Einführung

in die

Objektorientierte Programmierung

Bearbeitung: Dieter Lindenberg

Version 2011

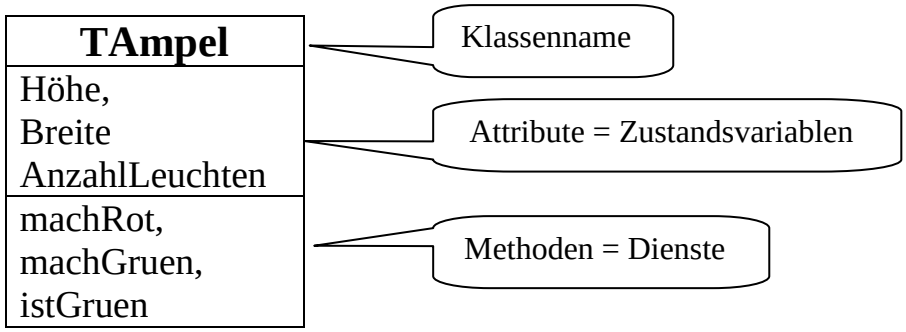
Inhalt

Klassen und Objekte, Grundlagen

Wir werden in diesem Kurs Programme erstellen, die mehrere Units und teilweise sogar mehrere Formblätter enthalten. Außerdem werden die Programme durchaus ziemlich umfangreich sein. Damit alle Beteiligten zusammenarbeiten können, sind u.a. auch gemeinsame Regeln für Namensgebungen notwendig.

Das Hauptformblatt lautet ***Main***, falls kein besserer Name gefunden wird. Die zugehörige ***Unit***, bzw. das zugehörige ***Modul***, wird unter dem Namen ***mMain.pas*** gespeichert. Alle *Unitnamen* beginnen mit dem Kleinbuchstaben *m*. Der Quelltext für das Hauptprogramm erhält automatisch den Namen der gespeicherten Datei. Projekte werden unter dem Namen ***pMain.dpr*** gespeichert.

Eine Klasse wird durch ein sog. UML-Klassendiagramm (**U**nified **M**odelling **L**anguage) dargestellt. Dieses ist dreigeteilt und enthält den Namen, die Attribute und die Dienste der Klasse. UML-Diagramme enthalten keine Typbezeichnungen (weil diese von der gewählten Programmiersprache abhängig wären).



Eine Klasse besitzt Attribute und Methoden (ein anderer Name für *Methoden* ist *Dienste*). Ein Objekt einer Klasse wird auch Instanz genannt.

Attribute sind Variablen, die zu jedem Objekt einer Klasse gehören. Verschiedene Objekte derselben Klasse haben üblicherweise unterschiedliche Attributwerte. Aus diesem Grund nennt man die Attribute auch *Zustandsvariablen*.

Eine Methode ist eine Prozedur oder Funktion, die zu einer Klasse gehört und von allen Klassenobjekten gemeinsam genutzt werden kann.

In Delphi gibt es sehr viele schon vordefinierte Klassen, z.B. *TForm*, *TEdit*, *TButton*, *TImage*, *TTimer*. In diesem Kurs werden wir eigene neue Klassen

definieren. Alle Klassennamen sollten mit dem Buchstaben T (für Type) beginnen.

Die Namen der von Delphi zur Verfügung gestellten Objekte sollen alle mit einem oder mehreren kennzeichnenden Buchstaben beginnen:
EdName: Editfeld, RbName: Radiobutton, BtName: Button, LbName: Listbox, ImName: Bilder, MemName: Memofeld, LName: Label
Alle gewählten Namen müssen sinnvoll sein.

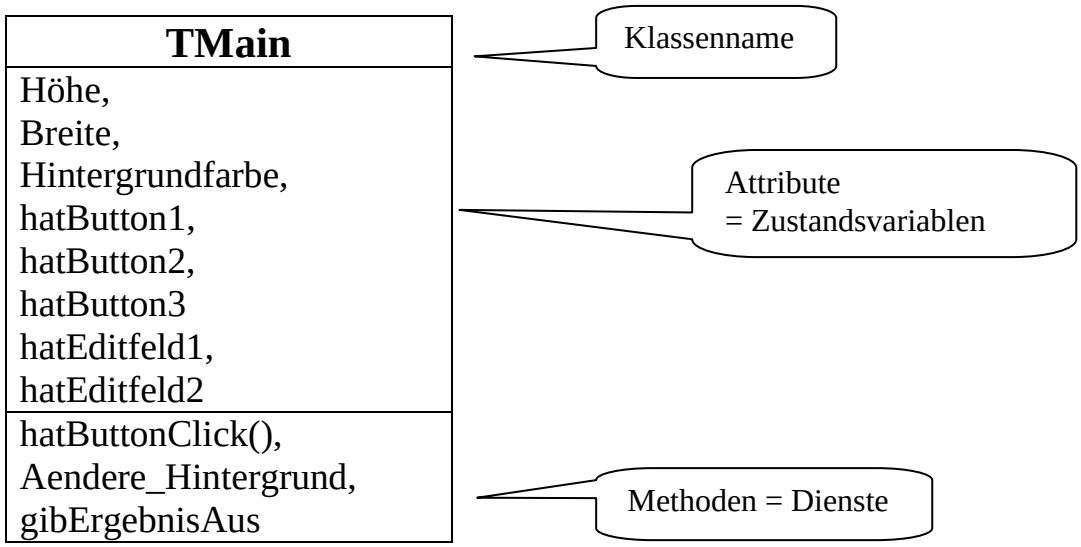
Durch die Existenz des Namens einer Objektvariablen gibt es aber das Objekt selbst noch nicht. Dieses wird entweder durch die Delphi-Programmier-umgebung schon aufgrund der Gestaltung des Formblattes automatisch erzeugt, oder aber erst im laufenden Programm durch Aufruf eines entsprechenden Konstruktorbefehls:

```
procedure Beispiel;
VAR A: TAmpel;
BEGIN
  A := TAmpel.create;
  ..............
END;
```

Sehr oft „haben“ Objekte einer Klasse wiederum andere Objekte. Zum Beispiel besitzt üblicherweise unser Hauptformblatt ein oder mehrere Objekte vom Typ TButton, TEdit, TLabel, TTimer usw.
Diese sog. ***Hat-Beziehung*** wird dann so dargestellt:

Üblicherweise werden, wenn die Beziehungen zwischen mehreren Klassen dargestellt werden soll, die jeweiligen Attribute und Dienste nicht nochmal dargestellt.

Innerhalb einer einzigen Klasse wird diese Hat-Beziehung durch ein oder mehrere entsprechende Attribute realisiert:



Schutzklassen

In größeren Programmen werden Teilmodule üblicherweise von vielen unterschiedlichen Programmierern erstellt. Ein fertig gestelltes Modul wird anschließend allen anderen Programmierern, die es gebrauchen könnten, zur Verfügung gestellt. Dabei darf es natürlich nicht erlaubt sein, dass jeder Programmierer für seine Zwecke kleine Änderungen an diesem Modul vornimmt. Ansonsten gäbe es in kürzester Zeit viele unterschiedliche Versionen dieses Moduls und niemand wüsste mehr, was dieses Modul denn nun wirklich macht.

Der Ersteller des Moduls bestimmt nun, welche Methoden des Moduls **benutzt** werden dürfen (durch Aufruf der entsprechenden Dienste). Diese Methoden werden unter der Schutzklasse ***public*** eingetragen. Ein Benutzer dieses Moduls (üblicherweise ein anderer Programmierer) kann nur diese Methoden aufrufen; an deren Programmierung ändern kann er natürlich auch nichts.

Davon abgesehen benötigt das Modul natürlich auch Prozeduren, Funktionen und Variablen, die von einem anderen Programmierer nicht benutzt werden dürfen, bzw. von deren Existenz ein anderer Programmierer überhaupt nichts wissen muss (weil es ihn normalerweise auch nicht interessiert, wie das Modul intern funktioniert). Derartige Attribute und Methoden werden unter der Schutzklasse ***private*** eingetragen.

Insbesondere sollten grundsätzlich alle Attribute der Schutzklasse *private* angehören.
Beispiel: Angenommen, man hätte eine Klasse *TRechteck* mit den Attributen *Länge, Breite* und *Flächeninhalt*. Wenn jemand willkürlich nur das Attribut *Länge* ändern würde, so hätte dies Auswirkungen auf das Attribut *Flächeninhalt*. Eine mögliche Änderung des Attributes *Länge* darf deshalb nur über eine öffentlich zur Verfügung gestellte Methode *setzeLaenge(n:INTEGER)* vorgenommen werden. Diese vom Ersteller der Klasse *TRechteck* implementierte Methode berücksichtigt gleichzeitige Änderungen des Attributes *Flächeninhalt*.

Benutzt man die Delphi-Programmierumgebung, so werden alle automatisch erzeugten Attribute und Methoden oberhalb des ***private***-Teils eingetragen, also scheinbar außerhalb der Schutzklassen. Dies ist eine Besonderheit der Programmierumgebung von Delphi. Aber alle diese automatisch vorgenommenen Eintragungen sind leider öffentlich (Schutzklasse *public*).
Es stellt sich natürlich die Frage, warum diese Eintragungen nicht unter *public* gemacht werden. Das liegt daran, dass Delphi keine rein objektorientierte Sprache ist, sondern eine Erweiterung der Sprache Pascal mit der Möglichkeit,

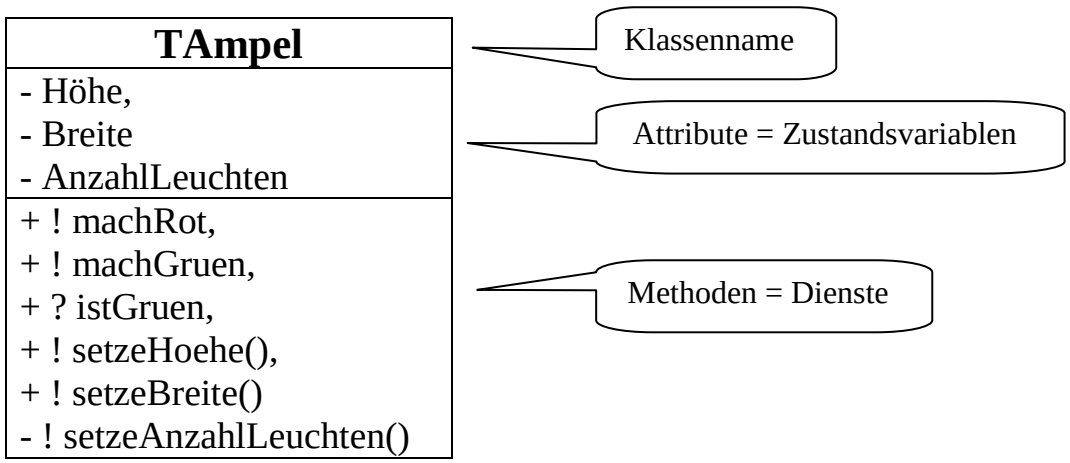
auch objektorientiert programmieren zu können.

Man kann innerhalb einer Unit (= eines Moduls) mehrere Klassen definieren. Innerhalb einer Delphi-Unit kann man leider auf alles zugreifen. Die Schutzklasse *private* schützt also nur dann, wenn versucht wird, auf Attribute oder Methoden einer Klasse zuzugreifen, die in einem anderen Modul (Unit) definiert ist.

Im Klassendiagramm wird die Schutzklasse hinzugefügt. Dabei gilt:
„+“ = public, “–“ = private

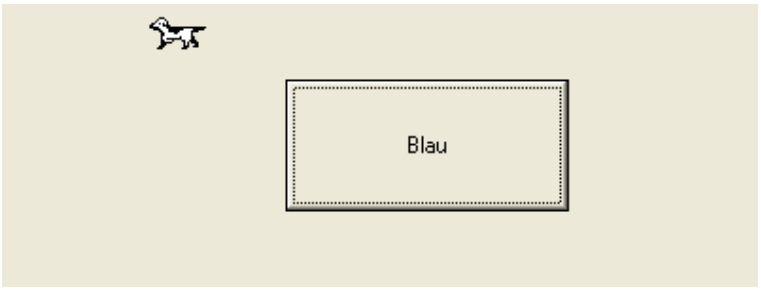
Zusätzlich unterscheidet man bei den Diensten zwischen Aufträgen (in Delphi: Prozeduren), gekennzeichnet durch ein Ausrufungszeichen und Anfragen (in Delphi: Funktionen), die mit einem Fragezeichen gekennzeichnet werden.

Übergabeparameter werden oft nur durch zwei Klammern angedeutet.

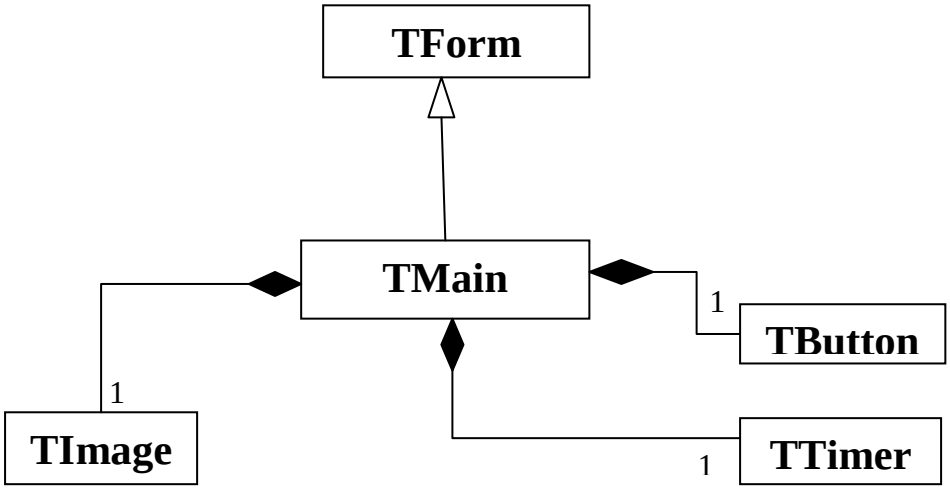


Wenn man Objekte erst während der Programmlaufzeit erzeugen will, so muß man dafür sorgen, dass diejenigen Module (Units), in denen die, den Objekten entsprechenden Klassen definiert sind, unter USES eingebunden werden. Die Units haben unter Windows leider oft andere Namen als unter Linux.
Beispiel: die Klassen *TEdit*, *TLabel* und *TButton* werden in Windows in der Unit *StdCtrls* definiert, in Linux in der Unit *QExtCtrls*. Die Klassen *TImage* und *TTimer* werden unter Windows in *ExtCtrls* definiert.

Im folgenden Beispielprogramm werden alle Objekte erst zur Laufzeit erzeugt. Das Formblatt, also das Objekt namens *Main* vom Typ *TMain*, enthält einen Button, mit dem man die Hintergrundfarbe einstellen kann und ein kleines Bild, welches sich timergesteuert hin und her bewegt.



Das zugehörige Klassen-Beziehungs-Diagramm sieht so aus:



Das etwas ausführlichere Klassendiagramm von TMain:

TMain
− Color, left, top, width, − height, delta
+ ! Create(), − ! setbounds(), − ! machHintergrundBlau, − ! bewegeBild

Der ***Konstruktor*** *Create* der Klasse *TMain* muss von außen aufgerufen werden können, damit ein solches Objekt (Formblatt) erst erzeugt wird.

Man benötigt in diesem Beispielsprogramm eine Variable namens *Main*, also ein Objekt der Klasse *TMain*. Falls ein Objekt dieser Klasse von einer anderen, fremden Unit benötigt würde, so müsste der entsprechende Objekt-Variablenname in der anderen Unit stehen. In dieser Unit unseres Beispielprogrammes dürfte er dann allerdings nicht mehr stehen.

Bemerkung: wir benutzen in diesem Beispielsprogramm Objekte (Formblatt, Image, Timer), die von der Delphi-Programmierumgebung schon bereitgestellt werden und demnach Standardobjekte sind. Diese Standardobjekte haben bereits eine Vielzahl von Attributen und Methoden, die wir für unser Beispiel eigentlich gar nicht benötigten. Später werden wir auch von Grund auf eigene Objekte kreieren.

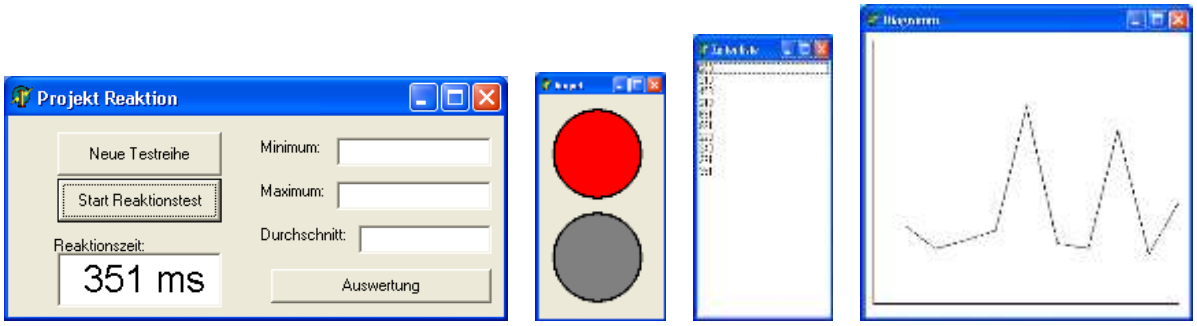
```
unit mOOP1;

interface

uses  Windows, ..... , StdCtrls, ExtCtrls;

type
  TMain = class(TForm)
   private
     hatButton: TButton;
     hatBild:   TImage;
     hatTimer:  TTimer;
     delta:     INTEGER;
     procedure  machHintergrundBlau(Sender:TObject);
     procedure  bewegeBild(Sender:TObject);
   public
     Constructor Create(AOwner: TComponent); override;
    {Damit wird der Konstruktor der Oberklasse über-
     schrieben. Das geht nur bei gleicher Parameter-
     übergabe. Voraussetzung: in der Oberklasse
     besitzt der Konstruktor die Eigenschaft virtual}
  end;

var  Main: TMain;

implementation
{$R *.dfm}
```

```
procedure TMain.machHintergrundBlau(Sender:TObject);
begin
  Main.Color := $00FF0000
end;

procedure TMain.bewegeBild(Sender:TObject);
begin
 IF (hatBild.Left + hatBild.Width > Main.Width-10) OR
    (hatBild.Left < 0) THEN delta := -delta;
 hatBild.Left := hatBild.Left + delta
end;

Constructor TMain.Create(AOwner: TComponent);
Begin
  Inherited Create(AOwner);
  {wichtig, damit überhaupt erst einmal ein leeres
   Formblatt erzeugt wird.}
  SetBounds(100,0,400,150); // left, top, width,
                            // height von Main
  hatButton := TButton.Create(Main);
  {Damit steht hatButton in der Komponentenliste des
   Formblattes Main}
  With hatButton DO BEGIN
    Parent := Main;  {Damit wird festgelegt, in
          welchem Elternfenster der Button erscheint}
    SetBounds(150, 40, 150, 70); //innerhalb von Main
    Caption := 'Blau';
    OnClick := machHintergrundBlau
  END;

  hatBild := TImage.Create(Main);
  hatBild.Picture.LoadFromFile('Dog.ico');
  hatBild.AutoSize := True;
  hatBild.Parent := Main;

  hatTimer := TTimer.Create(Main);
  hatTimer.Interval := 10;
  hatTimer.OnTimer := bewegeBild;
  {Timer sind keine visuellen Objekte, benötigen also
   auch keine Parentangabe}
  delta := 1;
end;  // of TMain.FormCreate
end.  // of Hauptprogramm
```

Projekt: Messung der Reaktionszeit

In diesem Projekt werden sämtliche Objekte erst während der Laufzeit des Programms erzeugt. Die einzige Ausnahme ist das leere Hauptformblatt namens *Main*. Natürlich kann man dann auch die verschiedenen Ereignisse (Mausklick auf Button bzw. Tastaturereignis) erst zur Laufzeit mit bestimmten Prozeduren verbinden.

In seiner Endfassung sollte das Projekt folgendermaßen aussehen:



In einer Testreihe ermittelt man mehrere Male seine Reaktionsgeschwindigkeit, indem man möglichst schnell nach dem Umschalten der Ampel von rot auf grün irgendeine beliebige Taste drückt. Die benötigte Zeit wird gemessen, in eine Liste eingefügt und in einem Diagramm dargestellt.
Zum Schluss der Testreihe kann man sich die beste, schlechteste und durchschnittlich benötigte Zeit angeben lassen.

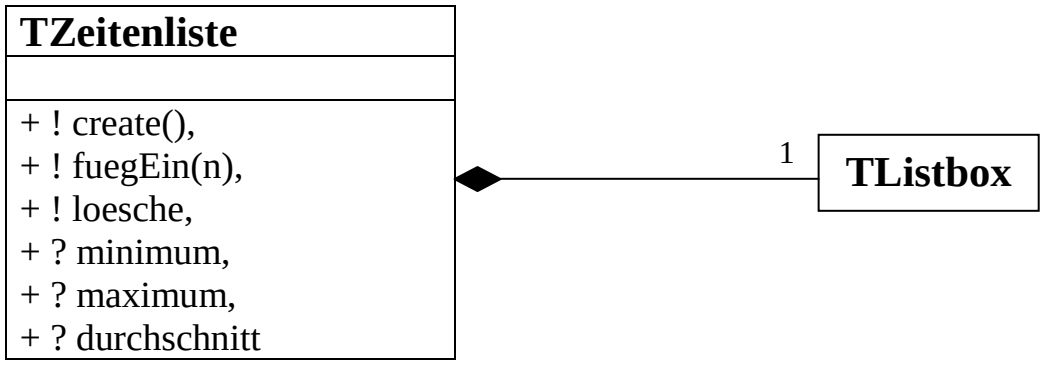
TAmpel
- AmpelGruen
+ ! Create(), + ! machRot, + ! machGruen, + ? istGruen, - ! FormPaint(), - ! FormResize()

```
unit mAmpel;

uses ….
type
  TAmpel = class(TForm)
  private
    AmpelGruen: Boolean;
    procedure FormPaint(Sender: TObject);
    procedure FormResize(Sender: TObject);
  public
    constructor Create(AOwner: TComponent); override;
    procedure machRot;
    procedure machGruen;
    function IstGruen: Boolean;
  end;

implementation….

procedure TAmpel.machRot;
begin
  with Canvas Do begin
    pen.Width := 3;
    Brush.Color := clRed;
    Ellipse(20,20,140,140);
    Brush.Color := clGray;
    Ellipse(20,160,140,280)
  end;
  AmpelGruen := False
end;
```

```
procedure TAmpel.machGruen;
begin
  with Canvas Do begin
    pen.Width := 3;
    Brush.Color := clLime;
    Ellipse(20,160,140,280);
    Brush.Color := clGray;
    Ellipse(20,20,140,140)
  end;
  AmpelGruen := True
end;


function TAmpel.IstGruen: Boolean;
begin
  IstGruen := AmpelGruen
end;


procedure TAmpel.FormPaint(Sender: TObject);
{Diese Prozedur wird vom Betriebssystem automatisch immmer dann
aufgerufen, wenn das sog. OnPaint-Ereignis ausgelöst wurde. Dies wird immer
dann ausgelöst, wenn das Betriebssystem feststellt, dass ein Teil der Darstellung
auf dem Bildschirm nicht mehr gültig ist, z.B. wenn ein anderes Fenster, das
vorher die Komponente überdeckte, nun verschoben wird. Deshalb muß die
Ampel nun neu gezeichnet werden.}
begin
 IF istgruen THEN machGruen ELSE machRot
end;


procedure TAmpel.FormResize(Sender: TObject);
{ Diese Prozedur wird vom Betriebssystem automatisch aufgerufen, wenn das
sog. OnResize-Ereignis ausgelöst wurde. Falls die Größe der Ampel vom
Benutzer während des Spieles geändert wurde, wird sie neu gezeichnet.}
begin
  IF istgruen THEN machGruen ELSE machRot
end;
```

Beachte im Folgenden, dass diejenigen Methoden, welche man mit den Ereignissen verknüpfen will, in ihrer (obigen) Deklaration den Parameter (Sender: TObject) haben müssen, obwohl jetzt bei dem eigentlichen Ereignis-Methode-Verknüpfungsbefehl dieser Parameter gar nicht erwähnt wird.

```
constructor TAmpel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  OnPaint := FormPaint;
  OnResize := FormResize
end;

end.
```



unit mZeitenliste;

```
……..
type
  TZeitenliste = class(TForm)
    private
      LBox: TListBox;
    public
      constructor Create(AOwner: TComponent); override;
      procedure Fuegein(n:Integer);
      procedure Loesche;
      function Minimum: INTEGER;
      function Maximum: INTEGER;
      function Durchschnitt: INTEGER;
    end;
……..
procedure TZeitenliste.Loesche;
BEGIN
  LBox.Clear
END;
```

```
procedure TZeitenliste.Fuegein(n:Integer);
BEGIN
  LBox.Items.Add(IntToStr(n))
END;


function TZeitenliste.Minimum: INTEGER;
VAR min, help, i: INTEGER;
BEGIN
   min := StrToInt(LBox.Items[0]);
   FOR i := 0 TO LBox.Items.Count - 1 DO Begin
     help := StrToInt(LBox.Items[i]);
     IF help < min THEN min := help
   End;
   result := min
END;


function TZeitenliste.Maximum: INTEGER;
 VAR max, help, i: INTEGER;
 BEGIN
   max := StrToInt(LBox.Items[0]);
   FOR i := 0 TO LBox.Items.Count - 1 DO Begin
     help := StrToInt(LBox.Items[i]);
     IF help > max THEN max := help
   End;
   result := max
 END;

function TZeitenliste.Durchschnitt: INTEGER;
 VAR summe, i: INTEGER;
 BEGIN
   summe := 0;
   FOR i := 0 TO LBox.Items.Count - 1 DO
     summe := summe + StrToInt(LBox.Items[i]);
   result := Round(summe / LBox.Items.Count)
 END;

constructor TZeitenliste.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  LBox := TListbox.Create(self);
  LBox.Parent := self;
  LBox.SetBounds(0,0,self.width-25,self.height-30);
end;
```

TDiagramm
- Ymax, - Rand
+ ! create(), + ! tragEin(n), - ! zeichneAchsen, + ! loesche,

unit mDiagramm;

```
type
  TDiagramm = class(TForm)
    private
      Nummer, ymax, Randunten, Randoben, Randlinks,
      Randrechts: INTEGER;
      procedure ZeichneAchsen;
    public
      constructor Create(AOwner: TComponent); override;
      procedure TragEin(Zeit:INTEGER);
      procedure loesche;
  end;
……..
```

```
procedure TDiagramm.ZeichneAchsen;
BEGIN
 With Canvas DO BEGIN
   MoveTo(Randlinks,Randoben);
   LineTo(Randlinks,Height-Randunten);
   LineTo(Width - Randrechts,Height - Randunten);
 END
END;
```

```
procedure TDiagramm.loesche;
BEGIN
  With Canvas DO BEGIN
    Brush.Color := clwhite;
    Rectangle(0,0, width, height)
  END;
  Nummer := 0;
  ZeichneAchsen
END;
```

```
procedure TDiagramm.TragEin(Zeit:INTEGER);
VAR xb, yb: INTEGER;
BEGIN
 INC(Nummer);
 xb := Round(Randlinks+Nummer*(Width-Randlinks-Randrechts)/10);
 yb := Round(Zeit*(Randoben+Randunten-Height)/ymax+Height-
       Randunten);
 With Canvas DO
    IF Nummer = 1 THEN BEGIN
      MoveTO(xb, yb);
      Pixels[xb, yb] := clblack
    END
    ELSE LineTo(xb, yb )
END;


constructor TDiagramm.Create(AOwner: TComponent);
begin
 inherited create(AOwner);
 ymax := 1000;
 Randlinks := 10;  Randrechts := 20;
 Randoben := 10;  Randunten := 50;
 Loesche
end;
```

```
Unit mReaktion1;          // Hauptprogramm
……
interface
uses  Windows, … mAmpel, mDiagramm, mZeitenliste;

type
  TMain = class(TForm)
  private
    BtTestreihe : TButton;
    BtReaktionstest : TButton;
    EdReaktionszeit : TEdit;
    LbReaktionszeit :TLabel;
    TiReaktion :TTimer;

    LbMinimum: TLabel;
    LbMaximum: TLabel;
    LbDurchschnitt: TLabel;
    EdMinimum: TEdit;
    EdMaximum: TEdit;
    EdDurchschnitt: TEdit;
    BtAuswertung: TButton;


    Startzeit, Stopzeit: TDateTime;
    Zeitenliste: TZeitenliste;
    Diagramm: TDiagramm;
    Ampel: TAmpel;
    procedure TiReaktionTimer(Sender: TObject);
    procedure BtTestreiheClick(Sender: TObject);
    procedure BtReaktionstestClick(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key:
                          Word; Shift: TShiftState);
    procedure BtAuswertungClick(Sender: TObject);
  public
    constructor Create(AOwner: TComponent); override;
  end;

var  Main: TMain;
implementation
….
```

```
procedure TMain.TiReaktionTimer(Sender: TObject);
begin
  Ampel.machGruen;
  Startzeit := now;
  TiReaktion.Enabled := False;
end;


procedure TMain.BtReaktionstestClick(Sender: TObject);
begin
  EdReaktionsZeit.Text := ' ';
  Ampel.machRot;
  TiReaktion.Interval := 2000 + Random(3000);
  TiReaktion.Enabled := True
end;


procedure TMain.BtTestreiheClick(Sender: TObject);
begin
  IF Ampel = NIL THEN BEGIN
    Ampel :=TAmpel.Create(Main);
    Ampel.Left := Main.Left + Main.width;
    Ampel.Top :=0;
    Ampel.Show;
  END;
  Ampel.machRot;

  BtReaktionstest.enabled := True;
  BtReaktionstest.Visible := True
  BtTestreihe.Visible := False;

  IF Zeitenliste = NIL THEN BEGIN
    Zeitenliste :=TZeitenliste.Create(Main);
    Zeitenliste.Top := 0;
    Zeitenliste.Left := Left+Width+Ampel.Width;
    Zeitenliste.Show
  END;
  Zeitenliste.loesche;

  IF Diagramm = NIL THEN BEGIN
    Diagramm :=TDiagramm.Create(Main);
    Diagramm.SetBounds(0,Top+Height,Width,350);
    Diagramm.Show;
  END;
  Diagramm.loesche;
```

```
  EdReaktionszeit.Text := '';
  EdMinimum.Text := '';
  EdMaximum.Text := '';
  EdDurchschnitt.Text := '';
end;


procedure TMain.BtAuswertungClick(Sender: TObject);
begin
 IF EdReaktionszeit.Text <> '' THEN BEGIN
   Zeitenliste.Show;
   EdMinimum.Text := IntToStr(Zeitenliste.Minimum);
   EdMaximum.Text := IntToStr(Zeitenliste.Maximum);
   EdDurchschnitt.Text:= IntToStr(Zeitenliste.Durchschnitt)
 END
 ELSE Showmessage('nicht möglich !')
end;


procedure TMain.FormKeyDown(Sender: TObject; var Key:
                            Word; Shift: TShiftState);
{wichtig: ein Tastendruck wird nur von dem aktiven Formblatt registriert. Das
Drücken der Leertaste entspricht dem Mausklick auf denjenigen Button, der den
Fokus hat.}
VAR Zeitdifferenz: TDateTime;
    stunde, minute, sekunde, ms: WORD;
begin
  Stopzeit := now;
  Zeitdifferenz := Stopzeit - Startzeit;
  DecodeTime(Zeitdifferenz, stunde, minute, sekunde, ms);
  IF Ampel.IstGruen THEN BEGIN
    EdReaktionszeit.Text:=IntToStr(1000*sekunde + ms)+' ms';
    Zeitenliste.Fuegein(sekunde * 1000 + ms);
    Diagramm.TragEin(sekunde * 1000 + ms);
    Ampel.machRot
  END
  ELSE showmessage('Ampel ist rot')
end;
```

```
constructor Create(AOwner: TComponent);
begin
  inherited Create(AOwner);

  BorderStyle := bsSingle;  //Größe unveränderbar;
  Left := 0;
  Top := 0;
  Width := 340;
  Height := 250;
  Caption := 'Projekt Reaktionstest';
  Keypreview := True;

  BtTestreihe := TButton.Create(Main);
  BtTestreihe.Parent := Main;
  BtTestreihe.SetBounds(0,20,120,32);
  BtTestreihe.Caption := 'Neue Testreihe';
  BtTestreihe.OnClick := BtTestreiheClick;

  BtReaktionstest := TButton.Create(Main);
  BtReaktionstest.Parent := Main;
  BtReaktionstest.SetBounds(0,70,120,32);
  BtReaktionstest.Caption := 'Start Reaktionstest';
  BtReaktionstest.OnClick := BtReaktionstestClick;
  BtReaktionstest.Visible := False;

  LbReaktionszeit := TLabel.Create(Main);
  with LbReaktionszeit do begin
    Parent := Main;
    SetBounds(5,120,120,32);
    Caption := 'Reaktionszeit'
  end;

  EdReaktionszeit := TEdit.Create(Main);
  with EdReaktionszeit do begin
    Parent := Main;
    SetBounds(0,140,120,60);
    Font.Name := 'Arial';
    Font.Size := 20;
    Text := ''
  end;
```

```
LbMinimum := TLabel.Create(Main);
with LbMinimum do begin
  Parent := Main;
  SetBounds(140,25,60,20);//Left, Top, Width, Height
  Caption := 'Minimum'
end;

EdMinimum := TEdit.Create(Main);
with EdMinimum do begin
  Parent := Main;
  SetBounds(210,20,120,20);
  Font.Name := 'Arial';
  Font.Size := 20;
  Text := ''
end;

LbMaximum := TLabel.Create(Main);
with LbMaximum do begin
  Parent := Main;
  SetBounds(140,60,60,20);//Left, Top, Width, Height
  Caption := 'Maximum'
end;

EdMaximum := TEdit.Create(Main);
with EdMaximum do begin
  Parent := Main;
  SetBounds(210,50,120,20);
  Font.Name := 'Arial';
  Font.Size := 20;
  Text := ''
end;

LbDurchschnitt := TLabel.Create(Main);
with LbDurchschnitt do begin
  Parent := Main;
  SetBounds(140,100,60,20);//Left, Top, Width, Height
  Caption := 'Durchschnitt'
end;
```

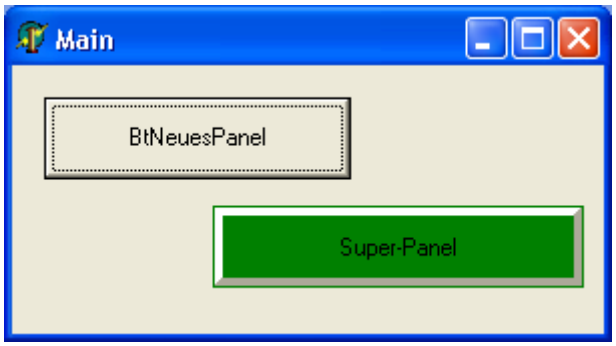
```
  EdDurchschnitt := TEdit.Create(Main);
  with EdDurchschnitt do begin
    Parent := Main;
    SetBounds(210,80,120,20);
    Font.Name := 'Arial';
    Font.Size := 20;
    Text := ''
  end;

  BtAuswertung := TButton.Create(Main);
  With BtAuswertung DO BEGIN
    Parent := Main;
    SetBounds(140,150,120,30);//Left, Top, Width, Height
    Caption := 'Auswertung';
    OnClick := BtAuswertungClick
  END;

  TiReaktion := TTimer.Create(Main);
  TiReaktion.Enabled := False;
  TiReaktion.OnTimer := TiReaktionTimer;
  self.OnKeyDown := FormKeyDown;
end;

end.
```

Erzeugung neuartiger Objekte

Bisher wurden ausschließlich neue Klassen definiert, die nur spezielle Formblätter waren, wie z.B. Diagramm, Ampel oder Zeitenliste. Im Folgenden sollen neue Objekte erzeugt werden, die keine Formblätter sind, z.B. neuartige Panels.
Dieser neue Paneltyp mit all seinen neuen Eigenschaften wird natürlich auch in einem eigenen Modul (Unit) festgelegt. Wähle dazu (in Delphi 6) *File – New – Unit* ! (Bei neuen Formblättern werden gleich einige Zusatzdateien mit Informationen über das neue Formblatt miterzeugt. Das geschieht bei neuen *Units* nicht).

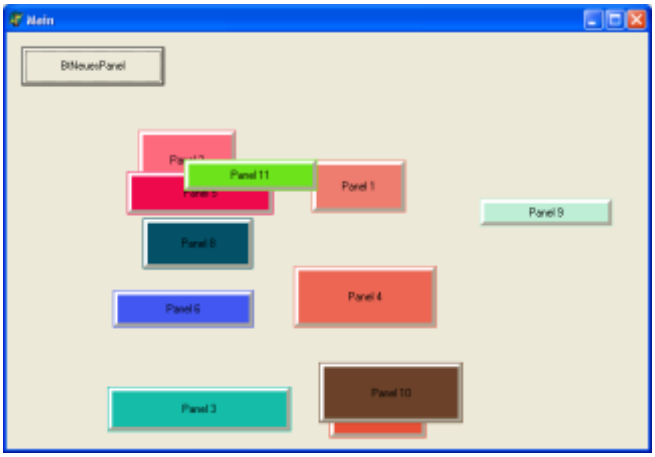
```
unit mPanel;

interface

uses ExtCtrls, Classes, Graphics, Controls, Forms;

type
  TGruenPanel = class(TPanel)
    public
    constructor Create(AOwner: TComponent); override;
    {Damit wird der Konstruktor der Oberklasse überschrieben. Das geht nur
    bei gleicher Parameterübergabe. Voraussetzung: in der Oberklasse besitzt
    der Konstruktor die Eigenschaft virtual}
  end;

implementation

constructor TGruenPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  {wichtig, damit überhaupt erst einmal ein Panel (der
   Oberklasse) erzeugt wird.}
  Color := clGreen;
  BorderStyle := bsNone;
  BorderWidth := 1;
  BevelOuter := bvNone;
  BevelInner := bvRaised;
  BevelWidth := 4
end;
end.
```

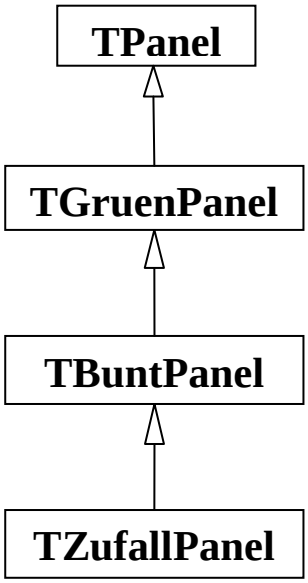
Wenn man im nachfolgenden Hauptprogramm *mPanel1* auf den Button klickt, wird ein neues Objekt der Klasse *GruenPanel* erzeugt.




```
unit mPanel1;

interface

uses…….  Dialogs, mPanel, StdCtrls;

type
  TMain = class(TForm)
        BtNeuesPanel: TButton;
        procedure BtNeuesPanelClick(Sender: TObject);
        private
         GP : TGruenPanel;
  end;

var  Main: TMain;

implementation……..

procedure TMain.BtNeuesPanelClick(Sender: TObject);
begin
  GP := TGruenPanel.Create(main);
  with GP DO BEGIN
    Parent := main;
    Left := 100;
    Top := 70;
    Caption := 'Super-Panel'
  END
end;

end.
```

Im folgenden Programm wird jedesmal, wenn man den Button anklickt, ein neues Objekt der Klasse *ZufallPanel* mit Zufallsfarbe, Zufallsgröße und nummerierter Beschriftung erzeugt. Diese Eigenschaften besitzt das neue Panel selbst. Den zufälligen Ort des neuen Panels kann man leider nur im Hauptprogramm bestimmen.





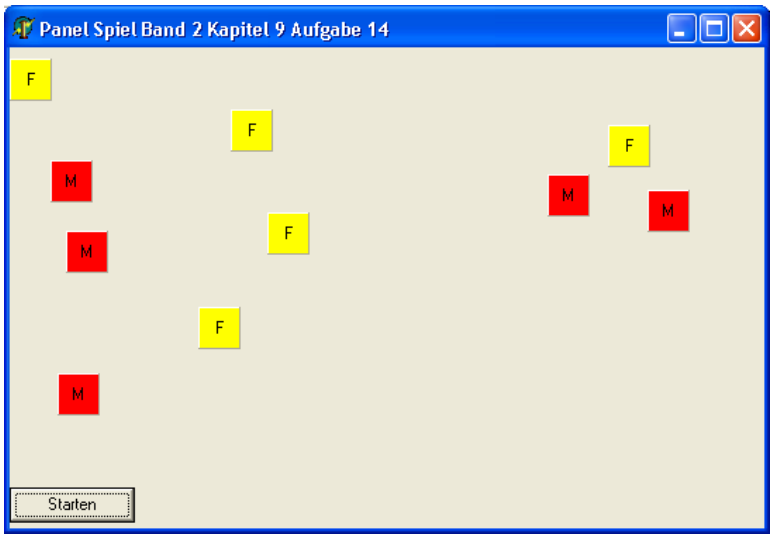

```
unit mPanelDemo;

interface

uses  ExtCtrls, Classes, Graphics, Controls, Forms,
SysUtils;

type
  TGruenPanel = class(TPanel)
     public
     constructor Create(AOwner: TComponent);override;
  end;

  TBuntPanel = class(TGruenPanel)
   public
   constructor Create(AOwner: TComponent; Farbe:
                                          TColor); virtual;
    {der neue Konstruktor hat mehr Parameter als der
     Konstruktor in der Oberklasse. Deshalb kann er
     nicht mehr die Eigenschaft override haben.}
  end;

  TZufallPanel = class(TBuntPanel)
   public
   constructor Create(AOwner: TComponent; Farbe:
                          TColor; nr:INTEGER); virtual;
  end;
```

```
implementation

constructor TGruenPanel.Create(AOwner: TComponent);
 begin.....end;  // siehe weiter oben!


constructor TBuntPanel.Create (AOwner: TComponent;
                                          Farbe: TColor);
begin
  inherited Create(AOwner);
  {wichtig, damit überhaupt erst einmal ein Panel (der Oberklasse
   TGruenPanel) erzeugt wird.}
  Color := Farbe
end;


constructor TZufallPanel.Create (AOwner: TComponent;
                           Farbe: TColor; nr: INTEGER);
begin
  inherited Create(AOwner,Farbe);
  {wichtig, damit überhaupt erst einmal ein Panel (der Oberklasse TBuntPanel)
   erzeugt wird.}
  Width := Random(100)+100;
  Height := Random(50)+30;
  Caption := 'Panel '+IntToStr(nr);    // benötigt unter
                                       // unter Windows die Unit SysUtils
 // Left := Random(Parent.Width);
 // Top := Random(Parent.Height - Height)
 // geht nicht, weil parent jetzt noch unbekannt
end;

end.
```

Das eigentliche Hauptprogramm folgt nun:

```
unit mPanel4; ……..

uses ……, mPanelDemo, StdCtrls;

type  TMain = class(TForm)
        BtNeuesPanel: TButton;
        procedure BtNeuesPanelClick(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        private
          BP: TBuntPanel;
          ZP: TZufallPanel;
          Nummer: INTEGER;
      end;

var  Main: TMain;

implementation……

procedure TMain.BtNeuesPanelClick(Sender: TObject);
Var Farbe: TColor;
begin
  Randomize;
  Farbe := Random($FFFFFF);
  INC(Nummer);
  ZP := TZufallPanel.Create(main,Farbe,Nummer);
  ZP.Parent := main;
  ZP.Left := Random(Width-ZP.Width);
  ZP.Top := Random(Height-ZP.Height)
end;


procedure TMain.FormCreate(Sender: TObject);
begin
 Nummer := 0;
end;

end.
```

Band 2, Kapitel 9, Aufgabe 14: Projekt: Fang das Panel




```
unit mAnzeige;
.....
type  TAnzeigeForm = class(TForm)
        Edit1: TEdit;
        public
         procedure SetPunkte(n:Integer);
      end;

implementation  .............

procedure TAnzeigeForm.SetPunkte(n:INTEGER);
 BEGIN
  Edit1.Text := IntToStr(n)
 END;
end.
```

```
unit mFangPanel;

interface

USES ExtCtrls, Classes, Graphics, Controls, Forms;

Type
  TFangPanel = class(TPanel)
    public
    constructor Create(AOwner: TComponent); override;
  end;

  TMine = class(TPanel)
    public
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TFangPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Color := clYellow;
  width := 30;
  height := 30;
  Caption := 'F'
end;

constructor TMine.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Color := clRed;
  width := 30;
  height := 30;
  Caption := 'M'
end;
end.
```

```
unit mSpiel;

interface

uses   ……, mFangPanel, mAnzeige, StdCtrls;

type
  TMain = class(TForm)
    Timer1: TTimer;
    BtStart: TButton;
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure BtStartClick(Sender: TObject);
    private
      panelAnzahl: INTEGER;
      AFenster: TAnzeigeForm;
      procedure PanelErzeugen;
      procedure LoeschVerstecktePanels;
      procedure PanelWeg(Sender: TObject);
      procedure Spielauswertung;
      procedure NeuPositionieren;
      procedure MineErzeugen;
      procedure MineGetroffen(Sender: TObject);
  end;

var Main: TMain;

implementation  …….

procedure TMain.NeuPositionieren;
VAR i: INTEGER;
begin
 FOR i := 0 TO ComponentCount - 1 DO
   IF Components[i] is TFangPanel THEN BEGIN
      TFangPanel(Components[i]).Left:= Random(Main.Width-50);
      TFangPanel(Components[i]).Top:= Random(Main.Height-100)
   END
   ELSE IF Components[i] is TMine THEN BEGIN
      TMine(Components[i]).Left := Random(Main.Width - 50);
      TMine(Components[i]).Top := Random(Main.Height - 100)
   END
end;
```

```
procedure TMain.Spielauswertung;
begin
 Timer1.Enabled := FALSE;
 IF PanelAnzahl = 0 THEN Showmessage('Du hast gewonnen!')
 ELSE Showmessage('Du hast verloren !');
 Main.Close
end;

procedure TMain.MineGetroffen(Sender: TObject);
begin
 Main.Close
end;

procedure TMain.PanelWeg(Sender: TObject);
begin
 TFangPanel(Sender).Hide;
 PanelAnzahl := PanelAnzahl - 1;
 AFenster.SetPunkte(PanelAnzahl);
 IF (PanelAnzahl=0) OR (PanelAnzahl =10) THEN Spielauswertung;
 NeuPositionieren
end;


procedure TMain.LoeschVersteckтePanels;
VAR i: INTEGER;
    K: TComponent;
begin
 FOR i := ComponentCount - 1 DOWNTO 0 DO BEGIN
   K := Components[i];
   IF K is TFangPanel THEN
      IF TFangPanel(K).Visible = False THEN K.Destroy
 END
end;


procedure TMain.PanelErzeugen;
VAR FP: TFangPanel;
BEGIN
   FP := TFangPanel.Create(Main);
   FP.Left := Random(Main.Width - 50);
   FP.Top := Random(Main.Height - 100);
   FP.Parent := Main;
   FP.OnClick := PanelWeg;
   PanelAnzahl := PanelAnzahl + 1
 END;
```

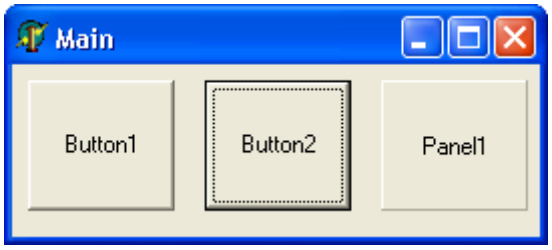
```
procedure TMain.MineErzeugen;
VAR M: TMine;
BEGIN
   M := TMine.Create(Main);
   M.Left := Random(Main.Width - 50);
   M.Top := Random(Main.Height - 100);
   M.Parent := Main;
   M.OnClick := MineGetroffen
 END;


procedure TMain.FormCreate(Sender: TObject);
VAR i: INTEGER;
begin
 PanelAnzahl := 0;
 FOR i := 1 TO 5 DO Begin
                     PanelErzeugen;
                     MineErzeugen
                    END;
 AFenster := TAnzeigeForm.Create(Main);
 AFenster.SetPunkte(PanelAnzahl);
 AFenster.Show
end;


procedure TMain.Timer1Timer(Sender: TObject);
begin
 LoeschVersteckтePanels;
 PanelErzeugen;
 AFenster.SetPunkte(PanelAnzahl);
 IF (PanelAnzahl=0) OR (PanelAnzahl =10) THEN Spielauswertung;
 NeuPositionieren
end;


procedure TMain.BtStartClick(Sender: TObject);
begin
 Timer1.Enabled := TRUE;
 BtStart.Visible := FALSE
end;

end.
```

Taster und Schalter

In praktisch jedem von uns geschriebenen Programm haben wir z.B. eine *procedure TMain.Button1Click(Sender: TObject)* geschrieben. Diese wurde ausgeführt, wenn ein Mausklick (d.h. die Maus wurde gedrückt und wieder losgelassen) über dem Button1 getätigt wurde. Analog kann ein Button aber auch schon allein auf das *MouseDown*- oder *MouseUp*- Ereignis reagieren.

Wenn man genau hinschaut, sieht man, dass bei einem Mausklick nicht nur die von uns geschriebene Prozedur ausgeführt wird, sondern dass zusätzlich auch die Gestalt bzw. Form des Buttons kurzzeitig (beim Niederdrücken der Maus) geändert wird. Ebenso ändert sich auch die Form wieder beim Loslassen der Maus. Falls man mehrere Buttons benutzt, ändert sich auch die Form des zuletzt aktiven Buttons.

Diese Formänderung tritt bei normalen Panels nicht auf.



Bemerkung: Die beiden Mausereignisse „*OnClick*“ und „*OnDblClick*“ sind nur Kombinationen der beiden elementaren Mausereignisse „*OnMouseDown*“ und „*OnMouseUp*“.

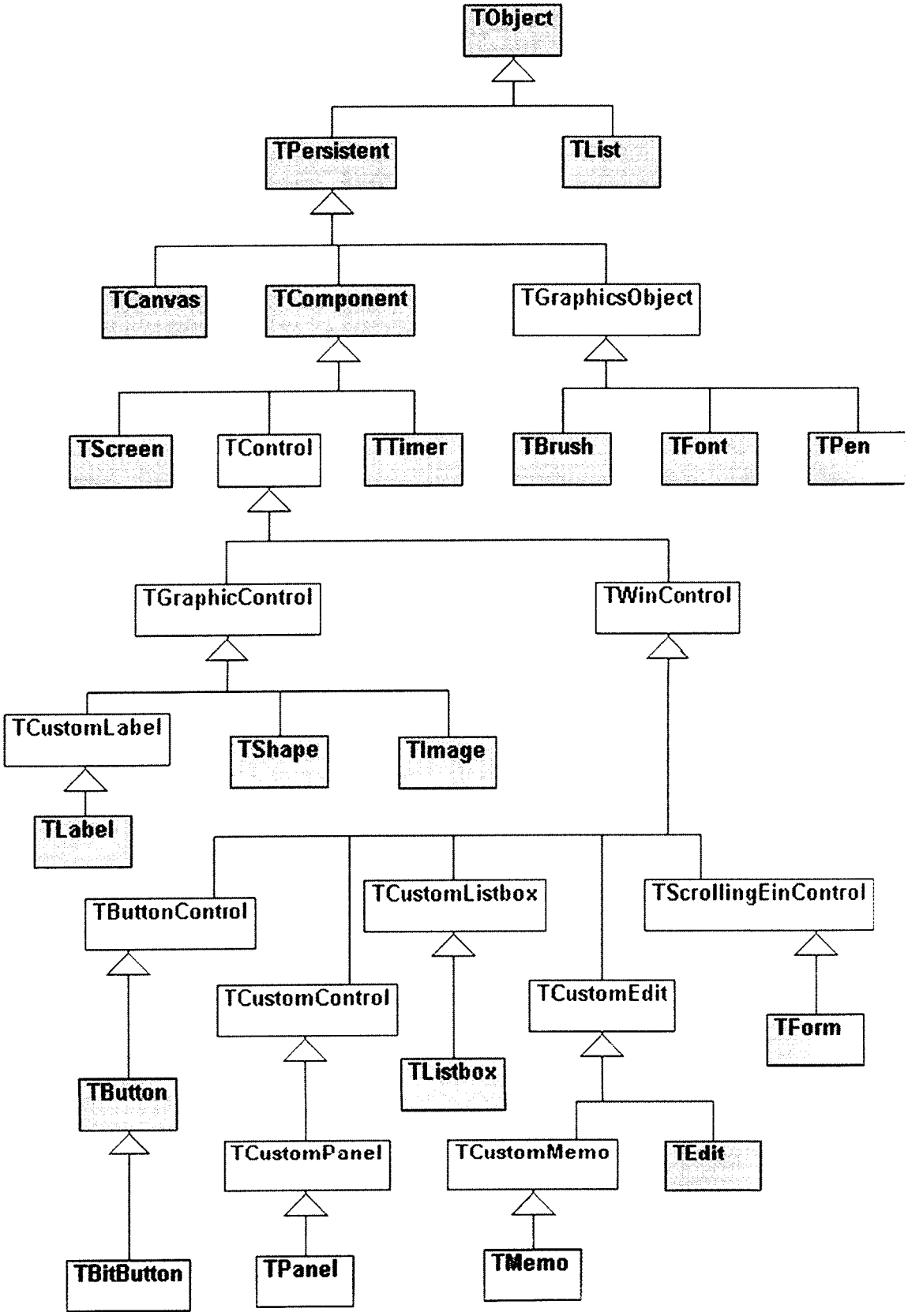
Erklärung für die Formänderungen: Wenn die Maus gedrückt wird, schickt das Windows-Betriebssystem eine entsprechende Meldung (genannt *WMMouseDown*, dabei steht *WM* für *WindowsMessage*) an Delphi.

Die Klassen *TButton* und *TPanel* sind Unterklassen von *TControl*, liegen in der Hierarchie aber deutlich tiefer.
Schon in der Oberklasse *TControl* ist die *WMMouseDown*-Meldung mit der **internen** Methode (von *TControl*) *procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X,Y: INTEGER); virtual;* verknüpft. Diese interne Methode erledigt zunächst einige hier unwesentliche Aufgaben und ruft zum Schluss die **externe** Ereignismethode *OnMouseDown* aufruft, die üblicherweise entweder leer ist oder vom Programmierer geschrieben wurde.

Die in *TControl* interne Methode *MouseDown* wird auf alle Unterklassen vererbt (*protected* Methode). In der Klasse *TPanel* allerdings wird sie wieder durch eine fast leere Prozedur überschrieben, d.h. *MouseDown* bewirkt hier selbst nichts, sondern ruft nur die externe (leere oder vom Programmierer geschriebene) Ereignismethode *OnMouseDown* auf.

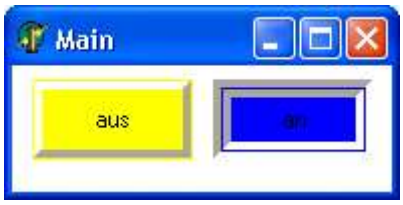
Die interne Panel-Methode *MouseDown* kann aber in noch tiefer liegenden Unterklassen leicht wieder überschrieben bzw. ergänzt werden. Dies soll im Folgenden geschehen.

Übersicht: Delphi-Klassenhierarchie (Auszug)



Im folgenden Programm werden ein Taster und ein Schalter simuliert. Ein elektrischer Taster schließt einen Kontakt nur solange man ihn gedrückt hält. Beispiel: Haustür-Klingel-Taster.
Ein elektrischer Schalter schließt einen Kontakt und öffnet ihn erst wieder beim nächsten schalten. Beispiel: Lichtschalter.

Im folgenden Programm soll sich einerseits die Form des Tasters bzw. des Schalters bei Betätigung ändern. Dafür soll nur die *MouseDown*- bzw. *MouseUp*-Prozedur von entsprechenden Panels manipuliert werden. Andererseits soll sich bei Betätigung die Hintergrundfarbe des Hauptformulars ändern. Dies wird entsprechend im Hauptprogramm programmiert.



Taster werden gedrückt, indem man die Maustaste über der Schaltfläche drückt. Sobald die Maustaste losgelassen wird, nimmt der Taster wieder seinen alten Zustand an. Wir programmieren zunächst also nur, dass sich **das Aussehen** des Tasters ändert. Bei dem *MouseUp*-Ereignis sollte der Taster sofort wieder seine alte Gestalt annehmen, ohne dass eine Aktion erfolgt.

Schalter hingegen besitzen die beiden Zustände *„an“* und *„aus“*. Durch Anklicken des Schalters wird er niedergedrückt (sein Aussehen ändert sich) *„an“*. Dieser Zustand (und sein Aussehen) ändert sich erst, wenn der Schalter ein zweites Mal angeklickt wird (*„aus“*).

Außer den beiden Schutzklassen *private* und *public* gibt es noch die Schutzklasse *protected*. Deren Attribute und Methoden sind nur für abgeleitete Klassen nutzbar (also praktisch *public*), für fremde Klassen aber nur *private*. Gekennzeichnet wird die Schutzklasse *protected* durch eine Raute # .

unit mPanel;

```
uses  ExtCtrls, Classes, Graphics, Controls, Forms, SysUtils;

type

 TGruenPanel = class(TPanel)
  public
    constructor Create(AOwner: TComponent); virtual;
 end;


 TBuntPanel = class(TGruenPanel)
  public
    constructor Create(AOwner: TComponent; Farbe: TColor); virtual;
 end;

 TBuntTaster = class(TBuntPanel)
  protected
   procedure MouseDown (Button: TMouseButton; Shift: TShiftState; X,Y: Integer); override;
   procedure MouseUp (Button: TMouseButton; Shift: TShiftState; X,Y: Integer); override;
 end;

 TBuntSchalter = class(TBuntTaster)
    Protected
   procedure MouseDown (Button: TMouseButton; Shift: TShiftState; X,Y: Integer); override;
   procedure MouseUp (Button: TMouseButton; Shift: TShiftState; X,Y: Integer); override;
    public
      gedrueckt: Boolean;
      // Variablen müssen müssen vor den Methoden deklariert werden!
      constructor Create(AOwner: TComponent; Farbe: TColor); override;
 end;


implementation
constructor TGruenPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Color := clGreen;
  BorderStyle := bsNone;   BorderWidth := 1;
  BevelOuter := bvNone;
  BevelInner := bvRaised;
  BevelWidth := 4
end;
```

```
constructor TBuntPanel.Create (AOwner: TComponent; Farbe: TColor);
begin
  inherited Create(AOwner);
  Color := Farbe
end;


procedure TBuntTaster.MouseDown (Button: TMouseButton; Shift: TShiftState; X,Y:
                                                                    Integer);
begin
  BevelInner := bvLowered;     Caption := 'an';
  inherited MouseDown(Button,Shift,X,Y)      //Damit auch die externe Ereignis-
end;                                            methode aufgerufen wird.


procedure TBuntTaster.MouseUp(Button: TMouseButton; Shift: TShiftState; X,Y:Integer);
begin
  BevelInner := bvRaised;     Caption := 'aus';
  inherited MouseUp(Button,Shift,X,Y)          //wegen externer Ereignismethode
end;


constructor TBuntSchalter.Create(AOwner: TComponent; Farbe: TColor);
begin
  inherited Create(AOwner, Farbe);
  Caption := 'aus';      Gedrueckt := False
end;


procedure TBuntSchalter.MouseDown (Button: TMouseButton; Shift: TShiftState; X,Y:
                                                                    Integer);
begin
  IF Gedrueckt THEN begin
    BevelOuter := bvRaised;
    caption := 'aus';
    inherited MouseUp(Button, Shift, X,Y)
  end
  ELSE begin
    BevelOuter := bvLowered;
    caption := 'an';
    inherited MouseDown(Button, Shift, X,Y)
  end;
  Gedrueckt := Not Gedrueckt
end;
```

```
procedure TBuntSchalter.MouseUp (Button: TMouseButton; Shift: TShiftState; X,Y:
                                                                    Integer);
begin
// Damit wird die übergeordnete Prozedur deaktiviert
end;

end.
```

unit mTaster; *// Hauptprogramm*

```
Uses   Windows, ….., Controls, mPanel, StdCtrls;

type
  TMain = class(TForm)
          procedure TasterMouseDown(Sender: TObject; Button: TMouseButton;
                                                    Shift: TShiftState; X,Y: Integer);
          procedure TasterMouseUp(Sender: …….);
          procedure SchalterMouseDown(Sender: ……);
          procedure SchalterMouseUp(Sender: ……);
          procedure FormCreate(Sender: TObject);
       end;

var
  Main: TMain;
  BT: TBuntTaster;
  BS: TBuntSchalter;

Implementation……


procedure TMain.TasterMouseDown(Sender: TObject; Button: TMouseButton;
                                                    Shift: TShiftState; X,Y: Integer);
begin
  Color := clRed
end;

procedure TMain.TasterMouseUp(Sender: …….);
begin
  Color := clWhite
end;
```

```
procedure TMain.SchalterMouseDown(Sender: ……);
begin
   Color := clsilver
end;


procedure TMain.SchalterMouseUp(Sender: ……);
begin
   Color := clblack
end;


procedure TMain.FormCreate(Sender: TObject);
begin
  BT := TBuntTaster.Create(Main, clYellow);
  BT.Parent := main;
  BT.Width :=80;
  BT.Height := 40;
  BT.Left := 10;
  BT.Top := Main.Height - 90;
  BT.OnMouseDown := TasterMouseDown;
  BT.OnMouseUp := TasterMouseUp;
  BS := TBuntSchalter.Create(Main, clBlue);
  With BS DO BEGIN
    Parent := main;
    Width :=80;
    Height := 40;
    show;
    Left := 10 + BT.Width + 10;
    Top := Main.Height - 90;
    OnMouseDown := SchalterMouseDown;
    OnMouseUp := SchalterMouseUp;
  END;
end;

end.
```

Aufgaben

1. Stelle die Ableitungshierarchien für die Klassen *TPanel*, *TGruenPanel*, *TBuntpanel*, *TBuntTaster* und *TBuntSchalter* in einem Klassendiagramm dar! Orientiere dich an den Beispielen im Lehrbuch auf den Seiten 62 und 66 !

2. Erkläre genau, warum die Taster nach ihrer Erzeugung noch keine Aufschrift "an" oder "aus" besitzen, die Schalter hingegen schon!

3. Zwei unterschiedliche Bilder, *BildA* und *BildB*, sollen von links nach rechts über den Bildschirm wandern. Wenn sie rechts angekommen sind, sollen sie links neu erscheinen. *BildA* wird von einem Taster gesteuert, *BildB* von einem Schalter.

4. Ändere sowohl für die Taster als auch für die Schalter die Schaltzustände so, dass sich je nach Schaltzustand auch die Farbe des Tasters bzw. des Schalters ändert !

5. Ändere sowohl für die Taster als auch für die Schalter die Schaltzustände so, dass sich je nach Schaltzustand auch die Größe des Tasters bzw. des Schalters ändert !

public, private, protected, virtual, override

Das sinnvolle Konzept der Schutzklassen *private* und *public* (und *protected*) ist in Delphi leider nicht konsequent umgesetzt worden. Der Grund dafür liegt in der beabsichtigten, möglichen Kompatibilität bzw. Übertragbarkeit von und zu den älteren Pascal-Programmen.

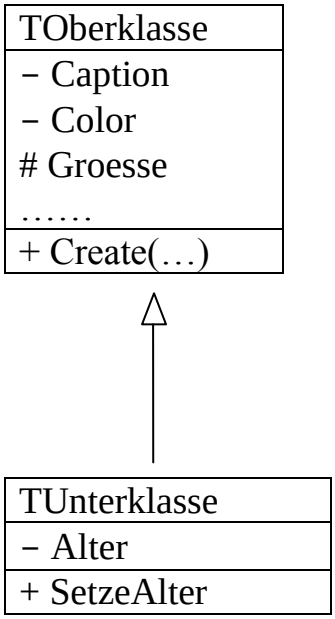
Leider gilt deshalb folgende Einschränkung in Delphi:
Innerhalb einer Unit kann man immer auf alle, insbesondere auch auf private Bestandteile einer Klasse zugreifen, auch wenn sich diese in einer anderen Klasse (in derselben Unit) befinden.
Die Schutzklassen private bzw. protected schützen also immer nur dann, wenn versucht wird, auf Attribute oder Methoden einer Klasse zuzugreifen, die in einer anderen Unit definiert ist.

Unabhängig von obiger Einschränkung werden wir alle unsere Programme so schreiben, als würde obige Einschränkung nicht gelten !

Wenn man in der Unterklasse eine Methode (neu) definiert, die denselben Namen wie eine Methode der Oberklasse besitzt, so existieren in der Unterklasse beide Methoden, allerdings wird die ererbte Methode durch die neu definierte verdeckt (ähnlich verhält es sich z.B. mit lokalen und globalen Variablen gleichen Namens in Prozeduren). Üblicherweise gibt es dann beim Compilieren eine entsprechende Warnung: *[Warning] Method 'Create' hides virtual method of base type'*

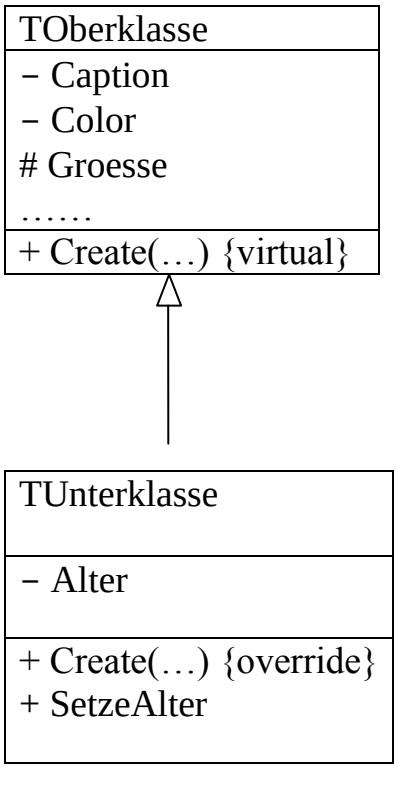
Das Erzeugen dieser Warnung lässt sich allerdings unterdrücken, wenn man hinter dem Namen der neu definierten Methode den Zusatz ***reintroduce;*** angibt.
Beispiel: procedure Methodenname(…..); reintroduce; virtual;

Im Folgenden betrachten wir nun die beiden Klassen *TOberklasse* und *TUnterklasse*. Die Unterklasse leitet sich von der Oberklasse ab. Sie beinhaltet noch eine (*private*) Eigenschaft *Alter: INTEGER*, die sich also nicht noch weiter vererben kann.



Private Eigenschaften und Methoden der Oberklasse werden **nicht** auf die Unterklasse vererbt. Jedes Objekt der Unterklasse besitzt also nur diejenigen Eigenschaften und Methoden der Oberklasse, die entweder *public* oder *protected* sind (beachte oben genannte Einschränkung in Delphi).
Zusätzlich gibt es aber noch weitere, nur in der Unterklasse definierte Eigenschaften und Methoden.

Falls die Objekte der Unterklasse nur **zusätzliche** Attribute oder Methoden haben sollen (und nicht schon bei der Erzeugung andere Attributswerte als ein Objekt der Oberklasse), so genügt derselbe Konstruktor. Das Delphisystem hängt die zusätzlichen Attribute und Methoden automatisch bei der Erzeugung mit an.



Falls die Objekte der Unterklasse schon bei der Erzeugung andere oder zusätzliche Eigenschaftswerte haben sollen, so muss man die ererbte Version *Create(...)* überschreiben.

Dies geschieht dadurch, dass man in der Unterklasse hinter der Deklaration des Konstruktors *Create* den Zusatz *override* angibt. Der Konstruktor der Oberklasse lässt sich allerdings nur überschreiben, wenn bei dessen Deklaration der Zusatz *virtual* angegeben wird.

Die Benutzung von *override* setzt voraus, dass die Methode in der Unterklasse exakt dieselben Parameter besitzt wie die Methode in der Oberklasse.
Benötigt man in der Unterklasse mehr oder andere Parameter, so muß man die entsprechende Methode auch in der Unterklasse als *virtual* deklarieren!

Virtuelle Methoden der Oberklasse, die in der Unterklasse überschrieben werden (die also in der Unterklasse hinter dem Namen den Zusatz *override* haben), bleiben in der Unterklasse virtuelle Methoden. Sie können also von (in der Ableitungshierarchie) noch tiefer liegenden Unterklassen wiederum überschrieben werden!

Interessanterweise lässt sich in der Unterklasse auch dann noch auf den ererbten Konstruktor zugreifen, wenn man diesen mit *override* überschrieben hat. Das liegt daran, dass jede Klasse intern auch einen Zeiger besitzt, der auf die direkte Oberklasse verweist; und dort kann man auf die vererbte Methode zugreifen.

```
constructor TUnterklasse.Create(AOwner: TComponent); //override
BEGIN
  Inherited Create(AOwner);
  Alter := 70
END;
```

Virtuelle Methoden-Tabelle (VMT)

Statische Methoden werden vom Compiler direkt übersetzt. Sämtliche **statische** Methoden vom Typ *public* und *protected* werden von Unterklassen **völlig identisch** übernommen. *Private* Methoden werden nicht vererbt!

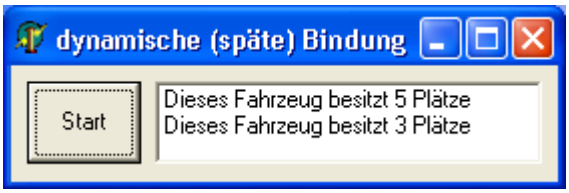
Für **jede** im Programm benutzte Klasse wird jedoch eine Tabelle der virtuellen Methoden angelegt. Diese Tabelle ordnet den Methodennamen die zugehörige Speicheradresse zu. Außerdem werden die virtuellen Methoden durchnummeriert, sodaß man mithilfe eines Indexes auf sie zugreifen kann.

Diese VMT enthält zunächst sämtliche von allen Oberklassen übernommenen virtuellen Methoden und zusätzlich deren Adressen, falls diese Methoden nicht mit *override* überschrieben wurden. Falls eine dieser virtuellen Methoden mit *override* überschrieben wurde, so enthält die VMT die dadurch aktualisierte Adresse. Das bedeutet, dass die Indizes (nicht die Adressen!) der ersten virtuellen Methoden in Ober- und Unterklasse übereinstimmen. Erst danach, am Schluß der Tabelle, werden in der aktuellen Unterklasse neu hinzukommende virtuelle Methoden angehängt.

Beispiel:
Vu sei ein Objekt der Unterklasse, *statMeth* sei eine statische Methode (vom Typ *public* oder *protected*) der Oberklasse (die also in der Unterklasse geerbt wurde), welche u.a. die virtuelle Methode mit dem Index 17 aufruft. Dann wird durch die Anweisung *Vu.statMeth* die in der VMT der Unterklasse stehende Methode mit dem Index 17 gesucht und ausgeführt.

Sollte in der Unterklasse hinter einem Methodennamen nicht *override* sondern wiederum *virtual* stehen, so wird in der VMT eine zweite gleichnamige Methode angehängt (die dann natürlich einen anderen Index erhält).

Welche Methode nun gegebenenfalls ausgeführt wird, hängt in jedem Fall davon ab, ob der Methodenname von einem Objekt der Ober- oder von einem Objekt der Unterklasse aufgerufen wird.




```
unit mMain;
……..
type
  TMain = class(TForm)
    BtStart: TButton;
    ListBox1: TListBox;
    procedure BtStartClick(Sender: TObject);
  end;

  TAuto = class(TObject)
    public
      procedure InformationsBereitstellung;
      function SitzplatzAnzahl: INTEGER; virtual;
  end;

  TLKW = class(TAuto)
    public
      function SitzplatzAnzahl: INTEGER; override;
  end;

var  Main: TMain;
     Auto: TAuto;
     LKW: TLKW;


Implementation…….
{$R *.dfm}

procedure TAuto.InformationsBereitstellung;
BEGIN
  Main.ListBox1.Items.Add('Dieses Fahrzeug besitzt '
            + IntToStr(Sitzplatzanzahl) + ' Plätze')
END;

function TAuto.SitzplatzAnzahl: INTEGER;
BEGIN
  RESULT := 5
END;
```

```
function TLKW.SitzplatzAnzahl: INTEGER;
BEGIN
  RESULT := 3
END;

procedure TMain.BtStartClick(Sender: TObject);
begin
  Auto := TAuto.Create;
  LKW := TLKW.Create;
  Auto.InformationsBereitstellung;
  LKW.InformationsBereitstellung
end;

end.
```

Beachte: Die Methode *Informationsbereitstellung* ist in der Unterklasse TLKW identisch mit der gleichnamigen Methode *Informationsbereitstellung* in der Oberklasse TAuto.
Wenn allerdings innerhalb der Methode *Informationsbeschaffung* die Hilfsmethode *Sitzplatzanzahl* aufgerufen wird, dann wird erst geprüft, welches Objekt gerade vorliegt. In Abhängigkeit vom Ergebnis dieser Prüfung wird entschieden, ob die Hilfsmethode *Sitzplatzanzahl* aus der Ober- oder aus der Unterklasse verwendet wird.

Der Delphi-Quelltext wird also so compiliert, dass erst zur Laufzeit des Programms die richtige Methodenauswahl getroffen wird. Diese Programm- oder Compilereigenschaft nennt man ***dynamische*** oder ***späte Bindung***.

Aus der Sicht des Programmbenutzers zeigen die verschiedenen Objekte ein ***polymorphes*** (=vielgestaltiges) ***Verhalten***: Objekte verschiedener Klassen reagieren unterschiedlich auf identische Befehle (*Informationsbereitstellung*). Man bezeichnet dies als ***Polymorphie***.

Aufgabe: Lösche im obigen Programm das Wort *override* oder (zweite Aufgabe) ersetze es durch *virtual*! Erkläre jeweils das Resultat!
Hinweis: In den VMTs der Ober- und Unterklasse haben vererbte Methoden dieselben Nummern, egal ob sie überschrieben werden oder nicht.

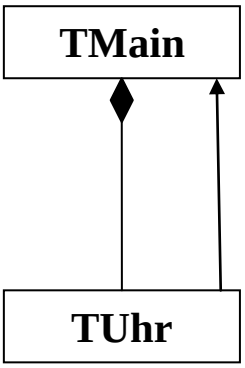
Man kann einer Variablen vom Typ einer bestimmten Klasse beliebige Objekte von abgeleiteten Klassen zuordnen. Das Umgekehrte geht nicht!
Von dieser Möglichkeit werden wir im Schulunterricht jedoch keinen Gebrauch machen. Beispiel:

```
type  T1 = class(TObject)
        .............
      end;
      T2 = class(T1)
        .............
      end;

var  A: T1;
     B: T2;
procedure TMain.BtStartClick(Sender: TObject);
begin
  A := T2.Create;  // Das ist möglich !
  B := T1.Create;  // Das ist nicht möglich !
end;
```

Aufgabe: Untersuche, welche Methode jeweils ausgeführt wird !
Untersuche drei Fälle: die Methode Act der Klasse T2 wird als statisch, als override oder als virtual deklariert! Warum ist die Erklärung in jedem Fall sehr einfach?

```
type  T1 = class(TObject)
        procedure Act; virtual;
      end;

      T2 = class(T1)
        procedure Act; // override;
      end;

var  A: T1;
     B: T2;
procedure TMain.BtStartClick(Sender: TObject);
begin
  A := T1.Create;
  A.Act;
  B := T2.Create;  B.Act;
end;
```

Projekt: Uhren




```
unit mUhr;
interface
USES Forms, Graphics, ExtCtrls;
Type
  TUhr = class(TObject)
           private
             Xul, Yul, Breite, XM, YM, Minute,
             Sekunde, Minutenzeigerlaenge,
             Sekundenzeigerlaenge:   Integer;
             hatTimer: TTimer;
             kenntContainer: TForm;
             procedure zeichneUhr;
             procedure zeichneZeiger;
             procedure zeichneZeigerNeu(Sender:
                                        TObject);
                                 //Parameter erforderlich
           public
             constructor Create(X, Y, Width:Integer;
                          Container: TForm); virtual;
         end;

implementation

constructor TUhr.Create(X, Y, Width:Integer;
                                    Container: TForm);
begin
  inherited Create;
  kenntContainer := Container;
  Xul := X;      // Koordinaten unten links
  Yul := Y;
```

```
  Breite := Width;
  XM := Xul + Breite DIV 2;
  YM := Yul - Breite DIV 2;
  Sekundenzeigerlaenge := Round(0.45*Breite);
  Minutenzeigerlaenge := Round(0.3*Breite);
  Sekunde := 15;     // willkürliche Anfangswerte
  Minute := 0;
  ZeichneUhr;
  hatTimer := TTimer.Create(kenntContainer); // Owner
                                        // wird benötigt
  hatTimer.Interval := 1000;
  hatTimer.OnTimer := zeichneZeigerNeu
  // Parameterliste muß übereinstimmen
end;

procedure TUhr.zeichneZeigerNeu(Sender: TObject);
begin
 With kenntContainer.Canvas Do BEGIN
   zeichneZeiger;
   Sekunde := (Sekunde+2) MOD 60;
   // damit Sekundenzeiger schneller ist
   Minute := (Minute+1) MOD 60;
   zeichneZeiger;
 END;
end;

procedure TUhr.zeichneZeiger;
VAR XSek, YSek, XMin, YMin: Integer;
begin
 With kenntContainer.Canvas Do BEGIN
   pen.Width := 1;
   MoveTo (XM, YM);
   XSek := XM+Round(Sekundenzeigerlaenge*sin(Pi*Sekunde/30));
   YSek := YM-Round(Sekundenzeigerlaenge*cos(Pi*Sekunde/30));
   LineTo(XSek, YSek);

   pen.Width := 2;
   MoveTo (XM, YM);
   XMin := XM+Round(Minutenzeigerlaenge*sin(Pi*Minute/30));
   YMin := YM-Round(Minutenzeigerlaenge*cos(Pi*Minute/30));
   LineTo(XMin, YMin);
  END
end;
```

```
procedure TUhr.zeichneUhr;
begin
 With kenntContainer.Canvas Do BEGIN
   pen.Mode := pmNotXor;
   pen.Color := clBlack;
   Brush.Color := clYellow;
   Ellipse(Xul, Yul, Xul + Breite, Yul - Breite);
   zeichneZeiger
  END;
end;


end.
```

unit mHaupt;

```
interface

uses  Windows,…., mUhr, StdCtrls;

type
  TMain = class(TForm)
            BtErzeugen: TButton;
            procedure BtErzeugenClick(Sender: TObject);
            private
             U1 : TUhr;
          end;

var  Main: TMain;
implementation
……
procedure TMain.BtErzeugenClick(Sender: TObject);
begin
  U1 := TUhr.Create(20,170,100,Main)
end;

end.
```

Aufgaben

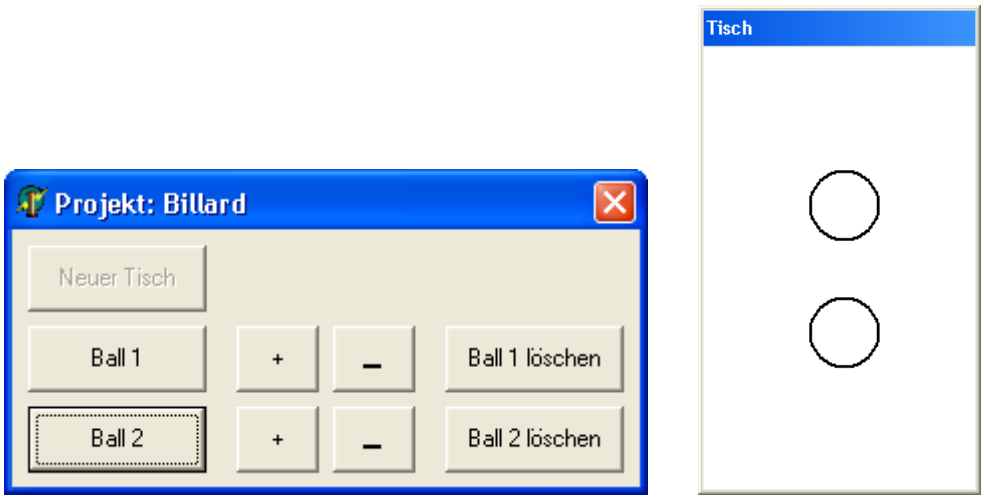
1. Sorge dafür, dass die Uhr einen dritten Zeiger für die Stundenangabe erhält!
2. Die Uhr soll von außen gestellt werden können. Sie benötigt also eine öffentliche Methode
   procedure setzen(Stunde, Minute, Sekunde: INTEGER).
3. Man soll die Uhrzeit von außen abfragen können. Dazu benötigt man eine öffentliche Methode
   procedure Uhrzeit(VAR Stunde, Minute, Sekunde: INTEGER)
   Beachte die Art der Parameterübergabe!
4. Zusätzlich soll unterhalb der analogen Zeigeruhr noch die Zeit in digitaler Form angezeigt werden. Implementiere folgende Lösung: Unterhalb der Zeigeruhr wird auf der Canvas ein Rechteck gezeichnet (damit man die digitale Zeitangabe immer wieder löschen kann, bevor man sie neu schreibt). Mit dem Befehl ***textout(x, y: INTEGER; angabe: STRING)*** läßt sich auf der Canvas am angegebenen Ort ein Text ausgeben.
5. Die Uhr soll nun die aktuelle Rechnerzeit anzeigen. Diese läßt sich z.B. folgendermaßen ermitteln:

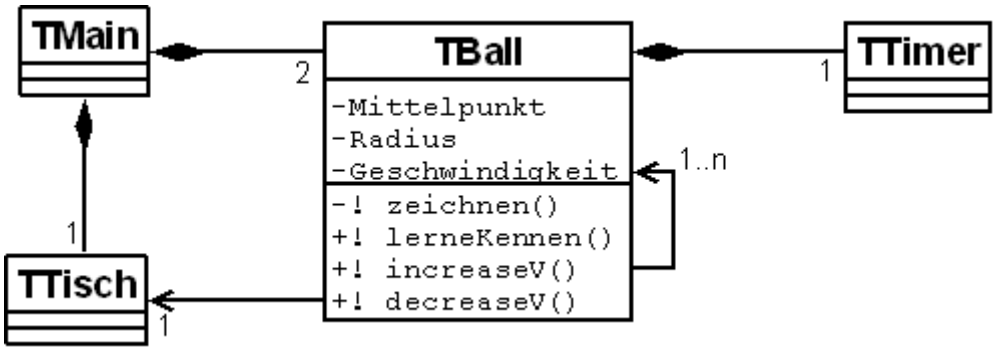
```
USES    …… , SysUtils;
……..
var Present: TDateTime;
    Hour, Min, Sec, MSec: Word;
begin
  Present:= Now;
  DecodeTime(Present, Hour, Min, Sec, MSec);
  Label1.Caption := IntToStr(Hour) + ': ' +
                                     IntToStr(Min);
end;
```

Projekt: Billard

Im Folgenden bewegen sich auf einem Tisch zwei Bälle aufeinander zu. Sobald sie sich berühren, bleiben sie beide liegen.

Die Bälle sollen möglichst „intelligent“ sein. Insbesondere kennen sie ihre eigene Größe und Geschwindigkeit. Sie können sich selber zeichnen. Dafür müssen sie natürlich den Tisch kennen, auf dessen Canvas sie sich zeichnen sollen. Außerdem sollen sie sich gegenseitig kennen, damit sie bei Berührung liegen bleiben.
In diesem Fall ist es nicht möglich, dass z.B. der zweite Ball bei der Erzeugung des ersten Balles als Parameter übergeben wird, weil der zweite Ball noch gar nicht existiert. Die beiden Bälle müssen von jemandem dritten (dem Hauptprogramm) miteinander bekannt gemacht werden. Zu diesem Zweck muß die Klasse *TBall* eine öffentliche Kennenlernmethode zur Verfügung stellen.





unit mBall;

```
interface
USES ExtCtrls, Graphics, Classes, Forms;
Type
  TBall = class(TObject)
            private
              kContainer: TForm;
              kBall: TBall;
              hatTimer: TTimer;
              XM,YM,Radius, Geschwindigkeit: Integer;
              procedure Zeichnen(Sender: TObject);
                    // Parameter wird benötigt
            public
              constructor Create (X,Y,r,v: Integer;
                          Container: TForm); virtual;
              destructor Destroy;  override;
              procedure LerneKennen(Ball:TBall);
              procedure IncreaseV;
              procedure DecreaseV;
            end;

implementation

procedure TBall.LerneKennen(Ball: TBall);
begin
  kBall := Ball
end;

procedure TBall.Zeichnen(Sender: TObject);
begin
  With kContainer.Canvas DO begin
    Ellipse(XM-Radius, YM-Radius, XM+Radius, YM+Radius);
    // alter Ball wird durch überzeichnen gelöscht
    YM := YM+Geschwindigkeit;
    Ellipse(XM-Radius, YM-Radius, XM+Radius, YM+Radius);
    IF kBall <> NIL THEN
      IF Sqrt(Sqr(XM-kBall.XM) + Sqr(YM-kBall.YM)) <
                                  Radius + kBall.Radius
        THEN hatTimer.Enabled := False;
  end;
end;
```

```
constructor TBall.Create (X,Y,r,v: Integer;
                                  Container: TForm);
begin
  inherited Create;
  kContainer := Container;
  XM := X;
  YM := Y;
  Radius := r;
  Geschwindigkeit := v;
  hatTimer := TTimer.Create(kContainer);
    // Owner wird benötigt
  hatTimer.Interval := 50;
  hatTimer.Enabled := True;
  hatTimer.OnTimer := Zeichnen;
                          // Parameterliste muß übereinstimmen
  With kContainer.Canvas DO begin
    pen.Mode := pmNotXor;
    pen.Color := clBlack;
    pen.Width := 2;
    Ellipse(XM-Radius, YM-Radius, XM+Radius, YM+Radius)
  end;
end;


destructor TBall.Destroy;
begin
  hatTimer.Destroy;
  With kContainer.Canvas DO
    Ellipse(XM-Radius, YM-Radius, XM+Radius, YM+Radius);
  inherited Destroy
end;

procedure TBall.IncreaseV;
BEGIN
  INC(Geschwindigkeit)
END;

procedure TBall.DecreaseV;
BEGIN
  DEC(Geschwindigkeit)
END;

end.
```

```
unit Haupt;
interface
uses  ……..Classes, Forms, mBall;

type
  TMain = class(TForm)
    BtErzeugeBall1, BtErzeugeBall2, BtNeuerTisch,
    Bt1Plus, Bt1Minus, Bt2Plus, Bt2Minus, Bt1Loesch,
    Bt2Loesch: TButton;
    procedure BtErzeugeBall1Click(Sender: TObject);
    procedure BtErzeugeBall2Click(Sender: TObject);
    procedure BtNeuerTischClick(Sender: TObject);
    procedure Bt1PlusClick(Sender: TObject);
    procedure Bt1MinusClick(Sender: TObject);
    procedure Bt2PlusClick(Sender: TObject);
    procedure Bt2MinusClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Bt1LoeschClick(Sender: TObject);
    procedure Bt2LoeschClick(Sender: TObject);
    private
      Ball1, Ball2: TBall;
      Tisch: TForm;
  end;

var  Main: TMain;

implementation
{$R *.dfm}

procedure TMain.BtErzeugeBall1Click(Sender: TObject);
begin
  IF Ball1 <> NIL THEN Ball1.Destroy;
  // es gibt immer nur einen Ball1
  Ball1 := TBall.Create(90,30,25,1,Tisch);
  Bt1Minus.Enabled := TRUE;
  Bt1Plus.Enabled := TRUE;
  Bt1Loesch.Enabled := TRUE;
  // Wenn Ball2 schon existiert, dann lernen die Bälle sich kennen
  IF Ball2 <> NIL THEN BEGIN
    Ball1.LerneKennen(Ball2);
    Ball2.LerneKennen(Ball1)
  end;
end;
```

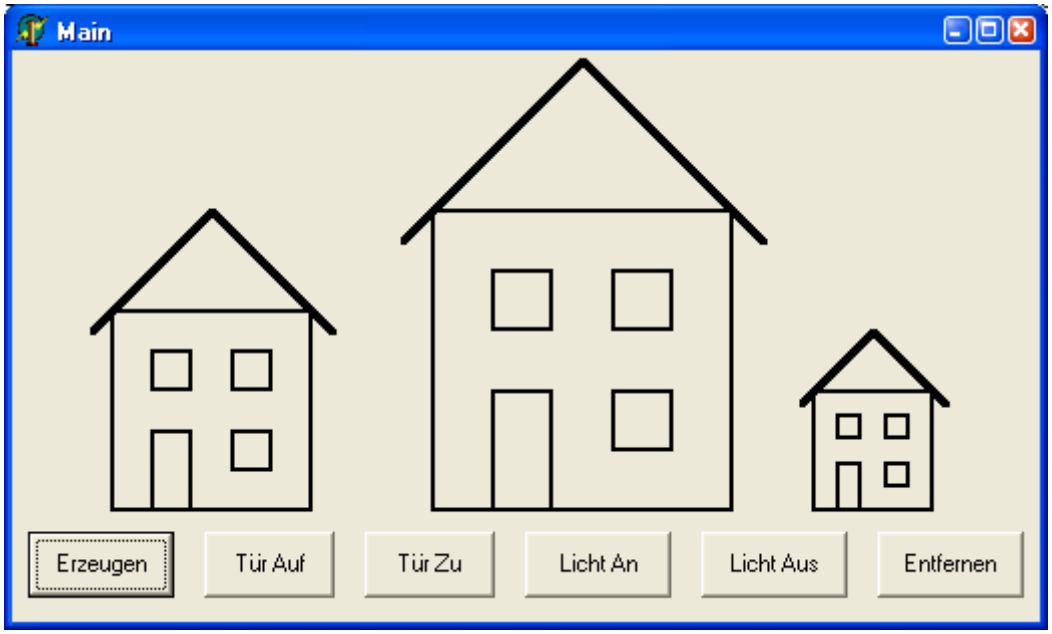
```
procedure TMain.BtErzeugeBall2Click(Sender: TObject);
begin
  IF Ball2 <> NIL THEN Ball2.Destroy;
  // es gibt immer nur einen Ball2
  Ball2 := TBall.Create(90,250,25,-2,Tisch);
  Bt2Minus.Enabled := TRUE;
  Bt2Plus.Enabled := TRUE;
  Bt2Loesch.Enabled := TRUE;
  // Wenn Ball1 schon existiert, dann lernen die Bälle sich kennen
  IF Ball1 <> NIL THEN BEGIN
    Ball1.LerneKennen(Ball2);
    Ball2.LerneKennen(Ball1)
  end
end;


procedure TMain.BtNeuerTischClick(Sender: TObject);
begin
  Tisch := TForm.Create(Main);
  Tisch.Left := 10;
  Tisch.Top := 100;
  Tisch.Height := 300;
  Tisch.Width := 200;
  Tisch.Show;
  BtNeuerTisch.Enabled := False;
  BtErzeugeBall1.Enabled := True;
  BtErzeugeBall2.Enabled := True
end;

procedure TMain.Bt1PlusClick(Sender: TObject);
begin
  Ball1.IncreaseV
end;

procedure TMain.Bt1MinusClick(Sender: TObject);
begin
  Ball1.DecreaseV
end;

procedure TMain.Bt2PlusClick(Sender: TObject);
begin
  Ball2.IncreaseV
end;
```

```
procedure TMain.Bt2MinusClick(Sender: TObject);
begin
  Ball2.DecreaseV
end;


procedure TMain.FormCreate(Sender: TObject);
begin
  BtErzeugeBall1.Enabled := False;
  BtErzeugeBall2.Enabled := False;
  Bt1Plus.Enabled := False;
  Bt2Plus.Enabled := False;
  Bt1Minus.Enabled := False;
  Bt2Minus.Enabled := False;
  Bt1Loesch.Enabled := False;
  Bt2Loesch.Enabled := False;
end;


procedure TMain.Bt1LoeschClick(Sender: TObject);
begin
  IF Ball1 <> NIL THEN Ball1.Destroy;
  Ball1 := NIL;    // sehr wichtig! wird leider nicht von Destroy
                      automatisch gesetzt.
  Bt1Plus.Enabled := False;
  Bt1Minus.Enabled := False;
  Bt1Loesch.Enabled := False;
  IF Ball2 <> NIL THEN Ball2.LerneKennen(NIL)
end;


procedure TMain.Bt2LoeschClick(Sender: TObject);
begin
  IF Ball2 <> NIL THEN Ball2.Destroy;
  Ball2 := NIL;    // sehr wichtig! vgl. oben!
  Bt2Plus.Enabled := False;
  Bt2Minus.Enabled := False;
  Bt2Loesch.Enabled := False;
  IF Ball1 <> NIL THEN Ball1.LerneKennen(NIL)
end;


end.
```

Aufgaben

1. Die Bälle sollen sich nun gegenseitig und auch an der oberen und unteren Tischkante elastisch stoßen. Diese Reflektion ist leicht zu simulieren, indem man einfach die Geschwindigkeitsrichtung der am Stoß beteiligten Bälle ändert.
2. Die obigen Methoden *TBall.IncreaseV* und *TBall.DecreaseV* berücksichtigen nicht das Vorzeichen der Geschwindigkeit, d.h. eventuell verlangsamt die Methode *IncreaseV* auch den Ball. Schreibe diese Methoden so um, dass jeweils der Absolutbetrag der Geschwindigkeit entsprechend geändert wird.
3. Wenn sich zwei Kugeln gleicher Masse zentral elastisch stoßen, tauschen sie gegenseitig ihre Geschwindigkeiten aus. Um dies programmieren zu können, muss die Klasse *TBall* öffentliche Methoden zum Setzen bzw. Auslesen der Geschwindigkeit haben. Programmiere dies!
4. Die Bälle sollen nun in beliebige Richtungen rollen. Dafür erhält die Klasse *TBall* ein zusätzliches Attribut namens *winkel*. Ein Winkel von *0* entspricht der Richtung nach Osten, der Winkel *pi/2* entspricht der Nordrichtung usw. Erzeugt wird ein Ball nun mit dem *constructor TBall.Create (X,Y,r,v: Integer; alpha: REAL; Container: TForm);* Die Reflektionsmethoden müssen angepaßt werden. Am linken und rechten Tischrand wird nun ebenfalls reflektiert. Überlege, wie sich die Winkel bei einer Reflektion ändern!
5. Aufgrund der Rollreibung sollen die Bälle langsamer werden. Simuliere dies, indem du bei jedem Timeraufruf die Geschwindigkeit um den *Reibungsfaktor* 0.995 langsamer machst. Beachte: die Geschwindigkeit sollte nun anfangs deutlich größer gewählt werden. Außerdem muss sie nun vom Typ REAL sein.
6. ***(mathematisch anspruchsvoll)*** Normalerweise stoßen sich zwei Kugeln nicht zentral sondern schräg. Ein Stoß ist zentral, wenn sich beide Kugeln entlang ihrer Verbindungsgeraden M_1M_2 bewegen. In diesem Falle tauschen sie einfach ihre Geschwindigkeitsvektoren aus (Voraussetzung: gleiche Masse). Falls der Stoß nicht zentral war, muss man die Geschwindigkeiten beider Kugeln zerlegen in jeweils eine Komponente parallel zu M_1M_2 und eine Komponente senkrecht dazu. Nur die parallele Komponente wird ausgetauscht, die andere bleibt erhalten.
7. ***(informatisch anspruchsvoll)***
   a) Drei Kugeln sollen auf dem Tisch herumrollen und sich gegenseitig und an den Tischkanten stoßen.
   b) Eine beliebige, aber konstante Anzahl n von Kugeln sollen auf dem Tisch herumrollen und sich gegenseitig und an den Tischkanten stoßen.

Projekt: Gebäude




```
unit mGebaeude;

interface

USES Graphics, Forms;
Type
  THaus = class(TObject)
            protected
              zX, zY, zBreite, zHoehe, zR: Integer;
              zLichtAn, zTuerAuf: Boolean;
              kContainer: TForm;
              procedure FensterZeichnen; virtual;
              procedure WandZeichnen;
              procedure DachZeichnen;
              procedure TuerZeichnen;
            public
              constructor Create(XPos, YPos, Breite:
                      Integer; container: TForm); virtual;
              procedure Zeichnen;
          end;
```

```
implementation

constructor THaus.Create(XPos, YPos, Breite: Integer;
                                    Container: TForm);
begin
  inherited Create; // Die Create-Methode der obersten Delphiklasse
                                    TObject besitzt keine Parameter!
  kContainer := Container;
  zX := XPos;
  zY := YPos;
  zBreite := Breite;
  zHoehe := zBreite;
  zR := Breite DIV 5;
  zLichtAn := False;
  zTuerAuf := False
end;

procedure THaus.Zeichnen;
begin
  WandZeichnen;
  Dachzeichnen;
  TuerZeichnen;
  FensterZeichnen
end;

procedure THaus.WandZeichnen;
begin
  With kcontainer.Canvas Do begin
    brush.Color := clBtnFace;
    pen.Width := 2;
    Rectangle(zX, zY, zX+zBreite, zY-zHoehe)
  end
end;

procedure THaus.DachZeichnen;
begin
   With kcontainer.Canvas Do begin
     pen.Width := 4;
     MoveTo(zX+ zBreite DIV 2, zY-zHoehe-5*zR DIV 2);
     LineTo(zX-zR DIV 2, zY-zHoehe+zR DIV 2);
     MoveTo(zX+ zBreite DIV 2, zY-zHoehe-5*zR DIV 2);
     LineTo(zX+zBreite+zR DIV 2, zY-zHoehe+zR DIV 2);
  end
end;
```

```
procedure THaus.TuerZeichnen;
begin
  With kcontainer.Canvas Do begin
    brush.Color := clBtnFace;
    pen.Width := 2;
    Rectangle(zX+zR, zY, zX+2*zR, zY-2*zR)
  end
end;



procedure THaus.FensterZeichnen;
begin
  With kcontainer.Canvas Do begin
    brush.Color := clBtnFace;
    pen.Width := 2;
    Rectangle(zX+zR, zY-3*zR, zX+2*zR, zY-4*zR);
    Rectangle(zX+3*zR, zY-3*zR, zX+4*zR, zY-4*zR);
    Rectangle(zX+3*zR, zY-zR, zX+4*zR, zY-2*zR);
  end
end;

end.
```

unit mHaupt;

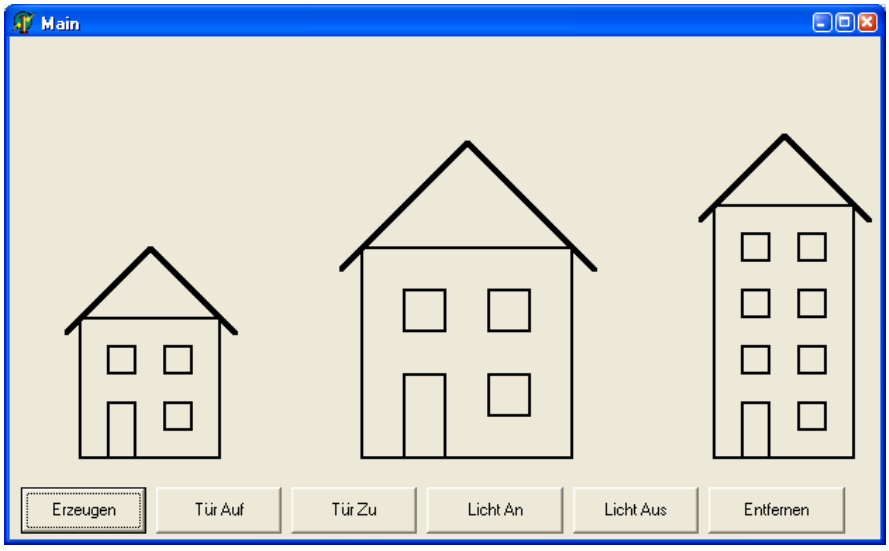
```
interface
uses .....;

type
  TMain = class(TForm)
    BtErzeugen, BtTuerAuf, BtTuerZu: TButton;
    BtLichtAn, BtLichtAus, BtEntfernen: TButton;
    procedure BtErzeugenClick(Sender: TObject);
    procedure BtEntfernenClick(Sender: TObject);
    private
     Haus1, Haus2, Haus3: THaus;
  end;
```

```
var  Main: TMain;

implementation ……

procedure TMain.BtErzeugenClick(Sender: TObject);
begin
 Haus1 := THaus.Create(50,230,100,Main);
 Haus1.Zeichnen;
 Haus2 := THaus.Create(210,230,150,Main);
 Haus2.Zeichnen;
 Haus3 := THaus.Create(400,230,60,Main);
 Haus3.Zeichnen
end;


procedure TMain.BtEntfernenClick(Sender: TObject);
begin
 IF Haus1 <> NIL THEN BEGIN
   Haus1.Destroy;
   Haus1 := NIL
 END;
 IF Haus2 <> NIL THEN BEGIN
   Haus2.Destroy;
   Haus2 := NIL
 END;
 IF Haus3 <> NIL THEN BEGIN
   Haus3.Destroy;
   Haus3 := NIL
 END;
 Canvas.Brush.Color := clwindow;
 Canvas.Rectangle(0,0,Width, Height);
end;

end.
```

Aufgaben

1. Die Türen der Häuser sollen mit Klinke gezeichnet werden. Beachte, dass der Ort der Klinke auch von der Hausgröße abhängt!
2. Die Klasse *THaus* soll noch zwei öffentliche Methoden namens *machLichtAn* und *machLichtAus* erhalten. Programmiere dann auch zusätzlich noch die entsprechenden Buttons im Hauptprogramm.
3. Die Klasse *THaus* soll noch zwei öffentliche Methoden namens *machTuerAuf* und *machTuerZu* erhalten. Programmiere dann auch zusätzlich noch die entsprechenden Buttons im Hauptprogramm.
4. Die Methode *Zeichnen* der Klasse *THaus* soll den Status *protected* erhalten. Das bedeutet, dass sich das Haus selber zeichnen soll.
5. Die Methode *Destroy* der Klasse *THaus* soll nun auch seine eigene Zeichnung auf dem Formblatt entfernen.

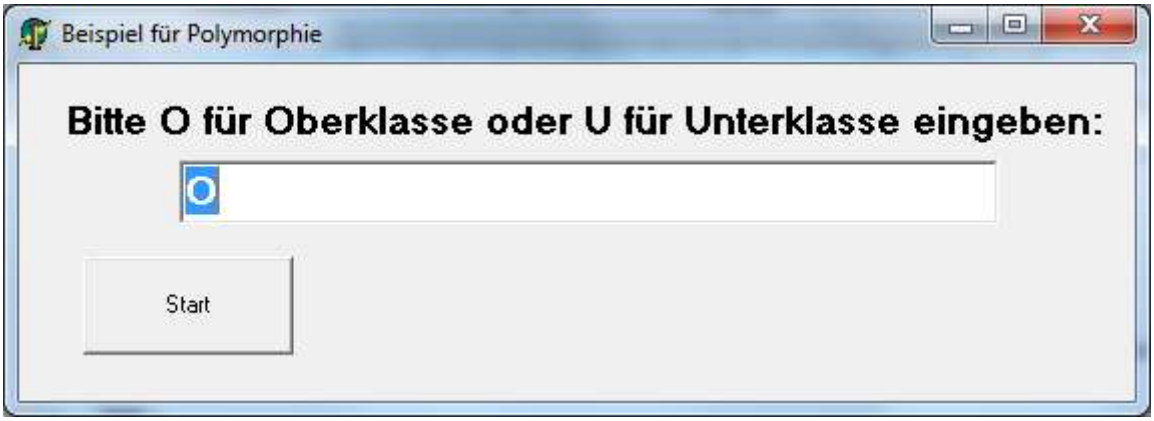
Polymorphie



THochhaus
…….
!Fensterzeichnen + !Create

Die Klasse *THochhaus* lässt sich von der Klasse *THaus* ableiten. Es müssen nur zwei Methoden geändert (d.h. überschrieben) werden: die Methode *Fensterzeichnen* (weil mehr Fenster vorhanden sind) und die Methode *Create* (weil jetzt Höhe und Breite nicht mehr identisch sind). Alle anderen Methoden sind mit den entsprechenden Methoden der Oberklasse identisch.

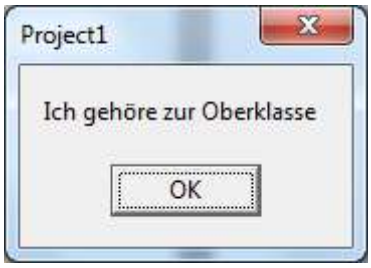
Nehmen wir an, dass im Programm ein normales Haus und ein Hochhaus gezeichnet werden soll. Daß die jeweils passende Methode *Fensterzeichnen* ausgeführt wird, bezeichnet man als ***dynamische*** oder ***späte Bindung*** von Methoden. Erst zur Laufzeit des Programms wird geprüft, welches Objekt vorliegt (hier: *THaus* oder *THochhaus*), und in Abhängigkeit von dieser Prüfung wird entschieden, welche der beiden gleichnamigen Methoden *Fensterzeichnen* ausgeführt wird.
Diese Prüfung findet natürlich nur für virtuelle und überschriebene (override) Methoden statt.
Das Ergebnis dieser Prüfung ist übrigens nicht abhängig von der Deklaration der Typvariablen (bekanntlich kann eine Variable der Oberklasse auch ein Objekt einer Unterklasse beinhalten). Betrachte dazu folgendes Programm:



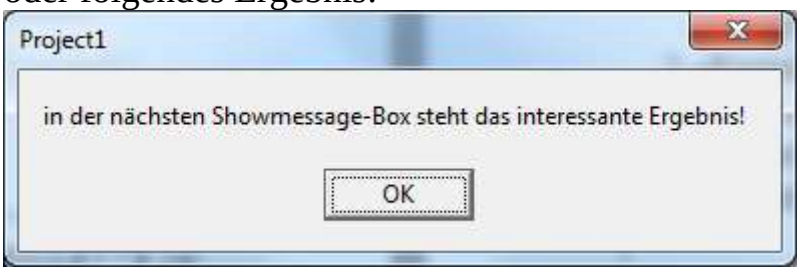

```
unit Polymorphie;
interface
uses ……  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    BtStart: TButton;
    procedure BtStartClick(Sender: TObject);
   private
     { Private declarations }
   public
     { Public declarations }
  end;

  TOberklasse = class(TObject)
    public
      procedure meldeAlles;
    protected
      procedure meldeKlasse; virtual;
    end;
```

```
  TUnterklasse = class(TOberklasse)
    protected
      procedure meldeKlasse; override;
    end;

var
  Form1: TForm1;
  OKObjekt: TOberklasse;
  UKObjekt: TUnterklasse;
  VarObjekt: TOberklasse;

implementation
{$R *.dfm}

procedure TOberklasse.meldeAlles;
BEGIN
 Showmessage('in der nächsten Showmessage-Box steht
                        das interessante Ergebnis!');
 meldeKlasse
END;


procedure TOberklasse.meldeKlasse;
BEGIN
  Showmessage('Ich gehöre zur Oberklasse')
END;


procedure TUnterklasse.meldeKlasse;
BEGIN
  Showmessage('Ich gehöre zur Unterklasse')
END;


procedure TForm1.BtStartClick(Sender: TObject);
begin
  UKObjekt := TUnterklasse.create;
  OKObjekt := TOberklasse.create;
  If Edit1.Text = 'O' THEN VarObjekt := OKObjekt
  ELSE VarObjekt := UKObjekt;
  VarObjekt.meldeAlles;
end;

end.
```

Obiges Programm liefert abhängig vom Inhalt des Edit-Fensters entweder folgendes Ergebnis:



oder folgendes Ergebnis:



Wichtig: erst zur Laufzeit des Programms entscheidet der Benutzer, was im Edit-Fenster steht. Davon hängt ab, ob die Variable *VarObjekt* vom Typ der Ober- oder der Unterklasse definiert wird (beides ist möglich). Unabhängig vom Klassentyp wird auf jeden Fall die Methode *meldeAlles* aufgerufen. Innerhalb dieser Methode wird jedoch erst während der Laufzeit des Programms entschieden, welche *meldeKlasse*-Methode (diejenige der Unterklasse oder diejenige der Oberklasse?) aufgerufen wird. Das hängt davon ab, von welcher Klasse die Variable *VarObject* zum Zeitpunkt des Aufrufes ist.

Dass Objekte verschiedener Klassen unterschiedlich auf gleichnamige Aufträge reagieren können, bezeichnet man als ***polymorphes Verhalten*** oder ***Polymorphie.***
Im Zusammenhang mit der *Polymorphie* spricht man auch von der sog. *dynamischen* oder *späten Bindung* von Methoden an die Variablen. Damit ist, wie oben beschrieben, gemeint, dass erst zur Laufzeit entschieden wird, welche Methode eine Variable ausführen kann.

Aufgaben

1. Erstelle die Klasse *THochhaus* und erzeuge ein Hochhaus auf dem Hauptformular!
2. Erstelle eine Klasse *TUhrenhaus*! Objekte dieser Klasse haben im Dach eine laufende Uhr.
3. Erstelle eine Klasse *TFarbhaus!* Objekte dieser Klasse sind farbig.
4. Erstelle eine Klasse *TFlachbau*!

Klassenbildung durch Generalisation

Im Unterricht haben wir Objekte vom Typ THaus oder THochhaus konstruiert. Analog könnte man weitere Typen TTurm, THausMitUhr, THalle, TBungalow usw. kreieren. Im Prinzip könnte man alle Typen von der Klasse THaus ableiten.

Dabei würden allerdings einige Unterklassen z.B. Dinge erben, die sie gar nicht benötigen. Vielleicht müssten sogar einige Methoden einfach nur mit *begin end* überschrieben werden, damit sie in der Unterklasse nichts ausrichten.

Ein besseres Vorgehen wäre sicherlich, wenn man aus allen gemeinsam benötigten Attributen und Methoden eine eigene Oberklasse abstrahiert. Dieses Verfahren nennt man auch ***Generalisation***.

Oft ist es auch zweckmäßig, nur aus einigen wenigen Teilklassen eine (Zwischen-)Oberklasse zu abstrahieren.

Aufgabe: In einer Schule werden teils gemeinsame, teils unterschiedliche Daten über Lehrer, Schüler, Sekretärinnen und Hausmeister benötigt. Dasselbe gilt für Daten über Klassenräume, Abstellkammern, Kellerräume usw.
Erstelle entsprechende Klassenhierarchien!