Einführung in die objektorientierte Programmierung mit Delphi

Klassen in Delphi

Michael Puff
mail@michael-puff.de

2010-03-26

Inhaltsverzeichnis

Vorwort

Die Objektorientierter Programmierung (OOP) ist im Prinzip nur eine Definition, wie der Programmierer zur Entwurfszeit seine Programmschnittstellen zu definieren hat. Die OOP basiert auf der prozeduralen Programmierung und stellt einen Überbau da, der es ermöglicht strukturierte Schnittstellen zu entwerfen. Dieser Überbau muss die Fähigkeit besitzen, dass die Schnittstellen zur Laufzeit selbst über die verwendeten Schnittstellen Auskunft zu geben. In Delphi geschieht dies mit über die Typinformation (RTTI = Run Time Type Information). Dadurch wird es möglich eine strenge Typprüfung und somit mehr Sicherheit in den Schnittstellen zu implementieren. Die Mittel OOP umzusetzen basieren also teilweise auf prozeduraler Programmierung und auf den Fähigkeiten des Compilers und der IDE.

Grundgedanke von OOP war es Programmierer dazu zu zwingen, strukturierten, wiederverwendbaren und erweiterbaren Code zu entwickeln. Da dies nur in einer restriktiven und strukturierten Programmiersprache wird dadurch auch erklärbar. Die Grundlage der OOP ist das Klassenkonzept. Objekte selber sind nur die zur Laufzeit existierenden Datencontainer deren Schnittstellen durch die Klasse klar definiert wurden.

Die Objektorientierung führt nun ganz neue Programmiertechniken ein: Kapselung, verbergen von Code, Zugriff nur über definierte Schnittstellen; Vererbung, Erweiterung von bestehen den Code; Wiederverwendbarkeit, Code kann einfach in anderen Projekten wiederverwendet werden. Auf der einen Seite erfordert dies ein gesteigertes Abstraktionsvermögen des Programmierers. Auf der anderen Seite ist eine Klasse auch der Versuch die Wirklichkeit programmiertechnisch abzubilden.

Danksagung

An dieser Stelle will ich mich recht herzlich bei allen aus der Delphi-Praxis bedanken, die mir bei der Erarbeitung dieses Tutorials helfend bei Seite gestanden haben. Mein Dank gilt ins besondere choose für sein Beispiel zur Polymorphie, Motzi für weitere Erläuterungen und den Beiträgen und Code Beispielen von Chewie, Bernd Ua und NicoDe zu Klassenmethoden. Natürlich darf an dieser Stelle auch nicht meine liebe Freundin Micky vergessen werden, die sich die Mühe gemacht hat, das ganze Korrektur zu lesen, obwohl sie wahrscheinlich kein Wort von dem verstanden hat, was sie gelesen hat.

Insbesondere Danke ich Andreas Hausladen dafür, dass ich seine Texte bezüglich der Interna was beim Aufruf des Konstruktors und Destruktors passiert und wie Delphi Methodenzeiger realisiert übernehmen durfte (siehe dazu die Kapitel B.1, B.2 und B.3 ab Seite 33). Ursprünglich erschienen diese Texte auf der Delphi Seite „Delphi-Source – Der Delphi-Treff“[1].

Michael Puff, Kassel den 26. November 2007.

[1] http://www.dsdt.info/

1. Eine Klasse in Delphi

1.1. Grundlegendes

Eine Klasse (auch Klassentyp) definiert eine Struktur von Feldern, Methoden und Eigenschaften. Die Instanz eines Klassentyps heißt Objekt. Die Felder, Methoden und Eigenschaften nennt man auch ihre Komponenten oder Elemente.

- Felder sind im Wesentlichen Variablen, die zu einem Objekt gehören.
- Eine Methode ist eine Prozedur oder Funktion, die zu einer Klasse gehört.
- Eine Eigenschaft ist eine Schnittstelle zu den Daten (Feldern) eines Objektes. Eigenschaften verfügen über Zugriffsbezeichner, die bestimmen, wie ihre Daten gelesen oder geändert werden sollen (Getter/Setter).

Objekte sind dynamisch zugewiesene Speicherblöcke, deren Struktur durch die Klasse festgelegt wird. Jedes Objekt verfügt über eine Kopie der in der Klasse definierten Felder. Die Methoden werden jedoch von jeder Instanz gemeinsam genutzt. Das Erstellen und Freigeben von Objekten erfolgt mit Hilfe spezieller Methoden, dem Konstruktor und dem Destruktor.

Eine Klassentypvariable ist ein Zeiger auf ein Objekt. Deshalb können mehrere Variablen auf dasselbe Objekt verweisen. Sie können also auch, wie andere Zeigertypen, den Wert nil annehmen. Sie müssen aber nicht explizit dereferenziert werden, um auf das betreffende Objekt zuzugreifen.

Ein Klassentyp muss deklariert und benannt werden, bevor er instantiiert werden kann. Die Deklaration einer Klasse steht im Interface-Abschnitt einer Unit. Die Implementation im Implementations-Abschnitt. Eingeleitet wird die Deklaration einer Klasse mit dem Schlüsselwort type, gefolgt von dem Bezeichner der Klasse, einem Gleichheitszeichen, dem Schlüsselwort class und in Klammern eventuell die Angabe einer Klasse, von der die neue Klasse abgeleitet werden soll.

```
type
  TMyClass = class(TParent)
    Elementliste
  end;
```

`TMyClass` ist ein beliebiger, gültiger Bezeichner. Die Angabe einer Vorfahrenklasse (`TParent`) ist optional. Gibt man keinen Vorfahren an, so nimmt Delphi automatisch die Klasse `TObject` als Vorfahre der Klasse an. Die in der Unit `System` deklarierte Klasse `TObject` ist der absolute Vorfahre aller anderen Klassentypen. In ihr sind alle Methoden implementiert, die nötig sind zur Erstellung und Verwaltung einer Klasse. Das betrifft unter anderem den Prozess

des Erzeugens und die Verwaltung eines Objektes zur Laufzeit. Methoden werden in einer Klassendeklaration als Funktions- oder Prozedurköpfe ohne Rumpf angegeben.

Die Methoden einer Klasse werden im Implementations–Abschnitt einer Unit implementiert. Dabei wird die Schlüsselwörter `procedure` oder `function` vorangestellt. Es folgt der Klassenname und durch den Gültigkeitsoperator „.“ gefolgt die Methode.

```
procedure TMyClass.Something;
begin

end;
```

Tipps und Tricks:

- Bei Klassennamen gilt als Konvention, dass man den Klassennamen mit einem großem „T“ beginnen lässt, welches für den Begriff „Type“ steht.
- Aus Gründen der Übersichtlichkeit sollte man auch bei Klassen ohne Vorfahre `TObject` als Vorfahre angeben.
- Ein Klassentyp ist zu seinen Vorfahren zuweisungskompatibel.

Siehe dazu Demo *einfache Klasse*.

1.2. Sichtbarkeiten

Sichtbarkeit bedeutet, dass man den Zugriff nach außen hin einschränkt, das heißt, der Anwender einer Klasse sieht nur das, was er sehen muss, um die Klasse einzusetzen. Er sieht also nur die von der Klasse definierten und zur Verfügung gestellten Schnittstellen. Auf interne Methoden der Klasse, die die Klasse zur Manipulation der Daten implementiert, hat der Anwender keinen Zugriff, womit die Klasse die Sicherheit der Eigenschaften wahren kann. In einer Klasse hat jedes Element ein Sichtbarkeitsattribut, das durch die reservierten Schlüsselwörter `private`, `protected`, `public` angegeben wird. Ein Element ohne Attribut erhält automatisch die Sichtbarkeit des vorhergehenden Elementes in der Deklaration. Die Elemente am Anfang einer Klassendeklaration ohne Sichtbarkeitsangabe werden standardmäßig als `published` bzw. `public` deklariert. Grundsätzlich unterscheidet man zwei Arten von Sichtbarkeiten: private und öffentliche.

private Auf private-Elemente kann nur innerhalb des Moduls (Unit oder Programm) zugegriffen werden, in dem die Klasse deklariert ist. Im `private`-Abschnitt werden Felder und Methoden deklariert, die strengen Zugriffsbeschränkungen unterliegen. Ein `private`-Element steht nur in der Unit zur Verfügung.

protected Ein `protected`-Element ist innerhalb des Moduls mit der Klassendeklaration und in allen abgeleiteten Klassen sichtbar. Mit diesem Sichtbarkeitsattribut werden also Elemente deklariert, die nur in den Implementierungen abgeleiteter Klassen verwendet werden sollen.

public Ein `public`-Element unterliegt keiner Zugriffsbeschränkung. Klasseninstanzen und abgeleitete Klassen können, auf diese Felder und Methoden zugreifen. Ein `public`-Element steht überall da zur Verfügung, wo die Klasse sichtbar ist, zu der es gehört.

published Published steuert die Sichtbarkeit im Objektinspektor und bewirkt weitere Informationen für RTTI.

Die Sichtbarkeit eines Elements kann in einer untergeordneten Klasse durch Redeklaration erhöht, jedoch nicht verringert werden. So kann eine `protected`-Eigenschaft in einer abgeleiteten Klasse als `public` deklariert werden, aber nicht als private.

Tipps und Tricks:

- Alle Felder (Variablen einer Klasse) sollten privat deklariert werden. Zugriff sollte nur über eine höhere Zugriffsebene (protected, published, public) über Properties erfolgen. Dies gibt einem die Flexibilität zu entscheiden, wie auf ein Feld zugegriffen werden soll, ohne an der Art und Weise, wie die Klasse benutzt wird, etwas zu ändern.
- Öffentliche Methoden und Eigenschaften definieren das Verhalten einer Klasse. Wird eine Klasse erst einmal benutzt, sollte man davon Abstand nehmen, öffentliche Methoden und Eigenschaften zu ändern. Statt dessen sollte eine neuen Methode deklariert werden mit den neuen Eigenschaften.
- Es ist sinnvoll, die Elemente ihrer Sichtbarkeit nach in der Klasse zu deklarieren, wobei man mit den Elementen der geringsten Sichtbarkeit beginnt und sie gruppiert. Bei dieser Vorgehensweise braucht das Sichtbarkeitsattribut nur einmal angegeben zu werden, und es markiert immer den Anfang eines neuen Deklarationsabschnittes.
- Die meisten Methoden, die nicht public sind, sollten im protected-Abschnitt deklariert werden, da man nicht vorhersehen kann wie eine Klasse in Zukunft genutzt wird. Eventuell muss sie vererbt werden, und die abgeleiteten Klassen müssen interne Methoden ändern.
- Methoden im protected-Abschnitt einer Klasse sollten im Allgemeinen als virtual deklariert sein, damit sie gegebenenfalls überschrieben werden können. Ansonsten würde eine Deklaration einer Methode im protected-Abschnitt wenig Sinn machen.

Siehe dazu Demo: *einfache Klasse*.

1.3. Konstruktor, Destruktor

1.3.1. Konstruktor

Eine Klasse hat zwei besondere Methoden, den Konstruktor und den Destruktor. Wird explizit kein Konstruktor implementiert, wird der Konstruktor der übergeordneten Klasse aufgerufen. Obwohl die Deklaration keinen Rückgabewert enthält, gibt ein Konstruktor immer einen Verweis auf das Objekt, das er erstellt bzw. in dem er aufgerufen wird zurück. Eine Klasse kann auch mehrere Konstruktoren haben. Im Normalfall hat sie jedoch nur einen mit der Bezeichnung `Create`.

```
MyObject := TMyClass.Create;
```

Diese Anweisung reserviert zunächst Speicher für das neue Objekt auf dem Heap. Anschließend werden alle Felder initialisiert. Ordinalfelder werden mit dem Wert Null, alle Zeiger mit nil und alle Strings mit einem Leerstring initialisiert. Aus diesem Gund brauchen nur die Felder initialisiert zu werden, denen ein bestimmter Anfangswert zugewiesen werden soll. Dann werden alle weiteren Aktionen in der Implementierung des Konstruktors ausgeführt. Am Ende gibt der Konstruktor eine Referenz auf das neu erstellte und initialisierte Objekt zurück. Tritt in einem mit einer Klassenreferenz aufgerufenen Konstruktor eine Exception

auf, wird das unvollständig initialisierte Objekt automatisch durch einen Aufruf des Destruktors Destroy wieder freigegeben. Die Deklaration gleicht einer normalen Prozedurdeklaration, nur steht statt des Schlüsselwortes `procedure` oder `function` das Schlüsselwort `constructor`.

1.3.2. Destruktor

Der Destruktor ist das Gegenstück zum Konstruktor und entfernt das Objekt wieder aus dem Speicher. Die Deklaration gleicht einer normalen Prozedurdeklaration, beginnt aber mit dem Schlüsselwort `destructor`. Ein Destruktor kann nur über ein Instanzobjekt aufgerufen werden. Beispiel:

```
MyObject.Destroy;
```

Beim Aufruf eines Destruktors werden zuerst die in der Implementierung angegebenen Aktionen ausgeführt. Normalerweise werden hier untergeordnete Objekte und zugewiesene Ressourcen freigegeben. Danach wird das Objekt aus dem Speicher entfernt. Da der Destruktor in der Lage sein muss, Objekte freizugeben, die unvollständig erstellt wurden und deshalb `nil` sind, sollte beim Freigeben eines solchen Objektes unbedingt vorher auf `nil` getestet werden. Wird ein Objekt mit der Methode `Free` freigegeben, wird die Prüfung automatisch durchgeführt.

Tipps und Tricks:
Ein häufiger Anfängerfehler ist es den Konstruktor mit einer Variable vom Objekttyp aufzurufen- Solch ein Aufruf endet meist in einer Zugriffsverletzung, da die Objektreferenz meist ungültig ist.

```
var ObRef: TSomething;
begin
  ObjRef.Create;  // falsch
  ObjRef := TSomething.Create;  // richtig
```

- `inherited` sollte im Konstruktor immer aufgerufen werden.
- Ein Objekt sollte immer mit `Free` zerstört werden. Der Destruktor sollte nie direkt aufgerufen werden, denn `Free` prüft, ob das Objekt gültig ist (nicht `nil`) und ruft dann den Destruktor auf. `Free` endet deshalb nicht in einem Fehler, wenn das Objekt nie initialisiert wurde.
- Nach dem Aufruf von `Free` wurde das Objekt zwar aus dem Speicher entfernt, der Zeiger enthält aber immer noch die Adresse des Objektes. Eine Abfrage mit `Assigned` liefert also nach einem Aufruf von `Free` immer noch `True`. Entweder setzt man den Zeiger explizit auf `nil` oder man verwendet die Prozedur `FreeAndNil`.
- Überschreibt man den Destruktor, sollte diese Methode immer als `override` deklariert sein, da es sonst zu Speicherlöchern kommen kann, wenn `Free` zur Freigabe genutzt wird.

Siehe dazu Demo: *einfache Klasse*.

1.4. Eigenschaften

Eine Eigenschaft definiert wie ein Feld ein Objektattribut. Felder sind jedoch nur Speicherbereiche, die gelesen und geändert werden können, während Eigenschaften mit Hilfe bestimmter Aktionen gelesen und geschrieben werden können. Sie erlauben eine größere Kontrolle über den Zugriff auf die Attribute eines Objekts und ermöglichen das Berechnen von Feldern (Attributen).

Die Deklaration einer Eigenschaft muss einen Namen, einen Typ und mindestens eine Zugriffsangabe enthalten. Die Syntax lautet folgendermaßen:

```
property Eigenschaftsname[Indizes]: Typ Index Integer-Konstante Bezeichner;
```

- Eigenschaftsname ist ein gültiger Bezeichner.
- `[Indizes]` ist optional und besteht aus einer Folge von durch Semikola getrennten Parameterdeklarationen. Jede Deklaration hat die Form `Bezeichner1, ..., Bezeichnern: Typ`.
- `Typ` muss ein vordefinierter oder vorher deklarierter Typenbezeichner sein.
- Die Klausel `Index` Integer-Konstante ist optional. Bezeichner ist eine Folge von `read`-, `write`-, `stored`- `default`- und `implements`-Angaben.

Eigenschaften werden durch ihren Zugriffsbezeichner definiert.

1.4.1. Auf Eigenschaften zugreifen

Jede Eigenschaft verfügt über eine read- oder eine write- Angabe oder über beide. Diese Zugriffsbezeichner werden wie folgt angegeben:

```
read FeldOderMethode
write FeldOderMethode
```

`FeldOderMethode` ist der Name eines Feldes oder einer Methode, welche in der Klasse oder in einer Vorfahrenklasse deklariert ist. Beispiel:

```
property Color: TColor read GetColor write SetColor;
```

Die Methode GetColor muss hier folgendermaßen deklariert werden:

```
function GetColor: TColor;
```

Die Deklaration der Methode SetColor muss eine folgende Form haben:

```
procedure SetColor(Value: TColor);
```

Wenn ein Eigenschaft in einem Ausdruck verwendet wird, erfolgt der Zugriff mittels den mit `read` angegebenen Elements (Feld oder Methode). Bei Zuweisungen wird das mit `write` angegebene Element verwendet. Eine Eigenschaft, die nur mit einer `read`-Angabe deklariert ist, nennt man Nur-Lesen-Eigenschaft. Ist nur ein `write`-Bezeichner vorhanden, spricht man

von einer Nur-Schreiben-Eigenschaft. Erfolgt auf eine Nur-Lesen-Eigenschaft ein Schreibzugriff oder auf eine Nur-Schreiben-Eigenschaft ein Lesezugriff, tritt ein Fehler auf.

Siehe dazu Demo: *einfache Klasse*.

2. Vererbung, Polymorphie und abstrakte Klassen

Was ist nun Vererbung und Polymorphie?

Bei der Vererbung geht es darum, dass man eine Klasse von einer anderen ableiten kann. Somit erbt der Nachfolger alles vom Vorfahren. Das bringt uns aber noch nicht viel weiter. Der Gag bei der Vererbung ist, dass man jetzt der neuen Klasse neue Methoden/Properties hinzufügen und somit die alte Klasse erweitern kann. Das wäre der erste Schritt, die Vererbung. Man kann somit vorhandenen Code als Grundlage nehmen und muss nicht alles noch mal neu erfinden. Dies spielt bei der Komponentenentwicklung eine wichtige Rolle. Es wäre Unsinn für eine neue Komponente alles noch mal neu zu implementieren. Stattdessen nimmt man sich eine bestehende und erweitert sie entsprechend.

Der nächste Schritt wäre jetzt die Polymorphie. Sie bietet die Möglichkeit, geerbte Methoden zu überschreiben und zu verändern. Man kann die neue Klasse so den Erfordernissen anpassen. Dies setzt allerdings voraus, dass die Ursprungsklasse dies zulässt. Das heißt, Methoden müssen mit dem Schlüsselwort `virtual` zumindest im `protected`-Abschnitt der Ursprungsklasse deklariert sein. So kann man sie in der neuen Klasse mit `override` überschreiben.

Abstrakte Klassen führen dieses Konzept noch weiter. Eine abstrakte Klasse ist eine Klasse, die lediglich die Deklaration der Methoden und Felder enthält, sie aber nicht implementiert. Von dieser abstrakten Klasse kann man nun Klassen ableiten, die die schon bereitgestellten Methoden jede auf ihre Weise implementieren. Nehmen wir mal ein Beispiel aus dem wirklichen Leben. Gegeben sei eine Klasse `TNahrung`. Welche die Methode/Eigenschaft Geschmack bereitstellt. Diese zu implementieren wäre nicht sehr sinnvoll. Denn wie schmeckt Nahrung? Nahrung hat keinen Geschmack. Sehr wohl können aber davon abgeleitete Klassen einen Geschmack haben. Also deklariert man die Methode/Eigenschaft Geschmack der Klasse `TNahrung` abstrakt und überlässt die Implementierung den Nachfahren. Man könnte nun von `TNahrung` eine Klasse `TGemüse` ableiten, welche auch noch nicht die Methode/Eigenschaft Geschmack implementiert. Erst ein weiterer Nachfahre, zum Beispiel `TSpinat`, könnte sie dann entsprechend implementieren. Man hat also somit die Möglichkeit sich eine übersichtliche, strukturierte Klassenhierarchie aufzubauen, die je nach Belieben erweiterbar und anpassbar ist. Diese Techniken erfordern allerdings eine erweiterte Methodendeklaration und -implementierung wie die, die wir bereits kennen gelernt haben. Und darum geht es in diesem Kapitel.

2.1. Erweiterte Methodendeklaration und -implementierung

Wie schon erwähnt, ist eine Methode eine Prozedur oder Funktion, die zu einer bestimmten Klasse gehört. Daher wird auch beim Aufruf einer Methode das Objekt angegeben, mit dem die Operation ausgeführt werden soll. Wie die Deklaration und die Implementierung im Einzelnen aussieht, habe ich weiter oben schon erklärt.

Jetzt können in einer Methodendeklaration noch spezielle Direktiven enthalten sein, die in anderen Funktionen/Prozeduren nicht verwendet werden. Diese Direktiven müssen in der Klassendeklaration enthalten sein und in der folgenden Reihenfolge angegeben werden: reintroduce; overload; Bindung; Aufrufkonvention; abstract; Warnung Hierbei gilt: Bindung ist `virtual`, `dynamic` oder `override`. Aufrufkonvention ist `register`, `pascal`, `cdecl`, `stdcall` oder `safecall`. Warnung ist `platform`, `depracted`, oder `library`.

2.1.1. inherited

Das Schlüsselwort `inherited` ist für die Polymorphie von großer Bedeutung.

Folgt auf `inherited` der Name eines Elements, entspricht dies einem normalen Methodenaufruf bzw. einer Referenz auf eine Eigenschaft oder ein Feld. Der einzige Unterschied besteht darin, dass die Suche nach dem referenzierten Element bei dem direkten Vorfahren der Klasse beginnt, zu der die Methode gehört.

Die Anweisung `inherited` ohne Bezeichner verweist auf die geerbte Methode mit dem selben Namen wie die aufrufende Methode. Handelt es sich dabei um eine Botschaftsbehandlung, verweist diese auf die geerbte Botschaftsbehandlung für die selbe Botschaft.

Tipps und Tricks:

- Die meisten Konstruktoren rufen inherited am Anfang und die meisten Destruktoren am Ende auf.
- Wird inherited alleine benutzt und hat der Vorfahre keine Methode mit dem selben Namen, ignoriert Delphi den Aufruf von inherited.

2.1.2. Self

In jeder Methode deklariert Delphi die Variable `Self` als versteckten Parameter. Der Bezeichner `Self` verweist in der Implementierung einer Methode auf das Objekt, in dem die Methode aufgerufen wird.

2.2. Methodenbinding

Methodenbindungen können statisch (Standard), virtuell oder dynamisch sein. Virtuelle und dynamische Methoden können überschrieben werden.

2.2.1. Statische Methoden

Methoden sind standardmäßig statisch. Beim Aufruf bestimmt der zur Compilierzeit festgelegte Typ, der im Aufruf verwendeten Klassenvariablen, welche Implementierung verwendet wird. am deutlichsten wird dies an Hand eines Beispiels:

```
type
  TFigure = class(TObject)
    procedure Draw(Caption: String);
  end;

  TRectangle = class(TFigure)
    procedure Draw(Caption: String);
  end;
```

Dies sind also die Klassen. Wobei `TRectangle` von `TFigure` abgeleitet ist. Beide Klassen stellen die Methode `Draw` zur Verfügung. Was passiert nun bei den Aufrufen, wenn sie wie folgt aussehen:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  try
    Figure.Draw; // Ruft TFigure.Draw auf
  finally
    FreeAndNil(Figure);
  end;

  Figure := TRectangle.Create;
  try
    Figure.Draw; // Ruft TFigure.Draw auf
  finally
    FreeAndNil(Figure);
  end;

  Figure := TFigure.Create;
  try
    TRectangle(Figure).Draw; // Ruft TRectangle.Draw auf
  finally
    FreeAndNil(Figure);
  end;

  Rectangle := TRectangle.Create;
  try
    Rectangle.Draw;  // Ruft TRectangle.Draw auf
  finally
    FreeAndNil(Rectangle);
  end;
end;
```

Im ersten Aufruf passiert nichts ungewöhnliches. Wie erwartet wird die Methode `Draw` von `TFigure` aufgerufen. Im zweiten Aufruf jedoch wird auch die Methode `Draw` von `TFigure`

aufgerufen, obwohl die Variable `Figure` ein Objekt der Klasse `TRectangle` referenziert. Es wird jedoch die `Draw`-Implementation in `TFigure` aufgerufen, weil `Figure` als `TFigure` deklariert ist.

Siehe dazu Demo: *einfache Vererbung*.
Siehe dazu Demo: *statische Methoden*.

2.2.2. Virtuelle und dynamische Methoden

Mit Hilfe der Direktiven virtual und dynamic können Methoden als virtuell oder dynamisch deklariert werden. Diese können im Gegensatz zu statischen in abgeleiteten Klassen überschrieben werden. Beim Aufruf bestimmt nicht der deklarierte, sondern der aktuelle Typ (also der zur Laufzeit) der im Aufruf verwendeten Klassenvariable welche Implementierung aktiviert wird.

Um eine Methode zu überschreiben, braucht sie nur mit der Direktive `override` erneut deklariert zu werden. Dabei müssen Reihenfolge und und Typ der Parameter sowie der Typ des Rückgabewertes mit der Deklaration in der Vorfahrenklasse übereinstimmen. Die Direktive `override` sagt dem Compiler außerdem, dass man die geerbte Methode willentlich überschreibt und stellt sicher, dass man die Methode des Vorfahren nicht überdeckt.

```
type
  TMyClass = class(TObject)
  public
    Destructor Destroy;
  end;

var
  Form1: TForm1;

implementation

{\$R *.dfm}

Destructor TMyClass.Destroy;
begin

end;
```

In obigen Beispiel warnt einem der Compiler davor, dass die Methode Destroy von der Vorfahren Klasse von der eigenen überdeckt wird:

```
[Warnung] Unit1.pas(20): Methode 'Destroy' verbirgt virtuelle Methode vom
    Basistyp 'TObject'.
```

Deklariert man nun die eigenen Methode mit `override`, sagt man dem Compiler, dass man die Methode des Vorfahren überschreiben will, aber nicht verdecken will.

```
type
  TFruit = class
  public
    function GetTaste: string; virtual; abstract;
```

```
end;

TCitrusFruit = class(TFruit)
public
  procedure Squeeze; virtual; abstract;
end;

TLemon = class(TCitrusFruit)
public
  function GetTaste: string; override;
  procedure Squeeze; override;
end;

TGrapefruit = class(TCitrusFruit)
public
  function GetTaste: string; override;
  procedure Squeeze; override;
end;

TBanana = class(TFruit)
public
  function GetTaste: string; override;
end;
```

Ausgehend von diesen Deklarationen zeigt der folgende Programmcode, wie sich der Aufruf einer virtuellen Methode durch eine Variable auswirkt, deren aktueller Typ erst zur Laufzeit festgelegt wird.

```
var
  MyFruit: TFruit;
  idx: Integer;
begin
  idx := RadioGroup1.ItemIndex;
  case idx of
    0: MyFruit := TLemon.Create;
    1: MyFruit := TGrapefruit.Create;
    2: MyFruit := TBanana.Create;
  end;
  if Assigned(MyFruit) then
  begin
    try
      if MyFruit is TCitrusFruit then
        (MyFruit as TCitrusFruit).Squeeze
      else
        ShowMessage('Keine Zitrusfrucht.');
    finally
      FreeAndNil(MyFruit);
    end;
  end;
```

Nur virtuelle und dynamische Methoden können überschrieben werden. Alle Methoden können jedoch überladen werden.

Virtuelle und dynamische Methoden sind von der Semantik her identisch. Sie unterscheiden sich nur bei der Implementierung der Aufrufverteilung zur Laufzeit. Virtuelle Methoden werden auf Geschwindigkeit, dynamische auf Code-Größe optimiert.

Tipps und Tricks:

- Mit virtuellen Methoden kann polymorphes Verhalten am besten implementiert werden.
- Dynamische Methoden sind hilfreich, wenn in einer Basisklasse eine große Anzahl überschreibbarer Methoden deklariert sind, die von vielen Klassen geerbt, aber nur selten überschrieben werden.
- Dynamische Methoden sollten nur verwendet werden, wenn sich dadurch ein nachweisbarer Nutzen ergibt. allgemein sollte man virtuelle Methoden verwenden.

Siehe dazu Demo: *Polymorphie abstrakte Klassen*.

Unterschiede zwischen Überschreiben und Verdecken
Wenn in einer Methodendeklaration dieselben Bezeichner- und Parameterangaben wie bei der geerbten Methode ohne die Anweisung override angegeben werden, wird die geerbte Methode durch die neue Verdeckt. Beide Methoden sind jedoch in der abgeleiteten Klasse vorhanden, in der die Methode statisch gebunden wird.

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

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



procedure TForm1.Button1Click(Sender: TObject);
var
  Foo: T1;  // T2
begin
  Foo := T2.Create;
  try
    Foo.Act;  // Ruft T1.Act auf
  finally
    FreeAndNil(Foo);
  end;
end;
```

Er setzt man bei der Variablendeklaration `T1` durch `T2` wird auch die Methode `Act` von der Klasse `T2` aufgerufen.

reintroduce
Mit Hilfe der Anweisung reintroduce kann verhindert werden, dass der Compiler Warnungen dieser Art:

```
[Warnung] Unit1.pas(37): Methode 'Act' verbirgt virtuelle Methode vom Basistyp '
    T1'
```

ausgibt, wenn eine zuvor deklarierte Methode verdeckt wird. `Reintroduce` sollte also dann verwendet werden, wenn eine geerbte virtuelle Methode durch eine neue Deklaration verdeckt werden soll.

Tipps und Tricks:

- `reintroduce` muss immer die erste Direktive sein, wenn sie verwendet wird.

Abstrakte Methoden
Eine abstrakte Methode ist eine virtuelle oder dynamische Methode, die nicht in der Klasse implementiert wird, in der sie deklariert ist. Die Implementierung wird erst spätere in einer abgeleiteten Klasse durchgeführt. Bei der Deklaration abstrakter Methoden muss die Anweisung `abstract` nach `virtual` oder `dynamic` angegeben werden. Sie dazu auch das Beispiel auf Seite 4. Eine abstrakte Methode kann nur in einer Klasse (bzw. Instanz einer Klasse) aufgerufen werden, in der sie überschreiben wurde.

Siehe dazu Demo: *Reintroduce*.

2.3. Methoden überladen

Eine Methode kann auch mit der Direktive overload deklariert werden. Damit ermöglicht man es einer abgeleiteten Klasse eine Methode zu überschreiben und die Datentypen der Parameter zu ändern. Wenn sich die Parameterangaben von denen ihres Vorfahren unterscheiden, wird die geerbte Methode überladen, ohne dass sie dadurch verdeckt wird. Bei einem Aufruf der Methode in einer abgeleiteten Klasse wird dann diejenige aufgerufen, bei der die Parameter übereinstimmen.

Beispiel:

```
Definition und Implementierung der Klassen:

type
  T1 = class(TObject)
    procedure Test(i: Integer); overload; virtual;
  end;

  T2 = class(T1)
    procedure Test(s: String); reintroduce; overload;
  end;

procedure T1.Test(i: Integer);
begin
  ShowMessage(IntToStr(i));
end;

procedure T2.Test(s: String);
begin
  ShowMessage(s);
end;
```

Und Aufruf im Programm:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  T: T2;
begin
  T := T2.Create;
  try
    T.Test(5);
  finally
    FreeAndNil(T);
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  T: T2;
begin
    T := T2.Create;
  try
    T.Test('Hello');
  finally
    FreeAndNil(T);
  end;
end;
```

Tipps und Tricks:

- Verwenden Sie beim Überladen einer virtuellen Methode die Direktive `reintroduce`, wenn die Methode in einer abgeleiteten Klasse neu deklariert wird.
- Methoden, die als read- oder write-Bezeichner für Eigenschaften fungieren, können nicht überladen werden.
- Die `overload` Direktive muss vor den Direktiven `virtual`, `dynamic` oder `abstract` stehen.
- Man kann auch Methoden überladen, die in der Basisklasse nicht mit overload deklariert wurden.
- Der Compiler benutzt den Typ und die Anzahl der Parameter, um zu entscheiden welche überladenen Methode aufgerufen werden soll .Um zwischen verschiedenen Integer-Typen zu unterscheiden wird der Typ genommen, der am ehesten dem angegebenen Typen entspricht.

3. Klassenreferenzen

Jede Klasse erbt von `TObject` die Methoden `ClassType` und `ClassParent`, mit denen die Klasse eines Objektes und seines direkten Vorfahren ermittelt werden kann. Beide Methoden geben einen Wert vom Typ `TClass (TClass = class of TObject)` zurück, der in einen spezielleren Typ umgewandelt werden kann. Außerdem erben alle Klassen die Methode `InheritsFrom`, mit der ermittelt werden kann, ob ein Objekt von einer bestimmten Klasse abgeleitet ist. Diese Methoden werden von den Operatoren `is` und `as` verwendet und normalerweise nicht direkt aufgerufen.

Siehe dazu Demo: *Klassenreferenzen*.

3.1. Klassenoperatoren

3.1.1. Der Operator is

Der Operator `is` führt eine dynamische Typprüfung durch. Mit ihm kann der der aktuelle Laufzeittyp eines Objektes ermittelt werden.

```
Object is Klasse
```

Dieser Ausdruck gib `True` zurück, wenn ein Objekt eine Instanz der angegebenen Klasse oder eines ihrer Nachkommen ist. Anderen Falls ist der Rückgabewert `False`. Er ist auch `False`, wenn das Objekt den Wert `nil` hat.

Tipps und Tricks:

- Der is Operator ruft die Methode InheritsFrom der Klasse auf.
- Wurde der Objekttyp schon mit is geprüft, ist es überflüssig den Operator as zu benutzen. Ein einfacher Typecast reicht aus und erhöht die Performance.

3.1.2. Der Operator as

Der Operator `as` führt eine Typenumwandlung mit Laufzeitprüfung durch.

```
Object as Klasse
```

Dieser Ausdruck gibt eine Referenz auf das selbe Objekt wie `Object`, aber mit dem von Klasse angegebenen Typ zurück. Zur Laufzeit muss `Object` eine Instanz von Klasse sein oder einem ihrer Nachkommen. Andernfalls wird eine Exception ausgelöst. Ist der Typ von `Object` nicht mit der Klasse verwandt, gibt der Compiler eine Fehlermeldung aus.

Die Regeln der Auswertungsreihenfolge machen es häufig erforderlich, `as`-Typenumwandlungen in Klammern zu setzten.

```
(MyFruit as TcitrusFruit).Squeeze
```

Siehe dazu Demo: *Polymorphie abstrakte Klasse*.

3.2. Klassenmethoden

Klassenmethoden sind Methoden, die nicht mit Objekten, sondern mit Klassen arbeiten, es sind also statische Funktionen die zu einer Klasse gehören. Das heißt man muss, um sie benutzen zu können keine Instanz der Klasse erzeugen. Der Vorteil von Klassenmethoden ist, dass sie Zugriff auf statische Methoden der Klasse haben, wie den Klassentyp oder den Vorfahren der Klasse. Allerdings haben sie auch Einschränkungen. Mit ihnen kann man nicht auf auf `Self` der Klasse zugreifen.

Der Sinn liegt nun darin, Schnittstellen, Routinen und Klasse, die von dieser verwendet wird zu einer logischen Einheit zusammenfassen. Die Zugehörigkeit zur Klasse macht den Kontext der Funktion deutlich, erfordert aber, wie schon gesagt keine Instanziierung von außen. Sie dient lediglich zur Kapselung und Strukturierung des Codes. Anders gesagt, dient die Klasse lediglich als Container.

Die Definition einer Klassenmethode erfolgt mit dem einleitenden Schlüsselwort `class`:

```
class function GetUserInput(out Data: TSomeData): Boolean;
```

Auch die definierende Deklaration einer Klassemethode muss mit class eingeleitet werden:

```
class function TForm2.GetUserInput(out Data: TSomeData): Boolean;
begin
  with TForm2.Create(nil) do
  begin
    ShowModal();
    Data.FData := Edit1.Text;
    Result := length(Edit1.Text) <> 0;
    Free();
  end;
end;
```

siehe dazu Demo: *Klassenmethoden*.

A. Demos

A.1. Einfache Klasse

```
uses
  TTemp;

procedure TForm1.btnFahrenheitClick(Sender: TObject);
var
  Temp: TTemperatur;
begin
  Temp := TTemperatur.create(StrToFloat(edtTemp.Text));
  try
    stcErgebniss.Caption := FloatToStrF(Temp.Fahrenheit, ffNumber, 8, 2);
  finally
    FreeAndNil(Temp);
  end;
end;

procedure TForm1.bntCelsiusClick(Sender: TObject);
var
  Temp: TTemperatur;
begin
  Temp := TTemperatur.create(StrToFloat(edtTemp.Text));
  try
    stcErgebniss.Caption := FloatToStrF(Temp.Celsius, ffNumber, 8, 2);
  finally
    FreeAndNil(Temp);
  end;
end;
```

```
unit TTemp;

interface

type TTemperatur = class
  private
    FTemperatur : double;
    function GetTemp: double;
    procedure SetTemp(Value: double);
  protected
    function C2F: double; virtual;
    function F2C: double; virtual;
  public
    constructor create(Value: double);
    property Value: double read GetTemp write SetTemp;
    property Celsius: double read F2C;
    property Fahrenheit: double read C2F;
```

```
  end;

implementation

constructor TTemperatur.create(Value: double);
begin
  inherited Create;
  FTemperatur := Value;
end;

procedure TTemperatur.SetTemp(Value: double);
begin
  FTemperatur := Value;
end;

function TTemperatur.GetTemp: Double;
begin
  result := FTemperatur;
end;

function TTemperatur.C2F: double;
begin
  result := 9.0/5.0 * FTemperatur + 32.0;
end;

function TTemperatur.F2C: double;
begin
  result := 5.0/9.0*(FTemperatur - 32.0);
end;

end.
```

A.2. Einfache Vererbung

```
uses
  Family;

procedure TForm1.btnParentsClick(Sender: TObject);
var
  Parents: TParent;
begin
  Parents := TParent.Create;
  try
    ShowMessage(Parents.GetType + #13#10 + Parents.GetFamilyName + #13#10 +
      IntToStr(Parents.GetChildrenCount));
  finally
    FreeAndNil(Parents);
  end;
end;

procedure TForm1.btnSonClick(Sender: TObject);
var
  Son: TSon;
begin
```

```
  Son := TSon.Create;
  try
    ShowMessage(Son.GetType + #13#10 + Son.GetFamilyName + ', ' +
      Son.GetChristianName + #13#10 + IntToStr(Son.GetChildrenCount));
  finally
    FreeAndNil(Son);
  end;
end;
```

```
unit Family;

interface

type
  TParent = class(TObject)
  private
    FFamilyName: String;
    FChildrenCount: Cardinal;
  public
    constructor Create;
    function GetType: String;
    function GetFamilyName: String;
    function GetChildrenCount: Cardinal;
  end;

type
  TSon = class(TParent)
  public
    constructor Create;
    function GetType: String;
    function GetChristianName: String;  // Klasse TParent erweitert
  end;

implementation

constructor TParent.Create;
begin
  FFamilyName := 'Müller';
  FChildrenCount := 1;
end;

function TParent.GetType: String;
begin
  result := 'Eltern';
end;

function TParent.GetFamilyName: String;
begin
  result := FFamilyName;
end;

function TParent.GetChildrenCount: Cardinal;
begin
  result := FChildrenCount;
end;

constructor TSon.Create;
```

```
begin
  inherited;
  FChildrenCount := 0;
end;

function TSon.GetType: String;
begin
  result := 'Sohn';
end;

function TSon.GetChristianName: String;
begin
  result := 'Otto';
end;

end.
```

A.3. Statische Methoden

```
type
  TFigure = class(TObject)
    procedure Draw;
  end;

  TRectangle = class(TFigure)
    procedure Draw;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TFigure.Draw;
begin
  Messagebox(0, 'Draw Methode von TFigure', 'Draw', 0);
end;

procedure TRectangle.Draw;
begin
  Messagebox(0, 'Draw Methode von TRectangle', 'Draw', 0);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  try
    Figure.Draw; // Ruft TFigure.Draw auf
  finally
```

```
    FreeAndNil(Figure);
  end;

  Figure := TRectangle.Create;
  try
    Figure.Draw; // Ruft TFigure.Draw auf
  finally
    FreeAndNil(Figure);
  end;

  Figure := TFigure.Create;
  try
    TRectangle(Figure).Draw; // Ruft TRectangle.Draw auf
  finally
    FreeAndNil(Figure);
  end;

  Rectangle := TRectangle.Create;
  try
    Rectangle.Draw;  // Ruft TRectangle.Draw auf
  finally
    FreeAndNil(Rectangle);
  end;
end;
```

A.4. Polymorphie

```
uses
  Fruits;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyFruit: TFruit;
  idx: Integer;
begin
  idx := RadioGroup1.ItemIndex;
  case idx of
    0: MyFruit := TLemon.Create;
    1: MyFruit := TGrapefruit.Create;
    2: MyFruit := TBanana.Create;
  end;
  if Assigned(MyFruit) then
  begin
    try
      if MyFruit is TCitrusFruit then
        (MyFruit as TCitrusFruit).Squeeze
      else
        ShowMessage('Keine Zitrusfrucht.');
    finally
      FreeAndNil(MyFruit);
    end;
  end;
end;
```

```
unit Fruits;

interface

uses
  Dialogs;

type
  TFruit = class //pure abstract class
  public
    function GetTaste: string; virtual; abstract;
  end;

  TCitrusFruit = class(TFruit) //pure abstract class introduces new ops
  public
    procedure Squeeze; virtual; abstract;
  end;

  TLemon = class(TCitrusFruit)  // same taste as citrus fruit
  public
    function GetTaste: string; override;  // acerbic
    procedure Squeeze; override;
  end;

  TGrapefruit = class(TCitrusFruit)
  public
    function GetTaste: string; override;  // disgustful ;)
    procedure Squeeze; override;
  end;

  TBanana = class(TFruit)
  public
    function GetTaste: string; override;  // sugary
  end;

implementation

function TLemon.GetTaste: String;
begin
  result := 'sauer';
end;

procedure TLemon.Squeeze;
begin
  ShowMessage('Von Hand ausdrücken.');
end;

function TGrapefruit.GetTaste: String;
begin
  result := 'süß';
end;

procedure TGrapefruit.Squeeze;
begin
  ShowMessage('Mit der Maschine ausdrücken.');
end;
```

```
function TBanana.GetTaste: String;
begin
  result := 'fruchtig';
end;

end.
```

A.5. Reintroduce

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

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

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure T1.Act;
begin
  ShowMessage('Act Methode von T1');
end;

procedure T2.Act;
begin
  ShowMessage('Act Methode von T2');
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Foo: T1;  // T2
begin
  Foo := T2.Create;
  try
    Foo.Act;  // Ruft T1.Act auf
  finally
    FreeAndNil(Foo);
  end;
end;
```

A.6. Klassenreferenzen

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ClassRef: TClass;
begin
  ListBox1.Clear;
  ClassRef := Sender.ClassType;
  while ClassRef <> nil do
  begin
    ListBox1.Items.Add(ClassRef.ClassName);
    ClassRef := ClassRef.ClassParent;
  end;
end;
```

A.7. Klassenmethoden

```
uses
  Unit2;

procedure TForm1.Button1Click(Sender: TObject);
var
  Data: TSomeData;
begin
  if TForm2.GetUserInput(Data) then
    Label2.Caption := Data.FData;
end;
```

```
unit Unit2;

interface

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

type
  TSomeData = record
    FData: String[255]
  end;

type
  TForm2 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
    class function GetUserInput(out Data: TSomeData): Boolean;
  end;
```

```
var
  Form2: TForm2;

implementation

{$R *.dfm}

class function TForm2.GetUserInput(out Data: TSomeData): Boolean;
begin
  with TForm2.Create(nil) do
  begin
    ShowModal();
    Data.FData := Edit1.Text;
    Result := length(Edit1.Text) <> 0;
    Free();
  end;
end;

procedure TForm2.Button1Click(Sender: TObject);
begin
  ModalResult := mrOK;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  ModalResult := mrCancel;
end;

end.
```

B. Anhang

B.1. Vorgänge beim Aufruf des Konstruktors

Anders als in C++ übernimmt der Konstruktor in Delphi neben der Initialisierung der Klasse auch die Aufgabe der Speicherreservierung. Der Vorfahrkonstruktor steht nun vor dem Problem, dass er wissen muss, dass die Speicherreservierung bereits durchgeführt wurde und somit kein weiterer Speicher zu reservieren ist. In Delphi wurde dies gelöst indem der Konstruktor eine „Hybrid-Methode“ ist. Im Gegensatz zu normalen Methoden generiert der Compiler Code, der dem Konstruktor zwei versteckte Parameter übergibt. Der Typ des ersten Parameters „Self“ hängt dabei vom Wert des Zweiten ab. Dieser wird im CPU Register `DL` übergeben und gibt den Aufrufmodus des Konstruktors an. Delphi kennt drei unterschiedliche Aufrufmodi für Konstruktoren.

Der erste Aufrufmodus ist **ClassCreate with Allocation** und wird beim Erzeugen einer Instanz der Klasse genutzt.

```
Reference := MyClass.Create;
```

Der Aufrufmodus-Parameter hat in diesem Fall den Wert 1. Dies führt dazu, dass als erstes die System-Funktion `_ClassCreate`, mit dem ClassType Self-Parameter, vom Konstruktor aus aufgerufen wird. Diese ruft ihrerseits die Klasssenmethode `NewInstance` auf, die den notwendigen Speicher reserviert und die Methode `InitInstance` aufruft. Diese wiederum initialisiert alle Felder des Objekts mit deren Null-Wert und baut die „Interface Table“ auf. Zudem setzt sie den Zeiger auf die VMT (Virtual Method Table) in den dafür vorgesehenen Speicherbereich der Instanz ein. `_ClassCreate` richtet des Weiteren einen Exception-Block ein, der bei einer Exception innerhalb des Konstruktor-Codes automatisch den Destruktor aufgeruft. Dieser Exception-Block endet mit dem verlassen des äußersten Konstruktors. Als Rückgabewert liefert `_ClassCreate` den mit `NewInstance` erzeugten Instanz-Zeiger, auch bekannt als „Self“, der nun im Konstruktor wie bei einer normalen Methode Verwendung findet.

Der zweite Aufrufmodus ist **Inherited Call**. Er dient dazu, den Vorfahrkonstruktor aufzurufen.

```
inherited Create;
```

Dieser darf natürlich keinen neuen Speicherplatz reservieren und auch die Klassen nicht neu initialisieren. Ein weiterer Exception-Block ist ebenfalls nicht notwendig. Damit bei dem `inherited` Aufruf dies alles nicht geschieht, setzt der Compiler den Aufrufmodus-Parameter

auf den Wert 0, was den Vorfahrkonstruktor veranlasst, den `_ClassCreate` Aufruf zu übergehen. Der Self-Parameter des Konstruktors ist in diesem Modus ein ganz gewöhnlicher Instanz-Zeiger, da das Objekt ja bereits erzeugt ist.

Der dritte Aufrufmodus ist **Initialization without Allocation**, bei dem zwar ein Exception-Block eingerichtet wird, aber keine Speicherreservierung stattfindet. Dieser Modus dient dem Aufruf des Konstruktors als gewöhnliche Methode.

```
Variable.Create;
```

Hierbei enthält der Aufrufmodus-Parameter den Wert -1 und „Self“ ist ein Instanz-Zeiger. Dieser Aufrufmodus wird sehr selten eingesetzt, da es normalerweise nicht notwendig ist, den Konstruktor als reine Initialisierungsmethode zu verwenden, weil er bereits beim Erzeugen der Instanz ausgeführt wurde. Sie ist auf das TurboPascal `object` zurückzuführen, bei dem man, wie unter C++, den Speicher für dynamische Instanzen selbst reservieren musste. Zudem birgt diese Art des Aufrufs die Gefahr von Speicherlecks, die dadurch entstehen, dass bereits reservierter Speicher und Referenzen auf Objekte überschrieben und somit nicht mehr freigegeben werden können.

Bei allen drei Aufrufmodi wird nach dem Ausführen des Konstruktor-Codes die virtuelle Methode `AfterConstruction` ausgeführt. Da dem Konstruktor zwei versteckte Parameter übergeben werden, welche die CPU-Register `EAX` (ClassType bzw. Self) und `DH` (Aufrufmodus) belegen, bleibt nur noch das `ECX` Register für einen dritten Parameter übrig. Der Rest muss über den langsameren Stack übergeben werden.

B.2. Vorgänge beim Aufruf des Destruktors

Wie beim Konstruktor unterscheidet sich Delphi auch hier von C++. Bei Delphi kümmert sich der Destruktor um die Speicherfreigabe. Damit es möglich ist den Vorfahrdestruktor aufzurufen, wird hier, wie beim Konstruktor, ein zweiter versteckter Aufrufmodus-Parameter im `DH` CPU-Register übergeben. Der Destruktor kennt im Vergleich zum Konstruktor jedoch nur zwei Aufrufmodi.

Der erste Modus ist **Destruction with Deallocation**, bei dem `DH` den Wert 1 enthält. Dieser wird beim direkten Aufruf des Destruktors benutzt und führt zum Aufruf von `BeforeDestruction` gefolgt von einem `_ClassDestroy`. Dieses gibt das Objekt mittels `FreeInstance` frei. `FreeInstance` seinerseits ruft vor dem Freigeben die Methode `CleanupInstance` auf, um alle referenzgezählten Felder (String, dyn. Array, Interface) ordnungsgemäß aufzuräumen.

Der zweite Aufrufmodus ist **Inherited Destruction**, bei dem `DH` den Wert 0 enthält. Er findet beim Aufruf des Vorfahrdestruktor mittels `inherited` Verwendung. Hierbei wird der `_ClassDestroy` Aufruf übersprungen und somit wird das Objekt nicht vorzeitig freigegeben. Wie oben bereits erwähnt, wird der Destruktor automatisch aufgerufen, wenn im Konstruktor eine Exception auftritt. Aus diesem Grund sollte man beim Erzeugen einer Instanz den try/finally-Block wie folgt schreiben.

```
Reference := TMyClass.Create;
try
...
finally
  Reference.Free;
end;
```

Die Schreibweise, bei der das `try` vor dem Konstruktor-Aufruf steht, führt bei einer Exception im Konstruktor zu einer zusätzlichen Schutzverletzung, da `Reference.Free` dann auch aufgerufen wird, wenn in `Reference` ein undefinierter Wert steht.

B.3. Funktionsweise von Methodenzeigern

Was ist ein Methodenzeiger? Genau genommen ist ein Methodenzeiger eine Referenz auf eine Methode einer Instanz einer Klasse. Das klingt jetzt erstmal kompliziert. Es handelt sich hierbei aber um das einfache `procedure() of object`. Ein solcher Methodezeiger kann jede Methode eines jeden Objekts aufnehmen, die dieselbe Signatur (Rückgabewert, Parameter) besitzt.
Doch wie schafft es Delphi, dass die Methode für die passende Instanz aufgerufen wird? In einem Zeiger ist ja nur Platz für die Adresse der Methode oder des Objekts, nicht aber für beide. Delphi geht hier wieder einmal sehr trickreich vor. In Wirklichkeit ist ein Methodenzeiger gar kein Zeiger, sondern ein Record vom Typ `TMethod`. Dieser Record enthält zwei Felder. `TMethod.Code` enthält die Adresse der Methode und `TMethod.Data` einen Zeiger auf das Objekt, an das die Methode gebunden ist. Beim Aufruf eines Methodenzeigers übergibt Delphi die in `TMethod.Data` liegende Referenz als ersten versteckten Self-Parameter an die in `TMethode.Code` verankerte Methode, wodurch die Methode für das richtige Objekt aufrufen wird.

Mit diesem Wissen kann man Delphi nun austricksen und die Methodenzeiger für andere Dinge einsetzen. Zum Beispiel ist es möglich eine gewöhnliche Prozedur als Methode zu missbrauchen. Die Prozedur muss in diesem Fall dieselbe Signatur wie der Methodenzeiger haben. Zudem muss aber noch der versteckte Self-Parameter als erster Parameter eintragen sein. Danach ist es nur noch eine Sache des richtigen Typecasts, der die Prozedur in einen Methodenzeiger packt.

```
procedure SimulatedMethod(Self: TForm1; Sender: TObject);
begin
  Self.Caption := 'Du hast mich ausgerufen';
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  Event: TNotifyEvent;
begin
  TMethod(Event).Code := @SimulatedMethod;
  TMethod(Event).Data := Self;
  Button1.OnClick := Event;
end;
```

Mit dieser Technik kann man auch die Methode bzw. das Objekt austauschen.

B.4. Windows-API Callback-Funktion als Methode einer Klasse

Was kann man damit nun anfangen? Nehmen wir mal an, dass man sich eine schöne Klasse entworfen hat, in der man eine Windows-API Funktion benutzen will, die aber eine Callback Funktion benötigt. Zum Beispiel `EnumWindows` zum Enumerieren der Fenster. Das Problem ist nun, dass wir diese API-Funktion nicht einfach eine Methode unserer Klasse übergeben können, da ein Funktionszeiger und ein Methodenzeiger inkompatibel zu einander sind.

Wie wir oben gesehen haben, ist ein Methodenzeiger nichts anderes ist als ein Zeiger auf ein Record mit zwei Feldern. Ein Feld mit einem Zeiger für den Code und ein Feld mit einem Zeiger auf die Instanz. Wird eine Methode nun aufgerufen, wird sie ganz normal mit Hilfe des ersten Zeigers aufgerufen. Der zweite Zeiger wird als „unsichtbarer“ Parameter in die Parameterliste der Methode eingefügt. Dies ist der sogenannte Self-Parameter, über den man auf die Instanz der Klasse zugreifen kann.

Um nun eine Methode für eine bestimmte Instanz über einen nnormalenFFunktionszeiger aufrufen zu können, kann man folgendes tun: Man legt einen ausführbaren Speicherbereich an und schreibt in diesen die beiden Zeiger, den Aufruf der Methode und die Return-Anweisung. Den Zeiger auf diesen Speicherbereich kann man nun als Funktionszeiger der Callback-Funktion übergeben.

```
function TEnumWindows.MakeProcInstance(M: TMethod): Pointer;
begin
  // Ausführbaren Speicher alloziieren fü 15 Byte an Code
  Result := VirtualAlloc(nil, 15, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  asm
    // MOV ECX,
    MOV BYTE PTR [EAX], $B9
    MOV ECX, M.Data
    MOV DWORD PTR [EAX+$1], ECX
    // POP EDX (bisherige Rücksprungadresse nach edx)
    MOV BYTE PTR [EAX+$5], $5A
    // PUSH ECX (self als Parameter 0 anfügen)
    MOV BYTE PTR [EAX+$6], $51
    // PUSH EDX (Rücksprungadresse zurück auf den Stack)
    MOV BYTE PTR [EAX+$7], $52
    // MOV ECX, (Adresse nach ecx laden)
    MOV BYTE PTR [EAX+$8], $B9
    MOV ECX, M.Code
    MOV DWORD PTR [EAX+$9], ECX
    // JMP ECX (Sprung an den ersten abgelegten Befehl und Methode aufrufen)
    MOV BYTE PTR [EAX+$D], $FF
    MOV BYTE PTR [EAX+$E], $E1
    // hier kein Call, ansonsten würde noch eine Rücksprungadresse auf den Stack
       gelegt
  end;
end;
```

Die Methode MakeProcInstance schreibt eine neue Funktion in den Speicher, die vorher im Quelltext noch nicht vorhanden war. Aufgabe dieser Funktion ist es den Self-Parameter mit auf den Stack zu legen, so dass er als zusätzlicher „unsichtbarer“ Parameter an die Methode über geben wird. Das heißt vorher sah der Stack so aus:

```
Rücksprungadresse
Parameter 1
Parameter 2
Parameter 3
...
```

Nach dem Code sieht der Stack dann so aus:

```
Rücksprungadresse
Parameter 0 = Self
Parameter 1
Parameter 2
Parameter 3
...
```

MakeProcInstance liefert jetzt die Adresse dieser neu implementierten Funktion zurück. Und diese Funktion macht eben nix weiteres, als self (welches ja in den 15 Bytes mit enthalten ist) als Parameter vorne dran zu hängen und die eigentliche Funktion (siehe Funktionszeiger) aufzurufen. (Danke an sirius für die Erklärung.)

Diese Lösung hat allerdings ein paar Nachteile:

1. Die Methode muss als `stdcall` deklariert sein.
2. Der Rückgabewert darf nicht vom Typ String, dynamic array, method pointer oder Variant sein.

→ Demo: A_1 CallBackMethod

Literaturverzeichnis

[Lischner, 2000] Ray Lischner: *Delphi in a Nutshell*. O'Reilly, 1. Auflage, 2000

[Delphi 7 Handbuchsatz] Delphi 7 Handbuchsatz: *Delphi 7 Handbuchsatz – Sprachreferenz*, Borland

[Schumann, 2000] Schumann, Hans-Georg: *Delphi für Kids*, MITP, 2000