HANSER

Borland Delphi 2005

Walter Doberenz, Thomas Gewinnus

Microsoft .NET Framework-Entwicklung

ISBN 3-446-40202-0

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/3-446-40202-0 sowie im Buchhandel

2.2 Die wichtigsten Neuheiten in Delphi .NET

Der Übergang von Delphi 7 nach Delphi 8 Ende des Jahres 2003 markierte den bisher größten Sprung in der Geschichte von Delphi, vergleichbar nur mit dem Übergang von Object Pascal nach Delphi 1 vor mehr als zehn Jahren[1]. Zahlreiche gravierende Änderungen waren notwendig, um maximale Kompatibilität mit der darunter liegenden .NET-Architektur zu erreichen. Damit wurde gleichzeitig auch Kompatibilität zu den anderen .NET-Programmiersprachen, wie VB .NET oder C#, erreicht und es können Assemblies genutzt werden, die in anderen .NET-Sprachen geschrieben wurden.

Im vorliegenden Abschnitt wollen wir den vom alten Win32-Delphi kommenden Umsteiger mit den wichtigsten sprachlichen Änderungen vertraut machen.

2.2.1 Bezeichner (Identifiers)

Damit die verschiedenen .NET-Sprachen konfliktfrei mit der CLR zusammenarbeiten können, muss vermieden werden, dass reservierte Wörter miteinander kollidieren. Delphi .NET führt dazu ein spezielles Symbol (*&*) ein, welches Sie als Präfix für ein reserviertes Wort benutzen können, um es dann ganz legitim als Bezeichner zu verwenden (*Qualified Identifier*).

Genauer gesagt erlaubt Ihnen das *&* den Zugriff auf CLR-Symbole (oder auf andere definierte Symbole), aber es verbietet Ihnen die Deklaration von Bezeichnern in Ihrem Code, die die eigenen Syntaxregeln von Delphi verletzen würden.

Beispiel: Der Gebrauch des *Label*-Steuerelements der *Windows Forms*-Bibliothek. Weil das Wort *Label* zu den reservierten Delphi-Wörtern gehört, können Sie entweder den vollen Namespace oder aber nur ein *&* voranstellen. Die folgenden zwei Deklarationen sind deshalb identisch:

```
Label1: System.Windows.Forms.Label;
Label1: &Label;
```

2.2.2 Was passierte mit den Units?

Units sind nach wie vor die Grundbausteine eines Delphi-Programms. Unter .NET scheinen sie allerdings keinen rechten Sinn mehr zu ergeben, denn ab sofort ist sämtlicher Code in Klassen gekapselt. Um zu verstehen, wie Borland dieses Problem gelöst hat, sei ein kurzer Vergleich mit C# – der wichtigsten .NET-Programmiersprache – gestattet[2].

---

[1] Die Unterschiede zwischen Delphi 8 und Delphi 2005 für .NET sind demgegenüber vergleichsweise gering.

2 Die meisten neuen Delphi-Implementierungen folgen denen von C#.

Delphi versus C#

Als einer der wichtigsten Unterschiede zwischen Delphi und C# gilt, dass C# eine pure OOP-Sprache ist, d.h., der gesamte Code ist in Methoden verpackt, was globale Funktionen/Prozeduren praktisch ausschließt. Delphi aber ermöglicht (wie z.B. auch C++) beide Programmier-Paradigmen.

Die Frage, ob eine pure OOP-Sprache grundsätzlich besser ist als reines prozedurales Programmieren, würde zu endlosen Diskussionen um des Kaisers Bart führen. Deshalb hier nur so viel: In jeder OOP-Sprache findet man bestimmt auch Bibliotheken mit Klassen, die nicht instanziiert werden, sondern nur zum Benutzen von Klassenfunktionen bestimmt sind. In diesem Fall sind die Klassen lediglich Container von globalen Routinen, machen also auch nichts anderes als Module in fortgeschrittenen prozeduralen Sprachen.

Units mutieren zu Klassen

Beim Überführen von Delphi nach .NET musste Borland einen Weg finden, um das existierende Unit-Modell mit seinen globalen Variablen/Funktionen/Prozeduren zu unterstützen. Erreicht wurde dies dadurch, dass im Hintergrund eine Klasse für jede Unit erzeugt wird, wobei die globalen Member als Klassenvariablen bzw. Klassenmethoden in Erscheinung treten.

Sie als Programmierer merken davon nichts, Sie verwenden weiter Ihre Units und überlassen das Generieren der entsprechenden Klasse dem Compiler.

Hinweis: Um sich davon zu überzeugen, dass aus jeder Unit tatsächlich eine Klasse geworden ist, können Sie Ihre Assemblies mit einem Tool wie z.B. dem MSIL-Disassembler (*Ildasm.exe*) untersuchen, den Sie zusammen mit zahlreichen anderen Tools im Verzeichnis *c:\Programme\Microsoft.NET\SDK\v1.1\Bin* finden.

Units als Namespaces

Units werden unter Delphi.NET auch als Namespaces benutzt. Eine mit Delphi compilierte Assembly hat so viele Namespaces wie sie Units im Quellcode verwendet. Neu ist auch, dass Units lange Namen haben können, die die Dot-Notation nutzen.

Beispiel:

```
uses System.IO.Text, System.Web;

uses DoKo.Test;
```

Hinweis: Achten Sie auch darauf, dass den klassischen Delphi Win32 Bibliotheken-Units wie (z.B. *SysUtils*, *Classes*) der Präfix *Borland.Vcl* voranzustellen ist.

Beispiel: Statt

```
uses Classes, SysUtils;
```

gilt

```
uses Borland.Vcl.Classes, Borland.Vcl.SysUtils;
```

Achten Sie auch auf folgende Verwechslungsmöglichkeit:

Hinweis: Assembly-Dateinamen sehen oft aus wie Namespace-Namen, aber die Dateinamen müssen nicht notwendigerweise den/die Namensraum/Namensräume wiedergeben, zu denen die Assembly gehört. *System.Windows.Forms.dll* gehört z.B. zu anderen Namensräumen als zu System.*Windows.Forms*.

Unit Initialisierung

Was wird aus den *initalization* und *finalization* Sektionen einer Unit, welche (bei Delphi für Win32) globalen Code zu Beginn und Ende eines Programms ausführen?

Weil sich in Delphi für .NET Units in Klassen verwandeln, wird der Initialisierungscode zu einer statischen Methode der Klasse, die vom Konstruktor aufgerufen wird. Da die Konstruktoren einer Klasse automatisch durch die CLR angestoßen werden, bevor jede Klasse verwendet wird, ist das resultierende Verhalten ähnlich dem gewohnten.

Einziger wichtiger Unterschied ist, dass es keine bestimmte Reihenfolge der Ausführung der Klassen-Konstruktoren eines Programms mehr gibt. Das bedeutet, dass die Reihenfolge der Ausführung der verschiedenen *initialization*-Sektionen unbekannt ist und sich während der Programmausführung ändern kann.

2.2.3 Grundlegende Datentypen

Musste man früher die Delphi-Datentypen mitunter an die darunter liegende CPU anpassen (z.B. bei Gleitkommazahlen), muss die Sprache nun mit der *Common Type System* (CTS) Spezifikation der CLR klarkommen. Natürlich sind die meisten der Basistypen noch vorhanden, andere hat Borland auf Basis der verfügbaren nachgebaut.

Die CLR trifft eine klare Unterscheidung zwischen zwei unterschiedlichen Familien von Datentypen:

- **Wertetypen** (*Value Types*)
  Diese liegen direkt auf dem Stack. Wenn Sie einen Wertetyp kopieren oder zuweisen wird vom System eine komplette Kopie dieses Wertes angelegt. Wertetypen sind alle primitiven Datentypen (Integer und Gleitkommazahlen, Zeichen, boole'sche Werte) und Records.
- **Verweistypen** (*Reference Types*)
  Diese liegen auf dem Heap und unterliegen dem Garbage Collector. Dazu gehören alle anderen Typen wie Objekte, Strings, dynamische Arrays ...

Man könnte natürlich argwöhnen, dass eine pure OOP-Sprache den Gebrauch von primitiven Typen ausschließen und generell für alles Objekte verwenden müsste[1]. Es sind aber vor allem Effizienzgründe, die fordern, primitive Typen systemnah zu verwalten.

Primitive Datentypen

Die folgende Tabelle zeigt eine Zusammenstellung der "primitiven" CLR-Typen, die nicht unmittelbar Objekten zugeordnet werden können. Zwecks Vergleich werden auch die äquivalenten C#-Datentypen mit angegeben:

Delphi.NET-Datentyp	**C#.NET-Datentyp**	**.NET-CLR-Typ**
Byte	*byte*	*System.Byte*
ShortInt		*System.SByte*
SmallInt	*short*	*System.Int16*
Integer	*int*	*System.Int32*
Word	*ushort*	*System.UInt16*
LongWord / Cardinal	*uint*	*System.UInt32*
Int64 / Comp	*long*	*System.Int64*
Single	*float*	*System.Single*
Double / Real	*double*	*System.Double*
Decimal	*decimal*	*System.Decimal*
Char / WideChar	*char*	*System.Char*
Boolean	*bool*	*System.Boolean*
Extended		*Borland.Delphi.System.Extended*
Currency		*Borland.Delphi.System.Currency*
ByteBool		*Borland.Delphi.System.ByteBool*
WordBool		*Borland.Delphi.System.WordBool*
LongBool		*Borland.Delphi.System.LongBool*

Hinweis: Der alte *Real48* Typ (6 Byte Gleitkommazahl) ist nicht mehr verfügbar. Der *Real* Type hingegen wird vom Compiler direkt *Double* zugeordnet.

Wie Sie sehen, haben die letzten fünf in der Tabelle aufgelisteten Delphi-Typen kein CLR-Äquivalent. Wir empfehlen Ihnen daher, diese Typen nicht zu verwenden, wenn sie wollen, dass Ihre Klassen auch von anderen .NET-Sprachen benutzt werden sollen. Nichts spricht

[1] Wie z.B. SmallTalk, die "Mutter aller OOP-Sprachen".

aber gegen eine Benutzung innerhalb Ihres Programms zwecks Kompatibilität mit existierendem Delphi Code!

Hinweis: Wenn immer es möglich ist, sollten Sie die Typen mit direkter CLR Zuordnung benutzen!

Boxing von primitiven Typen

Um primitive Typen und Objekte gemeinsam zu verwalten (z.B. in einer Container-Klasse) können sie in Object Wrappers "geboxt" werden. Das entspricht quasi einer expliziten Typkonvertierung nach *TObject* alias *System.Object*.

Beispiel: Boxing eines Integer-Wertes

```
var x1: Integer;
    o1: TObject;
begin
    x1 := 10;
    o1 := TObject (x1);            // Boxing bzw. explizite Typkonvertierung
```

Boxing ermöglicht es, dass sich eine Objektreferenz auch auf einen primitiven Wert beziehen kann.

Hinweis: Durch Boxing können einfache Zahlen quasi wie Objekte behandelt werden!

Beispiel: Die vordefinierte *ToString*-Methode der Basisklasse *TObject* wird auf einen Integer angewendet.

```
var  x1: Integer;
     st: String;
begin
     x1 := 10;
     st := TObject (x1).ToString;
```

2.2.4 Records

Wir kommen jetzt zu einer anderen wichtigen Familie von Wertetypen, den Records (in C# als Strukturen bezeichnet). Obwohl Records schon lange Bestandteil von Delphi sind und mit zum Pascal-Urgestein gehören, haben sie beträchtliche Erweiterungen erfahren, so können sie Methoden haben, Konstruktoren und auch Operatoren.

Record versus Klasse

Ein mit Methoden ausgestatteter Record ähnelt einer Klasse, der wichtigste Unterschied (außer dem Fehlen von Vererbung und Polymorphie) ist, dass die Record-Variablen den

lokalen Speicher nutzen (Stack) und ein "Werte-Kopier-Verhalten" (*by value*) bei Zuweisungen bzw. bei der Parameterübergabe an Funktionen zeigen.

Im Unterschied dazu werden Variablen vom Typ einer Klasse auf dem Heap (dynamischer Speicher) angelegt und zeigen bei Zuweisungen ein "Referenz-Kopier-Verhalten" (*by reference*), sie kopieren nicht das Objekt selbst, sondern nur einen Verweis (eine Adresse) zum Objekt.

Wenn Sie eine Record-Variable deklarieren können Sie diese – so wie jeden anderen Wertetyp – sofort benutzen, ohne dazu einen Konstruktor bemühen zu müssen.

Hinweis: Record-Variablen sind schlanker und effizienter als reguläre Objekte, weil sie am dynamischen Speichermanagement nicht teilhaben.

Dies ist auch der Hauptgrund dafür, dass Sie für kleine und einfache Datenstrukturen Records den Vorzug geben sollten.

Deklaration und Verwendung

Beispiel: Ein Record, wie Sie ihn jetzt in Delphi.NET deklarieren können.

```
type  TPerson = record
   private
    _name: string;  _wohnort: string;
   public
    function Adresse: String;
    constructor Create (name, wohnort: string);
end;

function TPerson.Adresse: string;
begin  Result := _name + Environment.NewLine + _wohnort  end;

constructor TPerson.Create (name, wohnort: string);
begin _name := name;  _wohnort := wohnort end;
...
var
    person: TPerson;
begin
  person := TPerson.Create('Müller', 'Berlin');
  Label1.Text := person.Adresse;
...
```

Wie Sie sehen, kann ein Record auch einen Konstruktor besitzen, aber dieser muss Parameter haben. Wenn Sie es trotzdem mit *Create()* versuchen, erhalten Sie eine Fehlermeldung.

Neue vordefinierte Records in Delphi

Wegen der komplizierter gewordenen Record-Definition und wegen der Option zur Operatorenüberladung hat Borland einige der vordefinierten Delphi-Typen in Records verwandelt (*Variant, DateTime, Currency* ...).

Insbesondere betrifft dies den *Variant*-Datentyp, denn es gibt z.B. keine Unit *VarUtils* und keinen Typ *TVarData* mehr. Delphi für .NET implementiert Varianten mittels Operatorenüberladung und ClassHelpers.

Die Implementierung von *Currency* ist ein Record mit Operatorenüberladung und Unterstützung bestimmter .NET-Interfaces wie *IConvertible*.

Varianten Records

Dies sind Datenstrukturen, deren Felder unterschiedliche Datentypen haben können in Abhängigkeit von einem gegebenen Feld.

Die wichtigste Regel für Varianten-Records in Delphi .NET ist, dass Sie im überlappenden Teil der Typdeklaration keine Verweistypen benutzen können.

Beispiel: Die folgende (unsichere) Typdeklaration funktioniert nicht, wenn *feld4* vom *String*-Datentyp ist.

```
type TVarRecord = record
      public
        feld1: Integer;
        case test: Boolean of
          True: (feld2: Integer; feld3: Char);
          False: (feld4: Integer);       // (feld4: String); => funktioniert leider nicht!
end;
```

2.2.5 Strings und andere Verweistypen

Zu dieser Kategorie gehört so ziemlich alles, was außerhalb der primitiven Datentypen noch Rang und Namen hat, also beliebige Objekte, wie sie aus dem *class*-Type instanziiert werden, aber vor allem auch Strings, Arrays und die zahlreichen Collections.

Wie bereits erwähnt, ist der Hauptunterschied zwischen Werte- und Verweistypen der, dass Referenztypen dem Heap zugewiesen werden und dem Garbage Collector unterliegen.

Strings

Der größte Unterschied zwischen Delphi für Win32- und Delphi für .NET-Strings besteht darin, dass der *String*-Typ nun ein Unicode Wide-String und kein Ansi-String mehr ist.

- Win32-Strings liegen auf dem Heap (Long Strings), werden durch einen Referenzzähler bedient und unterliegen der so genannten *copy on write*-Technik (mit jeder Änderung an einer String-Variablen wird ein neuer String erzeugt).

- .NET-Strings sind ziemlich ähnlich, benutzen aber den UTF16 Unicode Zeichensatz (jedes Zeichen wird mit 16Bit dargestellt = 2Bytes). Der Index richtet sich nach dem Zeichen und nicht nach dem Byte!

Hinweis: Ein .NET-String belegt demzufolge doppelt so viel Speicherplatz wie ein Win32-String (es sei denn, Sie benutzen den WideString-Typ von Win32).

Allgemein gilt für einen Delphi.NET-String: *String = WideString = System.String.* Im Gegensatz dazu werden ein *AnsiString* und ein *ShortString* nun als *Array of Byte* betrachtet.

Die folgende Tabelle zeigt die Zuordnung der verfügbaren Delphi-Strings zu den korrespondierenden CLR-Typen:

Delphi.NET-Datentyp	.NET-CLR-Typ
String/WideString	*System.String*
AnsiChar	*Borland.Delphi.System.AnsiChar*
ShortString	*Borland.Delphi.System.ShortString*
AnsiString	*Borland.Delphi.System.AnsiString*

Die grundsätzliche Verwendung des UTF16-Unicode Zeichensatzes unter .NET vereinfacht die Programmierung, da Sie sich nun nicht mehr um Multibyte-Zeichensätze kümmern müssen. Dennoch muss der Code bei der Übernahme auf die *Char*-Größe überprüft werden, da es sich nun nicht mehr um Ein-Byte-, sondern um Zwei-Byte-Zeichen handelt.

Hinweis: Sie können weiterhin Strings mit Ein-Byte-Zeichen verwenden, müssen diese aber nun als *AnsiString* anstatt als *String* deklarieren.

Der Compiler führt eine Konvertierung zwischen diesen Strings durch, wenn Sie eine explizite Typumwandlung verwenden oder wenn Sie die Strings implizit umwandeln, indem Sie sie einer Variablen oder einem Parameter des jeweils anderen Typs zuweisen.

Stringaddition

Ein anderer bemerkenswerte Fakt ist, dass auf Grund der *copy on write* Technik auch unter .NET das Zusammenfügen von Strings sehr umständlich ist, wenn es traditionell mit + gemacht wird (die *AppendStr*-Methode gibt es nicht mehr). Dabei muss selbst dann ein neuer String im Speicher erzeugt werden, wenn nur ein paar Zeichen zu einem String zu addieren sind. Um diese langsame Implementation zu überwinden, stellt das .NET Framework eine spezielle Klasse *StringBuilder* zum Zusammenfügen von Strings bereit.

Hinweis: Einen sehr aussagekräftigen Beweis für den Geschwindigkeitsvorteil des *StringBuilder* -Objekts finden Sie im Rezept R 6.1.

Nicht initialisierte Strings

In Delphi .NET hat ein nicht initialisierter String den Wert *nil*. Der Compiler berücksichtigt diese Tatsache, wenn Sie einen nicht initialisierten String mit einem leeren String vergleichen. Für die Code-Zeile

```
if S <> '' then ...
```

führt der Compiler beispielsweise den Vergleich durch und behandelt den nicht initialisierten String als leeren String. Bei anderen String-Operationen wird ein nicht initialisierter String nicht automatisch als leerer String behandelt (darin besteht ein Unterschied zu Code, der auf der Win32-Plattform compiliert wurde). Dies kann zu einer Laufzeit-Exception aufgrund eines leeren Objekts führen.

ToString-Methode

Und zum Abschluss unserer Betrachtungen zum *String*-Datentyp soll ein wichtiges und nützliches Feature nicht vergessen werden:

Hinweis: In der .NET FCL (*Framework Class Library*) – und als Konsequenz auch in Delphis RTL – hat jedes Objekt eine String-Repräsentation, verfügbar durch Aufruf der virtuellen *ToString*-Methode.

Beispiel: Anzeige einer Zahl

```
var i: Integer;
begin
  i := 5;
  Label1.Text := i.ToString;
```

2.2.6 Benutzen des Typs PChar

In Delphi .NET ist die Benutzung von Pointern und anderen unsicheren Typen zwar verpönt, aber immer noch möglich, Sie brauchen einfach nur den Code als *unsafe* zu markieren.

Nachdem Sie den Compilerschalter

```
{$UNSAFECODE ON}
```

gesetzt haben, können Sie globale Routinen oder Methoden mit der *unsafe* Direktive markieren. Ihre Applikation ist dann zwar eine legale .NET Anwendung, allerdings keine sichere.

Wenn Sie den *PChar*-Typ anwenden wollen, erhalten Sie eine Compilerwarnung "Unsicherer Code ist nur in unsicheren Prozeduren erlaubt.." Das Problem ist lösbar, wenn Sie die *unsafe*-Direktive einsetzen, um damit die betreffende Prozedur zu markieren.

Beispiel: Der Inhalt einer TextBox wird zeichenweise auf Großbuchstaben getestet.

```
{$UNSAFECODE ON}
function isUpper(c: Char): Boolean;
begin
  Result := c in ['A'..'Z', 'Ä', 'Ö', 'Ü']
end;

procedure pCharGB(p: PChar); unsafe;
begin
  while (p^ <> #0) do begin
    if isUpper(p^) then MessageBox.Show(p^ + ' ist ein Großbuchstabe!');
    Inc(p)
  end
end;

procedure TWinForm1.Button1_Click(sender: System.Object; e: System.EventArgs); unsafe;
var ca: array of Char;
begin
  ca := TextBox1.Text.ToCharArray;
  pCharGB(@ca[0])
end;
```

Obiger Code funktioniert zwar, aber es ist schwer zu verstehen, warum man stattdessen nicht eine einfachere und pointerfreie Lösung wählt.

Daraus ergibt sich die generelle Frage, ob man den *PChar*-Typ überhaupt noch benötigt, denn sein Hauptanwendungsgebiet – die Parameterübergabe beim Direktzugriff auf die Windows-API – hat unter .NET so gut wie keine Bedeutung mehr.

Hinweis: Beachten Sie, dass viele RTL-Funktionen, die den Typ *PChar* unterstützt haben, aus der RTL entfernt wurden. Sie müssen diese Funktionen nun durch die entsprechenden Versionen vom *String*-Typ ersetzen.

2.2.7 Sichere Typkonvertierung

Da die .NET CLR höhere Anforderungen an die Typsicherheit stellt, wurden einige der in Delphi bislang möglichen direkten Typumwandlungen abgeschafft bzw. geändert.

Casting eines Objekts in einen anderen Objekttyp

Nach wie vor ist ein direktes Typecasting möglich:

```
ObjectA := TClassA (ObjectB);
```

Die .NET-konforme Regel ist aber, dass jedes Objekt auf Kompatibilität zu jedem seiner Basisklassentypen überprüft wird. Das ist zwar langsamer als eine direkte Typumwandlung, dafür aber sicherer:

```
ObjectA := ObjectB as TClassA;
```

Anders als bei Delphi für Win32 besteht in Delphi .NET also kein Unterschied zwischen einer expliziten Typumwandlung und dem Operator *as*. In beiden Fällen ist die Umwandlung nur dann erfolgreich, wenn die zu konvertierende Variable eine Instanz des Typs ist, in den sie umgewandelt werden soll. Dies bedeutet, dass Code, der vorher problemlos ausgeführt wurde (durch Umwandlungen zwischen nicht kompatiblen Datentypen), nun unter Umständen eine Laufzeit-Exception generiert.

Casting primitiver Datentypen

Etwas anders verhält es sich bei den primitiven Datentypen (z.B. Integer), die wir in einen *TObject*-Typ verpacken wollen. Hier landen wir – wie bereits beschrieben – beim Boxing des nativen Wertes:

```
ObjectA := TObject (aNr);
```

So erlangen Sie ein neues Objekt, welches den Wert der Umwandlung kapselt.

Die Rückumwandlung in den Originalwert:

```
aNr := Integer (ObjectA);
```

2.2.8 Neuigkeiten bei den Klassen

Obwohl das CLR-Klassenmodell voll unterstützt wird, bleiben die meisten der traditionellen Delphi-Klassen-Features unverändert. Wichtige Neuerungen betreffen z.B. die Sichtbarkeitsregeln, Änderungen bei statischen Mitgliedern, das Konzept der ClassHelpers und die Garbage Collection. Wir aber wollen bei einem scheinbar belanglosen Feature beginnen:

Geerbter Konstruktor muss aufgerufen werden

Delphi war bis jetzt eine der wenigen OOP-Programmiersprachen, die nicht verlangten, dass zunächst die Basisklasse im Konstruktor der erbenden Klasse initialisiert werden muss. Dadurch wurden Tür und Tor für gemeine Fehler und schwammiges Verhalten geöffnet.

Hinweis: Delphi .NET verlangt, dass Sie im Konstruktor der Subklasse den Konstruktor der Basisklasse aufrufen und zwar **bevor** Sie irgendein Feld oder irgendeine Methode der Basisklasse benutzen.

Beispiel: Der folgende triviale Code wird nicht mehr compiliert (Fehlermeldung: *'Self' ist nicht initialisiert. Ein geerbter Konstruktor muss aufgerufen werden*):

```
type TTestclass = class
private
   f1: Integer;
public
   constructor Create;
end;
constructor TTestclass.Create;
begin
  f1 := 10;
end;
```

TTestClass erbt von *TObject* und ist somit an dieselbe Regel wie auch jede andere Klasse gebunden (obwohl der *TObject*-Konstruktor ja eigentlich nutzlos ist), also:

```
constructor TTestclass.Create;
begin
  f1 := 10;
  inherited Create;
end;
```

Bevorzugen sollten Sie aber die folgende sichere Reihenfolge:

```
constructor TTestclass.Create;
begin
  inherited Create;
  f1 := 10;
end;
```

Strengere Sichtbarkeitsregeln

Im Unterschied zu den meisten anderen OOP-Sprachen hat das "alte" Delphi eine ziemlich lockere private Sichtbarkeit. Diese wirkt nur zwischen Units, nicht aber zwischen Klassen innerhalb derselben Unit. Eine Klasse konnte also auf die privaten Felder bzw. Methoden einer anderen Klasse derselben Unit zugreifen. Gleiches galt für die Protected Members.

Um mit der CLR kompatibel zu sein, hat Borland zwei weitere Zugriffsspezifizierer hinzugefügt: *strict private* und *strict protected*. Diese verhalten sich erwartungsgemäß im Sinne der CLR: Andere Klassen innerhalb derselben Unit können nicht darauf zugreifen. Der Zugriff auf *strict protected* Symbole ist nur bei abgeleiteten Klassen möglich.

Die folgende Tabelle zeigt die Zuordnung der Sichtbarkeitsattribute:

Delphi.NET	**CLR**
private	*assembly*
strict private	*private*
protected	*family/assembly*

Delphi.NET	CLR
strict protected	*family*
public	*public*
published	*public*

Protected-Hacking funktioniert noch!

In Delphi .NET hat *protected* seine Bedeutung beibehalten, nämlich dass die damit deklarierten Elemente nur innerhalb der Klassendeklaration und in allen davon abgeleiteten Klassen sichtbar sind. Mit dieser Sichtbarkeit werden also Elemente deklariert, die nur in den Implementierungen abgeleiteter Klassen verwendet werden sollen.

Gestandene Delphianer wissen aber von einem Trick, wie man auf protected Daten einer anderen Klasse zugreifen kann, selbst wenn sich diese Klasse in einer anderen Unit befindet.

Die Kernidee des "protected hacking" ist die Fähigkeit, eine andere Klasse in eine vorgetäuschte Subklasse zu casten. Wenn auch die CLR das nicht erlaubt, so ignoriert der Compiler dies und lässt Sie trotzdem auf das protected Member zugreifen.

Beispiel: In einer Klasse *TKunde* gibt es ein geschütztes Feld *Kontostand*:

```
type  TKunde = class
  protected
    Kontostand: Currency;
  public
    Nachname: String;
 end;
```

Das Hauptformular benutzt *Protected Hacking* wie folgt für den Zugriff :

```
type THackKunde = class(TKunde);            // Referenz auf abgeleitete Klasse!

procedure TWinForm1.Button2_Click(sender: System.Object; e: System.EventArgs);
var ks: Decimal;
    kunde: TKunde;
begin
  kunde := TKunde.Create('Müller', 10000);
  ks := THackKunde(kunde).Kontostand;         // Typecasting nach abgeleiteter Klasse!
  Label1.Text := 'Der Kontostand von ' + kunde.Nachname + ' beträgt ' + ks.ToString('c');
end;
```

Hinweis: Den kompletten Quellcode finden Sie im Rezept R 6.2!

Die einzig gute Nachricht ist, dass das *Protected Hacking* nur innerhalb einer Assembly funktioniert. Falls sich die Unit, in welcher die Basisklasse definiert ist, in einer anderen Assembly befindet, wird es nicht funktionieren.

Klassendaten

Ein weiteres neues Feature von Delphi betrifft die Einführung von Klassendaten (*class data*). Erlaubt waren bisher nur Klassenmethoden (statische Methoden, die nicht an ein bestimmtes Objekt gebunden sind, sondern an die Klasse).

Klassendaten können gemeinsam von allen Objekten der Klasse benutzt werden. Früher musste man dieses Verhalten mit globalen Variablen in der *implementation*-Sektion simulieren, dies aber kann bei der Ableitung von Klassen Schwierigkeiten bereiten.

Klassendaten werden – ähnlich wie Klassenmethoden – durch Voranstellen der Schlüsselwörter *class var* markiert.

Beispiel:

```
type TTest = class
      private
         class var anzahl: Integer;
      public
         class procedure getAnzahl: Integer;
```

Statische Klassenmethoden

Außerdem können Sie jetzt auch *class properties* und *static class procedures* deklarieren.

Statische Klassenmethoden wurden zwecks CLS-Kompatibilität eingeführt. Weil relevante Unterschiede in der Implementierung bestehen, hat sich Borland für zwei getrennte Sprach-Features entschieden.

Der Unterschied besteht darin, dass statische Klassenmethoden keine Referenz zu ihrer eigenen Klasse besitzen (kein *Self*) und nicht virtuell sein können (also nicht überschrieben werden dürfen). Positiv ist, dass sie zur Definition von Klasseneigenschaften benutzt werden können.

Beispiel: Eine Demoklasse mit verschiedenen statischen Mitgliedern.

```
type TTestStatic = class
     private
        class var _name: string;
    public
        class procedure Test1;
        class procedure Test2; static;
        class function getName: string; static;
        class procedure setName (Value: string); static;
```

```
        class property Name: string
        read getName write setName;
end;
```

2.2.9 Class Helpers

Als Borland mit der Arbeit zu Delphi.NET begann, musste zunächst ein Hauptproblem gelöst werden: Wie kann man die Delphi-Basisklassen (*TObject*, *Exception, ...*) mit den korrespondierenden Klassen des .NET Frameworks in Einklang bringen?

Als trickreicher Ausweg entstanden die ClassHelpers. Ein Class Helper ist eine spezielle Klasse, die mit einer anderen Klasse verbunden wird, um dort zusätzliche Eigenschaften und Methoden – allerdings keine Daten! – verfügbar zu machen. Auf diese Weise können Sie eine Methode auf ein Objekt dieser Klasse anwenden, auch wenn diese Klasse keinerlei Ahnung von der Existenz einer solchen Methode hat.

Wie Sie sehen, erinnert dieses Konstrukt stark an die Mehrfachvererbung von Klassen, die allerdings bei OOP-Puristen nicht den besten Ruf genießt.

Beispiel: Der folgende Code deklariert eine Klasse *TKreis* und einen Helper für diese Klasse, der die Eigenschaft *Umfang* hinzufügt.

```
type  TKreis = class
  private
    _rad : Double;
  public
    constructor Create(rad : double);
    property Radius: Double read _rad write _rad;
  end;

type TKreisHelper = class Helper for CKreis
    function getUmfang: Double;
    procedure setUmfang(Value: Double);
  public
    property Umfang: Double read getUmfang write setUmfang;
  end;
```

Die Helper-Methode wird zu einer Methode der Klasse und kann wie jede andere Methode benutzt werden. Der Class Helper selbst wird nicht instanziiert!

```
  kreis1 := TKreis.Create(10);
  Label1.Text := kreis.Umfang.ToString;
```

Hinweis: Den ausführbaren und leicht modifizierten Quellcode finden Sie im Rezept R 6.3!