OOP und Delphi

Jeder, der das obligatorische „Hello World" Programm in Delphi erstellt hat, ist schon mit Klassen und Objekten in Berührung gekommen, er hat schon objektorientiert programmiert. Im folgenden Tutorial werde ich die Grundlagen von OOP erläutern und dabei auf die Besonderheiten von Delphi eingehen. Als Abschluss kommt dann ein kleines Beispielprojekt.

Klasse und Objekt - alles das selbe oder doch nicht?

Oft wird der Begriff „Klasse" und „Objekt" verwechselt bzw. missverstanden. Zwischen beiden besteht ein großer Unterschied, dieser ist aber sehr einfach zu merken: Das Verhältnis Klasse zu Objekt ist identisch wie das Verhältnis zwischen Variablentyp und Variable (also z.B. Integer und Variable i).

Was ist eine Klasse?

Die Klasse ist sozusagen eine Vorlage für Objekte. Ähnlich wie bei den einfachen Datentypen (Integer) muss ein Objekt erst deklariert werden, damit diese benutzt werden kann, also

```
var objekt:TKlasse;
```

Allerdings reicht das bei einem Objekt nicht ganz aus. Das Objekt muss nach der Deklaration erst mal initialisiert werden. Bestimmte Attribute müssen Werte zugewiesen werden usw. Die Initialisierung erledigt der *Konstruktor*.

Konstruktor und Destruktor

Der Konstruktor und sein Gegenspieler der Destruktor sind spezielle Methoden eines Objektes, die bei der Initialisierung (Konstruktor) bzw. beim Löschen (Destruktor) aufgerufen werden. Im Konstruktor werden z.B. die Konstruktoren anderer Objekte aufgerufen. Der Destruktor gibt beim Löschen den benutzen Speicher wieder frei, indem er die erzeugten Objekte wieder freigibt.

Eine Klasse kann man grundsätzlich mit einem Record verglichen werden. Dort werden unterschiedliche Datenarten zusammengefasst, um diese zu kombinieren. Bei einer Klasse gibt es aber zusätzlich Funktionen und Prozeduren, die die gespeicherten Daten bearbeiten können. In der OOP gibt es 3 Basisprinzipien, durch die die OOP so mächtig wird:

- Kapselung
- Vererbung
- Polymorphie

Auf diese 3 Prinzipien werde ich im folgenden eingehen.

Was ist nun ein Objekt?

Ein Objekt in der Softwareentwicklung ist letztendlich dasselbe wie in der realen Welt (z.B. Auto). Das Objekt hat bestimmte Eigenschaften (wie z.B. Farbe, Höchstgeschwindigkeit) und das Objekt kann bestimmte Tätigkeiten ausführen, bzw. das Objekt kann bestimmte Befehle ausführen (z.B. starten, Gas geben, bremsen).
Also: Ein Objekt besteht aus verschiedenen *Methoden* (Prozeduren und Funktionen) und *Attributen* (Variablen, die Daten enthalten). Die Attribute werden dabei durch die Methoden

sozusagen geschützt, d.h. dass nur über eine Methode auf ein Attribut zugegriffen werden kann.

Bsp. Tanken: Beim Auto kann normalerweise nur durch das Tankloch Sprit in den Tank eingefüllt werden. In den Tank passt zudem nur eine bestimmte Menge an Treibstoff. Ist der voll, läuft der Tank einfach über.
Als Methode Tanken in einem Objekt würde das ungefähr so aussehen:

```
procedure Tanken(fTankmenge:FLOAT);
begin
  if (fvorhandeneSpritmenge+fTankmenge)>fMaxTankinhalt then
    fvorhandeneSpritmenge=fMaxTankinhalt;
  else
    fVorhandeneSpritmenge:=fVorhandeneSpritmenge+fTankmenge;
end;
```

So, nun kann der imaginäre Tank unseres Autos nie mehr überlaufen. Wichtig ist jetzt nur noch, dass man von „aussen" nicht auf das Attribut fVorhandeneSpritmenge zugreifen kann. Sonst wäre ja folgendes ohne Probleme möglich:

```
fVorhandeneSpritmenge:=2*MaxTankinhalt;
```

Da das in der realen Welt nicht passieren kann, darf es auch in der abgebildeten Version in unserem PC nicht passieren. Damit das nicht möglich ist, gibt es die sogenannte **Kapselung**. Kapselung bedeutet, dass nur mit bestimmten Methoden auf Attribute zugegriffen und diese verändert werden können. Es bedeutet schlicht, dass bestimmte Attribute und Methoden vor der Außenwelt versteckt werden. Bei einem Auto interessiert es den Anwender (Autofahrer) überhaupt nicht, was genau passiert, wenn er auf das Gaspedal tritt, Hauptsache der Motor heult auf.

Wie funktioniert die Kapselung?

In der OOP gibt es die Schutzklassen *private* und *public*. Wie die Namen schon sagen, versteckt private alle folgenden Attribute und Methoden vor der Öffentlichkeit, während public den vollen Zugriff erlaubt. Was sich hier jetzt erstmal so toll anhört, macht sich nachher in der Entwicklung ziemlich heftig bemerkbar, da für jedes Attribut, das als private deklariert wird, eine oder mehrere Methoden benötigt wird, die den Zugriff auf diese Variable gestattet.
Bsp.: Da der Tankinhalt unseres Objektes Autos als private deklariert wurde, brauchen wir eine Methode (Tanken) die den Zugriff auf dieses Attribut erlaubt. Hier können dann z.B. Wertprüfungen (minimal, maximal) vorgenommen werden, um Fehler zu vermeiden.

Um diesen Zugriff auf private Attribute zu vereinfachen, gibt es in Delphi die *properties* (Eigenschaften). Diese werden so deklariert:

```
property Tankfuellung SINGLE read fVorhandeneSpritmenge;
```

Nun kann einfach über den Aufruf

```
  Edit1.Text:=FloatToStr(Objekt.Tankfüllung);
```

der Tankinhalt angezeigt werden. Will man auch einen Schreibzugriff auf die Eigenschaft erlauben muss sie folgendermaßen implementiert werden:

```
property Tankfuellung SINGLE read fVorhandeneSpritmenge write
Tanken;
```

Delphi kennt noch 2 weitere Schutzklassen *protected* und *published*. Im Bereich *published* können nur Attribute (also Daten) stehen. Diese können dann schon zur Entwurfszeit im Objektinspektor mit Werten belegt werden. Die Schutzklasse *protected* werde ich später bei der Vererbung erklären.

Vererbung

Bei der Vererbung wird ein sog. Basisobjekt implementiert und von diesem weitere Objekte abgeleitet. Der Vorteil: Diese abgeleiteten Objekte erben alle Attribute und Methoden der Basisklasse, d.h. diese müssen nicht noch einmal implementiert werden.
Als Beispiel soll wieder unser Auto dienen. Wir definieren eine Basisklasse *TFahrzeug*. Diese Klasse hat die Attribute

- Höchstgeschwindigkeit
- Reifenanzahl
- Sitzplätze

und die Methoden

- gasgeben
- bremsen
- schalten

Durch die Vererbung ist es jetzt möglich die Klasse *TKFZ* und die Klasse *TFahrrad* von der Basisklasse *TFahrzeug* abzuleiten. Die beiden abgeleiteten Klassen können dann die Methoden (gasgeben, bremsen, schalten) entsprechend ihrer Bauart implementieren. Bei der Klasse *TFahrrad* werden die Attribute im Konstruktor gesetzt. Nicht so bei der Klasse *TKFZ*, denn von dieser werden jetzt weiter Klassen *TAuto*, *TBus* und *TLKW* abgeleitet. Bei diesen Klassen müssen jetzt nur noch die Attribute im Konstruktor gesetzt werden. Die Methoden sind bei allen gleich. Es ist also sehr einfach, aus einer allgemein gehaltenen Basisklasse weitere Klasse abzuleiten, die oft auch nur als Vorlage für weitere Vererbungen dienen. Zudem ist es jetzt jederzeit möglich, weitere Klassen von *TFahrzeug* (z.B. Flugzeug) oder von *TKFZ* (z.B. Panzer) abzuleiten, ohne dass man wissen muss, wie die Methode *gasgeben* funktioniert.

Bei der Basisklasse *TFahrzeug* wurden die Methoden ja nicht implementiert, d.h. es wurden nur die Methodenköpfe (function gasgeben;) aufgeführt. Diese Methoden müssen in einer abgeleiteten Klasse mit „Leben" gefüllt werden, damit diese funktionieren. Solche Methoden werden als **abstrakt** bezeichnet, da sie in der Klasse nicht ausgeführt werden können. Solche Klassen, die abstrakte Methoden haben, nennt man dann **abstrakte Klassen** (logisch). Von diesen Klassen kann dann aber KEIN Objekt abgeleitet werden! Ist auch klar, denn was soll ich mit einem Fahrzeug ohne Bremsanlage?

Jetzt noch ein Wort zu der Schutzklasse *protected*. Oben habe ich ja ausgeführt, dass alles was unter *private* deklariert wird, für alle unsichtbar ist, auch für abgeleitete Klassen! Damit aber diese auf Methoden und Attribute der Elternklasse zurückgreifen können, ohne dass diese von außen einsehbar sind, wurde in Delphi die Schutzklasse *protected* eingeführt. Alles was hier deklariert wird, ist für Außenstehende unsichtbar, für die abgeleiteten Klassen jedoch sichtbar.

Polymorphie

Die Polymorphie (Vielgestaltigkeit) hängt sehr eng mit der Vererbung zusammen. Wir haben im der Klasse *TFahrzeu*g die Methoden gasgeben, bremsen und schalten erzeugt. Die abgeleiteten Klassen *TKFZ* und *TFahrrad* besitzen diese Methoden, müssen diese aber durch eigene Methoden „überschreiben". Der Aufruf ist aber immer der gleiche! Es ist also egal, ob man ein Objekt der Klasse *TFahrrad* oder der Klasse *TKFZ* hat, immer funktioniert der Aufruf

```
objekt.gasgeben;
```

Bei einem Fahrrad wird eben mehr gestrampelt, bei einem KFZ wird aus Gaspedal getreten.

Und genau das ist Polymorphie! Eine Methode mit gleichem Namen wird in abgeleiteten Klassen unterschiedlich implementiert (führt also was anderes aus).

So, die grundsätzlichen Dinge sind jetzt geklärt, alles weitere wird sich beim Beispiel erklären.

Das Beispiel

Es soll eine Klasse für Flächen implementiert werden. Diese soll die grundsätzlichen Methoden besitzen. Von dieser Basisklasse sollen dann Klassen für Rechtecke und Kreise abgeleitet werden, die die speziellen Methoden für diese Flächen bereitstellen.

Die Basisklasse *TFlaeche* soll folgende Methoden besitzen:
- berechneFlaeche
- berechneUmfang

und folgende Eigenschaften
- Flaeche
- Umfang

Die abgeleitete Klasse *TRechteck* erbt von TFlaeche und implementiert zusätzlich folgende Attribute:
- laenge
- breite

Die abgeleitete Klasse Kreis implementiert folgendes zusätzliche Attribut:
- radius

So, jetzt geht's an Delphi. Öffne ein neues Projekt und speichere das ganze gleich ab (Projekt oop_tuto, Unit1 als UMain). Füge dem Projekt eine neue Unit hinzu und speichere die Unit als UFlaeche gleich ab.

In die UFlaeche kommt nun die Klassendeklaration hinein:

```
unit UFlaeche;

interface

type

TFlaeche = class(TObject)
  protected
    function berechneFlaeche:double; virtual; abstract;
    function berechneUmfang:double; virtual; abstract;
  public
    property Flaeche:double read berechneFlaeche;
    property Umfang:double read berechneUmfang;
  end;

implementation
end.
```

Erläuterung:
Die beiden Methoden (berechneFlaeche und berechneUmfang) sind als abstrakt definiert, d.h. die Methodenrümpfe werden erst in den abgeleiteten Klassen implementiert. Abstrakte Methoden in Delphi gibt es in zwei Ausführungen, zum einem *virtual* und zum anderen *dynamic* sein. Virtuelle Methoden sind geschwindikeitsoptimiert und dynamische Methoden codeoptimiert (gibt weniger Code), sonst gibt's es keine Unterschiede.
Die beiden Eigenschaften (*Flaeche* und *Umfang*) sind schreibgeschützt und rufen einfach die Funktionen auf, die die Werte berechnen.
Hier passiert noch nicht sehr viel. Kommen wir deshalb gleich zu den Erben *TRechteck* und *TKreis* (am besten für jede Klasse eine extra Unit benutzen!):

```
unit URechteck;

interface
uses UFLaeche;
type
     TRechteck = class(TFlaeche)
     private
       lang, breit:integer;

     protected
       function berechneFlaeche:double; override;
       function berechneUmfang:double; override;

     public
       constructor create(Laenge, Breite:Integer);
     end;
implementation
constructor TRechteck.create(Laenge, Breite:Integer);
begin
  inherited create;
  lang:=laenge;
  breit:=breite;
end;

function TRechteck.berechneFlaeche:double;
begin
  result:=lang*breit;
end;
function TRechteck.berechneUmfang:double;
begin
  result:=2*lang+2*breit;
end;
end.
```

Hier sehen wir jetzt endlich mal, die komplette Funktionalität unseres Zieles.
Was hier auffällt, ist im Konstruktor / Destruktor der Aufruf

```
inherited create;
```

Mit diesem Aufruf wird der Konstruktor der Elternklasse von *TRechteck* also *TFlaeche* aufgerufen. Aber Moment mal, in *TRechteck* gibt's doch gar keinen Konstruktor create!

TObject - Die Basisklasse von Delphi

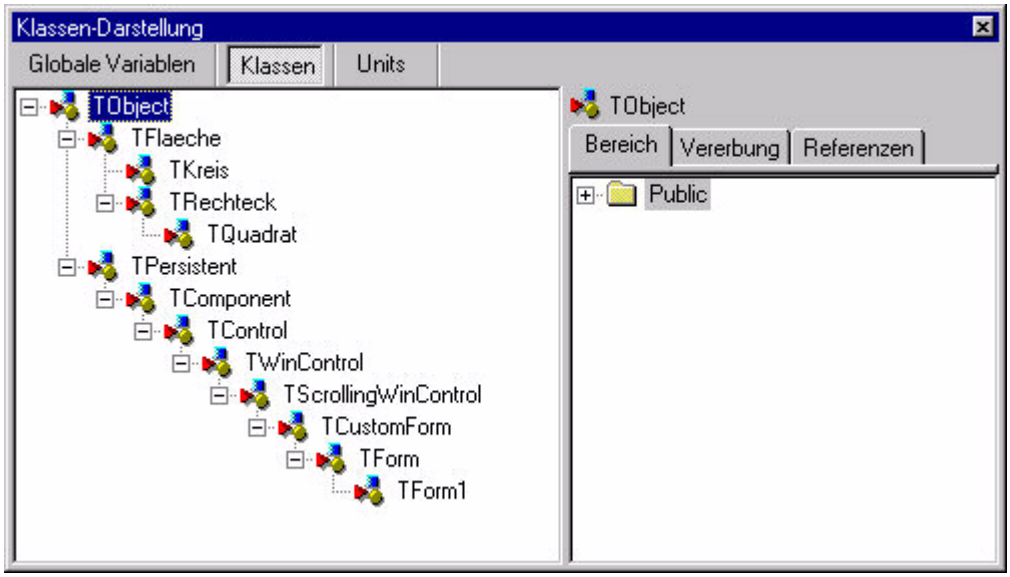
In Delphi gibt es die Basisklasse *TObject*. Von dieser Klasse erben alle anderen Klassen die Methoden, ob gewollt oder ungewollt. *TObject* sieht so aus:

```
TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

Da wird schon eine ganze Menge an Methoden erzeugt. Wo steht jetzt aber in unserem Code, dass von *TObject* geerbt werden soll? Wird hinter dem Aufruf

```
TFlaeche = class
```

nichts angegeben, dann wird automatisch von *TObject* geerbt. Letztendlich haben wir dann keine Basisklasse erzeugt, denn unsere Klasse *TFlaeche* erbt ja schon von *TO*bject.



Das hier ist die Klassendarstellung von Delphi (Hauptmenü: *Ansicht->Browser*). Man sieht hier die Klassen und wer von wem abstammt. Als letzte im Baum kommt TForm1, die Klasse unseres Hauptfensters.

Kommen wir nun wieder zu unserem Beispiel und zur Klasse *TKreis*:

```
unit UKreis;

interface
uses UFlaeche, Math;

type
    TKreis = class(TFlaeche)
    private
      radius:Integer;
    protected
      function berechneFlaeche:double; override;
      function berechneUmfang:double; override;

    public
      constructor create(KreisRadius:Integer);

    end;
implementation

constructor TKreis.create(KreisRadius:Integer);
begin
  inherited create;
  radius:=Kreisradius;
end;

function TKreis.berechneFlaeche;
begin
  result:=pi*SQR(radius);
end;

function TKreis.berechneUmfang;
begin
  result:=2*pi*radius
end;
end.
```

Um wirklich die letzten Zweifler von der Mächtigkeit von OOP zu überzeugen, leiten wir jetzt noch eine Klasse *TQuadrat* ab. Diesmal aber nicht von *TFlaeche* sondern von *TRechteck*:

```
unit UQuadrat;

interface
uses URechteck;

type TQuadrat = class (TRechteck)
     public
       constructor create(Breite:Integer);
     end;

implementation

constructor TQuadrat.create(Breite:Integer);
begin
  Inherited create(breite, breite);
end;

end.
```

Der Konstruktor ruft einfach den Konstruktor der Klasse *TRechteck* auf und übergibt als Parameter einfach die Seitenlänge des Quadrates zweimal.

Klassenmethoden und Klassenvariabeln

Als letzten Punkt werde ich den Bereich der Klassenmethoden und -variabeln ansprechen. Vorweg: Klassenvariabeln wie in C++/Java gibt es in Delphi nicht, man kann sich aber relativ einfach behelfen.
Klassenmethoden sind Methoden, die in einer Klasse definiert werden und KEIN Objekt brauchen um ausgeführt zu werden! Sie gehören also der Klasse und nicht irgendeinem deklarierten Objekt! Eine solche Klassenvariable kann z.B. dazu benutzt werden um zu zählen wie viele Objekte von dieser Klasse erzeugt wurden.
Eine Klassenvariable wird wie folgt erzeugt:

```
    class function count:integer;
```

Vor die Methode kommt einfach das Schlüsselwort *class*. Die Methode wird dann ganz normal implementiert. Hier der komplette Code unserer Klasse *TFlaeche*:

```
unit UFlaeche;

interface

type

TFlaeche = class(TObject)
  private

  protected

    function berechneFlaeche:double; virtual; abstract;
    function berechneUmfang:double; virtual; abstract;

  public

    constructor create;
    destructor free;
```

```
    class function count:integer;

    property Flaeche:double read berechneFlaeche;
    property Umfang:double read berechneUmfang;
  end;

implementation

var anzahl:integer;

constructor TFlaeche.create;
begin
  inc(anzahl);
end;

destructor TFlaeche.free;
begin
  dec(anzahl);
end;

class function TFlaeche.count:integer;
begin
  result:=anzahl;
end;
end.
```

Diesmal hat sogar der Konstruktor und der Destruktor was zu tun. Sie sollen beim Erzeugen / Löschen eines Objektes zur Variable *anzahl* eins addieren / subtrahieren, sodass dort die Anzahl aller erzeugten Objekte verzeichnet ist. Die Variable *anzahl* ist eine hausgemachte Klassenvariable, da auf sie nur durch die Klassenfunktion zugegriffen werden kann.

Das wars dann auch schon. Klar kann ich auf 10 Seiten nur die Oberfläche eines Themas ankratzen, zu dem Fachautoren Bücher schreiben. Bücher über OOP und Delphi gibts aber leider kaum. In fast allen Delphi-Büchern steckt etwas OOP drin, aber keines handelt nur über OOP. Deshalb gibts diesmal auch keine Literaturtipps.

Wie immer: Bei Problemen oder Anmerkungen:
wlemmermeyer@yahoo.de oder http://www.lemmermeyer.purespace.de. Dort gibts auch noch weitere Tutorials.
25.05.2001 Wolfgang Lemmermeyer