PC WURM
Computersysteme & Peripherie
SKRIPTEN
Akaziengasse 23; 2425 MÖCKELSDORF; Tel.: 02146/3123; Fax: 02146/2851
E-Mail: jowu@aon.at; Homepage: http://members.aon.at/pc-wurm

objektorientierte Programmierung in Turbo- & Object-Pascal

ab
Turbo Pascal 5.5 und Delphi 1.0

LEHRGANG

Verfasst und Herausgegeben von
Manuel Wurm

virtuelle Methoden

1. Auflage
Dezember 1997

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

Inhaltsverzeichnis

Literaturverzeichnis

Delphi 2 - Professionelle Anwendungen entwickeln
Herbers, Jens; Herbers, Jörg; Rensmann, Jörg
Data Becker, Düsseldorf 1996

ADIM-Skripten Band 47 - Turbo Pascal
Weissenböck, Martin
Eigenverlag, Wien 1992

Borland Pascal 7.0 - Das Kompendium
Bauder, Irene; Bär, Jürgen
Markt & Technik, Haar bei München 1993

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

Grundlagen der OOP in Turbo Pascal

Was ist objektorientiertes Programmieren?

Für das Verständnis der objektorientierten Programmierung (OOP abgekürzt), ist es notwendig, über Variablen, Konstanten, Funktionen, Prozeduren und andere Grundlagen von Pascal bescheid zu wissen.
Die objektorientierte Programmierung ist sicherlich eine der wichtigsten Neuerungen im Informatik-Bereich in den letzten Jahren. Sehr viele geläufige Programmiersprachen wurde um Möglichkeiten für die OOP erweitert. Das diese Ideen der Programmiersprache SMALLTALK entliehen sind, ist nur den wenigsten bekannt.
Borland hat für die Fortentwicklung von Turbo Pascal eigene Standards geschaffen: zunächste eine teilweise Implementierung der OOP in die Version 5.5, die dann in späteren Versionen verbessert wurde; schließlich liegt mit Objekt Pascal nun eine Programmiersprache vor, die dem objektorientierten Einsatz gerecht wird und auch durch die visuelle Programmierbarkeit im Trend der Zeit liegt.

Was zeichnet OOP aus?

Die Idee, die die Grundlage der objektorientierten Programmierung bildet, ist eigentlich recht einfach:
In der traditionellen (imperativen) Methode der Programmierung wurden jeweils Datenstrukturen gebildet. Diesen Datenstrukturen wurden dann mit Prozeduren oder Funktionen bearbeitet.

Beispiel:
In einem Array waren Mitarbeiterdaten gespeichert. Mit Hilfe der Funktion EINGABE konnte man neue Mitarbeiter eingeben, mit der Funktion LOESCHEN Datensätze löschen.

Wenn man diese Art der programmierung allerdings genauer betrachtet, fallen einige Unzulänglichkeiten auf:

- Prozeduren und Funktionen sind oft speziell für genau 1 Problem geschaffen (z. B. Mitarbeiter in das Array aufnehmen). Nach der imperativen Methode ist es aber im Programmcode nicht ersichtlich, daß diese Prozeduren (bzw. Funktionen) genau zu diesem Array gehören → **Daten und Strukturen sollten eine logische Einheit bilden**
- Rein theoretisch könnte ich in der traditionellen Programmierweise eine Prozedur, die genau für eine Datenstruktur geschrieben wurde, fälschlicherweise auch auf andere Daten anwenden → **Daten und Strukturen sollten "abgekapselt" sein**
- Wenn Sie z. B. Ihr Mitarbeiterarray auf eine verkettete Liste umstellen wollen (macht nichts, wenn Sie nicht wissen was das ist), müssen Sie alle Ihre Prozeduren und Funktionen umschreiben. Es wäre also günstiger, wenn die Anzeige- und Bearbeitungsprozeduren sich nicht um die Speicherungsart der Daten kümmern müßten → **implementationsunabhängige Strukturen**

Genau diese Nachteile versucht die objektorientierte Programmierung aus der Welt zu schaffen. Man erreicht durch ihre Anwendung daher eine strukturiertere und übersichtlichere Form, vor allem bei größeren Programmen.

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

Ein praktisches Beispiel zur OOP

Stellen Sie sich zum Beispiel vor, daß Sie in einer Ihrer Anwendungen verschiedene Grafikobjekte speichern und anschließend auf dem Bildschirm ausgeben wollen. Bei den Grafikobjekten handelt es sich um Punkte, Striche und Rechtecke. Mit einer traditionellen, imperativen Programmiersprache würden Sie jetzt wahrscheinlich damit beginnen, Variablen für die Koordinaten des Mittelpunkts Ihres Kreises, für den Radius desselben oder für die Kantenlängen des Rechtecks zu definieren. Die günstigste Art, dies zu programmieren, wäre sicherlich, Records anzulegen, daß die für das jeweilige Objekt wichtige Daten beinhaltet.
Anschließend würden Sie für jede Art von Grafikobjekt beispielsweise Prozeduren zum Belegen der Daten eines Records sowie Prozeduren zum Verändern der Größe oder zum Anzeigen auf dem Bildschirm programmieren.

Wenn Sie jedoch von der konkreten Anschauung der Grafikobjekte in der Realität ausgingen, würden Sie einen anderen Weg wählen: Sie würden ein Objekt vorsehen, in dem jeweils die Daten eines Punktes, eines Kreises etc. abgelegt würden (je nach Typ des Objekts eine andere Datenkonstelation). Innerhalb dieses Objekts würden Sie Prozeduren schreiben, die in diesem Fall tatsächlich zum Objekt selbst gehörten, und die das Objekt (d. h. die Daten davon) verändern. Schließlich würden Sie in jedes Objekt eine Prozedur implementieren, die das Objekt anzeigt. Dies würde dann "von außen" so aussehen:

Objekt Kreis:
Variablen: x, y, Radius
Prozeduren: Dateneingabe, Zeichnen

Damit haben Sie eine wesentlich bessere Struktur erziehlt:
Die Prozeduren bilden zusammen mit den Daten eine Einheit und jede Prozedur greift nur auf die ihr zugehörigen Daten zu. In der Sprache der OOP bezeichnet man die Daten übrigens als **Felder** oder **Attribute**, die Prozeduren, die zum Objekt (in anderen Programmiersprachen auch als **Klasse** bezeichnet) gehören, werden auch **Methoden** genannt.

Ein Beispiel:

```
TYPE
  KontoTyp = OBJECT
    Kontonr: INTEGER;          {Feld}
    Kontostand: REAL;          {Feld}
    PROCEDURE Auszug;          {Methode - Soll Kontostand ausgeben}
  END;
```

```
TYPE
  KreisTyp = OBJECT
    x, y, Radius: INTEGER;
    PROCEDURE Zeichnen;
  END;
```

Wenn wir unser Beispiel mit dem Kreis weiterführen: Die Methode ZEICHNEN muß noch irgendwo definiert werden. Dies geschieht irgendwo im **Vereinbarungsteil**, unbedingt vor dem Hauptprogramm:

```
PROCEDURE Kreis.Zeichnen;
BEGIN
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

```
  Circle(x, y, Radius);
END;
```

Wichtig:

- Wenn die Methode eines Objekt definiert wird, muß beim Namen immer zuerst der Objekttypname (in unserem Fall KREIS), gefolgt von einem Punkt und dem Namen der Prozedur (ZEICHNEN) angegeben werden.
  ```
  PROCEDURE Zeichnen;
  ```
  wäre also falsch (würde eine neue, vom Objekt völlig unabhängige Prozedur erstellen).
- In einer Methode eines Objekts können die Felder desselben Objekts ohne einer weiteren Angabe angegeben werden:
  Nicht:
  ```
  Kreis.x:=10;
  ```
  Sondern:
  ```
  x:=10;
  ```

Anschließend müssen wir natürlich noch eine Variable erzeugen, die das Objekt enthält (bis jetzt haben wir nur den Typ und die Methode festgelegt):

```
VAR Kreis: KreisTyp;
```

Eine Variable, die ein Objekt als Typ hat, heißt **Instanz** oder **Instanzvariable**. Auf die Komponenten einer Instanzvariablen wird genauso wie auf einen Record zugegriffen.
Im folgenden das vollständige Beispielprogramm:

```
PROGRAM KreisObjekt;
USES crt, graph;

TYPE
 KreisTyp = OBJECT
   x, y, Radius: INTEGER;
   PROCEDURE Zeichnen;
 END;

VAR gd, gm: INTEGER;
   Kreis: KreisTyp;

PROCEDURE KreisTyp.Zeichnen;
BEGIN
 Circle(x, y, Radius);
END;

BEGIN
 gd:=detect;
 InitGraph(gd, gm, 'C:\PASCAL');
 Kreis.x:=20;
 Kreis.y:=30;
 Kreis.Radius:=20;
 Kreis.Zeichnen;
 ReadLn;
 CloseGraph;
END.
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

Funktionen in Objekten

Natürlich können wir unser Objekt KREISTYP folgendermaßen erweitern:

```
TYPE
 KreisTyp = OBJECT
  x, y, Radius: INTEGER;
  PROCEDURE Zeichnen;
  FUNCTION PunktinKreis (xKor, yKor: INTEGER): BOOLEAN;
 END;
```

Die Funktion PUNKTINKREIS soll uns TRUE liefern, wenn der übergebene Punkt im Kreis liegt und FALSE, wenn er außerhalb liegt:

```
FUNCTION KreisTyp.PunktInKreis (xKor, YKor: INTEGER): BOOLEAN;
BEGIN
 PunktInKreis:=SQRT(SQR(x)-2*xKor*x+SQR(y)-
2*yKor*y+SQR(yKor)+SQR(xKor))<=Radius;
END;
...
{im Hauptprogramm:}
IF Kreis.PunktInKreis(20,20) THEN
  WriteLn('Punkt liegt im Kreis')
 ELSE
  WriteLn('Punkt liegt außerhalb');
```

Eine Funktion innerhalb eines Objektes nennt man **Eigenschaft**.

Konstruktoren und der typische Aufbau eines Objekts

Außer Feldern, Eigenschaften und Methoden ist in Pascal in einem Objekt ein ein **Konstruktor** vertreten. Er wird genauso wie eine Prozedur definiernt, außer daß das reservierte Wort CONTRUCTOR verwendet wird. Meistens bekommt er den Namen INIT.
Außerdem läßt man es aufgrund der Fehleranfälligkeit und Datensicherheit nicht zu, daß die Daten eines Objekte direkt verändert bzw. ausgelesen werden (z. B. KREIS.X:=10). Man erzeugt stattdessen Methoden, die die Arbeit übernehmen. Das hat den weiteren Vorteil, daß man innerhalb der Methoden bzw. der Datenstruktur des Objektes etwas ändern kann, ohne daß man den Aufruf des Objektes im restlichen Programm (**Schnittstelle**) modifizieren müßte.

```
TYPE
  KreisTyp = OBJECT
    ObjX, ObjY, Radius: INTEGER;
    CONSTRUCTOR Init(x, y: INTEGER);  {Startwerte setzen}
    PROCEDURE XYSetzen (x, y: INTEGER);  {Setzt ObjX und ObjY}
    FUNCTION Xkoordinate: INTEGER;  {Liefert den Wert von ObjX}
    FUNCTION Ykoordinate: INTEGER;  {Liefert den Wert von ObjY}
    PROCEDURE Zeigen;
    FUNCTION PunktInKreis(Xkor, Ykor: INTEGER): BOOLEAN;
  END;
...
CONSTRUCTOR Init(x, y: INTEGER);
BEGIN
  XYSetzen(x, y);
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

```
END;

PROCEDURE XYSetzen(x, y: INTEGER);
BEGIN
   ObjX:=x;
   ObjY:=y;
END;
...
VAR
 Kreis: KreisTyp;
...
{im Hauptprogramm:}
 Kreis.Init(20, 20);
 Write('Aktueller Mittelpunkt: ',Kreis.Xkoordinate,', ', Kreis.Ykoordinate);
```

Obwohl jetzt für den Zugriff auf OBJX oder OBJY eigene Methoden und Eigenschaften definiert wurden, wäre eine direkte Änderung noch immer zulässig. Daher wurde ab der Version 6.0 von Turbo Pascal das Schlüsselwort PRIVAT eingeführt.
In der Definition des Objektes sind Daten, Methoden etc. vor dem Schlüsselwort PRIVAT öffentlich; nach dem Schlüsselwort sind sie **abgekapselt**, d. h. auf sie kann nur innerhalb der Methoden und Eigenschaften des Objekts zugegriffen werden.

```
TYPE
   KreisTyp = OBJECT
     CONSTRUCTOR Init(x, y: INTEGER);          {Startwerte setzen}
     PROCEDURE XYSetzen (x, y: INTEGER);       {Setzt ObjX und ObjY}
     FUNCTION Xkoordinate: INTEGER;            {Liefert den Wert von ObjX}
     FUNCTION Ykoordinate: INTEGER;            {Liefert den Wert von ObjY}
     PROCEDURE Zeigen;
     FUNCTION PunktInKreis(Xkor, Ykor: INTEGER): BOOLEAN;
   PRIVATE
     ObjX, ObjY, Radius: INTEGER;
   END;
```

Nach dieser Definition ist im Hauptprogramm folgender Befehl nicht mehr zulässig:

```
Kreis.ObjX:=20;
Kreis.ObjY:=20;
```

Stattdessen ist die Methode XYSETZEN zu verwenden:

```
Kreis.XYSetzen(20, 20);
```

Vererbung

Verschiedene Grafikobjekte haben gemeinsam, daß man z. B. an einem Punkt mit den Koordinaten x und y die Position angeben kann. Man kann also die Methode XYSETZEN und die Eigenschaften XKOORDINATE und YKOORDINATE nicht nur für den Kreis verwenden, sondern z. B. auch, um einen Punkt darzustellen.
Daher erstellt man zuerst ein Objekt, daß die Methoden, Eigenschaften und Daten aller davon abgeleiteten Objekte beinhaltet (sogenanntes **Mutterobjekt**, **Oberklasse** oder **Superklasse**):

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

```
TYPE
  Position = OBJECT
    CONSTRUCTOR Init(x, y: INTEGER);
    PROCEDURE XYSetzen(x, y: INTEGER);
    FUNCTION Xkoordinate: INTEGER;
    FUNCTION Ykoordinate: INTEGER;
  PRIVATE
    ObjX, ObjY: INTEGER;
  END;
```

Anschließend werden die Objekte, die auch die Eigenschaften, Methoden und Daten des Mutterobjekts nutzen definiert. Dazu wird neben dem Schlüsselwort OBJECT die Oberklasse anzugeben. Zum Beispiel:

```
  Pixel = OBJECT (Position)
    CONSTRUCTOR Init(x, y, Farbe: INTEGER);
```

(Methoden bzw. der Konstrukteur mit dem gleichen Namen werden "überlagert").

```
    PROCEDURE FarbeSetzen(Farbe: INTEGER);
    FUNCTION FarbWert: INTEGER;
    PROCEDURE Anzeigen;
    PROCEDURE Ausblenden;
  PRIVATE
    ObjFarbe: INTEGER;
  END;

  Kreis = OBJECT (Position)
    CONSTRUCTOR Init(x, y, Radius: INTEGER);
    PROCEDURE SetzeRadius (Radius: INTEGER);
    PROCEDURE Anzeigen;
    FUNCTION RadiusWert: INTEGER;
  PRIVATE
    ObjRadius: INTEGER;
  END;
```

Im folgenden ein Beispielprogramm, daß die praktische Anwendung der Vererbung darstellt:

```
PROGRAM ObjBsp02;
USES crt, graph;
TYPE
 PositionTyp = OBJECT
  PROCEDURE XYSetzen (x, y: INTEGER);
  FUNCTION XKoordinate: INTEGER;
  FUNCTION YKoordinate: INTEGER;
 PRIVATE
  ObjX, ObjY: INTEGER;
 END;

 PixelTyp = OBJECT (PositionTyp)
  CONSTRUCTOR Init(x, y, Farbe: INTEGER);
  PROCEDURE FarbeSetzen(Farbe: INTEGER);
  PROCEDURE Anzeigen;
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

```
    PROCEDURE Ausblenden;
    FUNCTION FarbeWert: INTEGER;
  PRIVATE
    ObjFarbe: INTEGER;
  END;

  KreisTyp = OBJECT (PositionTyp)
    CONSTRUCTOR Init(x, y, Radius: INTEGER);
    PROCEDURE RadiusSetzen(Radius: INTEGER);
    PROCEDURE Anzeigen;
    FUNCTION RadiusWert: INTEGER;
  PRIVATE
    ObjRadius: INTEGER;
  END;

VAR
  Position: PositionTyp;
  Punkt: PixelTyp;
  Kreis: KreisTyp;
  gd, gm: INTEGER;
  Ausgabe: STRING;

PROCEDURE PositionTyp.XYSetzen(x, y: INTEGER);
BEGIN
  ObjX:=x;
  ObjY:=y;
END;

FUNCTION PositionTyp.XKoordinate: INTEGER;
BEGIN
  XKoordinate:=ObjX;
END;

FUNCTION PositionTyp.YKoordinate: INTEGER;
BEGIN
  YKoordinate:=ObjY;
END;

CONSTRUCTOR PixelTyp.Init(x, y, Farbe: INTEGER);
BEGIN
  XYSetzen(x, y);
  FarbeSetzen(Farbe);
END;

PROCEDURE PixelTyp.FarbeSetzen(Farbe: INTEGER);
BEGIN
  ObjFarbe:=Farbe;
END;

FUNCTION PixelTyp.FarbeWert: INTEGER;
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

```
 BEGIN
   FarbeWert:=ObjFarbe;
 END;

 PROCEDURE PixelTyp.Anzeigen;
 BEGIN
   PutPixel(XKoordinate, YKoordinate, FarbeWert);
 END;

 PROCEDURE PixelTyp.Ausblenden;
 BEGIN
   PutPixel(XKoordinate, YKoordinate, 0);
 END;

 CONSTRUCTOR KreisTyp.Init(x, y, Radius: INTEGER);
 BEGIN
   XYSetzen(x, y);
   RadiusSetzen(Radius);
 END;

 PROCEDURE KreisTyp.RadiusSetzen(Radius: INTEGER);
 BEGIN
   ObjRadius:=Radius;
 END;

 FUNCTION KreisTyp.RadiusWert: INTEGER;
 BEGIN
   RadiusWert:=ObjRadius;
 END;

 PROCEDURE KreisTyp.Anzeigen;
 BEGIN
   Circle(XKoordinate, YKoordinate, RadiusWert);
 END;

 {Hauptprogramm}
 BEGIN
   gd:=detect;
   InitGraph(gd, gm, 'C:\PASCAL');
   Kreis.Init(300, 200, 80);
   Kreis.Anzeigen;
   Punkt.Init(300, 200, 4);
   Punkt.Anzeigen;
   STR(Punkt.FarbeWert, Ausgabe);
   Ausgabe:='Farbwert des Punktes: '+Ausgabe;
   OutTextXY(10, 10, Ausgabe);
   ReadLn;
   CloseGraph;
 END.
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

Wie Sie im Programm sehen, muß schon vor dem Programmieren genau geplant werden, welche Objekte und Instanzen man braucht bzw. welche gemeinsame Methoden verschiedene Objekte benötigen.

Ergänzung

...zur Verdeutlichung der Arbeitsweise des Compilers:
Betrachten wir noch einmal folgende Prozedur:

```
PROCEDURE PositionTyp.XYSetzen(x, y: INTEGER);
BEGIN
   ObjX:=x;
   ObjY:=y;
END;
```

Stattdessen wäre auch folgendes möglich:

```
PROCEDURE PositionTyp.XYSetzen(x, y: INTEGER);
BEGIN
   Self.ObjX:=x;
   Self.ObjY:=y;
END;
```

Der Parameter SELF ist in jeder Prozedur oder Funktion vom jeweiligen Objekttyp. In der Praxis ist es allerdings nicht notwendig.

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

Das neue Objektmodell von Object Pascal

Bis jetzt haben wir (in Turbo- oder Borland-Pascal) ein Objekt immer mit dem Schlüsselwort OBJECT definiert.
Diese Syntax steht auch in Delphi zur Verfügung. Sie können also weiterhin Objekte dieses Typs in Delphi verwenden. Grund dafür ist, daß Sie somit Programme aus älteren Pascal-Versionen einfach in Object Pascal übernehmen können.
Um jedoch Objekte des neuen Typs zu verwenden, müssen Sie statt dem Schlüsselwort OBJECT das Wort CLASS verwenden, das schon auf eine genauere Unterscheidung von Klassen und Instanzen hinweist.
Durch die Benützung des neuen Objekttyps können Sie die neuen Möglichkeiten von Object Pascal nützen, wovon die wichtigsten im Folgenden aufgelistet sind:

- Zusätzlich zu den Schlüsselwörtern PRIVATE (ab Turbo Pascal 6.0) und PUBLIC (ab Turbo Pascal 7.0) kann man unter Object Pascal Deklarationsblöcke mit PROTECTED oder PUBLISHED betiteln. Bei PROTECTED kann man auch von Unterklassen auf diesen Block zugreifen; bei PUBLISHED werden Laufzeitinformationen erzeugt, d. h. diese Felder kann man im Objektinspektor verändern.
- Klassen, bei denen nicht explizit eine Mutterklasse angegeben wird, werden automatisch von *Tobject*, dem sogenannten "standardmäßigem Vorfahr", abgeleitet.
- In Objekt Pascal werden die Instanzen in Klassen dynamisch verwaltet, d. h. daß eine Variable eigentlich ein Zeiger ist, der auf den jeweiligen Speicherbereich zeigt. Als Programmierer merkt man allerdings nichts von dieser dynamischen Speicherverwaltung.

etc.

Die Objektdefinition

Die Syntax für eine neue Klasse lautet in Delphi folgendermaßen:

```
TYPE
  TKreis = CLASS
    x, y: INTEGER;
    Radius: INTEGER;
    PROCEDURE Anzeigen;
  END;
```

Wie schon im vorhergehenden Kapitel erwähnt, erbt diese Klasse auch alle Eigenschaften der Klasse TOBJECT, da wir keine spezielle Oberklasse angegeben haben.
Durch die Klassendeklaration haben wir also einen neuen Objekttyp angegeben. Um eine Instanz zu erzeugen, muß man anschließend in Delphi statt

```
VAR Kreis: TKreis;
```

zusätzlich im Hauptprogramm noch

```
Kreis:=Tkreis.Create;
```

angeben. (Warum nicht einfach die Variablendeklaration ausreicht, werden ich Ihnen bei den Konstrukteuren noch genauer schildern).
Anschließend kann man, wie schon in Turbo Pascal, auf die Datenfelder und Methoden des Objekts zugreifen:

```
Kreis.x:=20;
Kreis.y:=30;
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

```
Kreis.Radius:=10;
Kreis.Anzeigen;
```

Natürlich auch mit der WITH-Anweisung möglich:

```
WITH Kreis DO BEGIN
  x:=20;
  y:=30;
  Radius:=10;
  Anzeigen;
END;
```

Möglickeiten der Abkapselung etc.

Wie schon erwähnt, kennt Object Pascal außer PUBLIC und PRIVATE noch zwei andere Schlüsselwörter zur Deklarationsblockdefinition. Hier eine Übersicht:

```
TYPE
  Klassenname = CLASS
   {Felder und Methoden der Schutzebene PUBLISHED}
  PRIVATE
   {Felder und Methoden der Schutzebene PRIVATE}
  PROTECTED
   {Felder und Methoden der Schutzebene PROTECTED}
  PUBLIC
   {Felder und Methoden der Schutzebene PUBLIC}
  PUBLISHED
   {evtl. noch einmal Felder und Methoden der Schutzebene PUBLISHED}
  END;
```

PUBLISHED und PUBLIC:
Wenn keine Schutzebene angegeben wird, nimmt Object Pascal automatisch PUBLISHED an. Solche Felder und Methoden (sowie die der Schutzebene PUBLIC) stehen anderen Prozeduren und Programmteilen zur Verfügung, auf sie kann aus allen anderen Programmteilen zugegriffen werden. Diese Schnittstelle sollte möglichst klein gehalten werden, um umfangreiche Manipulationen des Objekts zu verhindern.
Im PUBLISHED-Modus stehen die Methoden und Datenfelder außer innerhalb des Programmes auch anderen Programmen zur Verfügung z. B. bei Komponenten.

PRIVATE und PROTECTED:
Bei PRIVATE und PROTECTED kann von außen (d. h. von Befehlen, die nicht in Methoden oder Funktionen des Objektes stehen) nicht zugegriffen werden. Zum Unterschied von PRIVATE, auf dessen Datenblock wirklich nur Methoden und Funktionen der eigenen Klasse zugreifen können, können bei PROTECTED auch die Unterklassen den Deklarationsblock dieser Klasse verwenden.

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

Unterklassen definieren

Unterklassen werden genauso wie in Turbo Pascal, nur mit dem Schlüsselwort CLASS definiert, also:

```
TYPE
  TPosition = CLASS
    PROCEDURE SetzeXY(x, y: INTEGER);
  PRIVATE
    ObjX, ObjY: INTEGER;
  END;

  TKreis = CLASS (TPosition)
    PROCEDURE SetzeRadius (Radius: INTEGER);
  PRIVATE
    ObjRadius: INTEGER;
  END;
```

Der Operator IS

Folgendes Beispiel (Fortsetzung des obigen):

```
VAR
  Position: Tposition;
  kleinerKreis: Tkreis;
  Ergebnis: BOOLEAN;

...
Ergebnis:=Position IS Tposition;      Ergebnis=TRUE
Ergebnis:=Position IS Tkreis;         Ergebnis=FALSE
Ergebnis:=Kreis IS Tkreis;            Ergebnis=TRUE
Ergebnis:=Kreis IS Tposition;         Ergebnis=TRUE (weil Unterklasse)
```

Der Konstrukteur & Destrukteur

Wenn man in Turbo Pascal eventuell auch ohne Konstrukteur ausgekommen ist, geht es in Object Pascal überhaupt nicht ohne. Bevor man überhaupt eine Instanz benutzen kann, muß man einen Konstrukteur aufrufen, der in Delphi normalerweise CREATE genannt wird. Da sich in Delphi alle Klassen ohne Mutterklasse von TOBJECT ableiten, ist in dieser Klasse auch schon ein Konstrukteur dieses Namens angelegt. Um diesen zu überschreiben, verwendet man das Schlüsselwort OVERRIDE, z. B.:

```
TYPE
  Grafikobjekt = CLASS
    {...}
  PUBLIC
    CONSTRUCTOR Create(x, y: INTEGER); override;
  END;
```

Da den Objektinstanzen in Object Pascal auch dynamisch Speicher zugewiesen wird, muß dieser nach der Verwendung des Objekts natürlich wieder freigegeben werden. Dazu verwendet man sogenannte **Destruktoren**. Der Standardname in TOBJECT heißt DESTROY.

```
TYPE
```

Tel (Privat): 02146/3123 Fax: 02146/2851 Manuel Wurm Tel (Firma): 02146/2851; E-Mail: jowu@aon.at

```
Pixel = CLASS
PRIVATE
  ObjX, ObjY: INTEGER;
  ObjFarbe: INTEGER;
PUBLIC
  CONSTRUCTOR Create (x, y: INTEGER);
  DESTRUCTOR Destroy;
END;
```

Im Normalfall müssen Sie sich jedoch nicht um die Programmierung des Destrukteurs (und um die Verwendete Methode FREE) kümmern, daher wird hier nicht näher auf ihn eingegangen.

Klassen in Klassen verwenden

Natürlich kann man in einer Klasse auch andere Objektinstanzen verwenden:

```
TYPE
  {...}
  Tgrafikobjekte = CLASS
    ObjKreis: TKreis;
    ObjRechteck: TRechteck;
    ObjX, ObjY: INTEGER;
  END;
```