Pascal i ensamt majestät.

*Jag kallar Delphis huvudspråk för "Object Pascal", även om jag vet att Borland numera rekommenderar benämningen "Delphi-språket" ("Delphi Language"). Men eftersom Delphi 2005 också är ett komplett utvecklingsverktyg för C#, tycker jag att "Object Pascal" är klarare. Det är också det namn som har använts fram till Delphi 7. Ett alternativ skulle kunna vara "Delphi Pascal".

C#

C# är ett modernt, typsäkert, interface-, objekt- och komponentorienterat språk, utvecklat av främst Anders Hejlsberg hos Microsoft. Anders, som tidigare var verksam hos Borland (1983-1996), är också känd för att ha skapat Turbo Pascal 1983 och Delphi 1995. Man kan ana dessa rötter när man arbetar med C#. Även om C# har lånat sin grundsyntax från C/C++/Java familjen, så ler man ofta igenkännande när man arbetar med C# på högre nivåer (som gränssnitt, klasser, komponenter, händelser, etc.) – det liknar ju snarare Object Pascal i Delphi!

C# är ett utmärkt val vid .NET-utveckling och källkoden till alla Microsofts standardbibliotek (assemblies) i .NET Framework Component Library (FCL) är också kodade i C#. Nu spelar det ju egentligen ingen roll när man använder FCL-biblioteken, eftersom .NET är språkneutralt – man kan använda kod från vilket .NET-språk som helst tillsammans med vilket *annat* .NET-språk som helst. Så ett assembly kodat i C#, kan användas tillsammans med Delphis Object Pascal, Visual Basic .NET, Visual C++ .NET osv. Och omvänt – ett assembly skrivet i Object Pascal kan användas i C#-projekt, etc.

Men mycket källkod och dokumentation i .NET är C#-orienterat, så det skadar det inte att lära sig åtminstone huvuddragen i detta språk. Sen är det mest tycke och smak vad man föredrar – C# eller Object Pascal. Det är ingen skillnad i möjligheter eller kodkvalitet. Om du ändå föredrar Object Pascal, så har Borland en experimentell webbtjänst, BabelCode, som översätter C# källkod till Object Pascal. Även om den inte är perfekt gör den ändå grovjobbet. Mer info om BabelCode hittar du här:

http://bdn.borland.com/article/0,1410,32338,00.html

Projekthanteringen i Delphi 2005 tillåter att man skapar projektgrupper med helt olika projekttyper – ett projekt kan vara skrivet i C#, ett annat i Object Pascal, etc. Editor och övriga verktyg i utvecklingsmiljön (objektinspektor, strukturfönster, verktygspalett, debugger, etc.) anpassar sig automatiskt till det projekt i projektgruppen som för tillfället är aktivt.

C#-kompilatorn i Delphi 2005 är faktiskt licensierad från Microsoft och därför 100% kompatibel med dito i Visual C# 2003 från Microsoft. Delphi 2005 har också wizards för att importera Microsoft Visual C#-projekt och exportera Delphi C#-projekt till Visual C#-format.

Det finns dock två nackdelar med C# jämfört med Object Pascal:

- C# kan inte användas vid Win32-utveckling, endast för .NET.
- C# kan inte användas tillsammans med Borlands portabla VCL.NET-bibliotek, endast med Microsofts FCL .NET och klasser baserade på detta.

I dessa fall är det Delphis Object Pascal som gäller ...

Delphi Object Pascal

Delphis Object Pascal söker sina rötter i Pascal från slutet av 60-talet, som i sin tur inspirerades av Algol från början av samma årtionde. Från 1983 och Turbo Pascal, är det Borland som har varit ledande vid vidareutvecklingen av Pascal. Ett tidigt tillägg var stränghanteringen (String-typen) och mot slutet av 80-talet tillkom modularisering (enheter/units) inspirerat av främst Modula-2 (också utvecklat ur Pascal). Under 90-talet kom så objektorientering och med Delphi (1995), komponentbaserad programmering. Senare introducerades programmering med gränssnitt/interface, först tillsammans med COM, men snart generaliserat som en del av språket. Idag är Delphis Object Pascal, defacto-standarden för Pascal.

Vi ska här titta på de viktigaste språknyheterna i Object Pascal hos Delphi 2005 (jämfört med Delphi 7 och 8). En del nyheter gäller både Win32 och .NET, medan andra enbart gäller .NET (åtminstone i denna Delphi-version).

Språknyheter i Delphi 2005 Object Pascal

UniCode (Win32 och .NET)

Vid editering av källkod använder Delphi 2005 alltid **UniCode**, dvs varje tecken i editorn upptar två bytes i minnet. Med UniCode kan alla internationella tecken (svenska, arabiska, ryska, japanska, etc.) användas utan problem. Men när källkoden sparas till disk, används fortfarande äldre 1-bytes ANSI-format, vilket gör att en internationella tecken kan gå förlorade (dock inga problem med svenska tecken så länge Windows är inställt på västeuropeiska tecken). När källkodsfilerna laddas igen expanderas texten åter till UniCode-format.

Om man garanterat vill att källkoden ska tolkas rätt, oavsett Windows språkinställningar, är det bättre att spara filerna på **UTF-8** format, som är ett kompakt lagringsformat för UniCode. Valet av lagringsformat måste tyvärr göras individuellt för varje källkodsfil och det hade ju varit trevligt om man hade kunnat ställa in UTF-8 som sitt standardformat. Men som sagt, med svenska tecken och västeuropeiska Windows-versioner är det inga problem med ANSI-formatet.

En fördel med att editor och kompilator använder UniCode, är att datatyper, variabler, metoder, etc numera kan innehålla internationella tecken (t ex 'åäö'). Här ett exempel där vi använder svenska tecken i symboler:

```
function TForm1.ProcentRäntaPåRänta(ProcentRäntesats: Double;
                                    AntalÅr         : Integer): Double;
var
  år: Integer;
  kapital: Double;
  räntefaktor: Double;
begin
  kapital:= 1;
  räntefaktor:= 1 + ProcentRäntesats/100;
  for år:= 1 to AntalÅr do
    kapital:= kapital*räntefaktor;
  Result:= 100*(kapital - 1);
end;
```

Koden ovan kan kompileras för såväl Win32 som .NET.

Undantaget är dock namn på publicerade egenskaper som fortfarande måste hålla sig till ANSI-tecken.

Char och String

En viktig skillnad mellan Object Pascal för Win32 och Object Pascal för .NET är att .NET-versionen använder även tecken (**Char**-typen) och strängar (**String**-typen) **UniCode**. Ett tecken upptar därför **2 bytes** i minnet. I Object Pascal för Win32 är det dock fortfarande 1 byte per tecken i Char och String. Normalt är det inga problem med denna skillnad, såvida man inte direkt utnyttjar UniCode-tecken i .NET och sen vill kompilera samma källkod för Win32, eller om man i ett program förutsätter att tecken lagras i t ex 1 byte. I det senare fallet är det bättre att avläsa teckenstorleken med SizeOf(Char), istället för att hårdkoda ett visst antal bytes/tecken.

Namndomäner/Name spaces (.NET)

Delphi är ett modulärt språk, där koden vanligen placeras i separatkompilerade **enheter** (**units**). Ibland händer det att två symboler i olika enheter råkar få samma namn (enheterna är kanske skapade av olika programmerare

och ovetande om varandra). Borlands Delphi-enhet **SysUtils** deklarerar t ex en procedur "Beep". I **Windows**-enheten, baserad på Microsoft WinAPI, finns emellertid en funktion med samma namn, "Beep". Borland-versionen är en procedur medan Microsoft-versionen är en funktion. Dessutom saknar Beep i SysUtils helt parametrar, medan Beep i Windows har två parametrar. Upplagt för bråk!

I Object Pascal gäller att om inget annat sägs, används den symbol som finns i den sist deklarerade enheten i användarens uses-lista. Så om deklarationen ser ut så här:

```
uses
  Windows, SysUtils,...;
...
  Beep;
```

så kompileras ett anrop av Beep i SysUtils.

Om vi behöver komma åt Beep i Windows, kan vi med punktnotation **kvalificera** Beep med enhetsnamnet:

```
if Windows.Beep(1000,2) then ...
```

Vi kan alternativt markera att det är Beep i SysUtils vi vill använda, genom att skriva

```
SysUtils.Beep;
```

Observera att enhetsnamnet (t ex unit SysUtils) måste vara detsamma som enhetens filnamn (SysUtils.pas, SysUtils.dcu, SysUtils.dcuil)!

.NET använder istället en friare gruppering av symboler som kallas för **namndomäner** (name spaces, namnrymder). Syftet är också här bl a att undvika namnkrockar mellan symboler som råkar ha samma namn. Namndomäner används också för att markera vem som har skapat koden och till vilket programbibliotek koden hör.

De mest grundläggande namndomänerna i .NET har Microsoft givit namn som System, System.Data, System.Data.OracleClient, System.Windows.Forms, etc. En namndomän kan alltså innehålla flera delnamn, åtskilda av punkter. En konvention är att inleda namndomäner med ett företagsnamn, t ex Microsoft.JScript, Borland.Delphi, Borland.Vcl, Databiten.Vip, Databiten.Vip.Vcl, etc.

Under .NET kompileras programbibliotek till s k assemblies, som är DLL:er (eller EXE-filer om program) med ytterligare information (metadata). Det är vanligt att använda samma namn på assemblyfiler och namndomäner (som ju är ett *måste* för Delphi-enheter), t ex namndomänen Databiten.Vip.Vcl finns i assemblyfilen Databiten.Vip.Vcl.dll. Men det är inget krav! Ett assembly kan innehålla flera namndomäner och omvänt kan en namndomän vara utspridd på flera assemblyfiler.

För att Delphi 2005 ska vara bakåtkompatibelt med tidigare versioner och plattformar (Win32, Kylix för Linux) och för att inte krångla till övergången till .NET, har man valt att behålla Delphis enheter/units som byggstenar i klassbibliotek och applikationer. Att också ge stöd åt .NET-assemblies var inget större problem, eftersom de i stort sett motsvarar Delphi's **paketfiler** (BPL:er under Windows). Men för att bli en fullvärdig .NET-medlem, måste Delphi .NET också ha stöd för namndomäner.

I Delphi 8 .NET användes en provisorisk lösning, där varje enhet/unit skapade en egen namndomän.
Så bildade enheten Borland.Vcl.SysUtils, också en namndomän med samma namn. Detsamma gjorde Borland.Vcl.Classes, Databiten.Vip.Vcl.VCPCal, Databiten.Vip.Vcl.VCPEnter, etc. Det blev ganska många namndomäner i varje assembly på så sätt! Dessutom kunde inte namndomäner omfatta flera assemblies.

Delphi 2005 har en bättre lösning. En namndomän skapas av alla namndelar i ett enhetsnamn, *utom* den sista namndelen. Enheten Borland.Vcl.SysUtils hör nu till namndomänen Borland.Vcl och detsamma gäller Borland.Vcl.Classes. Såväl Databiten.Vip.Vcl.VCPCal, som Databiten.Vip.Vcl.VCPEnter, hör nu till namndomänen Databiten.Vip.Vcl.

Inget hindrar nu heller att Borland.Vcl.SysUtils och Borland.Vcl.Classes skulle kunna placeras i olika assembly-filer, de hör fortfarande till samma namndomän, Borland.Vcl.

I Delphi 2005 använder vi fortfarande "uses enhetsnamn" för att importera symboler, så den nya namndomäntekniken är inget man behöver grubbla över vid vanlig Delphi-utveckling. Det blir däremot

tfn 026-256493
mail@databiten.se
www.databiten.se

intressant om man vill skapa generella .NET-assemblies, som ska kunna användas tillsammans med andra .NET-språk, t ex C#. Den nya tekniken gör Delphi en fullvärdig medlem av .NET-familjen!

for..in..do (Win32 och .NET)

Object Pascal i Delphi 2005 innehåller för första gången på länge en ny satskonstruktion, en utveckling av klassiska **for..to..do**-satsen. Med den nya **for..in..do**-satsen kan man enkelt loopa igenom en samling element i en mängd, en array, en lista, en kollektion, databas, etc.

Syntaxen ser ut så här

```
for variabel in samling do..
```

där samling är ett uttryck av mängd-, array- eller strängtyp, alternativt en typ som implementerar gränssnittet IEnumerable eller metoden GetEnumerator. Observera att det inte behövs en loopvariabel (räknare), utan **variabel** ovan är av elementtypen i **samling**. Här följer några exempel (vi leker också med stödet för svenska tecken i symboler):

for.. in mängd..do

```
type
 TGrönsak = (bönor,kål,morötter,rädisor,ärtor);
 TGrönsaker = set of TGrönsak;

function Räkna(const grönsaker: TGrönsaker): Integer;
var
  grönsak: TGrönsak;
begin
  Result:= 0;
  for grönsak in grönsaker do Inc(Result);
end;

var
 grönsallad: TGrönsaker = [kål,rädisor,ärtor];
begin
 WriteLn(Räkna(grönsallad),' grönsaker i salladen');
 ReadLn;
end.
```

Utskrift:

```
3 grönsaker i salladen;
```

Med Delphi 2005 för .NET kan man också utnyttja .NETs stöd för att lägga ett "objektomslag" ("boxing") runt data och direkt skriva ut grönsaksnamn med **ToString**-metod i .NET-versionen av TObject:

```
...
var
 grönsak: TGrönsak;
 grönsallad: TGrönsaker = [kål,rädisor,ärtor];
begin
 for grönsak in grönsallad do
   WriteLn(TObject(grönsak).ToString,' ');
 WriteLn(Räkna(grönsallad),' grönsaker i salladen');
 ReadLn;
end.
```

Utskrift:

```
kål
rädisor
ärtor
3 grönsaker i salladen;
```


tfn 026-256493
mail@databiten.se
www.databiten.se

for.. in sträng..do

```
var
  ch: Char;
  s: String;
..
s:= 'Delphi 2005';
for ch in s do
  Write(ch);
```

for.. in array..do

```
type
  TRGB = (red,green,blue);

var
  AArr: array[0..4] of String;
  BArr: array[-3..3] of Double;
  CArr: array[1..4,1..3] of Integer;
  DArr: array[TRGB] of Byte;
  EArr: array of String;

  a: String;
  b: Double;
  c: Integer;
  d: Byte;
  e: String;
...
  for a in AArr do
    WriteLn(a);
  for b in BArr do //index startar inte med 0
    WriteLn(b);
  for c in CArr do  //flerdimensionell array
    WriteLn(c);
  for d in DArr do //index är uppräknad typ
    WriteLn(d);
  SetLength(EArr,10); //dynamisk array
  for e in EArr do
    WriteLn(e);
```

for.. in klass med GetEnumerator-metod..do

for..in..do-satsen kan också användas på alla klasser (och härledda klasser) som implementerar antingen gränssnittet **IEnumerable** eller har en metod **GetEnumerator** som följer en viss standard. I VCL (Win32 och .NET) finns många sådana klasser. Några exempel är **TList**, **TInterfaceList**, **TCollection**, **TStrings**, **TComponent**, **TFields**, **TActnList**, **TMenuItem**, m fl. Exempel:

```
var
  sl: TStringList;
  s: String;
...
for s in sl do
  WriteLn(s);
```

Där behov finns, kan man också bygga in stöd för for..in..do i egna klasser där man behöver loopa igenom datasamlingar.

Förseglade och abstrakta klasser (Win32 och .NET)

Klasser kan nu deklareras som **sealed**, vilket gör det omöjligt att härleda nya klasser från en förseglad klass. En klass kan också deklareras som **abstract**, vilket innebär att man inte kan skapa instanser av klassen.

Nya sektioner (Win32 och .NET)

Klasser och poster har stöd för nya, "**strikta**" sektioner som deklareras enligt **strict private** och **strict protected**. Detta är i enlighet med .NET-standard och innebär att endast medlemmar i *samma* klass/post kan nå **strict**-deklarerade medlemmar. Utan **strict** kan också andra klasser/poster i samma enhet/unit använda medlemmarna.

Nästlade deklarationer (Win32 och .NET)

Poster och klasser kan nu innehålla interna konstant- och typdeklarationer, vilket gör att man bättre kan kapsla in interna delar i poster och klasser. Om dessa deklareras som private eller strict private kan man inte komma åt dem utifrån.

Poster med metoder, egenskaper, sektioner mm (.NET)

En annan nyhet i .NET-versionen är att **poster** (record-typer) kan innehålla **metoder**, **egenskaper** och **sektioner** (private, public, etc). Poster allokeras på stacken, till skillnad mot objektinstanser som allokeras på heapen. Poster är därför användbara där prestanda är av högsta prioritet. .NET-poster kan (men behöver inte) ha en konstruktor, som då används för att initiera data i posten. En nackdel jämfört med klasser är att posttyper inte kan ärvas. Dock kan posttyper användas för att implementera **gränssnitt** (interface).

Koden nedan illustrerar användningen av nya strict-sektioner, nästlade deklarationer samt poster med sektioner, metoder och egenskaper:

```
type
  TDator = class
  strict private
   type //nästlade typdeklarationer
     TCPUTillverkare = (AMD,Intel,TransMeta);

     TCPU = record //postdeklaration
     private //poster kan ha sektioner
      fTillverkare: TCPUTillverkare;
      fMHz: Single;
     public
      function Tillverkare: String; //poster (.NET) kan ha metoder och egenskaper
      property MHz: Single read fMHz;
     end; //TCPU

     TGränssnitt = (IDE,S_ATA,SCSI,USB2,FireWire);

     TDisk = class
     strict private
      fGränssnitt: TGränssnitt;
      fGBKapacitet: Integer;
     public
      constructor Create;
      function GBLedigt: Integer;
      property GBKapacitet: Integer read fGBKapacitet;
     end; //TDisk
     //slut nästlade deklarationer
   strict private
    fMBRAM: Integer;
    CPU: TCPU;
    Disk: TDisk;
   public
    constructor Create;
    property MBRAM: Integer read fMBRAM;
    function MHz: Single;
    function GBDiskLedigt: Integer;
  end;

function TDator.TCPU.Tillverkare: String; //nästlad postmetod
begin
 Result:= TOBject(fTillverkare).ToString; //.NET boxing
end;

constructor TDator.TDisk.Create; //nästlad konstruktor
begin
 inherited;
 fGränssnitt:= USB2; fGBKapacitet:= 250;
end;

function TDator.TDisk.GBLedigt: Integer; //nästlad metod
begin
  Result:= 100;
end;
```


tfn 026-256493
mail@databiten.se
www.databiten.se

```
constructor TDator.Create;
begin
 inherited;
 fMBRAM:= 512;
 //Poster ligger på stacken och instanser behöver inte allokeras dynamiskt
  with CPU do begin
   fTillverkare:= Intel; fMHz:= 2.8;
 end;
 //Objekt ligger på heapen och minne allokeras via konstruktor
 Disk:= TDisk.Create;
end;

function TDator.GBDiskLedigt: Integer;
begin
 Result:= Disk.GBLedigt;
end;

function TDator.MHz: Single;
begin
 Result:= CPU.MHz;
end;

var
 Dator: TDator;
begin
 Dator:= TDator.Create;
 WriteLn('MB RAM: ',Dator.MBRAM);
 WriteLn('MHz CPU: ',Dator.MHz:0:1);
 WriteLn('GB ledigt på disken: ',Dator.GBDiskLedigt);
 ReadLn;
end.
```

Klasskonstanter (Win32 och .NET)
Klassvariabler (.NET), **klasskonstruktorer** (.NET) och **klassegenskaper** (.NET)

Klassmetoder (statiska metoder) i tidigare versioner, har i Delphi 2005 kompletterats med **klasskonstanter**, **klassvariabler** (.NET), **klasskonstruktorer** (.NET) och **klassegenskaper** (.NET). Sådana klassmedlemmar är gemensamma för hela klassen och alla eventuella instanser. De kan direkt användas utan att man först behöver skapa en instans av klassen, men kan också användas via instansobjekt. Ett exempel:

```
type
  TKlass = class
    strict private
    const
     Version = '1.0';
    class var
     KlassData: String;
    public
     class constructor Create;
     class procedure KlassMetod;
     class property KlassEgenskap: String
           write KlassData;
    strict private
     var
      InstansData: String;  //fält
    public
     constructor Create(InstanceData0: String);
     procedure InstansMetod;
     property InstansEgenskap: String write InstansData;
  end;

class constructor TKlass.Create;
begin
 KlassData:= 'Version ' + Version;
end;

class procedure TKlass.KlassMetod;
begin
 WriteLn('KlassMetod:');
 WriteLn(KlassData);
 WriteLn('Slut KlassMetod ---------------');
end;

constructor TKlass.Create(InstanceData0: String);
begin
 InstansData:= InstanceData0;
 inherited Create;
end;
```


tfn 026-256493
mail@databiten.se
www.databiten.se

```
procedure TKlass.InstansMetod;
begin
 WriteLn('InstansMetod:');
 WriteLn(InstansData);
 WriteLn('InstansMetod anropar klassmetod...');
 KlassMetod;
 WriteLn('Slut InstansMetod =============');
end;

var
 Instans1,
 Instans2: TKlass;
begin
 //Anrop av klassmetoder och egenskaper - ingen instans behövs
 TKlass.KlassMetod;
 TKlass.KlassEgenskap:= 'Modifierad version 2';
 TKlass.KlassMetod;

//Anrop av vanliga metoder och egenskaper via instanser
 Instans1:= TKlass.Create('Instans1');
 Instans2:= TKlass.Create('Instans2');
 Instans1.InstansMetod;
 Instans2.InstansMetod;
 Instans1.InstansEgenskap:= 'Modifierad Instans1';
 Instans1.InstansMetod;

//Anrop av klassegenskap och klassmetod via instanser
 Instans1.KlassEgenskap:= 'Modifierad version 3';
 Instans2.KlassMetod;

 ReadLn;
end.
```

Operator overloading (.NET)

Nu är det möjligt att använda standardoperatorer som `'+', '-', '*', '/'`, etc. även tillsammans med egna datatyper och ge operatorerna betydelser som är avpassade för datatypen. Ett exempel på när detta skulle vara användbart, är komplexa tal, som kan definieras som poster med ett reellt och ett imaginärt fält. Visst vore det trevligt att kunna skriva komplexa uttryck med användning av vanliga operatorer? Inga problem – så här kan man göra i Delphi 2005:

```
uses
  Complex;

var
  a,b: TComplex;
...
  a.Real:= 1; a.Imaginary:= 2;
  b.Real:= 2; b.Imaginary:= 5;
  WriteLn(a + b);
```

Utskrift:
3 + 7i

Ett annat exempel är bråkräkning, som av en händelse råkar ingå DATABITENs **Visual Plus 2005 Toolbox** till Delphi 2005. Så här kan det se ut:

```
uses
  Databiten.VipVcl.VPFractions;

var
  f1,f2: Fraction;
...
  f1:= '4/12'; //omvandlas automatiskt till ett internt format
  f2:= '4/9';
  WriteLn(f1 + f2);
  WriteLn(f1 - f2);
  WriteLn(f1*f2);
  WriteLn(f1/f2);
  WriteLn(f1/f2+f2/f1);
```

Utskrift:
7/9
-1/9
4/27
3/4
2 1/12

tfn 026-256493
mail@databiten.se
www.databiten.se

Andra användningsområden är att definiera operatorer för vektorer och matriser, operatorer för filtrering av data (+ = union, /=snitt, etc), m m.

Class helpers (.NET)

Med "class helpers" kan man, utan att modifiera eller härleda från en befintlig klass, **förstärka** en klass så att det ser ut som den har ytterligare metoder och egenskaper. Class helpers används i Delphi .NET för att knyta ihop Delphis traditionella klasstruktur (som startar med TObject), med Microsofts .NETs klasstruktur, som startar med System.Object.

Istället för att låta Delphi 2005 inrymma två helt skilda klasshierarkier (som skulle vara ganska förvirrande), låter man Delphis TObject i grunden vara identisk med .NET System.Object. Så här deklareras TObject under .NET:

```
type
  TObject = System.Object;
```

Med hjälp av en class helper (TObjectHelper) förstärker man sedan System.Object med de metoder som vi förväntar oss från tidigare versioner av Delphis TObject.

```
type
  TObjectHelper = class helper for TObject
  public
    procedure Free;
    function ClassType: TClass;
    class function ClassName: string;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: System.Type;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const AName: string): TMethodCode;
    class function MethodName(ACode: TMethodCode): string;
    function FieldAddress(const AName: string): TObject;
    procedure Dispatch(var Message);
  end;
```

På samma sätt förfar man med vissa andra grundklasser som TComponent = System.ComponentModel.Component kompletterat med TComponentHelper, etc.
Borland har faktiskt tagit patent på class helper-tekniken!

Om inte Borland uppfunnit class helpers, hade det naturliga lösningen varit att härleda TObject från System.Object så här:

```
type
  TObject = class(System.Object)
    procedure Free;
    function ClassType: TClass;
    class function ClassName: string;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: System.Type;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const AName: string): TMethodCode;
    class function MethodName(ACode: TMethodCode): string;
    function FieldAddress(const AName: string): TObject;
    procedure Dispatch(var Message);
  end;
```

Men problemet med denna modell är att man skulle skapa en helt ny gren i .NET-klasssträdet, där alla klasser (härledda från TObject) i VCL-grenen, skulle vara helt inkompatibla med standard FCL-klasser i System.Object-grenen. Med class helpers är de två grenarna i hög grad kompatibla.

Det kanske kliar i fingrarna att använda class helpers till både det ena och det andra, men Borland avråder bestämt från lössläppthet på denna punkt – class helpers ska bara användas för att knyta ihop en extern klasshierarki (läs .NET FCL) med Delphis traditionella klasser. Man har t om hotat stänga av stödet för class helpers, utom för internt bruk, om tekniken skulle börja missbrukas!

En trevlig bieffekt av den nära kopplingen mellan grundklasserna i VCL och FCL, är att TObject i VCL .NET också har stöd för alla System.Object-metoder, t ex **ToString**-metoden som översätter objektdata till strängar. Eftersom .NET också har stöd för s k "boxing", dvs kan "slå in" alla datatyper (Boolean, Integer, Double, uppräkningstyper, etc) i "objektpaket", så kan man därför skriva kod i stil med:

```
type
  TRGB = (röd,grön,blå);
var
  i: Integer = 5;
  b: Boolean = True;
  Color: TRGB = grön;
begin
  WriteLn(i.ToString);
  WriteLn(b.ToString);
  Writeln(TObject(Color).ToString); //egen typ måste explicit typomtolkas först
```

Utskrift:
5
True
grön

Inlining av metoder (Win32, .NET)

Det är nu möjligt att snabba upp tidskritiska metodanrop genom att **inline**-deklarera dem. Istället för att kompilatorn lägger in ett anrop (call) till en metod, "kopieras" metodkoden direkt till anropsstället och man slipper därmed tidsödande anrop (kopiera parametrar till stacken, återgång, etc.).

```
procedure Tidskritisk(i: Integer); inline;
begin
...
end;

...
for i:= 1 to 1000 do
  Tidskritisk(i); //koden i Tidskritisk kopieras hit
```

I verkligheten är det inte riktigt så enkelt – även om en procedur är inline-deklarerad , så gör kompilatorn en egen bedömning om koden ska kopieras eller anropas. Valet beror bl a av tillgången på lediga CPU-register, kodstorlek, m m. Inline-deklarationer ska ses som en vink om vad man *eftersträvar*, inte vad man *kräver*.

Inlining lämpar sig enbart för korta, tidskritiska procedur/metodanrop och man bör alltid testa under verklig körning om inlining verkligen ger någon tidsvinst eller ej.

Fler nyheter

Det finns ytterligare nyheter i .NET, som stöd för **dynamisk allokerade flerdimensionella arrayer**, **custom attributes**, m m, som vi dock inte går in på i denna artikel.

Slutomdöme

Delphi 2005 har nu stöd för två kompletta programmeringsspråk – **C#** för .NET-utveckling och **Object Pascal** för såväl Win32 som .NET-utveckling. Object Pascal har vidareutvecklats så att det idag är ett komplett .NET-språk och en hel del av de nyheter som introducerats för att ge stöd åt .NET, har också lyfts över i Win32-versionen av Object Pascal. Helhetsomdömet är att Delphis Object Pascal är ett mycket kompetent val vid utveckling av komponenter och applikationer för såväl Win32 som .NET. Tillgång till C# i samma utvecklingsmiljö, gör inte saken sämre!