

Delphi-Sprache

1. Allgemeines zur Syntax

Über die Sprache

Bei der Delphi-Sprache (früher Object Pascal genannt) handelt es sich um eine höhere, objektorientierte Programmiersprache. Die wichtigsten Dinge aus dem zugrunde liegenden Pascal sind natürlich noch enthalten, allerdings hat sich die Sprache, die mit Delphi 1 eingeführt wurde, stark weiterentwickelt. Die Umbenennung von „Object Pascal Language“ in „Delphi Language“ erfolgte mit der Einführung von Delphi 7 im Jahr 2002.

Grundlegendes zur Syntax

Wie in vielen anderen Programmiersprachen werden mehrere Befehle durch Semikolon (;) getrennt. Im Gegensatz zu C/C++ unterscheidet die Delphi-Sprache nicht zwischen Groß- und Kleinschreibung. Während `meine_variable` und `Meine_Variable` in C/C++ unterschiedlich sind, sind sie für die Delphi-Sprache gleich. In dieser Hinsicht braucht der Entwickler also nicht so viel Disziplin beim Programmieren. Leerzeichen (natürlich nicht innerhalb eines Bezeichners) und Leerzeilen können verwendet werden, um die Übersichtlichkeit im Code zu erhöhen.

Das heißt natürlich nicht, dass man als Programmierer machen kann, was man will. Die Delphi-Sprache ist nämlich für ihre Typstrenge bekannt. Im Gegensatz zu Visual Basic muss eine Variable vor ihrer Verwendung in einem bestimmten Bereich deklariert werden. Ist ihr somit einmal ein Typ zugeordnet, kann sie keine Werte eines anderen Typs aufnehmen, nur Werte des gleichen Typs oder eines Untertyps. Auch Zuweisungen zwischen Variablen unterschiedlichen Typs lassen sich häufig nur über Konvertierfunktionen (wie z.B. `IntToStr`) bewerkstelligen.

2. Variablen und Konstanten

Variablen

Variablen sind jedem bekannt, der schon einmal Mathematik-Unterricht hatte. Variablen sind einfach dafür da, irgendwelche Daten (Eingaben, Berechnungsergebnisse usw.) im Arbeitsspeicher abzulegen. Über den Variablennamen kann man direkt auf den Wert zugreifen.

Hießen die Variablen in Mathe meist x oder y , sollte ein Programmierer solche Namen nicht verwenden. Sie erschweren die Lesbarkeit des Quellcodes sehr. Diese Erfahrung wird jeder schon einmal gemacht haben, der ein Programm mehrere Monate liegen gelassen hat, um es dann erneut anzugehen.



Wichtige Regel also: Variablen selbsterklärende Namen geben. Groß- und Kleinschreibung ist nicht von Bedeutung, jedoch dürfen nur Buchstaben (keine Umlaute!), Zahlen und der Unterstrich verwendet werden. Der Variablenname muss mit einem Buchstaben beginnen.

In dem Zusammenhang sollte auch auf die sog. ungarische Notation hingewiesen werden. Hierbei handelt es sich um einen Quasi-Standard. Variablennamen wird dabei eine kurze Vorsilbe hinzugefügt, die beschreibt, was für Werte darin gespeichert werden können. Im Delphi-Styleguide ist sie jedoch nicht vorgesehen.

Datentypen

Bevor eine Variable verwendet werden kann, sollte man sich darüber im Klaren sein, welche Werte sie aufnehmen soll.

Variablen sind also Platzhalter oder „Container“ für einen Wert, der Platz im Arbeitsspeicher belegt; der Datentyp ist dagegen der Bauplan, nach dem die Variable erstellt wird.

Folgendes sind die grundlegenden Datentypen in Delphi:

Typ	Wertebereich	Beispiel	
		Deklaration	Zuweisung
Integer (ganze Zahlen)	-2147483648 bis 2147483647	<code>var zahl: integer;</code>	<code>zahl := 14;</code>
Real (Gleitkommazahlen)	5.0×10^{324} bis 1.7×10^{308}	<code>var zahl: real;</code>	<code>zahl := 3.4;</code>
String (Zeichenketten)	ca. 2^{31} Zeichen	<code>var text: string;</code>	<code>text := 'Hallo Welt!';</code>
Char (Zeichen)	1 Zeichen	<code>var zeichen: char;</code>	<code>zeichen := 'a';</code>
Boolean (Wahrheitswert)	true, false	<code>var richtig: boolean;</code>	<code>richtig := true;</code>

Dies sind die Standardtypen, die generell verwendet werden können. Vor allem für Zahlen gibt es jedoch noch weitere Typen. Wenn z. B. sicher ist, dass nur ganze Zahlen von 1 bis 50 gespeichert werden sollen, so kann statt Integer auch der Typ Byte verwendet werden, der nur die Zahlen von 0 bis 255 aufnehmen kann. Das Gleiche gilt für reelle Zahlen. Die weiteren Typen sind unter „Reelle Typen“ bzw. „Integer-Typen“ in der Hilfe aufgeführt. Bei Strings gibt es einige grundlegende Unterschiede, weshalb Sie darüber einen extra Abschnitt finden.

Deklaration

Man kann eine Variable nicht einfach verwenden. Vorher muss sie dem Compiler bekannt gemacht werden, damit er beim Kompilieren entsprechende Typprüfungen durchführen kann. Diese Bekanntmachung nennt man Deklaration. Vor einer Deklaration wird das reservierte Wort `var` geschrieben. Als erstes kommt der Name der Variablen, hinter einem Doppelpunkt der Typ. Mehrere Variablennamen vom gleichen Typ können in einer Zeile, durch Kommata getrennt, stehen.

Beispiel:

```
var zahl1, zahl2, zahl3: integer;
    ergebnis: real;
    text, eingabe: string;
```

An folgenden Stellen im Code ist das möglich, je nach Gültigkeitsbereich:

Globale Variablen

Bei manchen Programmierern sind sie verpönt, trotzdem sind sie möglich: globale Variablen. Ihr Wert ist in der gesamten Unit verfügbar und in allen Units, die diese einbinden. Die Deklaration erfolgt am Anfang der Unit:

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
    TForm1 = class(TForm)
    private
        { Private-Deklarationen }
    public
        { Public-Deklarationen }
    end;

var
    Form1: TForm1;
    einezahl: integer;           // Diese Variable gilt in der ganzen Unit und
                                // in allen Units, die diese Unit einbinden

implementation

{$R *.DFM}

var eine_andere_zahl: real; // Diese Variable gilt nur in dieser Unit
```

Globale Variablen können bei ihrer Deklaration mit einem Startwert belegt werden:

```
einezahl: integer = 42;
```



Diese Art der Zuweisung verwendet ein einfaches Gleichheitszeichen, nicht := Bei lokalen Variablen ist diese Initialisierung nicht möglich.

Lokale Variablen

Das Gegenstück zu globalen Variablen sind die lokalen. Hierbei wird eine Variable zu Beginn einer Prozedur oder Funktion deklariert. Sie kann somit nur innerhalb dieses Abschnitts verwendet werden. Wird die Prozedur/Funktion verlassen, dann wird der Speicher für die Variablen wieder freigegeben, d. h. auf die Werte kann nicht mehr zugegriffen werden. So könnte eine lokale Variablendeklaration aussehen:

```
procedure IchMacheIrgendwas;
var text: string;
begin
    ... //Irgendwas Sinnvolles
end;
```

Initialisierung

Bevor auf eine Variable zugegriffen wird, sollte sie initialisiert werden, es sollte ihr also ein Anfangswert zugewiesen werden. Strings enthalten zwar einen leeren String; alle anderen Variablentypen enthalten jedoch irgendwelche zufälligen Werte.

Zuweisungen

Zuweisung von Werten an eine Variable erfolgt in Pascal durch die Symbolfolge := („ergibt sich aus“). Im Gegensatz zur Mathematik ist deshalb auch Folgendes möglich:

```
x := x+1;
```

Das Laufzeitsystem berechnet hier die Summe von x und 1 und legt das Ergebnis dann wieder in x ab. x ergibt sich aus dem bisherigen x plus 1. Kurz: x wird um 1 erhöht. Für diesen Fall gibt es auch eine eigene Prozedur: inc(x).

Typumwandlung

Im Gegensatz zu manch anderen Sprachen ist Delphi bei Typen sehr streng. Es ist also nicht möglich einer Integer-Variable eine Gleitkommazahl-Variable zuzuweisen. Dafür stehen eine große Auswahl an Konvertierungsfunktionen zur Verfügung:

von	nach	Funktion	Beispiel
Integer	Real	kein Problem, einfache Zuweisung	<pre>var zahl1: integer; zahl2: real; begin zahl2 := zahl1;</pre>
Real	Integer	Möglichkeiten: - Nachkommastellen abschneiden (trunc) - kaufm. Runden (round) - aufrunden (ceil, Unit Math) - abrunden (floor, Unit Math)	<pre>var zahl1: real; zahl2: integer; begin zahl2 := trunc(zahl1); zahl2 := round(zahl1);</pre>
Integer	String	IntToStr	<pre>var textzahl: string; zahl: integer; begin textzahl := IntToStr(zahl);</pre>
Real	String	FloatToStr FloatToStrF	<pre>var textzahl: string; zahl: real; begin textzahl := FloatToStr(zahl);</pre>
String	Integer	StrToInt StrToIntDef	<pre>var textzahl: string; zahl: Integer; begin zahl := StrToInt(textzahl);</pre>

String	Real	StrToFloat	<pre>var textzahl: string; zahl: real; begin zahl := StrToFloat(textzahl);</pre>
String	Char	Zugriff über Index (1 ist erstes Zeichen)	<pre>var text: string; zeichen: char; begin zeichen := text[1];</pre>
Char	String	kein Problem, einfache Zuweisung	<pre>var zeichen: char; text: string; begin text := zeichen;</pre>

Teilbereichstypen

Einen weiteren Datentyp, der eigentlich gar kein eigener Datentyp ist, gibt es noch, nämlich den Teilbereichstyp, auch Unterbereichstyp genannt. Hierüber kann man Variablen z. B. zwar den Typ Integer zuordnen, aber nicht den kompletten Definitionsbereich, sondern nur einen Teilbereich davon:

```
var kleineZahl: 0..200;
```

Der Variablen `kleineZahl` lassen sich jetzt nur ganze Zahlen zwischen 0 und 200 zuweisen. Ähnlich lassen sich auch Strings begrenzen:

```
var kleinerString: string[10];
```

Diese Variable kann nur zehn Zeichen aufnehmen, es handelt sich nun um einen ShortString, der auch Nachteile gegenüber einem „normalen“ AnsiString hat. Mehr dazu aber in einem extra Abschnitt über Strings.

Aufzählungstypen

Um das Ganze für einen Programmierer lesbarer zu machen, können statt Zahlen auch konstante Bezeichner verwendet werden:

```
var farben: (blau, gelb, gruen, rot);
```

Intern werden die Werte bei Null beginnend durchnummeriert und haben deshalb eine feste Reihenfolge. Über die Funktion `ord` lässt sich die Position bestimmen.

Wichtige Funktionen zum Arbeiten mit Aufzählungstypen sind:

<code>ord</code>	gibt die Position des Bezeichners zurück
<code>pred</code>	gibt den Vorgänger zurück
<code>succ</code>	gibt den Nachfolger zurück
<code>low</code>	gibt den niedrigsten Wert zurück
<code>high</code>	gibt den höchsten Wert zurück

Die Typen Integer und Char gehören ebenfalls zu den ordinalen (abzählbaren) Typen, d.h. die Werte lassen sich in einer Reihenfolge anordnen, weshalb o.g. Funktionen auf sie ebenfalls angewandt werden können.

Konstanten

Konstanten sind letztendlich Variablen, die innerhalb des Programms jedoch nur ausgelesen, nicht überschrieben werden dürfen. Deklariert werden sie an den gleichen Stellen wie Variablen, allerdings mit dem Schlüsselwort `const`, einem Gleichheitszeichen und ohne Angabe eines Datentyps:

```
const version = '1.23';
```

3. String-Typen

Unterschiedliche String-Typen

In Delphi gibt es verschiedene Möglichkeiten, Strings zu deklarieren:

```
-String  
-ShortString  
-AnsiString  
-WideString  
-Array[0..x] of Char  
-PChar
```

In diesem Abschnitt soll es darum gehen, welchen Typ man wann benutzt und wie sie sich überhaupt voneinander unterscheiden.

Pascal-Strings

Zunächst muss nach Delphi-Version unterschieden werden. Den Datentyp **String** gibt es in Pascal schon immer, jedoch hat sich mit der Umstellung auf 32 Bit (Delphi 2) sein Aufbau verändert.

In den 16-Bit-Versionen (bis Delphi 1) bestand ein String aus maximal 255 Zeichen, für den statisch Arbeitsspeicher reserviert wurde. Im ersten Byte des Strings befand sich die Längenangabe. Mit Delphi 2 hat sich das geändert, der alte String-Typ kann trotzdem noch verwendet werden, sein Typ ist **ShortString**:

```
var kurzertext: ShortString;
```

Ebenso wird ein String als ShortString angesehen, wenn man ihm bei der Deklaration eine feste Länge verordnet:

```
var kurzertext: String[20];
```

Die „altmodische“, aus DOS-Tagen stammende Begrenzung auf 255 Zeichen wurde mit Delphi 2 aufgehoben. Wird seitdem in Win32 eine Variable vom Typ String deklariert, handelt es sich um einen **AnsiString**. Die Besonderheit hiervon ist, dass es sich bei der Variable nur noch um einen Zeiger auf eine Stelle im Arbeitsspeicher handelt, die dynamisch vergeben wird. Der String kann somit (rein theoretisch) bis zu 2 GB groß werden, wobei sich der Speicherverbrauch automatisch der String-Länge anpasst. Enthält der String keine Zeichen, verbraucht er auch keinen Speicherplatz, abgesehen von 4 Byte für den Zeiger auf `nil`, also ins Leere. AnsiStrings enden mit einem Nullzeichen (#0), sind also voll kompatibel zu nullterminierten Strings, wie sie manchmal von API-Funktionen gefordert werden.

WideStrings werden in der .NET-Umgebung als Standard-String verwendet, existieren aber auch schon in früheren Delphi-Versionen. Sie bieten jedoch pro Zeichen zwei statt einem Byte Speicher, sind also Unicode-tauglich. AnsiStrings mit einem Byte pro Zeichen können nur Ansi-Zeichen aufnehmen. Mit Unicode sollen jedoch alle Schriftzeichen der Welt dargestellt werden können, weshalb auf zwei Byte pro Zeichen umgestellt wurde. WideStrings werden wie AnsiStrings dynamisch verwaltet. Unter Win32 unterliegen WideStrings nicht der Referenzzählung. Unter Linux und .NET jedoch schon.

Grundsätzlich ist es angebracht, in eigenen Programmen immer den generischen Typ String zur Deklaration von Strings zu verwenden. Kurze Strings (ShortString) sollten wegen ihrer statischen Größe nur in Ausnahmefällen (z.B. Datenaustausch mit DLLs) verwendet werden.

Nullterminierte Strings

Die folgenden String-Typen sind innerhalb eines normalen Delphi-Programms nicht erforderlich, gelegentlich werden sie jedoch zur Kommunikation mit der „Außenwelt“, der in C++ geschriebenen Win32-API benötigt.

Bei nullterminierten Strings handelt es sich um Zeichenketten, die ihr Ende durch eine ASCII-Null (#0) kennzeichnen. Man deklariert sie so:

```
var text: array [0..100] of Char;
```

Da es sich hierbei um keine normalen Pascal-Strings handelt, müssen solche nullterminierten Strings mit speziellen Funktionen bearbeitet werden, z.B. `StrPCopy`, um einen Pascal-String in einen nullterminierten String zu kopieren.

Bei **PChar** handelt es sich um einen Zeiger auf ein C-kompatibles Zeichenarray. Dieser Typ wird von einigen API-Funktionen gefordert. Man erhält ihn ganz einfach, indem man einen `AnsiString` mittels `PChar(langerText)` umwandelt.

Arbeiten mit Pascal-Strings

Die Arbeit mit den anfangs erwähnten Pascal-Strings (`ShortString`, `AnsiString`, `WideString`) ist recht einfach:

Zeichenketten zusammenhängen

```
var text1, text2: String;
begin
    text1 := 'toll';
    text2 := 'Ich finde Delphi '+text1+'!!!';
    // text2 enthält nun den Text 'Ich finde Delphi toll!!!'
```

Zugreifen auf ein bestimmtes Zeichen eines Strings

Der Zugriff auf ein einzelnes String-Zeichen erfolgt über dessen Index:

```
var text: String;
    zeichen: Char;
begin
    text := 'Ich finde Delphi toll!';
    zeichen := text[1];
    // zeichen enthält nun den Buchstaben 'I'
```

Vergleich zweier Strings

Das Vergleichen von zwei Strings erfolgt mit dem Gleichheitszeichen. Dabei wird Groß- und Kleinschreibung beachtet.

```
var text1, text2: string;
...
if text1 = text2 then ...
```


Die Delphi-Laufzeitumgebung bietet noch einige weitere Funktionen, z.B. `AnsiCompareText` zum Vergleich zweier `AnsiStrings` *ohne* Berücksichtigung der Groß- und Kleinschreibung. `pos` hilft beim Auffinden eines Teilstrings; `copy` zum Kopieren eines Teilstrings und `delete` zum Löschen eines Teilstrings sind ebenfalls wichtige Bearbeitungsmöglichkeiten.

Zur effektiven Arbeit mit Strings unter .NET bietet das .NET-Framework die Klasse `StringBuilder` (Namensraum `System.Text`). Das Zusammenfügen mehrerer Strings sollte unter .NET nicht mit dem Plus-Operator erfolgen, sondern so:

```
uses System.Text;

var s1, s2: string;
    sb: StringBuilder;
begin
    s1 := 'Hallo';
    s2 := 'Delphi';
    sb := StringBuilder.Create(s1);
    s1 := sb.Append(s2).ToString;
```

4. Zusammengesetzte Datentypen

Zum Pascal-Sprachumfang gehören nicht nur einfache Datentypen, wie sie im Abschnitt Variablen und Konstanten beschrieben sind, sondern auch zusammengesetzte.

Mengentypen

Um in einer einzigen Variablen eine unterschiedliche Menge an Werten des gleichen Typs zu speichern, gibt es Mengentypen. Es ist eine Menge an möglichen Werten vorgegeben, aus der eine beliebige Anzahl (keiner bis alle) in der Variablen abgelegt werden kann. Folgendermaßen wird eine solche Mengenvariable deklariert:

```
var zahlen: set of 1..10;
```

Damit können der Variablen zahlen Werte aus der Menge der Zahlen von 1 bis 10 zugewiesen werden – mehrere gleichzeitig oder auch gar keiner:

```
zahlen := [5, 9];           // zahlen enthält die Zahlen 5 und 9
zahlen := [];               // zahlen enthält überhaupt keine Werte
zahlen := [1..3];           // zahlen enthält die Zahlen von 1 bis 3
zahlen := zahlen + [5];      // zahlen enthält die Zahlen 1, 2, 3, 5
zahlen := zahlen - [3..10];  // die Zahlen von 3 bis 10 werden aus der Menge
                             // entfernt, es bleiben 1 und 2
```

Um nun zu prüfen, ob ein bestimmter Wert in der Menge enthalten ist, wird der Operator `in` verwendet:

```
if 7 in zahlen then ...
```

Arrays

Müssen mehrere Werte des gleichen Typs gespeichert werden, ist ein Array (zu deutsch Feld oder Liste) eine praktische Lösung. Ein Array hat einen Namen wie eine Variable, jedoch gefolgt von einem Index in eckigen Klammern. Über diesen Index kann man auf die einzelnen Werte zugreifen. Am einfachsten lässt sich das mit einer Straße vergleichen, in der sich lauter gleiche Häuser befinden, die sich jedoch durch ihre Hausnummer (den Index) unterscheiden.

Eine Deklaration der Art

```
var testwert: array [0..10] of integer;
```

bewirkt also, dass wir quasi elf verschiedene Integer-Variablen bekommen. Man kann auf sie über den Indexwert zugreifen:

```
testwert[0] := 15;
testwert[1] := 234;
```

usw.

Vorteil dieses Indexes ist, dass man ihn durch eine weitere Variable ersetzen kann, die dann in einer Schleife hochgezählt wird. Folgendes Beispiel belegt alle Elemente mit dem Wert 1:

```
for i := 0 to 10 do
  testwert[i] := 1;
```

Auf Schleifen wird jedoch in einem gesonderten Kapitel eingegangen.

Bei dem vorgestellten Array handelt es sich genauer gesagt um ein eindimensionales, statisches Array. „Eindimensional“, weil die Elemente über nur einen Index identifiziert werden, und „statisch“, weil Speicher für alle Elemente reserviert wird. Legt man also ein Array für Indexwerte von 1 bis 10000 an, so wird für 10000 Werte Speicher reserviert, auch wenn während des Programmablaufs nur auf zwei

Elemente zugegriffen wird. Außerdem kann die Array-Größe zur Programmlaufzeit nicht verändert werden.

Dynamische Arrays

Wenn schon so viel Wert auf die Eigenschaft statisch gelegt wird, muss es ja eigentlich auch etwas Dynamisches geben. Und das gibt es auch, zumindest seit Delphi 4: die dynamischen Arrays.

Der erste Unterschied findet sich in der Deklaration: Es werden keine Grenzen angegeben.

```
var dynArray: array of integer;
```

dynArray ist nun prinzipiell eine Liste von Integer-Werten, die bei Index Null beginnt.



Zum Hintergrundverständnis: Während statische Arrays direkt die einzelnen Werte beinhalten, enthält ein dynamisches Array nur Zeiger auf einen Arbeitsspeicherbereich. In der Anwendung ist das nicht zu merken, es ist keine spezielle Zeigerschreibweise nötig. Nur an einer Stelle bemerkt man das interne Vorgehen: Bevor man Werte in das Array stecken kann, muss man Speicher für die Elemente reservieren. Dabei gibt man an, wie groß das Array sein soll:

```
SetLength(dynArray, 5);
```

Nun kann das Array fünf Elemente (hier Integer-Zahlen) aufnehmen.



Man beachte: Da die Zählung bei Null beginnt, befindet sich das fünfte Element bei Indexposition 4!

Der Zugriff erfolgt ganz normal:

```
dynArray[0] := 321;
```

Damit es mit der Unter- und vor allem der Obergrenze, die ja jederzeit verändert werden kann, keine Zugriffe auf nicht (mehr) reservierten Speicher vorkommen, lassen sich Schleifen am einfachsten so realisieren:

```
for i := 0 to high(dynArray) do  
    dynArray[i] := 0;
```

Dadurch werden alle Elemente auf 0 gesetzt. `high(dynArray)` entspricht dem höchstmöglichen Index. Über `length(a)` lässt sich die Länge des Arrays ermitteln, welche immer `high(dynArray)+1` ist. Dabei handelt es sich um den Wert, den man mit `SetLength` gesetzt hat.

Da die Länge des Arrays jederzeit verändert werden kann, könnten wir sie jetzt mit

```
SetLength(dynArray, 2);
```

auf zwei verkleinern. Die drei hinteren Werte fallen dadurch weg. Würden wir das Array dagegen vergrößern, würden sich am Ende Elemente mit undefiniertem Wert befinden.

Mehrdimensionale Arrays

Wenn es eindimensionale Arrays gibt, muss es auch mehrdimensionale geben. Am häufigsten sind hier wohl die zweidimensionalen. Man kann mit ihnen z. B. ein Koordinatensystem oder Schachbrett abbilden. Die Deklaration ist wie folgt:

```
var koordinate: array [1..10, 1..10] of integer;
```

Es werden also 10x10=100 Elemente angelegt, die jeweils einen Integer-Wert aufnehmen können. Für den Zugriff auf einzelne Werte sind zwei Schreibweisen möglich:

```
koordinate[1,6] := 34;  
koordinate[7][3] := 42;
```

Records

Records entsprechen von der Struktur her einem Datensatz einer Datenbank - nur dass sie, wie alle bisher aufgeführten Variablen, nur zur Laufzeit vorhanden sind. Werden also mehrere unterschiedliche Daten, die immer zusammengehören, aufgenommen, werden Records verwendet. Beispiel: Adressen.

```
var Adresse: record
  name: string;
  plz: integer;
  ort: string;
end;
```

Die Variable Adresse wird also in drei „Untervariablen“ aufgegliedert.

Folgendermaßen greift man auf die einzelnen Felder zu:

```
adresse.name := 'Hans Müller';
adresse.plz := 12345;
adresse.ort := 'Irgendwo';
```

Damit das nicht (besonders bei vielen Elementen) in enorme Tipparbeit ausartet, gibt es dafür auch eine Abkürzung:

```
with adresse do begin
  name := 'Hans Müller';
  plz := 12345;
  ort := 'Irgendwo';
end;
```

Variante Records

Eine besondere Form der Records sind die varianten Records. Hier wird mittels case-Unterscheidung immer nur ein bestimmter Datenteil berücksichtigt.

Beispiel:

```
type
  person = record
    name: string;
    case erwachsen: boolean of
      true: (personalausweisnr: integer);
      false: (kinderausweisnr: integer;
              erziehungsberechtigte: string);
  end;
```

Das Record speichert in jedem Fall die Daten name und erwachsen. Abhängig von dem Wert erwachsen wird dann entweder die personalausweisnr (bei erwachsen=true) oder

kinderausweisnr und erziehungsberechtigte (bei erwachsen=false) gespeichert. Ist erwachsen=false, kann nicht auf den Wert personalausweisnr zugegriffen werden.

Dadurch hat man eindeutige Beschreibungen von Feldwerten und kommt nicht in Gefahr, die Kinderausweisnr versehentlich als Personalausweisnr zu speichern.

Wichtig ist noch zu bemerken, dass der variante Teil eines Records (also der case-Teil) immer am Ende stehen muss. Und case hat in diesem Fall auch kein schließendes end, wie sonst üblich.

Nun wäre es praktisch, eine größere Menge solcher Variablen zu haben, da eine Adressverwaltung üblicherweise mehr als nur eine Adresse enthält. Die Lösung: Wir kombinieren Array und Record. Damit wir unseren neuen Adresstyp aber einfach weiterverwenden können, wollen wir zuerst einen eigenen Typ anlegen.

Eigene Typen definieren

Wir erinnern uns: Variablendeklarationen enthalten auf der linken Seite einen Variablennamen und rechts vom Doppelpunkt einen zugehörigen Datentyp. Solche Typen können wir auch selbst definieren, beispielsweise obiges Record:

```
type TAdressRecord = record
  name: string;
  plz: integer;
  ort: string;
end;
```

Eine Typdefinition besteht also aus dem Schlüsselwort type gefolgt von einer Konstruktion, die einer Variablendeklaration ähnelt. Da aber statt einer Variablen ein Typ deklariert wird, ist das Folgende auch kein Variablen- sondern ein Typname. Typnamen beginnen in Delphi mit einem großen T, wenn sich der Programmierer an den Styleguide gehalten hat.

Statt eines Doppelpunkts wird ein Gleichheitszeichen verwendet.

Auf die folgende Weise kann solch ein Typ dann verwendet werden:

```
var adresse: TAdressRecord;
```

Diese Variablendeklaration entspricht der im obigen Beispiel für Records.

Wollen wir nun noch ein Array daraus machen, dann geht das so:

```
var adresse: array [1..50] of TAdressRecord;
```

Der Zugriff erfolgt streng nach den Regeln für Arrays und Records:

```
adresse[1].name := 'Hans Müller';
adresse[2].name := 'Susi Sorglos';
```

5. Zeiger

Bei Zeigern handelt es sich um Variablen, die eine Speicheradresse enthalten. Eine Zeigervariable zeigt also auf eine bestimmte Stelle im Arbeitsspeicher. Zum einen gibt es den Typ `Pointer`, der auf beliebige Daten zeigt und zum anderen spezialisierte Zeigertypen.

In der Delphi-Sprache kommen Zeiger (auch Referenzen genannt) häufig vor, so z.B. bei Objektreferenzen, bei langen Strings und bei dynamischen Arrays. An der Syntax fällt das an diesen Stellen nicht auf. Will man Zeiger auf selbstdefinierte Typen erstellen, muss dagegen die noch aus Pascal-Zeiten stammende Zeigersyntax mit `@` und `^` verwendet werden.

Unter .NET sollten Zeiger nicht mehr verwendet werden.

Doch zunächst ein Beispiel zur Deklaration von Zeigern:

```
type
  PAdressRecord = ^TAdressRecord;
  TAdressRecord = record
    name: string;
    plz: integer;
    ort: string;
end;

var
  adresse: TAdressRecord;
  adresszeiger: PAdressRecord;
```

`^` zur Zeigertypdefinition

Steht das Symbol `^` vor einem Typbezeichner, wird daraus ein Typ, der einen Zeiger auf den ursprünglichen Typ darstellt. `^TAdressRecord` ist ein Zeigertyp auf einen Speicherbereich, an dem sich etwas vom Typ `TAdressRecord` befinden muss.

`@` zur Ermittlung einer Speicheradresse

Anfänglich zeigt unsere Variable `adresszeiger` ins Leere - auf `nil` (not in list). Wir wollen nun, dass er auf die Variable `adresse` zeigt. Da ein Zeiger sich bekanntlich nur Speicheradressen merken kann, benötigen wir die Speicheradresse unserer Variable `adresse`. Diese ermitteln wir mit dem Operator `@`:

```
adresszeiger := @adresse;
```

`^` zur Dereferenzierung

Nun können wir auch über den Zeiger auf den Inhalt von `adresse` zugreifen. Wir wollen also, dass uns der Zeiger nicht die Adresse bekannt gibt, auf die er zeigt, sondern den Wert an dieser Speicherstelle. Diesen Vorgang nennt man Dereferenzieren. Verwendet wird dafür wieder das Symbol `^` - allerdings diesmal *hintereiner* Zeigervariablen:

```
var
  gesuchterName: string;
begin
  gesuchterName := adresszeiger^.name;
```

Eine Dereferenzierung ist nur mit spezialisierten Zeigern möglich, nicht mit dem Allround-Talent `Pointer`. Um auch mit dem Typ `Pointer` entsprechend arbeiten zu können, muss dieser in einen anderen Zeigertyp umgewandelt werden.

6. Schleifen

In einem Programmablauf kommt es öfters vor, dass eine bestimmte Befehlsfolge mehrmals hintereinander ausgeführt werden soll. So etwas nennt man „Schleife“.

Von ihnen gibt es unterschiedliche Arten: Sie unterscheiden sich darin, ob die Abbruchbedingung vor dem ersten Durchlauf geprüft werden soll oder erst danach und ob bereits feststeht, wie viele Male eine Schleife durchlaufen wird.

for-Schleife

Die for-Schleife hat folgenden Aufbau:

```
for i := 1 to 10 do begin
    ... // Befehlsfolge, die öfters ausgeführt werden soll
end;
```

Die Beispielschleife wird von 1 bis 10, also zehnmal durchlaufen. Nach jeder "Runde" wird die sog. Schleifenvariable automatisch um 1 erhöht. Die Schleifenvariable heißt im Beispiel i, das muss nicht so sein. Auf jeden Fall ist es aber ein Ordinalwert. Die Grenzen (1 bis 10) sind hier direkt als Zahlen vorgegeben, es können jedoch auch Integer-Variablen, Aufzählungen oder Chars sein. Es sind alle Typen erlaubt, bei denen es eine feste Reihenfolge gibt.

Schleifenvariablen dürfen grundsätzlich innerhalb der Schleife nicht verändert werden. Ist die Obergrenze kleiner als die Untergrenze, wird die Schleife nicht durchlaufen (z. B. `for i:=1 to 0`); sind die Grenzen identisch, wird sie einmal durchlaufen.

Alternativ zum Hochzählen der Schleifenvariable ist auch Folgendes möglich:

```
for i:=10 downto 1 do ...
```

while-Schleife

Im Gegensatz zur for-Schleife verwendet die while-Schleife keine Schleifenvariable, die automatisch hochgezählt wird. Hier wird vor jedem Durchlauf geprüft, ob eine bestimmte Bedingung erfüllt ist. Trifft diese nicht mehr zu, wird die Schleife nicht mehr durchlaufen und der Programmablauf danach fortgesetzt. Trifft die Bedingung bereits am Anfang nicht zu, wird die Schleife überhaupt nicht betreten. Man spricht hier von einer kopfgesteuerten Schleife.

Die while-Schleife hat folgende Struktur:

```
while x<>y do begin
    ... // Befehlsfolge, die öfters ausgeführt werden soll
end;
```

Solange also x ungleich y ist, wird die Schleife durchlaufen. Es ist also ratsam, x und/oder y innerhalb der Schleife zu verändern; andernfalls wäre das Durchlaufkriterium immer erfüllt, die Schleife würde nie zu einem Ende kommen. Der erfahrene Programmierer spricht hier von einer Endlosschleife, die ältere Betriebssysteme komplett in die Knie zwingen kann. Ist x bereits zu Beginn gleich y wird die Schleife überhaupt nicht durchlaufen.

Die Bedingung hinter `while` kann ein beliebiger Ausdruck sein, der einen Wahrheitswert (Boolean) ergibt.

repeat-until-Schleife

War bei der while-Schleife das Durchlaufkriterium anzugeben, ist es bei der repeat-until-Schleife das Abbruchkriterium. Außerdem wird dieses erst am Ende eines Schleifendurchlaufs geprüft. Ein Durchlauf findet also auf jeden Fall statt. Man spricht hier von einer fußgesteuerten Schleife.

```
repeat
    ... // Befehlsfolge, die öfters ausgeführt werden soll
until x=y;
```

Die Beispielschleife wird solange durchlaufen, bis x gleich y ist. Auch hier ist wieder darauf zu achten, dass keine Endlosschleife entsteht. Auch wenn x schon zu Beginn gleich y ist, wird die Schleife dennoch einmal durchlaufen.

Bedingungen

Bei den Abbruch- und Durchlaufbedingungen handelt es sich um logische Operationen. Wie diese in Pascal notiert werden und wann hierbei Klammern benötigt werden, ist in einem folgenden Kapitel beschrieben.

Schleifen abbrechen

Schleifen lassen sich natürlich auch vor dem regulären Ende verlassen. Dazu gibt es die Prozedur `break`. **break** kann nur innerhalb von Schleifen verwendet werden und setzt den Programmablauf mit der ersten Anweisung nach der Schleife fort.

Außerdem gibt es Situationen, in denen man schon zu Beginn eines Schleifendurchlaufs weiß, dass man gleich mit der nächsten „Runde“ fortfahren kann. Hier kann man **continue** verwenden. Dadurch wird die Durchführung eines Schleifendurchlaufs abgebrochen und mit dem nächsten Durchlauf begonnen. Bei for-Schleifen wird der Index erhöht.

7. Verzweigungen

if-else

Es gibt kaum ein Programm, bei dem immer alle Befehle hintereinander ausgeführt werden. Verzweigungen sind ein häufig eingesetztes Mittel. Es handelt sich hierbei um Fallunterscheidungen, die in der Regel mit `if` durchgeführt werden:

```
if <Boolean-Ausdruck> then <Anweisung>;
```

oder

```
if <Boolean-Ausdruck> then <Anweisung> else <Anweisung>;
```

Eine Anweisung kann dabei wiederum aus einer neuen `if`-Bedingung bestehen.

Beispiel:

```
if x>0 then ...  
else if x<0 then ...  
else ...;
```

Das Beispiel bedeutet Folgendes: Ist `x` größer als Null, wird das ausgeführt, was hinter dem ersten `then` steht (die drei Punkte). Handelt es sich dabei um mehr als einen Befehl, muss der Block mit `begin` und `end` umgeben werden.

`else`, zu Deutsch „sonst“, leitet eine Alternative ein. Wenn also die erste Bedingung nicht erfüllt ist, wird die zweite geprüft, hier, ob `x` vielleicht kleiner als Null ist. Trifft auch diese Bedingung nicht zu, bleibt noch ein `else` ohne Bedingung. Die letzten drei Punkte werden also immer dann ausgeführt, wenn die ersten beiden Bedingungen nicht zutreffen.

Wäre bereits die erste Bedingung erfüllt gewesen, so wären die folgenden `else`-Abschnitte gar nicht mehr geprüft worden.

Selbstverständlich muss so ein `if`-Block keine „`else if`“- oder „`else`“-Alternativen bieten. Das kommt immer auf die Situation an. Da sich das Ganze aber sehr stark an mathematische Logik anlehnt, dürfte die Notation nicht allzu schwer fallen.



Was allerdings zu beachten ist: In Pascal steht hinter dem letzten Befehl vor dem Wörtchen `else` kein Strichpunkt (Semikolon) wie sonst üblich (im Gegensatz zu C++).

Noch ein Beispiel, wobei jeweils mehrere Befehle ausgeführt werden:

```
var eingabe: integer;  
...  
if eingabe=1 then begin  
    eingabe := 0;  
    ausgabe := 'Sie haben eine 1 eingegeben';  
end //kein Strichpunkt!  
else if eingabe=2 then begin  
    eingabe := 0;  
    ausgabe := 'Sie haben eine 2 eingegeben';  
end  
else begin  
    eingabe := 0;
```

```
    ausgabe := 'Sie haben eine andere Zahl als 1 oder 2 eingegeben';  
end;
```

Hier sieht man besonders den Sinn von `else`. Wären die drei Fallunterscheidungen durch drei getrennte `if`-Abfragen dargestellt worden, dann hätten wir folgendes Problem: Angenommen `eingabe` ist 1, so ist die erste Bedingung erfüllt. Da hier `eingabe` jedoch auf 0 gesetzt wird, träfe nun auch die dritte Bedingung zu. Im obigen Beispiel mit `else` stellt das kein Problem dar.

case-Verzweigung

Müssten wir in obigem Beispiel mehr als nur zwei Zahlen prüfen, hätten wir ganz schön Tipparbeit. Für solche abzählbaren (ordinalen) Typen wie Integer und Char gibt es in Pascal eine Abkürzung:

```
case eingabe of  
  1: ausgabe := 'Sie haben 1 eingegeben';  
  2: ausgabe := 'Sie haben 2 eingegeben';  
  3: ausgabe := 'Sie haben 3 eingegeben';  
  else ausgabe := 'Sie haben nicht 1, 2 oder 3 eingegeben';  
end;
```

Zugegeben, das Beispiel ist nicht besonders sinnvoll, da die Variable `eingabe` direkt in einen String umgewandelt werden könnte. Allerdings stellt es gut die Funktionsweise von `case` dar. Zu beachten ist, dass am Ende eines Case-Blocks ein `end` stehen muss. Gehören mehrere Anweisungen zusammen, können sie wie bei `if` durch `begin` und `end` als zusammengehörig gekennzeichnet werden.



Bei `case` steht (im Gegensatz zu `if`) vor dem Schlüsselwort `else` ein Strichpunkt!

Mit `case` ist auch Folgendes möglich:

```
case eingabe of  
  1,3,5,7,9: ausgabe := 'Sie haben eine ungerade kleiner als 10 '+  
                      'eingegeben';  
  2,4,6,8,0: ausgabe := 'Sie haben eine gerade Zahl kleiner als 10 '+  
                      'eingegeben';  
  10..20:    ausgabe := 'Sie haben eine Zahl zwischen 10 und 20 '+  
                      'eingegeben';  
end;
```

So etwas ist natürlich auch bei `if`-Bedingungen möglich, dazu allerdings noch einen Einschub zum Thema Notation von logischen Bedingungen.

8. Logische Operatoren

Eine logische Bedingung wie $x > 0$ versteht jeder, der schon einmal Mathematikunterricht hatte. Allerdings kann eine Bedingung auch aus mehreren einzelnen Bedingungen bestehen: z. B. x größer als 0 und kleiner als 10. In Pascal kann man das leider nicht schreiben wie in Mathe ($0 < x < 10$).

Stattdessen müssen zwei einfache Bedingungen daraus gemacht werden: $x > 0$ und $x < 10$; mehrere einzelne Bedingungen müssen in Pascal jeweils von runden Klammern umgeben werden. Außerdem werden die folgenden logischen Operatoren verwendet: **and**, **or**, **not**, **xor**. Das Ergebnis einer logischen Bedingung ist immer ein Wahrheitswert (`true` oder `false`). Unser Beispiel lautet nun also

```
(x>0) and (x<10)
```

Mehrere solche Bedingungen können wie in der Mathematik mit Klammern gruppiert werden, z. B.

```
((x>0) and (x<10)) or ((y>0) and (y<10))
```

Solche Bedingungen können dann als Verzweigungsbedingungen (s.o.) oder auch als Abbruch- bzw. Durchlaufbedingungen bei Schleifen verwendet werden:

```
if ((x>0) and (x<10)) or ((y>0) and (y<10)) then ...
```

Kurzschlussverfahren

Standardmäßig verwendet der Delphi-Compiler das Kurzschlussverfahren bei der Auswertung der logischen Operatoren **and** und **or**. Das bedeutet, dass er von links nach rechts vorgeht. Eine **and**-Operation ergibt beispielsweise nur `true`, wenn beide Operanden `true` sind. Ist nur einer `false`, kann der Ausdruck nicht mehr `true` werden.

Steht in einem Quellcode also

```
if <Bedingung1> and <Bedingung2>
```

wird die Auswertung des Ausdrucks abgebrochen, wenn `Bedingung1` bereits `false` ist. Egal, welches Ergebnis `Bedingung2` liefert, der Ausdruck ergibt `false`. Deshalb wird im Kurzschlussverfahren `Bedingung2` in diesem Fall völlig ignoriert, was zu einer schnelleren Ausführung beiträgt.

Dennoch lässt sich über die Compiler-Direktive `$B` einstellen, dass eine vollständige Auswertung durchgeführt werden soll, falls das einmal nötig ist. Standardeinstellung ist `{ $B- }` für das Kurzschlussverfahren.

Operatoren überladen

Unter .NET ist es möglich, Operatoren zu überladen. Das bedeutet, dass der Programmierer festlegen kann, was z.B. passiert, wenn zwei Klassen mit dem `+` Operator addiert werden. Dazu wird mit der Syntax `class operator` eine Methode deklariert und implementiert. Der Name der Methode

richtet sich nach dem Operator, der überladen werden soll. So lautet der Name für den +Operator `Add`, für > `GreaterThan` und für = `Equal`. Die weiteren Operatoren-Methodennamen sind in der Hilfe zu finden.

Deklaration:

```
type
  TMyClass = class
    class operator Add(a, b: TMyClass): TMyClass;
  end;
```

Implementierung:

```
class operator TMyClass.Add(a, b: TMyClass): TMyClass;
begin
  ...
end;
```

Die Methode `Add` kann nicht direkt aufgerufen werden. Stattdessen muss das Pluszeichen verwendet werden.

9. Prozeduren und Funktionen

Prozeduren und Funktionen, auch „Unterprogramme“ oder Routinen genannt, haben die Aufgabe, öfter wiederkehrenden Programmcode sozusagen als Baustein zusammenzufassen. Dieser Baustein erhält einen eindeutigen Namen, über den er ausgeführt werden kann.

Aufbau einer Prozedur

Jede Prozedur besteht aus dem Schlüsselwort `procedure`, gefolgt von einem gültigen Namen und evtl. einer Parameterliste in runden Klammern. Sind keine Parameter vorhanden, werden die Klammern sowohl bei der Deklaration als auch beim Aufruf weggelassen. Diesen Teil nennt man Kopf der Prozedur. Es folgen Variablen- und Konstantendeklarationen und anschließend zwischen `begin` und `end` die Anweisungen, die die Prozedur durchführen soll:

```
procedure <Name>(<Parameter>);  
  <Variablen- und Konstanten>  
begin  
  <Anweisungen>  
end;
```

Beispiel: Die folgende Prozedur gibt so viele Töne über den PC-Lautsprecher aus, wie über den Parameter `Anzahl` angegeben.

```
procedure Toene(Anzahl: integer);  
var i: integer;  
begin  
  for i := 1 to Anzahl do  
    beep;  
end;
```

Der Aufruf für fünf Töne geschieht so:

```
Toene(5);
```

Aufbau einer Funktion

Eine Funktion unterscheidet sich nur geringfügig von einer Prozedur. Sie besitzt einen Rückgabewert und wird mit dem Schlüsselwort `function` deklariert anstelle von `procedure`.

```
function <Name>(<Parameter>): <Rückgabety>;  
  <Variablen- und Konstanten>  
begin  
  <Anweisungen>  
end;
```

Beispiel: Eine Funktion, die drei Zahlen addiert und das Ergebnis zurückliefert.

```
function SummeAusDrei(zahl1, zahl2, zahl3: integer): integer;  
begin  
    result := zahl1 + zahl2 + zahl3;  
end;
```

Bei `result` handelt es sich um eine vordefinierte Variable, der der Rückgabewert zugewiesen wird, den die Funktion haben soll. Alternativ kann dafür auch der Funktionsname verwendet werden. Mit `result` kann innerhalb der Funktion jedoch gerechnet werden, während der Funktionsname nur auf der linken Seite einer Zuweisung stehen darf, da ansonsten von einem rekursiven Aufruf der Funktion (Funktion ruft sich selbst auf) ausgegangen wird.

Es ist möglich, `result` oder dem Funktionsnamen öfters einen Wert zuzuweisen. Letztlich bildet der Wert den Rückgabewert der Funktion, der der Variablen `result` oder dem Funktionsnamen als letztes zugewiesen wurde.



Im Gegensatz zu `return` in andern Sprachen bricht `result` den Ablauf in einer Routine nicht an dieser Stelle ab.

Der Rückgabewert kann dann an der Aufrufstelle ausgewertet werden (`ergebnis` sei eine Integer-Variable):

```
ergebnis:=SummeAusDrei(3,5,9);
```

forward- und interface-Deklarationen

Prozeduren und Funktionen können nur aufgerufen werden, wenn ihr Name im Code bekannt ist. Und dieser Gültigkeitsbereich beginnt erst an der Stelle der Definition. Soll eine Routine bereits vorher bekannt sein, wird eine **forward**-Deklaration eingesetzt.

Dabei wird lediglich der Kopf einer Routine, versehen mit der Direktive `forward`, an eine frühere Stelle im Code gesetzt. Für die Funktion `SummeAusDrei` sähe das so aus:

```
function SummeAusDrei(zahl1, zahl2, zahl3: integer): integer; forward;
```

Die eigentliche Funktion, wie sie im letzten Abschnitt dargestellt ist, muss dann später im Code folgen.

Soll eine Prozedur oder Funktion auch aus einer anderen Unit aufrufbar sein, muss ihr Kopf im **Interface-Teil** der Unit stehen. Die Definition folgt im Implementation-Teil. Das Verhalten entspricht einer `forward`-Deklaration, die Direktive `forward` darf hierbei aber nicht verwendet werden.

Folgendes Beispiel zeigt eine interface-Deklaration, Implementierung sowie Aufruf einer Funktion:

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type

    TForm1 = class(TForm)
    private
        { Private-Deklarationen }
    public
        { Public-Deklarationen }
    end;
    function SummeAusDrei(zahl1, zahl2, zahl3: integer): integer;
    //Deklaration nur mit dem Kopf der Funktion

var
    Form1: TForm1;

implementation

{$R *.DFM}

function SummeAusDrei(zahl1, zahl2, zahl3: integer): integer;
begin
    result := zahl1 + zahl2 + zahl3;
end;

procedure TForm1.Button1Click(sender: TObject);
var ergebnis: integer;
begin
    ergebnis := SummeAusDrei(3, 5, 9); //Aufruf der Funktion
end;
```

Zu beachten ist: Das Gerüst der zweiten „Prozedur“ (TForm1.Button1Click usw.) haben wir nicht selbst erstellt. Hierbei handelt es sich um eine Ereignisbehandlungsmethode für das Ereignis OnClick eines Buttons. Dazu mehr in unseren Einsteiger-Tutorials. Von uns stammt nur der Funktionsaufruf darin.

Parameter

In den meisten Fällen wird eine Routine zwar öfters gebraucht, allerdings – z. B. bei Berechnungen – nicht immer mit den gleichen Werten. Deshalb gibt es, wie oben bereits gesehen, die Möglichkeit, Routinen Werte beim Aufruf zu übergeben.

Beim Aufruf einer Prozedur/Funktion mit Parametern muss beachtet werden, dass Anzahl und Typ der Werte übereinstimmen.

Anhand der Reihenfolge der Werte steht in obigem Beispiel fest, dass die Variable `zahl1` den Wert 3, `zahl2` den Wert 5 und `zahl3` den Wert 9 erhält. Diese Variablen werden nicht wie üblich über `var` deklariert. Ihre Deklaration erfolgt durch die Nennung im Funktionskopf. Außerdem gelten sie nur innerhalb der Funktion. Von außerhalb (z. B. nach Beendigung der Funktion) kann nicht mehr auf sie zugegriffen werden.

Um das Ganze etwas komplizierter zu machen, gibt es verschiedene Arten von Parametern, die durch `var`, `const` oder `out` gekennzeichnet werden.

Wert- und Variablenparameter

In obigen Beispielen wird immer eine Kopie eines Wertes an die Prozedur/Funktion übergeben. Wenn dieser Wert also innerhalb der Prozedur/Funktion verändert wird, ändert sich *nicht* die Variable, die beim Aufruf verwendet wurde:

```
procedure machwas(zahl: integer);
begin
    zahl:=zahl+5;
end;

procedure aufruf;
var einzahl: integer;
begin
    einzahl:=5;
    machwas(einzahl);
end;
```

Im Beispiel ruft also die Prozedur `aufruf` die Prozedur `machwas` mit dem Wert der Variable `einzahl` auf. In `machwas` wird dieser Wert über `zahl` angesprochen. Und obwohl `zahl` nun verändert wird, ändert sich der Wert in `einzahl` nicht. Er ist am Ende immer noch 5. Man spricht von einem Wertparameter, es wird nur der Inhalt der Variablen übergeben.

Im Fall des Variablenparameters wird das „Original“ übergeben. Ein solcher Parameter wird mit dem Schlüsselwort `var` gekennzeichnet.

```
procedure machwas(var zahl: integer);
begin
    zahl:=zahl+5;
    ...
end;
```



```
procedure aufruf;  
var einzahl: integer;  
begin  
    einzahl:=5;  
    machwas(einzahl);  
end;
```

Hier wird keine Kopie des Variableninhalts übergeben, sondern eine Referenz (also die Speicheradresse) der Variablen `einzahl`. Wird der Wert in `machwas` nun um 5 erhöht, geschieht dies auch mit der Variablen `einzahl`, weil es sich um dieselbe Variable im Speicher handelt. Sie wird nur von den beiden Prozeduren mit anderen Namen angesprochen. Im Gegensatz zum Fall der Wertparameter braucht ein Referenzparameter in einer Prozedur also keinen zusätzlichen Speicherplatz, da die „originale Variable“ weiterverwendet wird.

Über den Umweg des `var`-Parameters kann man sogar Prozeduren dazu bewegen, Werte zurückzugeben:

```
procedure machwas2(var wert1, wert2: integer);  
begin  
    wert1 := 2;  
    wert2 := 3;  
end;  
  
procedure aufrufen;  
var zahl1, zahl2: integer;  
begin  
    machwas2(zahl1, zahl2);  
end;
```

Dass die Variablen `zahl1` und `zahl2` vor der Übergabe an `machwas2` nicht initialisiert wurden, macht nichts, da sie dort sowieso nicht ausgelesen werden. In `machwas2` werden `wert1` und `wert2` Werte zugewiesen - und da es sich dabei um Referenzparameter handelt, automatisch auch `zahl1` und `zahl2`. Wenn `machwas2` abgearbeitet wurde, enthält `zahl1` also den Wert 2 und `zahl2` den Wert 3.

Konstantenparameter

Wird ein übergebener Wert in der Funktion/Prozedur nicht verändert und auch nicht als `var`-Parameter zum Aufruf einer weiteren Routine verwendet, kann man ihn als Konstantenparameter (`const`) deklarieren:

```
procedure machwas(const zahl: integer);
```

Das ermöglicht dem Compiler eine bessere Optimierung, außerdem wird nun nicht mehr zugelassen, dass der Wert innerhalb der Prozedur verändert wird.

Ausgabeparameter

Ausgabeparameter werden mit dem Schlüsselwort **out** deklariert. Wie der Name bereits sagt, können solche Parameter nur zur Zuweisung eines Ausgabewerts verwendet werden. Eine Übergabe von Werten an eine Routine ist damit nicht möglich. Ansonsten entspricht der Ausgabeparameter einem Variablenparameter.

Array-Parameter

Es ist natürlich auch möglich, Arrays als Parameter einer Routine zu verwenden. Jedoch nicht auf die Art, die man intuitiv wählen würde:

```
procedure MachWasAnderes(feld: array [1..20] of integer); //falsch!
```

Stattdessen muss zunächst ein eigener Typ definiert werden. Dieser kann dann in Prozedur- oder Funktionsköpfen verwendet werden:

```
type TMeinFeld = array [1..20] of integer;  
procedure MachWasAnderes(feld: TMeinFeld);
```

Prozeduren und Funktionen überladen

In Delphi ist es möglich, im selben Gültigkeitsbereich mehrere Routinen mit identischem Namen zu deklarieren. Dieses Verfahren wird „Überladen“ genannt.

Überladene Routinen müssen mit der Direktiven **overload** deklariert werden und unterschiedliche Parameterlisten haben.

```
function Divide(x, y: real): real; overload;  
begin  
    result := x / y;  
end;  
  
function Divide(x, y: integer): integer; overload;  
begin  
    result := x div y;  
end;
```

Diese Deklarationen definieren zwei Funktionen namens `Divide`, die Parameter unterschiedlicher Typen entgegennehmen.

Wenn `Divide` aufgerufen wird, ermittelt der Compiler die zu verwendende Funktion durch Prüfung des übergebenen Parametertyps.

`Divide(6.0, 3.0)` ruft beispielsweise die erste `Divide`-Funktion auf, da es sich bei den Argumenten um reelle Zahlen handelt, auch wenn der Nachkommateil Null ist.

Überladene Methoden müssen sich deshalb entweder in der Anzahl der Parameter oder in den Typen dieser Parameter signifikant unterscheiden.

Prozeduren und Funktionen abbrechen

Nach dem Ausführen einer Prozedur bzw. Funktion wird die Programmausführung an der aufrufenden Stelle fortgesetzt. Wenn man dort aber weitermachen will, bevor die Prozedur/Funktion vollständig ausgeführt wurde, kann man `exit` verwenden. **exit** bricht eine Prozedur/Funktion ab und setzt das Programm an der aufrufenden Stelle fort. Bei Funktionen ist darauf zu achten, dass bereits ein Rückgabewert definiert wurde.

10. Programmaufbau

Projektdatei: Anwendung

Eine in Delphi erstellte Anwendung (auch Projekt genannt) besteht aus einer Projektdatei mit der Endung `.dpr`, die das Hauptprogramm enthält, und evtl. einer oder mehreren Units. Der Aufbau der Projektdatei sieht so aus, wenn es sich um eine Anwendung mit grafischer Benutzeroberfläche (GUI) handelt:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Das Schlüsselwort **program** legt fest, dass aus dem Projekt nach dem Compilieren eine ausführbare Anwendung wird. Über Datei/Neu/Konsolenanwendung ist es jedoch auch möglich ein Programm mit einer DOS-ähnlichen Oberfläche zu erstellen. Dabei hat die dpr-Datei folgenden Aufbau:

```
program Project2;
{$APPTYPE CONSOLE}
uses sysutils;

begin
  // Hier Anwender-Code
end.
```

Jedes Delphi-Projekt besteht also aus einem Hauptprogramm, das sich in der Projektdatei befindet. Erstellt man Programme mit einer grafischen Oberfläche, muss diese Projektdatei normalerweise nicht bearbeitet werden. Ansehen kann man sie sich über das Menü „Projekt“ / „Quelltext anzeigen“.

Projektdatei: Library

Beginnt eine Projektdatei mit dem Schlüsselwort **library**, so wird nach dem Compilieren eine DLL (Dynamic Link Library unter Windows) bzw. `.so` (Shared Object unter Linux) daraus. Eine solche Bibliothek enthält Programmcode, der von anderen Anwendungen verwendet werden kann. Mehr darüber erfahren Sie in unserem Tutorial zu dem Thema.

Projektdatei: Package

Packages stellen eine Sammlung von Units dar und unterscheiden sich im Prinzip nicht von DLLs. Als Quelldatei hat die Projektdatei nicht die Endung .dpr, sondern .dpk für Delphi Package. Sie beginnt mit dem Schlüsselwort **package**. Der Aufbau sieht wie folgt aus:

```
package <Name>;

requires
  <Liste der benötigten externen Packages>

contains
  <Liste der Units, die zu dem Package gehören>

end.
```

Packages haben unter Win32 die Endung .bpl (Borland Package Library). Unter .NET sind es Assemblies mit der Endung .dll.

Units

Um nicht allen Code in eine einzige Datei schreiben zu müssen, gibt es das Konzept der Units. Der Entwickler kann seinen Code, z.B. nach Aufgaben geordnet, in verschiedene Programmmodule aufteilen. Diese Programmmodule (Units, auch Bibliotheken genannt) haben die Dateiendung .pas und folgenden Aufbau:

```
unit <Name der Unit>;

interface

uses <Liste der verwendeten Units>;

<Interface-Abschnitt zur Deklaration von Prozeduren, Funktionen usw.>

implementation

uses <Liste der verwendeten Units>

<Implementation-Abschnitt - der eigentliche Programmcode>

initialization
<Programmcode, der bei der Initialisierung ausgeführt wird>

finalization
<Programmcode, der beim Beenden ausgeführt wird>

end.
```

Solch ein Grundgerüst erhält man, wenn man in Delphi über das Menü Datei Neu/Unit auswählt.

Jede Unit beginnt in der ersten Zeile mit dem Schlüsselwort **unit**. Dahinter folgt der Name der Unit, der nicht von Hand bearbeitet werden darf. Er entspricht dem Dateinamen (ohne die Endung pas) und wird von Delphi beim Speichern der Unit automatisch angepasst.

Nun folgen zwei Schlüsselwörter, die jeweils einen neuen Abschnitt einleiten: der **interface**- und der **implementation**-Teil. Eine Unit endet mit dem Schlüsselwort **end** gefolgt von einem Punkt.

Wie im Kapitel über Prozeduren und Funktionen beschrieben, wird im **interface**-Teil lediglich der Kopf der im **implementation**-Teil befindlichen Prozeduren und Funktionen aufgeführt. So sieht man auf einen Blick, was sich in einer Unit alles befindet.

Als Beispiel schreiben wir nun eine Unit, die lediglich eine Funktion zur Mehrwertsteuerberechnung enthält:

```
unit Unit1;

interface
    function Brutto(netto: real): real;

implementation

function Brutto(netto: real): real;
begin
    result := netto * 1.16;
end;

end.
```

Wird einer Anwendung ein neues Fenster hinzugefügt, so gehört dazu immer auch eine Unit. Dagegen kann man zur besseren Strukturierung seines Codes beliebig viele Units einsetzen, die nicht mit einem Fenster in Verbindung stehen.

Beim Compilieren wird aus jeder .pas-Datei eine .dcu-Datei (Delphi Compiled Unit) bzw. eine .dcul (Delphi Compiled Unit Intermediate Language) in .NET erzeugt.

Die *Unit System* wird automatisch in jede Unit und jedes Hauptprogramm eingebunden, ohne dass sie extra erwähnt wird. Auf alle dort definierten Routinen kann also jederzeit zugegriffen werden.

Units verwenden

Mit solch einer Unit alleine kann man nicht viel anfangen. Wir müssen sie in eine Anwendung einbinden. Um die Funktion `Brutto` jedoch aus einer anderen Unit oder dem Hauptprogramm aufrufen zu können, müssen wir sie dort bekannt machen. Das geschieht über das Schlüsselwort **uses**, das bereits im ersten Beispiel zur Projektdatei oben zu sehen ist. Wichtig ist hierbei, dass jede Unit innerhalb eines Projekts einen eindeutigen Namen haben muss. Man kann nicht in `Unit1` eine weitere `Unit1` einbinden. Deshalb speichern wir unsere Beispiel-Unit unter dem Namen „beispiel.pas“. Die erste Zeile ändert sich nun automatisch in `unit beispiel;`.

Nun legen wir über das Datei-Menü eine neue Anwendung an. In die Unit1 binden wir unsere Unit ein:

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    Beispiel;

type

    TForm1 = class(TForm)
    private
        { Private-Deklarationen }
    public
        { Public-Deklarationen }
    end;

var

    Form1: TForm1;

implementation

    {$R *.DFM}

end.
```

Nun können wir in der gesamten Unit unsere Funktion `Brutto` verwenden, als wäre die Funktion in der gleichen Unit implementiert. Gibt es in einer anderen eingebunden Unit eine Funktion/Prozedur gleichen Namens, muss zuerst der Unit-Namen genannt werden, um klarzustellen, welche Funktion gemeint ist, z.B. `ergebnis := Beispiel.Brutto(1500);`

Positionen der uses-Klausel

Werden Units eingebunden, die nicht zu dem Projekt gehören, wie z. B. die Delphi-Standardunits, werden sie in der uses-Klausel im Interface-Teil eingefügt.

Soll jedoch eine andere Unit desselben Projekts eingebunden werden, wird am Anfang des implementation-Abschnitts eine weitere uses-Zeile eingefügt.

Haben wir z. B. ein Projekt mit zwei Formularen und den dazugehörigen Units `unit1` und `unit2`, und soll `Form2` (in `unit2`) aus `Form1` aufgerufen werden, so braucht die `Unit1` Zugriff auf die `Unit2`. Dazu wechseln wir in die Anzeige von `Unit1` und klicken im Datei-Menü von Delphi auf „Unit verwenden“. In dem erscheinenden Fenster sind alle Units des Projekts aufgelistet, die von der aktuellen Unit noch nicht verwendet werden. Hier wählen wir „Unit2“ aus und schließen das Fenster. Delphi hat nun automatisch eine Zeile direkt am Anfang des implementation-Abschnitts eingefügt. Folgendes

Beispiel zeigt diesen Fall, wobei Form1 einen Button (Button1) enthält, auf dessen Klick Form2 geöffnet wird:

```
unit Unit1;

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

uses Unit2;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;

end.
```

Die Verwendung der zweiten `uses`-Klausel hat ihren Grund darin, dass zwei Units sich nicht gegenseitig im `uses`-Abschnitt des Interface einbinden können (zirkuläre Unit-Referenz). Im Implementation-Teil ist dies jedoch möglich.

Allgemein gilt, dass Units, deren Routinen nur für den `implementation`-Abschnitt benötigt werden, auch im `implementation`-Abschnitt eingebunden werden. Wird dagegen ein Teil einer anderen

Unit (z.B. ein selbstdefinierter Datentyp) bereits im `interface`-Abschnitt benötigt, muss die Unit natürlich schon dort eingebunden werden.

interface und implementation

Der `interface`-Abschnitt (zu deutsch „Schnittstelle“) dient dazu, Funktionen, Prozeduren, Typen usw. dieser Unit anderen Units zur Verfügung zu stellen. Alles, was hier steht, kann von außen verwendet werden. Bei Prozeduren und Funktionen steht hier nur der Kopf der Routine (Name, Parameter und evtl. Rückgabewert). Die Definition folgt dann im `implementation`-Abschnitt. Der `interface`-Abschnitt endet mit Beginn des `implementation`-Abschnitts.

initialization und finalization

Bei Bedarf können am Ende einer Unit noch zwei Abschnitte stehen: `initialization` und `finalization`.

`Initialization` muss dabei als erstes aufgeführt werden. Hier werden alle Befehle aufgeführt, die bei Programmstart der Reihe nach ausgeführt werden sollen. Danach folgt das Ende der Unit (`end.`) oder der `finalization`-Abschnitt. Dieser ist das Gegenstück zu `initialization`. Hier können z. B. vor Programmende Objekte freigegeben werden.

Der Aufbau einer Units sieht dann so aus:

```
unit Unit1;  
  
interface  
  
implementation  
  
initialization  
  
finalization  
  
end.
```

`finalization` kann nur verwendet werden, wenn es auch einen `initialization`-Abschnitt gibt; `initialization` kann jedoch auch ohne `finalization` vorkommen. Eine Unit funktioniert allerdings auch ohne diese Abschnitte.

Namespaces

Im .NET-Framework wird das Konzept der Namespaces eingeführt. Hierdurch kann eine Struktur im Sinne von „ist enthalten in“ über den Namen angelegt werden. Einem Programm kann beispielsweise der Name `MeineFirma.Programme.MeinProgramm` gegeben werden. Diese Namen werden beim Speichern einer Datei vergeben. Einem Projekt kann über die Projektoptionen (Bereich Verzeichnisse/Bedingungen) ein Standard-Namespace vergeben werden. Unitnamen wird beim Compilieren dann automatisch der Namespace vor den Unit-Namen gesetzt. Aus `Unit1.pas` wird also `MeineFirma.Programme.MeinProgramm.Unit1.dcuil`.

Beim Einbinden einer Unit in eine andere Unit muss der voll qualifizierte Namen angegeben werden:

```
uses MeineFirma.Programme.MeinProgramm.EineUnit;
```

Wird nur der generische Namen (im Beispiel `EineUnit`) verwendet, wird nach der Unit in einer festgelegten Reihenfolge gesucht. Begonnen wird damit im aktuellen Unit-Namensraum, es folgt der in den Projektoptionen eingestellte Standard-Namespace. In der Projektdatei kann eine `namespaces`-Klausel verwendet werden, um die Suchreihenfolge in Namensräumen zu beeinflussen. Diese Liste wird durchsucht, wenn der Compiler im aktuellen Unit- und Projekt-Namensraum nicht fündig geworden ist.

Ein Namespace muss immer komplett angegeben werden. In obigem Beispiel würde es nicht ausreichen, `uses MeineFirma` zu schreiben, um auf alle Namensräume unterhalb zugreifen zu können.

Die Namensräume sind im Übrigen rein logisch zu sehen. Sie haben nichts mit einer festen Verzeichnisstruktur auf der Festplatte o.ä. zu tun.

11. Einführung in die Objektorientierung

Die objektorientierte Programmierung wurde in den 80er-Jahren entwickelt und soll die unstrukturierte Programmierung (am Anfang ein `begin`, dann Befehl auf Befehl und am Ende ein `end`, dazwischen möglichst viele Sprünge mit `goto`) ablösen.

Die Objektorientierung ist ein sehr komplexes Gebiet und soll hier deshalb nur in einfachen Worten für das grundlegende Verständnis geschildert werden.

Wie der Name bereits sagt, sind Objekte zentrale Elemente der objektorientierten Programmierung. Ein Objekt ist eine Einheit mit einem Zustand und einem Verhalten, was gleich an einem Beispiel verdeutlicht wird. Zuvor müssen jedoch noch ein paar Begriffe geklärt werden.

Klassen

Klassen stellen den „Bauplan“ eines Objekts dar. Sie definieren, welche Eigenschaften/Attribute und welche Methoden ein Objekt besitzt und wie es auf bestimmte Ereignisse reagiert. Klassennamen beginnen in Delphi üblicherweise mit einem großen T (Type). Dabei handelt es sich um eine Vereinbarung, nicht um eine Regel. Soll Programmcode weitergegeben werden, so sollte man sich an diesen Quasi-Standard halten.

Statt „Klasse“ werden gelegentlich auch die Begriffe „Klassentyp“, „Objektklasse“ oder „Objekttyp“ verwendet.

Instanzen

Instanzen sind nun „lebendige“ Objekte, die nach dem Bauplan einer Klasse erstellt wurden. Sie belegen bei der Programmausführung Arbeitsspeicher und können Daten aufnehmen. Von jeder Klasse kann es beliebig viele Instanzen geben, die alle gleich aufgebaut sind, aber unterschiedliche Daten enthalten können.

Objekte

Der Begriff „Objekt“ wird unterschiedlich verwendet. Manche gebrauchen ihn gleichbedeutend mit „Instanz“, andere verwenden ihn als Oberbegriff für Klassen und Instanzen. Beim Auftauchen dieses Begriffs muss also mit allem gerechnet werden.

Methoden

Methode ist der Überbegriff für Prozeduren und Funktionen, die Teil einer Klasse sind. Die Methoden stellen das Verhalten eines Objekts dar.

Attribute oder Felder

Attribute sind mehr oder weniger Variablen, die zu einem Objekt gehören und damit seinen Zustand beschreiben. Attributnamen beginnen in Delphi mit einem großen F (Field). Wie auch beim T vor Klassennamen handelt es sich hierbei um eine Vereinbarung.

Eigenschaften oder Properties

Eigenschaften sind keine eigenständigen Variablen, sie belegen zur Laufzeit keinen Speicherplatz. Über sie lassen sich Lese- und Schreibzugriffe auf Attribute regeln. Die Eigenschaften sind es auch, die im Objektinspektor angezeigt werden.

Doch nun ein **Beispiel einer Klassendefinition**:

```
type
  TAuto = class
  private
    FFarbe: string;
    FBaujahr: integer;
    procedure SetFarbe(Farbe: string);
  public
    property Farbe: string read FFarbe write SetFarbe;
  end;
```

Eine solche Klassendefinition kann überall dort stehen, wo auch Typ-Deklarationen vorgenommen werden können.

Die Klasse `TAuto` stellt nun den Bauplan für beliebig viele Instanzen dar. `FFarbe` und `FBaujahr` sind Attribute, wobei es zum guten Ton gehört, Attribute immer in den `private`-Abschnitt zu schreiben. Was diese Zugriffsrechte bedeuten, wird unter „Zugriff auf Objekte“ erläutert.

Zuerst wollen wir noch Instanzen von unserer Beispielklasse erzeugen.

Objektreferenzen

Variablen, deren Typ eine Klasse ist, heißen „Objektreferenzen“, z. B.

```
var MeinAuto: TAuto;
```

Dies deklariert eine Variable („Objektreferenz“) von der Klasse `TAuto`.

Die Werte von Objektreferenzen sind Zeigerwerte (Adressen im Hauptspeicher).

Das Deklarieren einer Objektreferenz wie oben reicht jedoch nicht aus, um eine Instanz zu erzeugen. Denn durch die reine Deklaration enthält `MeinAuto` nun den Wert `nil`. Es ist also noch kein Bereich im Hauptspeicher für unsere Autoinstanz reserviert worden. Wir haben lediglich mit dem Compiler vereinbart, dass es sich um etwas, das dem Bauplan von `TAuto` entspricht, handelt, wenn wir die Variable `MeinAuto` verwenden.

12. Instanzen erzeugen: Konstruktoren

Instanzen einer Klasse entstehen erst dadurch, dass sie ausdrücklich erzeugt werden. Dies geschieht mit dem Konstruktor **Create**. „Konstruktor“ ist ein Fremdwort und bezeichnet jemanden, der etwas konstruiert, also erzeugt. Man könnte `create` auch eine Methode nennen; es gibt jedoch einen großen Unterschied: Methoden können nur aufgerufen werden, wenn die Instanz bereits existiert. Der Konstruktor dagegen erzeugt die Instanz aus dem Nichts.

Der Konstruktor muss nicht von uns programmiert werden, da alle Klassen, die wir erfinden, automatisch den „Vorfahr“ `TObject` haben und damit alle grundlegenden Dinge von diesem erben. Weitere Informationen über `TObject`, den „Vater“ aller Delphi-Objekte, finden sich in der Delphi-Hilfe.

Und so wird eine Instanz erzeugt:

```
Objektreferenz := Klasse.Create;
```

In unserem konkreten Fall also:

```
MeinAuto := TAuto.Create;
```

Damit wird Speicherplatz für alle Attribute einer Auto-Instanz im Hauptspeicher reserviert und die zugehörige Adresse in der Variablen `MeinAuto` gespeichert.

Beim Erzeugen einer Klasse werden die jeweils vorhandenen Attribute mit folgenden Startwerten belegt:

- Alle Datenfelder mit einem ganzzahligen Datentyp (z.B. Integer) werden mit 0 initialisiert.
- Alle Datenfelder mit einem String-Typ werden durch eine leere Zeichenkette initialisiert.
- Alle Datenfelder mit einem Zeigertyp werden mit dem Wert `nil` initialisiert.
- Alle anderen Datenfelder bleiben undefiniert!

Wie auf Attribute und Methoden der neu erzeugten Instanz zugegriffen wird, folgt unter „Zugriff auf Objekte“.



Genau genommen ist es nicht `create`, das eine neue Instanz einer Klasse erzeugt. Vielmehr handelt es sich bei `create` um eine Methode, die die Klassenmethode `NewInstance` aufruft. Diese reserviert den Speicher auf dem Heap. Anschließend wird der Code in `create` zur Initialisierung ausgeführt. `create` kann übrigens auch an einer Objektreferenz aufgerufen werden. Dadurch wird dann keine neue Instanz erzeugt, sondern es wird das Objekt neu initialisiert.

Instanzen freigeben: Destruktoren

Wird eine Instanz einer Klasse nicht mehr benötigt, sollte der dafür verwendete Hauptspeicher wieder freigegeben werden. Dies geschieht mit dem Gegenstück zum Konstruktor, dem Destruktor („Zerstörer“). `Destroy` heißt dieser bei von `TObject` abgeleiteten Klassen. Um ein Objekt freizugeben, sollte jedoch die Methode `Free` verwendet werden. `Free` prüft, ob ein Verweis auf `nil` vorliegt, anschließend wird `Destroy` aufgerufen. Die Objektreferenz verweist jetzt allerdings nicht unbedingt auf `nil`. Soll das der Fall sein, kann auf die Prozedur `FreeAndNil` (Unit `SysUtils`) zurückgegriffen werden, die das freizugebende Objekt als Parameter erwartet – es handelt sich hier nicht um eine Methode.

Unter .NET arbeitet im Hintergrund ein Garbage Collector – eine Art Müllsammler, der alle nicht mehr benötigten Speicherbereiche ermittelt und frei gibt. Wann das genau ist, kann man nicht festlegen. Vorteil für den Programmierer: Er kann nicht mehr vergessen, selbst erstellte Objekte wieder freizugeben und damit Speicherlöcher zu produzieren. Stattdessen kümmert sich die .NET-Laufzeitumgebung (CLR) darum.

Dennoch sollte der Aufruf von `free` erfolgen, um andere Ressourcen freizugeben. Der Garbage Collector kümmert sich nur um den Speicher.

13. Zugriff auf Objekte

Zur Erinnerung noch einmal unsere Auto-Klasse von vorhin:

```
type
  TAuto = class
  private
    FFarbe: string;
    FBaujahr: integer;
    procedure SetFarbe(Farbe: string);
  public
    property Farbe: string read FFarbe write SetFarbe;
  end;
```

Diese Klasse enthält folgende Klassenkomponenten:

- Datenfelder (Attribute), hier: FFarbe, FBaujahr
- Methoden, hier: SetFarbe
- Eigenschaften (Property), hier: Farbe

Diese Komponenten sind im Bauplan (der Klasse) vorgegeben, können jedoch in allen Instanzen dieser Klasse unterschiedliche Werte haben. Ein Zugriff auf diese Komponenten ist deshalb nur über die Namen der Instanzen (Objektreferenzen) sinnvoll und möglich.

Wurde also über

```
var MeinAuto: TAuto;
```

eine Objektreferenz definiert und die Instanz mit `create` erzeugt, lassen sich die Komponenten folgendermaßen ansprechen:

```
var MeinAuto: TAuto;
begin
  MeinAuto := TAuto.Create;
  MeinAuto.FFarbe := 'rot';
  MeinAuto.FBaujahr := '1980';
  MeinAuto.SetFarbe('rot');
  MeinAuto.Farbe := 'rot';
```

Zugriffsrechte

Nun kommen wir zu den Zugriffsrechten, in unserem Fall **private** und **public**.

Private Komponenten dürfen nur innerhalb der Unit angesprochen werden, in dessen Interface-Teil die betreffende Klasse definiert wurde.

Öffentliche (`public`) Komponenten dürfen darüber hinaus auch überall dort angesprochen werden, wo die betreffende Unit (durch `uses`) zur Nutzung bereit gestellt wird.

Dies stellt auch den Grund der kompliziert erscheinenden Verwendung der **property** dar.

Soll ein Attribut (hier die Farbe) aus einer anderen Unit, die möglicherweise ein anderer Programmierer geschrieben hat, heraus verändert werden, kann dies zu Problemen führen. Die Zeile

```
property Farbe: string read FFarbe write SetFarbe;
```

bedeutet, dass „Farbe“ wie ein normales Attribut verwendet werden kann. Da es im `public`-Abschnitt steht, ist auch ein Zugriff aus anderen Units möglich. Wird der Wert von Farbe ausgelesen (`read`) gibt die Instanz automatisch den Wert von `FFarbe` zurück. Soll dagegen die Eigenschaft „Farbe“ neu gesetzt werden (`write`), wird `SetFarbe`, die Prozedur im `private`-Teil, aufgerufen. In ihr könnte nun geprüft werden, ob der übergebene Wert beispielsweise eine gültige Farbe für dieses Auto darstellt. Wenn ja, kann die Prozedur das Datenfeld `FFarbe` setzen:

```
procedure TAuto.SetFarbe(Farbe: string);  
begin  
    if (farbe = 'rot') or (farbe = 'blau') or (farbe = 'gruen') then  
        FFarbe := Farbe;  
end;
```

Darüber hinaus gibt es weitere Sichtbarkeitsattribute:

private	Ein <code>private</code> -Element kann nur innerhalb der gleichen Unit verwendet werden. Aus anderen Units ist ein Zugriff nicht möglich.
protected	Ein <code>protected</code> -Element ist wie ein <code>private</code> -Element innerhalb der gleichen Unit verwendbar. Darüberhinaus haben alle abgeleiteten Klassen darauf Zugriff, unabhängig davon, in welcher Unit sie sich befinden.
public	<code>public</code> -Elemente unterliegen keinen Zugriffsbeschränkungen.
published	<code>published</code> -Elemente haben dieselbe Sichtbarkeit wie <code>public</code> -Elemente. Zusätzlich können diese Element jedoch z. B. auch im Objektinspektor angezeigt werden, weshalb nicht alle Typen als <code>published</code> -Element eingesetzt werden können.
automated	<code>automated</code> ist nur noch aus Gründen der Abwärtskompatibilität vorhanden. Der Einsatz erfolgte in Zusammenhang mit OLE-Automatisierungsobjekten und nur in Klassen, die von <code>TAutoObject</code> (Unit <code>OleAuto</code>) abgeleitet waren. Für <code>TAutoObject</code> aus der Unit <code>ComObj</code> wird <code>automated</code> nicht mehr verwendet.

In Delphi 8 für .NET neu hinzugekommen sind folgende Attribute:

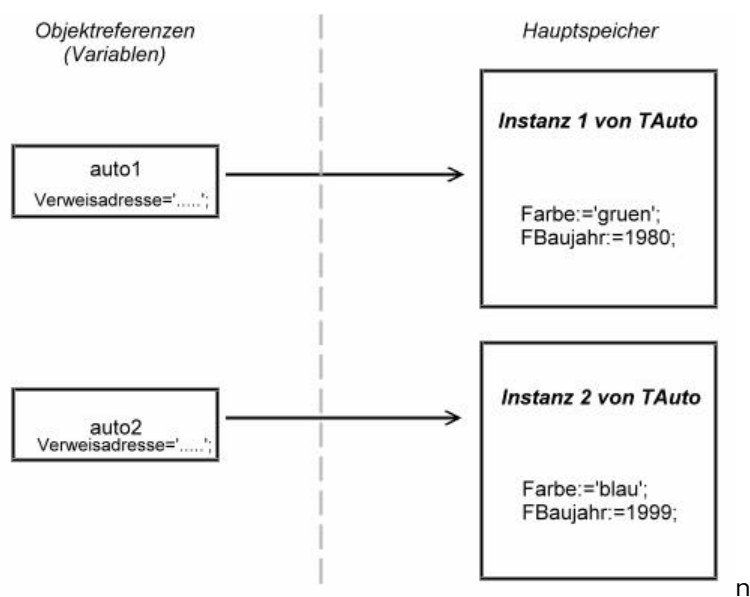
strict private	<code>strict private</code> -Elemente sind nur innerhalb der Klasse sichtbar. Andere Klassen können nicht darauf zugreifen, auch wenn sie sich in derselben Unit befinden.
strict protected	<code>strict protected</code> -Elemente sind innerhalb der Klasse und in allen davon abgeleiteten Klassen sichtbar, jedoch nicht in anderen (nicht verwandten) Klassen derselben Unit.

Zuweisungen für Objektreferenzen

Angenommen wir haben zwei unterschiedliche Auto-Instanzen in unserem Arbeitsspeicher:

```
var auto1, auto2: TAuto;  
...  
...  
auto1 := TAuto.Create;  
auto2 := TAuto.Create;
```

Beide sollen verschiedene Werte enthalten. Wir haben also folgende Situation:

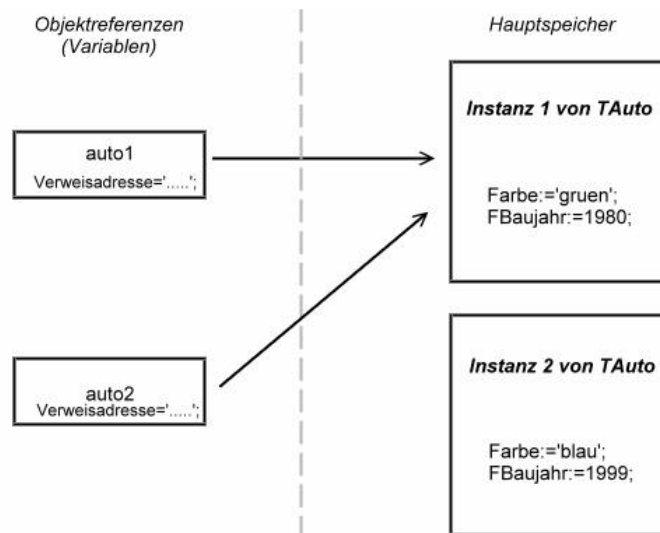


Was passiert nun, wenn wir die Zuweisung

```
auto2 := auto1;
```

vornehmen? Möglich ist das auf jeden Fall, da beide den gleichen Aufbau (nach dem Bauplan der Klasse TAuto) haben.

Im Arbeitsspeicher sieht es aber so aus:



D. h. im Hauptspeicher ändert sich gar nichts. Allerdings zeigen jetzt beide Referenzen auf das zuerst angesiedelte Objekt, das bisher **nur** unter dem Namen `auto1` bekannt war. Es wurde nur die Verweisadresse (wo sich die Instanz im Arbeitsspeicher befindet) von `auto2` auf die von `auto1` gesetzt.

Die zweite Instanz existiert immer noch, besitzt allerdings keine Referenz mehr, so dass ihr Platz im Arbeitsspeicher nicht mehr freigegeben werden kann (Speicherloch).

14. Felder

Bei Feldern handelt es sich um Variablen, die Teil eines Objekts sind. Genau wie Variablen können auch sie jeden Typ annehmen. Es gehört zum „guten Ton“, Felder immer im `private`-Teil einer Klasse zu deklarieren. Prinzipiell ist aber auch eine andere Position innerhalb der Klasse möglich. Jedoch müssen die Felddeklarationen vor den Methodendeklarationen stehen.

Beispiel:

```
type
  TAuto = class
  private
    FFarbe: string;
    FBaujahr: integer;
    procedure SetFarbe(Farbe: string);
  public
    property Farbe: string read FFarbe write SetFarbe;
  end;
```

Nach der Vereinbarung im Object Pascal-Styleguide beginnen alle Feldnamen mit einem großen F. Vom Compiler wird das allerdings nicht erzwungen. Es dient nur der Übersichtlichkeit.

Klassenfelder

Seit Delphi 8 für .NET gibt es in Delphi Klassenfelder. Diese sind nicht über einen Instanznamen, sondern über den Namen der Klasse ansprechbar und gelten deshalb für alle Instanzen dieser Klasse. Deklariert werden Klassenfelder über `class var`:

```
type
  TAuto = class
  public
    class var
      Eigenschaft: Integer
```

15. Methoden

Bei Methoden handelt es sich um Prozeduren und Funktionen, die zu einer Klasse gehören.

Deklaration, Implementierung und Aufruf

Methoden werden mit ihrem Kopf innerhalb der Klasse deklariert. An einer späteren Stelle im Code folgt die Implementierung der Methode. Diese erfolgt wie bei einer normalen Prozedur oder Funktion, außer dass vor den Methodennamen der Name der Klasse - getrennt durch einen Punkt - geschrieben wird.

Deklaration (im interface- oder implementation-Abschnitt einer Unit):

```
type
  TAuto = class
  private
    FFarbe: string;
    FBaujahr: integer;
    procedure SetFarbe(Farbe: string);
  public
    property Farbe: string read FFarbe write SetFarbe;
  end;
```

Implementierung (im implementation-Abschnitt einer Unit):

```
procedure TAuto.SetFarbe(Farbe: string);
begin
  ...
end;
```

Der Aufruf erfolgt dann am Name einer Instanz der Klasse:

```
var auto: TAuto;
...
auto.SetFarbe;
```

Self

Von einer Klasse können mehrere Instanzen gleichzeitig existieren. Innerhalb einer Methode kann über den Bezeichner `Self` auf die aufrufende Instanz zugegriffen werden. Beim Aufruf von weiteren Methoden derselben Klasse oder beim Zugriff auf Felder, kann `Self` auch weggelassen werden. `Self` steht innerhalb einer Methode also für den Namen des Objekts.

```
procedure TAuto.SetFarbe(Farbe: string);
begin
  FFarbe := 'rot';
  self.FFarbe := 'rot'; //gleichbedeutend
```



In Klassenmethoden enthält `Self` die Klasse. Über `Self.Create` kann man deshalb eine Instanz der Klasse erzeugen.

Sender

Wird über den Objektinspektor einer Komponente ein Ereignis zugeordnet, wird automatisch das Gerüst einer Ereignisbehandlungsmethode erzeugt. Für das `OnClick`-Ereignis von `Button1` auf `Form1` zum Beispiel:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  
end;
```

`Self` steht hier für `form1`, da `Button1Click` eine Methode von `TForm1` ist. Und `TForm1` hat nur die eine Instanz `form1`. Will man dagegen wissen, welche Komponente das Ereignis ausgelöst hat - es könnten ja mehrere Buttons mit dieser Ereignisbehandlungsmethode verknüpft sein -, verwendet man den Parameter `Sender`:

```
if Sender = Button1 then ...
```

Klassenmethoden

Methoden werden normalerweise an einer Instanz (einem Objekt) aufgerufen. Bei Klassenmethoden ist das anders. Wie der Name bereits sagt, werden diese an einer Klasse aufgerufen. Definiert werden Klassenmethoden wie normale Methoden, jedoch mit dem Zusatz `class`:

```
type  
  TKlasse = class(TObject)  
  public  
    class function GetInfo: string;  
  end;  
  
class function TKlasse.GetInfo: string;  
begin  
  ...  
end;
```

Botschaftsmethoden

Um auf Botschaften (eigene oder die vom Betriebssystem) zu reagieren, werden Botschaftsmethoden angelegt. Eine Botschaftsmethode hat immer einen einzigen `var`-Parameter. In Deklaration steht

dahinter die Direktive **message** und die ID der Botschaft. Folgende Methode reagiert auf die Windows-Botschaft WM_CHAR:

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ...
  end;
```

Weitere Informationen

Weitere Informationen über Methoden sind in den Abschnitten „Verbergen, überschreiben, überladen“ sowie „Virtuelle, dynamische und abstrakte Methoden“ zu finden.

16. Eigenschaften

Eigenschaften (properties) ermöglichen die Zugriffssteuerung auf Felder. Eigenschaften selbst enthalten keine Werte, sondern geben an, wie beim lesenden oder schreibenden Zugriff auf ein Feld vorzugehen ist.

Eigenschaften werden sinnvollerweise im `public`- oder `published`-Abschnitt einer Klasse definiert:

```
type TAuto = class
  private
    FFarbe: string;
    FBaujahr: integer;
    procedure SetFarbe(Farbe: string);
  public
    property Farbe: string read FFarbe write SetFarbe;
end;
```

Die Verwendung von Eigenschaften ist bereits im Abschnitt „Zugriff auf Objekte“ angesprochen worden.

Eine Eigenschaft verfügt über eine `read`- oder eine `write`-Angabe oder über beide. Dahinter folgt dann entweder direkt der Name des Feldes, auf das lesend oder schreibend zugegriffen werden soll, oder der Name einer Methode, die den Zugriff steuern soll.

Wird für `read` eine Methode verwendet, darf diese keinen Parameter haben und muss einen Rückgabewert liefern, der dem Typ der Eigenschaft entspricht. Bei einem Feld muss dieses natürlich auch den Typ der Eigenschaft haben.

Eine Methode für `write` muss genau einen Aufrufparameter besitzen, der ebenfalls dem Typ der Eigenschaft entspricht. Hinter `read` und `write` selbst wird jedoch immer nur der reine Methodenname ohne Parameter oder Rückgabewert angegeben.

Beispiel:

```
property Farbe: string read GetFarbe write SetFarbe;
...
function TAuto.GetFarbe: string;
procedure TAuto.SetFarbe(wert: string);
```

Enthält eine Eigenschaft nur eine `read`-Angabe, kann sie nur gelesen werden; enthält sie nur eine `write`-Angabe, kann sie nur geschrieben werden. Im jeweils anderen Fall tritt ein Fehler auf.

Properties können nicht als `var`-Parameter eingesetzt werden, da es sich bei ihnen ja nicht um direkte Zugriffe auf Felder handeln muss; sondern es können ja „nur“ Methoden dahinter stecken.

Array-Eigenschaften

Eigenschaften können auch wie Arrays arbeiten. Die Syntax sieht in einem solchen Fall wie folgt aus:

```
property Adressen[index: integer]: string read GetAdresse write SetAdresse;  
...  
function TKlasse.GetAdresse(index: integer): string;  
procedure TKlasse.SetAdresse(index: integer; wert: string);
```

Klasseneigenschaften

Seit Delphi 8 für .NET gibt es sog. Klasseneigenschaften. Diese Eigenschaften sind statisch und werden nicht an einer Instanz verwendet, sondern gelten für die Klasse. Deklariert werden sie im `class var`-Abschnitt einer Klassendefinition.

17. Funktions- und Methodenzeiger

Funktionen und Prozeduren können selbst in Variablen gespeichert werden (bzw. ein Verweis auf sie). Somit ist es möglich, einer Funktion eine andere Funktion als Parameter zu übergeben.

```
type
  TMeineFunktion = function: Integer;
```

In diesem Fall haben die Funktionen oder Prozeduren keinen Namen, sondern bestehen aus dem Schlüsselwort `function` oder `procedure`, bei Bedarf gefolgt von Parametern in runden Klammern und einem Rückgabewert. Handelt es sich um Methoden, folgt noch „of object“:

```
TMeineMethode = procedure of object;
```

Praktischer Nutzen der Sache: Man kann somit eine Ereignisbehandlungsroutine erstellen und erst zur Laufzeit einem Objekt (einer Komponente) zuordnen.

Beispiel:

```
type
  TMyOnClick = procedure(Sender: TObject) of object;
  TMyForm = class(TForm)
    procedure MyButtonClickEvent(Sender: TObject);
  end;

var
  MyOnClick: TMyOnClick;
  MyForm: TMyForm;
  ...
  MyOnClick := MyForm.MyButtonClickEvent;
```

Die OnClick-Events der Komponenten haben ebenfalls einen Methodenzeiger als Typ, den gleichen wie TMyOnClick im Beispiel. Deshalb ist es möglich, z.B. einem Button zur Laufzeit eine Methode mit dem gleichen Parameter zuzuweisen. Variablen vom Typ einer Funktion/Prozedur beinhalten die Speicheradresse, also einen Zeiger auf die Routine. Bei Methoden werden zwei Zeiger gespeichert: die Adresse der Methode und eine Referenz auf die Instanz.

Delegates

Das Konzept der Delegates in .NET entspricht diesen Funktions- und Methodenzeigern. Das Schlüsselwort `delegate` wird von Delphi nicht unterstützt.

Multicast-Events

In .NET ist es möglich, einem Event mehr als einen Eventhandler zuzuordnen. Die Zuordnung weiterer Methodenzeiger zu einem Event erfolgt über die Funktion `include`. Sie erwartet als ersten Parameter einen Event-Namen und als zweiten den Zeiger auf eine passende Methode. Das Gegenstück dazu ist die Funktion `exclude`.

18. Vererbung

Wir kommen nun zu einem wichtigen Punkt beim Arbeiten mit Klassen: der Vererbung.

Bisher haben wir unsere Klassen aus dem „Nichts“ komplett neu aufgebaut. Dennoch kannten sie bereits einige vordefinierte Befehle, die wir nicht programmiert haben wie z.B. `create` und `destroy`. Das liegt daran, dass in Delphi alle Klassen automatisch Nachkommen von `TObject` sind. Und wie im wirklichen Leben erben die Nachkommen das Verhalten ihrer Vorfahren.

Wird also ein Befehl an einer Instanz aufgerufen, der in der Klasse gar nicht definiert ist, sieht der Compiler beim direkten Vorfahren der Klasse nach usw., bis er den Befehl findet. Taucht er bei keinem der Vorfahren bis `TObject` auf, gibt es eine Fehlermeldung. Natürlich betrifft das nicht nur Methoden, sondern auch Attribute und Eigenschaften.

Wozu das Ganze gut sein kann, soll das folgende Beispiel zeigen.

```
type TMensch = class
  private
    FVorname: string;
    FNachname: string;
    FGeburtstag: TDate;
    FGeschlecht: string;
  end;

  TBerufstaetig = class(TMensch)
    private
      FKontoNr: integer;
      FBankleitzahl: integer;
    public
      procedure GehaltZahlen;
    end;

    TManager = class(TBerufstaetig)
      private
        FGehalt: real;
        FZulagen: real;
        FAnzahlMitarbeiter: integer;
      end;

      TSchueler = class(TMensch)
        private
          FKlasse: integer;
          FTaschengeld: real;
        end;
```

Hinter dem Schlüsselwort **class** steht in Klammern der Name des direkten Vorfahren. `TBerufstaetig` erbt also alle Attribute wie `FVorname` und `FNachname` von der Klasse `TMensch` und fügt noch eigene Attribute hinzu, die eben nur für Berufstätige gebraucht werden. `TManager` erbt von `TBerufstaetig` und somit auch von `TMensch`.

Wird von `TManager` eine Instanz gebildet

```
var Chef: TManager;  
...  
Chef := TManager.Create;
```

sind alle folgenden Aufrufe möglich:

```
Chef.FNachname := 'Schmidt';  
Chef.FBankleitzahl := 12345670;  
Chef.FGehalt := 10000;
```

obwohl alle drei Attribute in verschiedenen Klassen deklariert wurden.

Steht hinter dem Schlüsselwort keine Klasse (wie bei `TMensch`), wird `TObject` als direkter Vorfahr genommen.

Im Prinzip gilt: Alle Gemeinsamkeiten werden in einer Oberklasse (Basisklasse) zusammengefasst. Die Nachfahren (abgeleitete Klassen) enthalten dann nur noch Methoden und Attribute, in denen sie sich unterscheiden. Gäbe es keine Vererbung, müssten lauter gleichrangige Klassen erstellt werden, die größtenteils den gleichen Code enthalten.

Versiegelte Klassen

In .NET gibt es die Möglichkeit, dass ein Entwickler festlegen kann, dass eine Klasse nicht weiter abgeleitet werden darf. Dafür kennzeichnet er die betroffene Klasse als **sealed** (versiegelt):

```
TSchueler = class sealed (TMensch)  
  private  
    FKlasse: integer;  
    FTaschengeld: real;  
end;
```

Zuweisungskompatibilität von Objektreferenzen

Wie schon im Teil „Zugriff auf Objekte“ gesehen, ist das Zuweisen von Objektreferenzen möglich (`auto1 := auto2`). In diesem Fall waren beide Instanzen der gleichen Klasse. Kommt Vererbung ins Spiel, wird die Sache etwas komplizierter. Folgendes ist bei Zuweisungen möglich:

- Beide Seiten besitzen denselben Klassentyp (`auto1 := auto2`)
- Die rechte Seite referenziert einen Nachkommen der linken Seite. Beispiel:

```
var m: TMensch;  
    s: TSchueler;  
...  
m := s;
```

Andere Varianten sind nicht möglich! Wenn die Typen ungleich sind, muss die rechte Seite eine Präzisierung der linken Seite sein. An dieser Stelle mag man denken: Wie kann das denn funktionieren? Die Präzisierung hat doch häufig mehr Daten als die Basisklasse, wieso kann man sie dann zuweisen? Das liegt daran, dass wir es hier eben nur mit den Objektreferenzen zu tun haben. Diese sind genau genommen nur Variablen, die eine Zahl beinhalten, nämlich eine Adresse im Hauptspeicher. Die Variablen an sich sind also Zeiger und beinhalten alle die gleiche Datenmenge, nämlich eine Adresse. Deshalb funktioniert die Zuweisung.

19. Polymorphie

Typecasting - is und as

Durch voriges Beispiel ergibt sich allerdings ein Problem. Die Variable `m` ist vom Typ `TMensch`, verweist aber auf einen Schüler. Nach der Definition der Klasse `TSchueler` besitzt diese das Attribut `FTaschengeld`. Die Anweisung `m.FTaschengeld` führt jedoch zu einer Fehlermeldung, da der Compiler die Variable `m` nach wie vor für eine Instanz von `TMensch` hält. Nun wissen wir jedoch, dass `m` inzwischen auf einen Schüler verweist und deshalb auch das Taschengeld kennen muss. Der Delphi-Compiler ist so flexibel, dass wir ihm das mitteilen können. Dazu gibt es den Operator **as**.

```
(m as TSchueler).FTaschengeld
```

Hiermit sagen wir dem Compiler: Nimm das Objekt `m` und gehe davon aus, dass es gerade auf eine `TSchueler`-Instanz verweist. Der Compiler vertraut uns und stellt fest, dass `TSchueler` ja das Attribut `FTaschengeld` hat. Die Anweisung wird also akzeptiert. Löschen wir allerdings die Zuweisung `m:=s`, so dass `m` doch nur auf eine `TMensch`-Instanz und nicht auf einen Schüler verweist, gibt es zur Laufzeit logischerweise eine Bestrafung. Unter der Tatsache, dass der Compiler nicht vorhersagen kann, von welcher Klasse ein Objekt nun wirklich ist, das sich hinter einer Objektreferenz verbirgt, versteht man den Begriff Polymorphie (Vielgestaltigkeit).

Neben **as** gibt es auch noch den Operator **is**. Mit ihm können wir feststellen, welchen Typ eine Instanz hat, auf die eine Variable verweist.

```
if (m is TSchueler) then ...
```

So könnten wir an einer späteren Stelle im Programm prüfen, ob `m` immer noch auf die Schüler-Instanz verweist.

20. Verbergen, überschreiben, überladen

Das Hinzufügen neuer Attribute, Eigenschaften oder Methoden in Unterklassen ist eine praktische Sache. Was passiert aber, wenn ein vermeintlich neues Element den gleichen Namen verwendet wie ein Element der Vorfahrenklasse?

Komponenten gleichen Namens können einander innerhalb verwandter Klassen

- verbergen
- überschreiben
- überladen

Verbergen geerbter Komponenten

Das Prinzip „Verbergen“ besteht darin, in einer Nachkommenklasse den Namen einer geerbten Komponente neu zu vergeben. Dies kann jede Art von Klassen-Komponenten betreffen.

```
type
  TBerufstaetig = class(TMensch)
  private
    FKontoNr: integer;
    FBankleitzahl: integer;
  public
    procedure GehaltZahlen;
  end;

  TManager = class(TBerufstaetig)
  private
    FGehalt: real;
    FZulagen: real;
    FAnzahlMitarbeiter: integer;
  public
    procedure GehaltZahlen;
  end;
```

Durch die Deklaration von GehaltZahlen in der Klasse TManager wird die gleichnamige Methode von TBerufstaetig verborgen. Wird jetzt an einer Instanz von TManager diese Methode aufgerufen, wird nicht - wie bisher - der Code von TBerufstaetig ausgeführt, sondern die neue Prozedur GehaltZahlen, die nur für TManager-Instanzen und deren Nachkommen gilt.

Die Methoden müssen dazu nur im Namen übereinstimmen, die Parameter dürfen sich unterscheiden.

Falls die von TBerufstaetig geerbte Methode dennoch benötigt wird, kann sie mit

```
inherited manager.GehaltZahlen;
```

aufgerufen werden.

In der Regel wird in der neuen Methode damit die „alte“, geerbte Methode aufgerufen und anschließend (oder auch vorher) weitere Befehle ausgeführt, die in dieser Klasse benötigt werden.

Überschreiben

Das Überschreiben soll am Beispiel von Eigenschaften (s. Teil „Zugriff auf Objekte“) gezeigt werden. Dazu erweitern wird die oben verwendete Klasse TMensch um eine mögliche Eigenschaft:

```
type
  TMensch = class
  private
    FVorname: string;
    FNachname: string;
    FGeburtstag: TDate;
    FGeschlecht: string;
    procedure SetVorname(vorname: string);
  public
    property Vorname: string read FVorname write SetVorname;
  end;
```

Es besteht nun Lese- und Schreibzugriff auf die Eigenschaft Vorname. In der Realität wird der Vorname jedoch im Lauf des Lebens nicht mehr geändert. Wir verändern die Klasse also etwas und bilden eine Klasse, die nur noch Lesezugriff hat:

```
type
  TMensch = class
  private
    FVorname: string;
    FNachname: string;
    FGeburtstag: TDate;
    FGeschlecht: string;
    procedure SetVorname(vorname: string);
    property Vorname: string;
  end;
  TPerson = class(TMensch)
  public
    property Vorname read FVorname;
  end;
```

In TMensch ist die Eigenschaft Vorname nun nur noch private, d.h. Zugriffe aus anderen Units sind verboten; außerdem hat sie weder Lese- noch Schreibrechte. Sie dient nur als „Vorlage“ für Unterklassen und legt fest, dass Vorname vom Typ String sein muss. TPerson ist ein direkter Nachkomme von TMensch mit dem Unterschied, dass die Eigenschaft Vorname hier nun wieder öffentlich (public) ist, dafür nur noch Leserechte hat. Die Eigenschaft Vorname von TMensch wird dadurch **überschrieben**.

Der Vollständigkeit halber wäre nun noch eine Klasse möglich, in der die Eigenschaft `Vorname` auch Schreibrechte hat. Die Klasse entspricht der Klasse `TPerson`, allerdings mit

```
property Vorname write SetVorname;
```

Beim Überschreiben von Eigenschaften in Unterklassen darf das Spektrum der Zugriffsrechte (`read`, `write`) nur erweitert, aber nicht eingeschränkt werden. Daher hat die Oberklasse `TMensch` nur minimale Rechte, nämlich gar keine.

Übernommene Zugriffsrechte dürfen verändert werden. Es dürfte also in einer Unterklasse von `TPerson` statt `read FVorname` eine Funktion zum Lesen verwendet werden, z.B. `read GetVorname`.

Überladen von Methoden

Auch in Prozeduren und Funktionen, die zu einer Klasse gehören, ist das Überladen möglich. Eine Beschreibung dazu ist im Abschnitt „Prozeduren und Funktionen“ zu finden.

21. Eigene Konstruktoren und Destruktoren

Eigener Konstruktor

Es ist möglich, für jede Klasse einen eigenen Konstruktor sowie einen eigenen Destruktor zu schreiben. Besonders bei Konstruktoren kann das sinnvoll sein, weil der von `TObject` geerbte Konstruktor `create` nicht unbedingt das ausführt, was man sich wünscht.

Eigene Konstruktoren können ebenfalls `create` heißen, wodurch das ursprüngliche `create` überdeckt wird - sie können aber auch ganz andere Namen haben.

Das Kennzeichen eines Konstruktors ist, dass er an einer Klasse (nicht an einer Instanz) aufgerufen wird, wodurch eine Instanz erzeugt wird (Reservierung von Speicher usw.). Der Aufruf normaler Methoden, die mit `procedure` oder `function` beginnen, ist erst möglich, wenn eine Instanz existiert. Um dies zu unterscheiden, beginnt die Deklaration eines Konstruktors mit dem Schlüsselwort **constructor**:

```
interface

type
  TBerufstaetig = class(TMensch)
  private
    FKontoNr: integer;
    FBankleitzahl: integer;
  public
    procedure GehaltZahlen;
    constructor Create(Konto, BLZ: integer);
  end;

implementation

constructor TBerufstaetig.Create(Konto, BLZ: integer);
begin
  inherited Create; //hierdurch wird der ursprüngliche Konstruktor aufgerufen,
falls nötig
  {
    nun können Initialisierungen vorgenommen werden
    im Beispiel haben wir dem Konstruktor gleich Daten für Kontonr. und
Bankleitzahl übergeben
  }
  FKontoNr := Konto;
  FBankleitzahl := BLZ;
end;

...

var Mitarbeiter: TBerufstaetig;
```

```
...  
Mitarbeiter := TBerufstaetig.Create(12345, 1234567890); //Dies ruft den neuen  
Konstruktor auf
```

Wird der Konstruktor (`create`) nicht an einer Klasse, sondern an einer Instanz aufgerufen, werden dadurch alle Datenfelder des Objekts mit Standardwerten (s. „Einführung in die Objektorientierung“) überschrieben.

Eigener Destruktor

Ein eigener Destruktor wird auf ähnliche Art deklariert. Statt `constructor` wird hier `destructor` geschrieben.

Dies ist jedoch nur in wenigen Fällen sinnvoll, z. B. wenn beim Freigeben eines Objekts auch noch weitere Objekte freigegeben werden sollen.

Weder das Deklarieren von Konstruktoren noch das von Destrukturen ist unbedingt nötig, da jedes Objekt `create` und `destroy` von `TObject` erbt. Allerdings kann damit die Arbeit vereinfacht werden.

22. Virtuelle, dynamische und abstrakte Methoden

Neues Beispiel: Angenommen, wir haben eine Klasse `TKoerper`, die mehrere Unterklassen hat, z. B. `TKugel`, `TQuader` usw. Alle Klassen haben bestimmte Methoden gemeinsam, wie das Berechnen von Volumen und Oberfläche. Allerdings muss jede der Unterklassen eine andere Berechnung durchführen, da die Formeln für Kugeln und Quader unterschiedlich sind. Und obwohl alle Unterklassen diese gleichnamigen Methode besitzen, kann sie nicht ohne Weiteres in der Oberklasse aufgeführt werden, da hier ja keine Implementierung möglich ist (jeder rechnet anders).

Um dies aber zu ermöglichen, gibt es abstrakte Methoden. So kann z. B. in der Klasse `TKoerper` bereits eine function `Volumen` angegeben werden, die jedoch keinen Code enthält. Dieser folgt erst in den abgeleiteten Klassen. Im Code würde das so aussehen:

```
type
  TKoerper = class
  private
    FHoehe: real;
  public
    function Volumen: real; virtual; abstract;
    function Oberflaeche: real; virtual; abstract;
  end;

  TQuader = class(TKoerper)
  private
    FLaenge, FBreite: real;
  public
    function Volumen: real; override;
    function Oberflaeche: real; override;
  end;

implementation

function TQuader.Volumen: real;
begin
  result := FLaenge * FBreite * FHoehe;
end;

function TQuader.Oberflaeche: real;
begin
  result := (2*FLaenge*FBreite) + (2*FLaenge*FHoehe) + (2*FBreite*FHoehe);
end;
```

Hier ist zu sehen, dass `TKoerper` zwar die Methoden `Volumen` und `Oberflaeche` enthält, diese jedoch nirgends implementiert sind. Stattdessen ist dahinter **virtual; abstract;** vermerkt.

In der abgeleiteten Klasse `TQuader` dagegen sind diese beiden Methoden dann konkretisiert, jetzt ist ja bekannt, wie sie berechnet werden. Daher sind sie mit **override** markiert.

Wenn auch für weitere Unterklassen wie `TKugel`, `TZylinder` usw. so verfahren wird, dürfen für alle Objekte vom Typ `TKoerper` die Methoden `Volumen` und `Oberflaeche` aufgerufen werden, obwohl es nur für die konkreten Unterklassen jeweils spezifische Implementierungen gibt.

Steht in einem Code also

```
var K: TKoerper;
```

so ist `K.Volumen` auf jeden Fall erlaubt. Zur Laufzeit wird dann ermittelt, ob sich in `K` ein Objekt einer konkreten Unterklasse befindet, deren Methode dann ausgeführt wird. Handelt es sich allerdings tatsächlich um ein Objekt von `TKoerper`, so tritt ein Laufzeitfehler auf, weil hier `Volumen` ja nur abstrakt ist.

Jede abstrakte Methode muss auch als virtuell (Schlüsselwort `virtual`) deklariert sein. Umgekehrt muss eine virtuelle Methode nicht unbedingt abstrakt sein.

Im Unterschied zu abstrakten Methoden müssen virtuelle Methoden im betreffenden Klassentyp auch implementiert werden. Abstrakte Methoden dagegen werden nicht in der Klasse implementiert, in der sie deklariert sind.

Der Unterschied zu den „normalen“ (statisch identifizierten) Methoden besteht darin, dass virtuelle Methoden für Unterklassen (mit `override`) überschrieben und dadurch mit einer neuen Implementierung versehen werden können.



Wichtig:

- Eine virtuelle Methode (egal ob abstrakt oder nicht) und die sie (durch `override`) überschreibende Methode müssen die gleichen Köpfe (Prozedur- bzw. Funktionsköpfe) besitzen.
- Die überschreibende Methode ersetzt die überschriebene Methode vollständig (die überschriebene Methode steht ab der betreffenden Unterklasse nicht mehr zur Verfügung).

Dies sind große Unterschiede zum Verbergen einer Methode in einer Unterklasse durch eine andere: Zum Verbergen reicht der gleiche Methodenname (auf die Art und die Parameter und (evtl.) Ergebnistypen kommt es dabei nicht an). Außerdem kann eine verborgene Methode durch `inherited` wieder nutzbar gemacht werden.

Dynamische Methoden (`dynamic` statt `virtual`) werden wie virtuelle Methoden verwendet. Beide unterscheiden sich nur zur Laufzeit: Virtuelle Methoden werden auf Geschwindigkeit, dynamische auf Code-Größe optimiert. Borland empfiehlt, dynamische Methoden nur zu verwenden, wenn sich ein nachweisbarer Nutzen daraus ergibt. Im Allgemeinen ist `virtual` vorzuziehen.

Virtuelle Methoden verbergen: reintroduce

Wird eine als `virtual` deklarierte Methode verdeckt (d.h. in einer abgeleiteten Klasse wird eine gleichnamige Methode deklariert, die nicht mit der `override`-Direktive versehen ist), gibt der Delphi-Compiler Warnungen aus. Um die Warnung zu verhindern, wird die neue Methode, die die virtuelle Methode verdecken soll, mit `reintroduce` gekennzeichnet.

23. Helferklassen

Mit .NET wurde Delphi um das Konzept der unterstützenden Klassen (class helper) erweitert. Dabei handelt es sich um eine Klasse, die eine bereits bestehende Klasse erweitert bzw. abändert, ohne sie abzuleiten.

Beispiel:

```
type
  TMyClassHelper = class helper for TMyClass
    procedure MyProc;
  end;
...
procedure TMyClassHelper.MyProc;
begin
end;
```

Damit kann man auch die Prozedur MyProc an einer TMyClass-Instanz aufrufen. Hatte diese Klasse bereits eine gleichnamige Methode, wird die Methode der unterstützenden Klasse aufgerufen, da diese Vorrang hat.

Es ist programmiertechnisch sauberer, Vererbung anstelle von unterstützenden Klassen zu verwenden.

24. Exceptions

Bei einer objektorientierten Programmiersprache dürfen Exceptions nicht fehlen. Exceptions werden ausgelöst, wenn ein Fehler auftritt. Wird die Stelle im Code von `try...except` oder `try...finally` umgeben, so wird die Programmausführung an der Stelle abgebrochen, an der der Fehler aufgetreten ist, und im `except`- bzw. `finally`-Abschnitt fortgesetzt. Exceptions sind Klassen, die `Exception` als gemeinsame Basisklasse haben. Im Code, der im Fehlerfall ausgeführt wird, kann geprüft werden, was für eine Exception ausgelöst wurde.

Es ist auch möglich, eigene Exceptions zu erstellen und auszulösen. Die Namen von Exceptions beginnen dem Styleguide zufolge mit einem großen E.

```
type
    EMyError = class(Exception);
```

Exception auslösen:

```
raise EMyError.Create;
```

Fehler werden mit `try...except` oder `try...finally` abgefangen. Der `exception`-Abschnitt wird nur im Fehlerfall durchlaufen, der `finally`-Abschnitt auf jeden Fall.

```
try
    ...
except
    ...
end;
```

Im `except`-Abschnitt lässt sich prüfen, was für eine Exception aufgetreten ist:

```
except
    on EZeroDivide do ...;
    on EMathError do ...;
    on EMyError do ...;
else
    ...;
end;
```

Der `else`-Abschnitt wird immer dann ausgeführt, wenn die aufgetretene Exception nicht davor behandelt wurde.

Detailliertere Informationen zu Exceptions finden Sie in unserem Tutorial.

25. Interfaces

Ein Interface wird von der Syntax her ähnlich deklariert wie eine Klasse. Es besitzt allerdings nur öffentliche (public) Methoden, die nicht implementiert werden. Klassen können von einem oder mehreren Interfaces arbeiten und müssen dann die im Interface deklarierten Methoden auch implementieren. Dass eine Klasse ein bestimmtes Interface verwendet, bedeutet also, dass eine bestimmte Menge von Methoden zur Verfügung gestellt wird. Dies ist insbesondere bei verteilten Objektmodellen von Bedeutung (z.B. SOAP und CORBA).

Die Namen von Interfaces beginnen per Konvention mit einem großen I. Grundlegende Basisklasse aller Interfaces ist `IInterface`. Ähnlich wie Klassen von `TObject` erben alle Interfaces von `IInterface`. Da Interfaces aber keine Methodenimplementierung enthalten, wird nur die Verpflichtung geerbt, bestimmte Methoden anzubieten.

Beispiel für die Deklaration eines Interfaces:

```
type
  IMath = interface(IInterface)
    function Add(sum1, sum2: Integer): Integer;
    function Sub(sub1, sub2: Integer): Integer;
  end;
```

Das hier verwendete Schlüsselwort `interface` hat in diesem Fall nichts mit dem gleichnamigen Schlüsselwort beim Erstellen von Units zu tun.

Interfaces können einen GUID (Global Unique Identifier) verwenden, um eindeutig identifizierbar zu sein. In Delphi kann eine solche ID über die Tastenkombination Strg+Shift+G erzeugt werden:

```
type
  IMath = interface(IInterface)
    ['{A1229A9D-E207-4CBD-9B7E-FEFD9D3C1684}']
    function Add(sum1, sum2: Integer): Integer;
    function Sub(sub1, sub2: Integer): Integer;
  end;
```

Soll eine Klasse dieses Interface verwenden, wird wie folgt vorgegangen: Hinter dem Schlüsselwort `class` wird in Klammern die Basisklasse angegeben. Getrennt durch Kommata können hier Interfaces folgen. Beispiel:

```
type
  TMyClass = class(TObject, IMath)
```

Nun ist die Klasse gezwungen, die Methoden, die zu dem Interface und dessen Vorfahren gehören, zu implementieren. In unserem Fall sind das `Add` und `Sub` aus `IMath` und `_AddRef`, `_Release` und `QueryInterface` aus `IInterface` (unter Win32). Damit man die letzten drei Methoden (diejenigen aus `IInterface`) nicht selbst implementieren muss, sollte man eine Klasse, die Interfaces verwendet, nicht von `TObject`, sondern von `TInterfacedObject` ableiten. In dieser Klasse sind die `IInterface`-Methoden bereits implementiert, so dass diese einfach geerbt werden können:

```
type
  TMyClass = class(TInterfacedObject, IMath)
  public
    function Add(sum1, sum2: Integer): Integer;
    function Sub(sub1, sub2: Integer): Integer;
  end;
```

Unter .NET implementiert IInterface die Methoden AddRef, _Release und QueryInterface nicht, so dass diese Methoden nicht implementiert müssen und nicht auf TInterfacedObject zurückgegriffen werden muss.

Werden mehrere Interfaces mit gleichen Methoden implementiert, müssen die Namenskonflikte aufgelöst werden. Dies geschieht dadurch, dass einer Methode einer anderen zugeordnet wird:

```
type
  TMyClass = class(TInterfacedObject, IMath, IMySecondInterface)
    function IMath.Add = Addition;
```

Veröffentlichung von Delphi-Source.de
Stand März 2004

Autoren:

Martin Strohal
Andreas
Torsten Bäcker

Mitarbeit:

Johannes Tränkle
Nicolai Stawinoga