O O P

Objektorientiertes Programmieren

© Herbert Paukert

OBJEKTORIENTIERTES PROGRAMMIEREN

[01] Allgemeine Vorbemerkungen

Das vorliegende Kapitel beschäftigt sich mit den Grundelementen der objektorientierten Programmierung. Um diese einfach und klar darzustellen, befreien wir die Demonstrationsprogramme von allen ***Windows***-typischen Komponenten und beschränken uns nur auf das Wesentliche. Dazu wird zunächst im Menü <*Datei/Neue Anwendung*> ein neues Formular erstellt. Im nächsten Schritt entfernen wir das Formular samt zugehöriger Unit aus dem Projekt, indem wir in der Symbolleiste das Icon *"Datei aus Projekt entfernen"* anklicken und den nachfolgenden Dialog mit dem Schalter <*OK*> bestätigen. Nun öffnen wir im Menü <*Projekt/Quelltext anzeigen*> den Editor und ändern den darin gezeigten Quelltext des Hauptprogrammes wie folgt ab:

```
program project1;
{ Ein einfaches Hauptprogramm OHNE Objekte }

uses SysUtils;                  // Diese Standard-Unit
                                // muss eingebunden werden.
var Data : String;              // Globale String-Variable

begin
  Data := '  Hallo Welt! ';     // Einfache Wertzuweisung
  writeln;                      // Leere Textzeile am Bildschirm ausgeben
  writeln(Data);                // Ausgabe von DATA am Bildschirm
  writeln('  <Enter> ..... ');  // Textzeile am Bildschirm ausgeben
  Readln;                       // Wartet auf Betätigung von <ENTER>.
end.
```

Im Menü <*Projekt/Optionen*> wählen wir im Register *"Linker"* den Schalter *"Anwendung für Textbildschirm"*. Dadurch erstellt DELPHI eine so genannte Konsolenanwendung, d.h., das Eingabegerät ist die Tastatur und das Ausgabegerät ist der Bildschirm im Textmodus. Durch den Verzicht auf den Grafikmodus, in welchem DELPHI und WINDOWS automatisch arbeiten, und durch die Entfernung aller Standard-Units aus der *Uses*-Anweisung des Programmprojektes, ausgenommen *SysUtils*, befreien wir das Hauptprogramm vom gesamten Ballast einer komponentengesteuerten grafischen Bedienungsoberfläche. Fortan stehen uns nur zwei einfache Ein-/Ausgabe-Befehle zur Verfügung. *Writeln(**V**)* gibt die Variable ***V*** in einer Zeile am Textbildschirm aus. (*Writeln* allein bewirkt nur einen Zeilenvorschub). *Readln(**V**)* wartet auf eine Tastatureingabe, die mit <Enter> beendet werden muss, und weist der Variablen ***V*** den eingegebenen Wert zu. (*Readln* allein wartet nur auf die Betätigung der <Enter>-Taste).

Zum Schluss muss das Hauptprogramm im Menü <*Datei/Speichern unter*> mit dem neuen Namen "**oop0.dpr**" abgespeichert werden. Dadurch ändert DELPHI automatisch den Programmnamen im Listing von "**project1.dpr**" auf "**oop0.dpr**".

[02] Die Ausgangsbasis (*oop0*)

Das sehr einfache Programm "**oop0.dpr**" definiert nur die globale String-Variable *Data*, welche öffentlich zugänglich ist. Im Programm selbst erhält diese Variable einen Wert zugewiesen und wird sodann am Bildschirm ausgegeben.

Stellt man sich nun beispielsweise eine Textverarbeitung vor, so sind die Objekte der Verarbeitung einfache Textzeilen, deren Dateninhalte Strings sind. Verarbeitungsmethoden solcher Zeilen sind z.B. *Eingeben* und *Ausgeben* der Textdaten.

Ein solches Objekt *Zeile* einer Textverarbeitung umfasst somit den Datenspeicher *Data* und dessen Verarbeitungs-*Methoden*. Der direkte Zugriff auf den Datenspeicher soll von außen nicht erlaubt sein, sondern soll nur mehr durch den Aufruf einer einschlägigen Methode erfolgen (Prinzip der Datenkapselung).

In dieser Sichtweise stellen die einzelnen Anweisungen des Textverarbeitungsprogrammes Botschaften an das Objekt *Zeile* dar, seine Daten entsprechend zu verarbeiten (Prinzip der Kommunikation).

Natürlich wird das simple Objekt *Zeile* für eine Textverarbeitung nicht ausreichen. Es müssen also weitere Objekte vom Basisobjekt *Zeile* abgeleitet werden, die erstens alles können, was *Zeile* kann (Prinzip der Vererbung), und zweitens darüber hinaus noch einiges mehr leisten, z.B. eine Textzeile nach einem bestimmten Begriff durchsuchen oder den Text in Großschrift umwandeln (Prinzip der Erweiterung).

Die Byte-Welt des Computers wird durch den Programmierer in einer solchen Weise organisiert, dass sie sich als ein Netzwerk hierarchisch gegliederter Objekte darstellt. Diese Objekte umfassen Daten und Methoden, ihr Innenleben bleibt jedoch abgekapselt und verschlossen. Sie kommunizieren miteinander durch den Austausch von Nachrichten.

Das ist die Sichtweise der objektorientierten Programmierung, wie sie in den folgenden einfachen und übersichtlichen Beispielen dargestellt werden soll.

[03] Die Erzeugung von Objekten (*oop1*)

Ein Objekttyp ist ein Sprachgebilde, das aus Datenfeldern (bzw. **Eigenschaften**) und zugehörigen **Methoden** zur Verarbeitung dieser Daten besteht. Unter Datenkapselung versteht man den Zusammenschluss von Daten und zugehörigen Methoden. Ein Objekttyp wird durch den Bezeichner ***Class*** deklariert.

Die Konstruktion eines Objektes gliedert sich in zwei Abschnitte: Zuerst wird als Benutzerschnittstelle (Interface) der Objekttyp durch Angabe seiner Eigenschaften und Methoden deklariert. Danach werden die angeführten Methoden hintereinander codiert (Implementation). Ein Objekt ist somit dem klassischen Typ *Record* ähnlich und wird wie eine *Unit*-Struktur aufgebaut.

Eine Instanz ist eine konkrete Realisierung eines Objekttyps. Das geschieht in einer entsprechenden Variablendefinition.

Die Kommunikation mit den verschiedenen Objekten erfolgt durch Aufruf ihrer Methoden, was einer Botschaft (Nachricht) entspricht. Objekte werden grundsätzlich dynamisch im Hauptspeicher abgelegt, d.h., erst zur Laufzeit des Programmes wird der entsprechende Speicher reserviert, was eine so genannte *Construktor*-Methode bewerkstelligt. Am Ende des Programmlaufes wird der reservierte Speicher wieder freigegeben, was durch eine *Destruktor*-Methode erfolgt. Im Programm "**oop1.dpr**" wird der einfache Objekttyp *TEins* erzeugt, der nur ein Datenfeld und drei Methoden enthält.

```
program oop1;
{ Hauptprogramm mit einem erzeugten Objekt }

uses SysUtils;

type TEins = class(TObject)              // Objekt
       Data : String;                    // Datenfeld (bzw. Eigenschaft)
       procedure Eingeben(s: String);    // Methode
       procedure Ausgeben;               // Methode
       procedure Kennung;                // Methode
     end;
```

```
    procedure TEins.Eingeben(s: String);
    begin
      Data := s;
    end;

    procedure TEins.Ausgeben;
    begin
      Writeln;
      Writeln(Data);
    end;

    procedure TEins.Kennung;
    begin
      Writeln('  Ich bin OBJEKT [1], ');
      Writeln('  <Enter> ..... ');
      Readln;
    end;

var  Eins : TEins;                    // Definition des eigentlichen Objektes
                                      // (Instanz des Objekttyps)
begin
  Eins := TEins.Create;               // Von TObject übernommene Funktion
  Eins.Eingeben('  Hallo Welt!');
  Eins.Ausgeben;
  Eins.Kennung;
  Eins.Free;                          // Von TObject übernommene Prozedur
end.
```

Bildschirmausgabe:

```
Hallo Welt!
Ich bin Objekt [1],
<Enter> .....
```

Das Datenfeld *Data* ist ein String. Über die Methode *Eingeben* kann dem Datenfeld ein Wert zugewiesen werden. Die Methode *Ausgeben* zeigt den Wert des Datenfeldes am Bildschirm an. Die Methode *Kennung* dient der Identifikation des Objektes. An diesem Beispiel wird die Grundregel der objektorientierten Programmierung (OOP) ersichtlich, dass von außen niemals direkt auf ein Datenfeld eines Objektes zugegriffen werden soll, sondern immer nur über geeignete Methodenaufrufe (d.h. Botschaften).

Das Programm selbst besteht nur aus einfachen Botschaften an die Instanz *Eins* des Objekttyps *TEins*. Eigenschaften und Methoden eines Objektes werden dadurch angesprochen, indem man ihren Bezeichnern den Objektnamen voranstellt und dazwischen einen Punkt setzt.

Wichtig ist die erste Zeile *type TEins = class(TObject)* des Programmlistings. Diese Anweisung ist zunächst der Anfang einer Deklaration des Objekttyps *TEins*. Darüber hinaus wird aber auch der im System vordefinierte Typ *TObject* als Stammvater aller anderen Objekte der DELPHI-Welt ausgewiesen. Dieser allgemeinste Objekttyp ist in der System-Unit von DELPHI definiert und enthält alle wichtigen Routinen zur Erstellung, Verarbeitung und Entfernung von Objekten. Diese so genannten Klassen-Routinen werden von allen abgeleiteten Objekten übernommen und stehen ihnen zur Verfügung. Dazu gehören u.a. die vordefinierte *Construktor*-Funktion ***Create***, welche für die Objekte am Programmanfang Speicherplatz reserviert, und die *Destructor*-Prozedur ***Free***, welche am Programmende diesen Platz wieder freigibt. Das alles geschieht aber nicht bei der Compilation, sondern erst zu Laufzeit des Programmes. In diesem Zusammenhang spricht man auch von dynamischer Speicherverwaltung.

[04] Die Vererbung von Merkmalen (*oop2*)

Objekte können Nachfahren besitzen, welche die Eigenschaften und Methoden ihrer Vorfahren erben. In den Nachfahren können vererbte Merkmale verwendet oder auch neue Merkmale zusätzlich definiert werden (Vererbung bzw. Objekterweiterung). Die Deklaration eines Nachfahren erfolgt sprachlich mittels ***Class****(Vorfahre).* Als Grundregel der Vererbung gilt, dass ein Nachfahre alle Eigenschaften seiner Vorfahren übernimmt und alle Methoden seiner Vorfahren verwenden kann. Ruft ein Nachfahre eine vererbte Methode auf, dann wird sie grundsätzlich in der Form ausgeführt, wie sie im Vorfahren definiert ist. Man nennt solche Methoden daher auch statisch.

Im Programm "**oop2.dpr**" wird zum Objekttyp *TEins* ein Nachfahre *TZwei* erzeugt. Dieser enthält - neben dem von *TEins* vererbten Datenfeld *Data* und den drei vererbten Methoden *Eingeben*, *Ausgeben* und *Kennung* - ein neues Datenfeld *Data1* und eine neue Methode *Zusatz.* Diese beiden sind echte Erweiterungen des ursprünglichen Objektes.

```
program oop2;
{ Vererbung und Erweiterung von Objektmerkmalen }

uses SysUtils;

type TEins = class(TObject)         // Vorfahre
       Data : String;
       procedure Eingeben(s: String);
       procedure Ausgeben;
       procedure Kennung;
     end;

     TZwei = class(TEins)           // Nachfahre
      Data1 : String;               // Neues Datenfeld (bzw. Eigenschaft)
      procedure Zusatz;             // Neue Methode
     end;


     procedure TEins.Eingeben(s: String);
     begin
       Data := s;
     end;

     procedure TEins.Ausgeben;
     begin
       Writeln;
       Writeln(Data);
     end;

     procedure TEins.Kennung;
     begin
       Writeln('  Ich bin OBJEKT [1], ');
       Writeln(' <Enter> ..... ');
       Readln;
     end;


     procedure TZwei.Zusatz;
     begin
       Data1 := '  das ist ein Zusatzangebot. ';
       Writeln(Data1);
     end;


var  Eins : TEins;                    // Eins als Instanz von TEins
     Zwei : TZwei;                    // Zwei als Instanz von TZwei
```

```
begin
  Eins := TEins.Create;
  Zwei := TZwei.Create;
  Eins.Eingeben(' Hallo Welt! ');
  Eins.Ausgeben;
  Eins.Kennung;
  Zwei.Eingeben(' Liebe Freunde,');     // Vererbte Methode
  Zwei.Ausgeben;                        // Vererbte Methode
  Zwei.Zusatz;                          // Neue Methode (Objekterweiterung)
  Zwei.Kennung;                         // Vererbte Methode
  Eins.Free;
  Zwei.Free;
end.
```

Bildschirmausgabe:

```
Hallo Welt!
Ich bin Objekt [1],
<Enter> .....

Liebe Freunde,
das ist ein Zusatzangebot.
Ich bin Objekt [1],
 <Enter> .....
```

Die Bildschirmausgabe des Programmes zeigt deutlich, dass Objekt *Zwei* die vererbten Methoden *Eingeben*, *Ausgeben* und *Kennung* von seinem Vorfahren *Eins* erfolgreich verwendet. Außerdem enthält Objekt *Zwei* die neue Methode *Zusatz*, welche das neue Datenfeld *Data1* ebenfalls fehlerlos ausgibt. Damit sind **Vererbung** und **Erweiterung** demonstriert.

Zusatzbemerkung: In der Hierarchie der Objekttypen (Klassen) können ganz am Anfang Objekte mit leeren Methoden definiert werden, welche nur als Behälter (Container) für spätere Objektableitungen dienen. Jedoch wird von ihnen selbst nie eine Instanz (Vertreter) realisiert. Solche Objekte nennt man *Abstract*.

[05] Polymorphie und virtuelle Methoden (*oop3*)

Im Programm "**oop3.dpr"** ist die Methode *Kennung* unter dem gleichen Namen sowohl im Vorfahren *TEins* als auch im Nachfahren *TZwei* definiert. Sie wird nun, entsprechend dem Objekttyp, verschieden ausgestaltet. Soll beim Aufruf die vererbte Methode in jener Ausgestaltung ausgeführt werden, wie sie im Nachfahren definiert ist, dann muss diese Methode im Vorfahren mit ***virtuell*** und im Nachfahren mit ***override*** deklariert werden. Andernfalls würde die alte Methode *Kennung* des Vorfahren durch die gleichnamige neue Methode *Kennung* im Nachfahren **ohne** Vererbung ersetzt (verdeckt) werden. Der Nachfahre hat dann keine Möglichkeit, die ursprüngliche Methode seines Vorfahren aufzurufen.

In der Methode *Kennung* von *TZwei* soll es möglich sein, die gleichnamige Methode des Vorfahrens *TEins* unverändert aufzurufen. Damit diese in ihrer ursprünglichen Form ausgeführt werden kann, muss ihr der reservierte Bezeichner ***inherited*** vorangestellt werden.

Das nachfolgende Programm "**oop3.dpr"** demonstriert die Verwendung von virtuellen Methoden und deren Überschreibung (Overriding) in so genannten polymorphen Objekten. Mit "polymorph" wird der Sachverhalt der Vielgestaltigkeit bezeichnet.

```
program oop3;
{ Polymorphe Objekte mit virtuellen Methoden }

uses SysUtils;

type TEins = class(TObject)                  // Vorfahre
       Data : String;
       procedure Eingeben(s: String);
       procedure Ausgeben;
       procedure Kennung;  VIRTUAL;          // Virtuelle Methode
     end;

     TZwei = class(TEins)                    // Nachfahre
       Data1: String;                        // Neues Datenfeld (Eigenschaft)
       procedure Zusatz;                     // Neue Methode
       procedure Kennung;  OVERRIDE;         // Virtuelle Methode überschreiben
     end;


     procedure TEins.Eingeben(s: String);
     begin
       Data := s;
     end;

     procedure TEins.Ausgeben;
     begin
       Writeln;
       Writeln(Data);
     end;

     procedure TEins.Kennung;
     begin
       Writeln('  Ich bin OBJEKT [1], ');
       Writeln('  <Enter> ..... ');
       Readln;
     end;


     procedure TZwei.Zusatz;
     begin
       Data1 := '  das ist ein Zusatzangebot. ';
       Writeln(Data1);
     end;

     procedure TZwei.Kennung;
     begin
       INHERITED Kennung;                     // Vererbtes übernehmen
       Writeln('  und auch OBJEKT [2]. ');    // Neues hinzufügen
       Writeln('  <Enter> ..... ');
       Readln;
     end;


var  Eins : TEins;                           // Eins als Instanz von TEins
     Zwei : TZwei;                           // Zwei als Instanz von TZwei


begin
  Eins := TEins.Create;
  Zwei := TZwei.Create;
  Eins.Eingeben('  Hallo Welt! ');
  Eins.Ausgeben;
  Eins.Kennung;
  Zwei.Eingeben('  Liebe Freunde, ');         // Vererbte Methode
  Zwei.Ausgeben;                             // Vererbte Methode
  Zwei.Zusatz;                               // Neue Methode
  Zwei.Kennung;                              // Überschriebene,virtuelle Methode
  Eins.Free;
  Zwei.Free;
end.
```

Bildschirmausgabe:

```
Hallo Welt!
Ich bin Objekt [1],
<Enter> .....

Liebe Freunde,
das ist ein Zusatzangebot.
Ich bin Objekt [1],
<Enter> .....
und auch Objekt [2].
 <Enter> .....
```

Eine virtuelle Methode wird nicht zum Zeitpunkt der Kompilierung in den jeweiligen Programmcode direkt eingebunden, sondern erst bei der Ausführung des Programmes angesprungen (Prinzip der späten Bindung). Die Möglichkeit der selektiven Wahl einer Methode erst bei der Ausführung wird durch die Anlage einer Tabelle von Einsprungsadressen für die virtuellen Methoden eines Objektes ermöglicht (**VMT** = Virtuelle Methodentabelle bzw. **DMT** = Dynamische Tabelle). Diese Tabelle muss für jedes Objekt durch eine so genannte *Constructor*-Methode (z.B. *Create*) am Programmanfang erzeugt werden und wird dann beim Aufruf der virtuellen Methode angesprungen. Weil so jedes Objekt der Hierarchie seine eigene Methodentabelle besitzt, kann eine gleichnamige vererbte Methode in den Nachfahren verschiedengestaltig realisiert werden (Polymorphie). Das Gegenteil zu einer *Constructor*-Methode ist eine *Destructor*-Methode (z.B. *Free*). Diese dient zu Aufräumarbeiten im Speicher und entfernt das vom Constructor angelegte Objekt mitsamt seiner virtuellen bzw. dynamischen Methodentabelle. Auf den Unterschied zwischen virtuell (VMT) und dynamisch (DMT) soll hier nicht näher eingegangen werden.

In einem ähnlichen Sinn, aber ohne späte Bindung, realisiert auch der klassische Pascal-Compiler bereits eine Art von Polymorphie. Denken wir dabei nur an die Operation '+'. Sie kann sowohl zur Addition von Zahlen als auch zur Verknüpfung von Zeichenketten als auch zur Vereinigung von Mengen verwendet werden. Das ist bereits eine Form des Überschreibens einer gleichnamigen Operation in einer den jeweiligen Operanden entsprechenden Art und Weise.

[06] Wichtige Sprachelemente der OOP

Es sollen zum Schluss wichtige objektorientierte Sprachelemente als Ergänzung zu den klassischen Sprachelementen von DELPHI zusammengefasst werden.

CLASS	Deklariert einen Objekttyp (Klasse).
CONSTRUCTOR	Reserviert dynamischen Speicher zur Aufnahme eines Objektes, inklusive seiner virtuellen Methodentabelle (VMT).
DESTRUCTOR	Gibt den für ein Objekt reservierten Speicher wieder frei.
VIRTUAL	Deklariert eine virtuelle Methode im Vorfahren, d.h. trägt die Adresse der entsprechenden Routine in die virtuelle Methodentabelle des Objektes ein.
DYNAMIC	Wirkt ähnlich wie VIRTUAL, legt jedoch statt einer VMT eine DMT an.
OVERRIDE	Überschreibt eine vererbte, virtuelle Methode im Nachfahren.
INHERITED	Aktiviert im Nachfahren die originale, virtuelle Methode des Vorfahrens.
PRIVATE	Bezeichnet den lokalen, geschützten Teil in einer Objektdeklaration.
PUBLIC	Bezeichnet den globalen, öffentlichen Teil in einer Objektdeklaration.
PROPERTY	Deklariert eine Objekteigenschaft.
READ	Spezifiziert eine Eigenschaft zum Lesen eines Objektfeldes.
WRITE	Spezifiziert eine Eigenschaft zum Schreiben eines Objektfeldes.

[07] Der feine Unterschied: Feld oder Eigenschaft? (*farben*)

Ein Datenfeld eines Objektes belegt einen dafür reservierten Speicherplatz. Solche Datenfelder sollten im Sinne der OOP möglichst nicht öffentlich (public) deklariert werden, sondern privat und geschützt sein, um eine direkte Manipulation von außen zu vermeiden. Um solche private Felder zu verändern, müssen Routinen für kontrollierte Lesezugriffe und Schreibzugriffe definiert werden. Diese nennt man Eigenschaften (Properties). Demnach versteht man unter Eigenschaften einfache Methoden zum Lesen (read) und Schreiben (write) von Objektfeldern. Die Zuordnung dieser Eigenschaftsroutinen zu bestimmten Datenfeldern nennt man auch Kapselung. Properties sind somit die Schnittstellen zur Außenwelt, über welche auf die Datenfelder eines Objektes zugegriffen werden kann.

Wenn beispielsweise *FDat : FTyp* ein privates Datenfeld in einer Objektdeklaration ist, dann gibt es zwei Möglichkeiten, auf dieses Feld zuzugreifen. **Erstens:** Es können private Zugriffsroutinen definiert werden (*function GetFDat: FTyp bzw. procedure SetFDat(F: FTyp)*), denen dann eine öffentliche Property *Data* zugeordnet wird (*property Data: FTyp* ***read*** *GetFDat* ***write*** *SetFDat*). **Zweitens:** Es kann mit einer öffentlichen Property ohne eigene Zugriffsroutinen das private Datenfeld direkt gelesen oder geschrieben werden (*property Data: FTyp* ***read*** *FDat* ***write*** *FDat*). In beiden Fällen stellen *read* und *write* spezifische Schlüsselworte in der Propertydefinition dar.

In unserem letzten Programmprojekt "***farben***" wird ein Objekttyp *TCircle* definiert. Er besteht aus privaten Datenfeldern (Mittelpunkt *M*, Radius *R* und Farbe *F*) und privaten Lese- und Schreibroutinen für diese Felder. Öffentlich werden jene Eigenschaftsroutinen (Properties) deklariert, mit deren Hilfe von außen auf die internen Datenfelder zugegriffen werden kann. Zusätzlich gibt es eine Methode, welche den Kreis zeichnet. Die Konstruktormethode (*Erzeugen*) ermöglicht die Speicherreservierung und Initialisierung des Objektes. Eine entsprechende Destruktormethode (*Entfernen*) gibt den reservierten Speicher wieder frei. Die beiden Schlüsselworte *Constructor* und *Destructor* implementieren die Methoden *Create* und *Free* der allgemeinen Objektklasse *TObject*. Diese Methoden sind virtuell, denn sie werden polymorph im Nachfahren *TSect* verwendet.

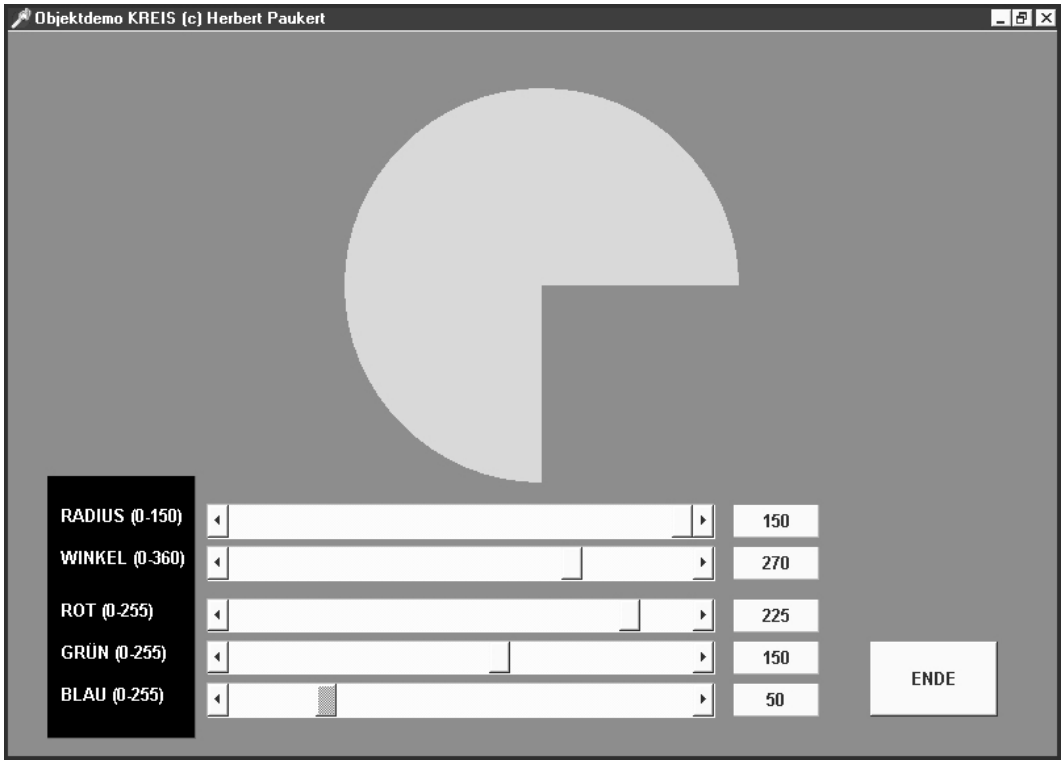
```
type TCircle = Class(TObject)    // TCircle als Nachfahre der allgemeinen Klasse

     Private                             // private Datenfelder
       M : TPoint;                       // privates Feld für Mittelpunkt
       R : Integer;                      // privates Feld für Radius
       F : TColor;                       // privates Feld für Farbe
       function GetCentre: TPoint;       // private Leseroutine (Mittelpunkt)
       procedure SetCentre(Z: TPoint);   // private Schreibroutine (Mittelpunkt)

     Public                              // öffentliche Eigenschaften
       property Zentrum : TPoint  read GetCentre write SetCentre;
       property Radius  : Integer read R write R;
       property Farbe   : TColor  read F write F;
       constructor Erzeugen(Form: TForm);  VIRTUAL;
       destructor  Entfernen;  VIRTUAL;
       procedure   Zeichnen(Canv: TCanvas);  VIRTUAL;
     end;
```

Als Nachfahre von *TCircle* ist der Objekttyp *TSect* konzipiert, der einen Kreissektor darstellt. Diese Klasse erbt alle Felder, Eigenschaften und Methoden ihres Vorfahrens *TCircle*. Sie enthält aber auch das zusätzliche private Datenfeld ***W***, welches den Zentriwinkel des Kreissektors bestimmt. Für den Zugriff auf ***W*** ist die öffentliche Property *Winkel* vorgesehen. Außerdem müssen die entsprechenden virtuellen Konstruktoren und Destruktoren und die virtuelle Methode *Zeichnen* überschrieben werden. Den kompletten Programmcode findet der Leser im nachfolgenden Listing.

Im eigentlichen Programm kann der Anwender über fünf Scrollbars die Länge des Radius, die Größe des Sektorwinkels und die Farbanteile Rot, Grün, Blau schrittweise verändern. Über die Methode Zeichnen werden dann diese Veränderungen sofort in der grafischen Darstellung des Objektes sichtbar. Das Programmprojekt "***farben***" demonstriert alle wichtigen Strukturen, welche für die OOP von Bedeutung sind.




```
unit farben_u;
{ Farben, Objektdemo (c) H.Paukert }

interface
uses Windows, Messages, SysUtils, Classes, Graphics,
     Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;                    // Beschriftung 'Radius'
    Label2: TLabel;                    // Beschriftung 'Rot'
    Label3: TLabel;                    // Beschriftung 'Gruen'
    Label4: TLabel;                    // Beschriftung 'Blau'
    Label5: TLabel;                    // Beschriftung 'Winkel'
    Panel1: TPanel;                    // Ausgabe von Radius
    Panel2: TPanel;                    // Ausgabe von Rot
    Panel3: TPanel;                    // Ausgabe von Gruen
    Panel4: TPanel;                    // Ausgabe von Blau
    Panel5: TPanel;                    // Ausgabe von Winkel
    ScrollBar1: TScrollBar;            // Einstellung von Radius
    ScrollBar2: TScrollBar;            // Einstellung von Rot
    ScrollBar3: TScrollBar;            // Einstellung von Gruen
    ScrollBar4: TScrollBar;            // Einstellung von Blau
    ScrollBar5: TScrollBar;            // Einstellung von Winkel
    Button1: TButton;                  // Programm beenden
    procedure FormCreate(Sender: TObject);
    procedure ScrollBar1Scroll(Sender: TObject; ScrollCode: TScrollCode;
                               var ScrollPos: Integer);
    procedure ScrollBar2Scroll(Sender: TObject; ScrollCode: TScrollCode;
                               var ScrollPos: Integer);
    procedure ScrollBar3Scroll(Sender: TObject; ScrollCode: TScrollCode;
                               var ScrollPos: Integer);
    procedure ScrollBar4Scroll(Sender: TObject; ScrollCode: TScrollCode;
                               var ScrollPos: Integer);
    procedure ScrollBar5Scroll(Sender: TObject; ScrollCode: TScrollCode;
                               var ScrollPos: Integer);
    procedure Button1Click(Sender: TObject);

    private { Private declarations }
    public  { Public declarations }
  end;
```

```
var Form1: TForm1;

implementation
{$R *.DFM}

type TCircle = Class(TObject)          // Nachfahre vom allgemeinen TObject
     Private
        M : TPoint;                    // privates Feld für Mittelpunkt
        R : Integer;                   // privates Feld für Radius
        F : TColor;                    // privates Feld für Farbe
        function  GetCentre: TPoint;   // private Leseroutine für Mittelpunkt
        procedure SetCentre(Z: TPoint); // private Schreibroutine für Mittelp.
     Public
        property Zentrum: TPoint  Read GetCentre Write SetCentre;
        property Radius : Integer Read R Write R;
        property Farbe  : TColor  Read F Write F;
        constructor Erzeugen(Form: TForm); VIRTUAL;
        destructor  Entfernen; VIRTUAL;
        procedure   Zeichnen(Canv: TCanvas); VIRTUAL;
     end;

     TSect = Class(TCircle)            // Nachfahre von TCircle
     Private
        W : Integer;                   // privates Feld für Zentriwinkel
     Public
        property Winkel: Integer Read W Write W;
        constructor Erzeugen(Form: TForm); OVERRIDE;
        destructor  Entfernen; OVERRIDE;
        procedure   Zeichnen(Canv: TCanvas); OVERRIDE;
     end;

     function TCircle.GetCentre : TPoint;
     { Liefert den Mittelpunkt des Kreisobjektes }
     begin
       Result := M;
     end;

     procedure TCircle.SetCentre(Z: TPoint);
     { Setzt den Mittelpunkt des Kreisobjekts }
     begin
       M := Z;
     end;

     constructor TCircle.Erzeugen(Form: TForm);
     { Reserviert Speicherplatz für ein Kreisobjekt und initialisiert es }
     var P : TPoint;
     begin
       P.X := Form.Width div 2;
       P.Y := Form.Height div 3;
       Zentrum := P;
       Radius  := 0;
       Farbe := RGB(0,0,0);
     end;

     destructor TCircle.Entfernen;
     { Entfernt das Kreisobjekt aus dem Hauptspeicher }
     begin
                                       // hier müssen KEINE Anweisungen stehen
     end;

     procedure TCircle.Zeichnen(Canv: TCanvas);
     { Zeichnet das Kreisobjekt auf der jeweiligen Formularleinwand }
     var X,Y,Ra: Integer;
           Col: TColor;
     begin
       X := Zentrum.X;
       Y := Zentrum.Y;
       Ra:= Radius;
       Col := Farbe;
```

```
      With Canv do begin
        Pen.Color   := Col;
        Brush.Color := Col;
        Brush.Style := bsSolid;
        Ellipse(X-Ra,Y-Ra,X+Ra,Y+Ra);
      end;
    end;

    Constructor TSect.Erzeugen(Form: TForm);
    { Reserviert Speicherplatz für ein Sektorobjekt und initialisiert es }
    begin
      INHERITED Erzeugen(Form);
      Winkel := 360;
    end;

    Destructor TSect.Entfernen;
    { Entfernt das Sektorobjekt aus dem Hauptspeicher }
    begin
      INHERITED Entfernen;
    end;

    procedure TSect.Zeichnen(Canv: TCanvas);
    { Zeichnet das Sektorobjekt auf der jeweiligen Formularleinwand }
    var X,Y,X1,Y1,Ra: Integer;
                 Wi: Real;
                Col: TColor;
    begin
      X := Zentrum.X;
      Y := Zentrum.Y;
      Ra:= Radius;
      Col:= Farbe;
      Wi := Pi*Winkel/180;
      X1 := X + Round(Ra * Cos(Wi));
      Y1 := Y - Round(Ra * Sin(Wi));
      With Canv do begin
        Pen.Color   := Col;
        Brush.Color := Col;
        Brush.Style := bsSolid;
        if Wi > 0 then Pie(X-Ra,Y-Ra,X+Ra,Y+Ra,X+Ra,Y,X1,Y1);
      end;             // "Pie" zeichnet einen Ellipsensektor
    end;

var  Sektor: TSect;                    // Objektinstanz
     Rot, Gruen, Blau : Byte;          // Farbbytes

procedure TForm1.FormCreate(Sender: TObject);
{ Objekt im Speicher anlegen }
begin
   Form1.Color := RGB(160,190,160);
   Sektor := TSect.Erzeugen(Form1);
end;

procedure TForm1.ScrollBar1Scroll(Sender: TObject; ScrollCode: TScrollCode;
                                  var ScrollPos: Integer);
{ Verändern des Radius }
var P : Integer;
    F : TColor;
    Ra: Integer;
begin
   P := ScrollPos;
   Panel1.Caption := IntToStr(P);
   Ra:= Sektor.Radius;
   F := Sektor.Farbe;
   if Ra > P then begin
      Sektor.Farbe := Form1.Color;
      Sektor.Zeichnen(Form1.Canvas);    // löscht den alten Sektor
   end;
   Sektor.Radius := P;
   Sektor.Farbe  := F;
   Sektor.Zeichnen(Form1.Canvas);       // zeichnet den neuen Sektor
end;
```

```
procedure TForm1.ScrollBar2Scroll(Sender: TObject; ScrollCode: TScrollCode;
                                  var ScrollPos: Integer);
{ Verändern des ROT-Anteils der Farbe }
var Col : TColor;
begin
   Rot := ScrollPos;
   Panel2.Caption := IntToStr(Rot);
   Col := RGB(Rot,Gruen,Blau);
   Sektor.Farbe := Col;
   Sektor.Zeichnen(Form1.Canvas);
end;

procedure TForm1.ScrollBar3Scroll(Sender: TObject; ScrollCode: TScrollCode;
                                  var ScrollPos: Integer);
{ Verändern des GRÜN-Anteils der Farbe }
var Col : TColor;
begin
   Gruen := ScrollPos;
   Panel3.Caption := IntToStr(Gruen);
   Col := RGB(Rot,Gruen,Blau);
   Sektor.Farbe := Col;
   Sektor.Zeichnen(Form1.Canvas);
end;

procedure TForm1.ScrollBar4Scroll(Sender: TObject; ScrollCode: TScrollCode;
                                  var ScrollPos: Integer);
{ Verändern des BLAU-Anteils der Farbe }
var Col : TColor;
begin
   Blau := ScrollPos;
   Panel4.Caption := IntToStr(Blau);
   Col := RGB(Rot,Gruen,Blau);
   Sektor.Farbe := Col;
   Sektor.Zeichnen(Form1.Canvas);
end;

procedure TForm1.ScrollBar5Scroll(Sender: TObject; ScrollCode: TScrollCode;
                                  var ScrollPos: Integer);
{ Verändern des Zentriwinkels }
var P : Integer;
    F : TColor;
    Wi: Integer;
begin
   P := ScrollPos;
   Panel5.Caption := IntToStr(P);
   Wi:= Sektor.Winkel;
   F := Sektor.Farbe;
   if Wi > P then begin
      Sektor.Farbe  := Form1.Color;
      Sektor.Zeichnen(Form1.Canvas);    // löscht den alten Sektor
   end;
   Sektor.Winkel := P;
   Sektor.Farbe  := F;
   Sektor.Zeichnen(Form1.Canvas);       // zeichnet den neuen Sektor
end;

procedure TForm1.Button1Click(Sender: TObject);
{ Objekt aus dem Speicher entfernen und Programm beenden }
begin
   Sektor.Entfernen;
   Application.Terminate;
end;

end.
```