Kurzreferenz

Delphi

Michael Puff

Erstellt: 2011-02-04

Homepage: http://www.michael-puff.de
E–Mail: mail@michael-puff.de

Inhaltsverzeichnis

Tabellenverzeichnis

1 Die Sprache

1.1 Lexikalische Elemente

Jede Zeile wird mit einem Semikolon abgeschlossen. Ausgenommen eine Befehlszeile auf die eine *else*-Anweisung folgt.

1.1.1 Kommentare

- Einzeilige Kommentare: //
- Blockkommentare: { } oder (* *)

1.1.2 Bezeichner

Bezeichner unterliegen folgenden Einschränkungen:

- Maximal 63 Zeichen lang.
- Sie müssen mit einem Buchstaben oder Unterstrich beginnen.
- Keine Sonderzeichen und Umlaute.
- Keine Leerzeichen.
- Er muss innerhalb der Sichtbarkeit einmalig sein.
- Es darf kein Schlüsselwort oder Unit (Dateiname) sein.

1.1.3 Literale

Ein Literal ist ein konstanter Ausdruck. Es gibt verschiedene Typen von Literalen:

- Die Wahrheitswerte *True* und *False*
- Integrale Literale für Zahlen, etwa 122
- Stringliterale für Zeichenketten wie „Paolo Pinkas“

1.1.4 Anweisungen und Blöcke

Delphi zählt zu den *imperativen Programmiersprachen*[1], in denen der Programmierer die Abarbeitungsschritte seiner Algorithmen durch Anweisungen vorgibt. Anweisungen können unter anderem sein:

[1] Es gibt auch Programmiersprachen, die zu einer Problembeschreibung selbstständig eine Lösung finden (Prolog)

1. Ausdrucksanweisungen etwa für Zuweisungen oder Funktionsaufrufe
2. Fallunterscheidungen (zum Beispiel mit *if*)
3. Schleifen für Wiederholungen (etwa mit *for* oder *do*-while)

Jede Anweisung wird mit einem Semikolon abgeschlossen. Vor *else* kommt kein Semikolon.

Atomare, also unteilbare Anweisungen, heißen auch *elementare Anweisungen*. Zu ihnen zählen zum Beispiel Funktionsaufrufe, Variablendeklarationen oder die leere Anweisung, die nur aus einem Semikolon besteht. Programme bestehen in der Regel aus mehreren Anweisungen, die eine Anweisungssequenz ergeben.

Ein *Block* fasst eine Gruppe von Anweisungen zusammen, die hintereinander ausgeführt werden. Anders gesagt: Ein Block ist eine Anweisung, die mit den Schlüsselwörtern *begin* und *end* eine Folge von Anweisungen zu einer neuen Anweisung zusammenfasst. Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann.

2 Aufbau der Quellcodedateien

2.1 Hauptprogrammdatei

```
program Project1;  // (1)

uses  // (2)
  Forms,
  Unit10 in 'Unit10.pas' {Form10};

// (3)

{$R *.res}

// (4)
begin
  Application.Initialize;
  Application.CreateForm(TForm10, Form10);
  Application.Run;
end.  // (5)
```

1. Programmname, muss identisch mit Dateinamen sein
2. uses-Abschnit, zum Einbinden weiterer/anderer benötiger Quellcodedateien
3. Deklarationsabschnitt für Variablen, Konstanten und für Unterfunktionen
4. Beginn Hauptprogramm
5. Abschluss Hauptprogramm

2.2 Quellcodedatei (Unit)

```
unit Unit10;  // (1)

interface  // (2)

uses  // (3)
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

// (4)

type
  TForm10 = class(TForm)
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;
```

```
var
  Form10: TForm10;

implementation  // (5)

{$R *.dfm}

end.  // (6)
```

1. Unitname, muss identisch mit Dateinamen sein
2. interface-Abschnitt für Unit übergreifende Deklarationen
3. uses-Abschnitt zum Einbinden weiterer benötigter Quellcodedateien
4. Deklarationsabschnitt für Variablen, Konstanten, Funktionsköpfen und Klassen
5. Implementationsabschnitt
6. Abschluss der Quellcodedatei

3 Variablen und Datentypen

3.1 Variablen

Eine Variable wird mit dem reservierten Wort *var* deklariert. Nach dem Variablenbezeichner folgt, durch Doppelpunkt getrennt, der Datentyp. Mehrere Variablen des gleichen Typs können durch Komma getrennt werden. Mehrere Deklarationen sind durch Semikolon zu trenne.

```
procedure Foo;
var
  i, j: Integer;
  s: string;
begin

end;
```

Außerhalb von Prozeduren und Funktionen deklarierte Variablen bezeichnet man als global. Sie sind in der ganzen Unit gültig. Innerhalb von Prozeduren und Funktionen deklarierte Variablen bezeichnet man als lokal. Sie sind nur innerhalb der Routine gültig.

Globale Variablen können initialisiert, also mit einem Wert vorbelegt, werden.

3.2 Konstanten

Mit dem reservierten Bezeichner *const* werden Konstanten deklariert. Diese können im Gegensatz zu Variablen im weiteren Programmverlauf keine anderen Werte zugewiesen werden.

```
const
  a = 1;
  flaeche = 100 * 25;
  title: string = 'Text'; // typisierte Konstante
```

3.3 Datentypen

Delphi unterstützt einige vordefinierte, einfache Datentypen, die im Folgenden beschrieben werden:

- ganze Zahlen: integer, word, byte, longint, shortint, cardinal, smallint, int64
- Gleitkommazahlen: single, double, extended
- logische Typen: boolean

Typ	Bereich	Größe [Byte]
byte	0 bis 255	1
word	0 bis 65535	2
shortint	-128 bis 127	1
smallint	-32768 bis 32767	2
integer	-2147483648 bis 2147483647	4
longint	-2147483648 bis 2147483647	4
cardinal	0 bis 4294967295	4
longword	0 bis 4294967295	4
int64	-2^{63} bis 2^{63-1}	8

Tab. 3.1: Ganze Zahlen

Typ	Bereich	Präzision	Größe [Byte]
single	$1.5*10^{-45}$ .. $3.4*10^{38}$	7	4
double	$5.0*10^{-324}$ .. $1.7*10^{308}$	15	8
extended	$3.6*10^{-4951}$ .. $1.1*10^{4932}$	19	10

Tab. 3.2: Gleitkommazahlen

- Zeichen und Zeichenketten: char, string

Hinweis: Ab Delphi 2009 sind alle Chars und Strings WideChars bzw. WideString, wenn nicht explizit als AnsiString oder AnsiChar deklariert.

Typ	Bereich	Größe [Byte]
boolean	True/False	1

Tab. 3.3: Logische Typen

Typ	Länge
Char	1
AnsiChar	1
WideChar	2
AnsiString	2GB
ShortString	255 Zeichen
string	Je nach Compiler Einstellung AnsiString ($H+) oder ShortString ($H-)
WideString	-

Tab. 3.4: Zeichen und Zeichenketten

Typ Erklärung		Größe
Pointer	Zeiger allgemein	4
PChar	Zeiger auf eine nullterminierende Zeichenkette	4

Tab. 3.5: Zeiger

3.4 Arrays

3.4.1 Statische Arrays

Die Größe statischer Arrays kann zur Laufzeit nicht geändert werden.

```
type
  TIntList = array[0..9] of Integer;  // Integer Array
  TMatrix = array[0..4, 0..4] of Byte  // zweidimensionales Byte Array
  tage = array[1..7] of string = ('Mo', 'Di', 'Mi', 'Do', 'Fr', 'Sa', 'So');  //
      Konstanten Array

var
  MyIntList: TIntList;

begin
  MyIntList[0] := 5;  // Zuweisung erstes Element
```

3.4.2 Dynamische Arrays

```
var
  a: array of Integer;
begin
  SetLength(a, 6);  // Länge auf 6 Elemente setzen
  for i := 0 to High(a) do
    a[i] := i;
```

- Festlegung der Größe erfolgt zur Laufzeit mit dem Aufruf von *SetLength*.
- Nicht mehr benötigte dynamische Arrays werden mit der Zuweisung von *nil* freigegeben.
- Dynamische Arrays sind nullbasierend.
- Die Funktion *Length* gibt die Anzahl der Array Elemente zurück.

Wichtig: Dynamische Arrays sind immer nullbasierend.

3.5 Benutzerdefinierte Datentypen (Records)

Zusammengesetzte Datentypen bezeichnet man als *Records*:

```
type
  TPerson = packed record
    Name: ShortString;
    alter: Integer;
  end;

  //...;

var
  Person: TPerson;
begin
```

```
  Person.Name := 'Hans';
```

Das Schlüsselwort *packed* ist optional. Wird es verwendet, werden die Felder des Records nicht an Datentypgrenzen ausgerichtet. Der Record wird dadurch kleiner, aber der Zugriff langsamer.

3.6 Aufzählungsdatentypen

Aufzählungstypen definieren eine Menge von Werten mit eindeutiger Reihenfolge, indem einfach die einzelnen Bezeichner dieser Werte aneinander gereiht werden. Die Werte selbst haben keine eigene Bedeutung.

```
type
  TFarbe = (rot, orange, gelb, gruen, blau, violett);
...
var
  farbe : TFarbe;
```

- Intern nummeriert Delphi die Werte bei 0 beginnend durch. Mit TFarbe(2) erhält man den Wert gelb, mit Low(TFarbe) den Wert rot, mit High(TFarbe) den Wert violett. Da die Werte geordnet sind, kann mit Succ(gruen) der Nachfolger (Successor)-Wert blau erreicht werden. Entsprechend liefert Pred(gelb) (Preducer) den Wert orange.
- An die interne Nummerierung kommt man über die Funktion Ord heran, so liefert z.B. Ord(blau) den integer-Wert 4.
- Aufzählungstypen sind skalare Typen, sie können in for-Schleifen und case-Anweisungen verwendet werden:

  ```
  for farbe := rot to blau do ...
  ```

- Man kann den Wert eines Aufzähltyps mit der Funktion GetEnumName in einen String verwandeln.
- Die interne Nummerierung kann beeinflusst werden.

  ```
  type
    TAmpel = (rot = 1, gelb = 2, gruen = 4);
  ```

4 Kontrollstrukturen

4.1 Schleifen

4.1.1 for-Schleife

```
for i := 0 to 9 do
begin

end;
```

```
// Rückwärts laufende for-Schleife
for i := 9 downto 0 do
begin

end;
```

- i nennt man Laufvariable.
- Die Laufvariable muss eine lokale Variable sein.
- Die Laufvariable muss ein ganzzahliger Datentyp, wie Integer oder Cardinal sein.
- Nach Verlassen der Schleife ist die Laufvariable unbestimmt und darf nicht weiter verwendet werden.
- Die Laufvariable kann innerhalb der Schleife nicht verändert werden.

4.1.2 repeat-until-Schleife

```
repeat

until i = 0;
```

4.1.3 while-Schleife

```
while i > 9 do
begin

end;
```

- Bei beiden Schleifen (repeat-until und while ist darauf zu achten, dass eine Abbruchbedingung existiert und auch erfüllt wird, sonst hat man eine Endlosschleife.

4.1.4 Continue, Break

- Mit dem Schlüsselwort *Continue* wird an den Anfang einer Schleife gesprungen.
- Mit dem Schlüsselwort *Break* wird eine Schleife verlassen.

4.2 Verzweigungen

4.2.1 if-else

```
if Bedingung then
begin
  Anweisung1;
end
else  // optional
begin
  Anweisung2;
end;
```

4.2.2 else if

```
if Bedingung1 then
begin
  Anweisung1;
end
else if Bedingung2 then
begin
  Anweisung2;
end
else
begin
  Anweisung3;
end;
```

4.2.3 case

```
case i of
  1:
  begin
    Anweisung1a;
    Anweisung1b;
  end;
  2: Anweisung2;
  3..5: Anweisung3;
else  // wenn keine Bedingung zutrifft
  begin
    Anweisung4;
  end;
end;
```

- Der else-Zweig ist optional.

5 Prozeduren und Funktionen

5.1 Deklaration und Aufbau

- Prozeduren haben keinen Rückgabewert.
- Funktionen haben einen Rückgabewert. Die Zuweisung des Rückgabewertes erfolgt im Funktionsrumpf an die Variable *Result*.
- Prozeduren und Funktionen können keine, einen oder mehrere Parameter übergeben werden.
- Optionale Parameter stehen am Ende der Parameterliste und sind mit einem Wert vorbelegt. Sie müssen beim Aufruf nicht mit angegeben werden.

```
function DoSomething(i: Integer; s: String; opt: string = 'Test'): Boolean;
begin
  // ...;
  // ...;
  Result := ...;
end;
```

5.2 Parameter

- Parameter werden standardmäßig als Kopie übergeben.
- Parameter, die mit dem Schlüsselwort *var* gekennzeichnet sind, werden als Zeiger auf die original Speicheradresse übergeben. Sie können innerhalb der Prozedur/Funktion verändert werden.
- Objekte und dynamische Arrays werden standardmäßig als var-Parameter übergeben.
- Parameter, die mit dem Schlüsselwort *const* übergeben werden, können innerhalb der Prozedur/Funktion nicht verändert werden.
- Arrays-Parameter müssen als eigener Typ deklariert werden.
- Prozeduren/Funktionen könne mit demSchlüsselwort *exit* an jeder beliebigen Stelle verlassen werden. Erfolgt der Aufruf von *exit* innerhalb eines Ressourcenschutzblockes[1], wird der Code im finally-Abschnitt noch ausgeführt.

[1] try-finally

6 Klassen

6.1 Grundlagen

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 Variable 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.

6.2 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.

6.3 Konstruktor, Destruktor

6.3.1 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`.

6.3.2 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.
- Des weiteren kann und darf es nur einen Destruktor geben. Konstruktoren kann es mehrere geben.
- Der Destruktor muss grundsätzlich *Destroy* heißen.

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 als letzte Anweisung 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.

6.4 Eigenschaften

6.4.1 Deklaration

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.

6.4.2 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.

6.5 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.

6.6 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`.

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.

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.

6.6.1 Methodenbinding

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

6.6.2 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.

6.6.3 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.

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.

6.7 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.

6.8 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.

6.9 Klassenoperatoren

6.9.1 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.

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
```

6.9.2 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;
```

7 Ausnahmebehandlung und Abschlussbehandlung

7.1 Ausnahmebehandlung mit try-except

Eine Exception wird ausgelöst, wenn die normale Programmausführung durch einen Fehler oder ein anderes Ereignis unterbrochen wird. Exceptions kann man auch selber auslösen. Die Steuerung wird dann an die Exception-Behandlungsroutine übergeben. Mit Hilfe dieser Routine kann die normale Programmlogik von der Fehlerbehandlung getrennt werden oder eine umständliche Bedingungsanweisungen. Exceptions sind Objekte und können somit vererbert werden, so dass man eigene Exceptions definieren kann, die von der übergeordneten Exception abgeleitet werden. Sie bringen bestimmte Informationen (z. B. eine Fehlermeldung) von der Stelle im Programm, an der sie ausgelöst wurden, zu dem Punkt, an dem sie behandelt werden.

Die Exception-Behandlung eignet sich für Fehler, die selten auftreten oder sich nur schwer eingrenzen oder vermeiden lassen, die aber schwerwiegende Folgen haben können. Sie kann auch für Fehlerbedingungen eingesetzt werden, die sich nur mit großem Aufwand in if..then-Anweisungen testen lassen.

Normalerweise kommen Exceptions bei Hardware-, Speicher- E/A- und Betriebssystemfehlern zum Einsatz. Wobei Bedingungsanweisungen (if-Verzweigungen) oftmals die beste Möglichkeit für einen Fehlertest. Beispiel:

```
try
  AssignFile(F, Filename);
  Reset(F);  // wenn die Datei nicht existiert wird eine EInOutError Exception
      ausgelöst
except
  on Exception do...
```

Im Vergleichdazu eine Fehlerbehandlung mit Bedingungsanweisungen:

```
if FileExist(FileName) then  // löst keine Exception aus
begin
  AssignFile(F, FileName);
  Reset(F);
end;
```

Exceptions auslösen und behandeln

Um eine Exception auszulösen wird das Schlüsselwort `raise` , das Exception-Objekt und das Konstruktor verwendet:

```
raise Exception.Create(...);
```

Beispiel:

```
function StrToIntRange(const s. string); Min, Max: LongInt): LongInt;
begin
  Result := StrToInt(s);
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt('%d ist nicht innerhalb des definierten Bereiches
        : %d..%d', [Result, Min, Max]);
```

Eine ausgelöste Exception wird nach ihrer Behandlung automatisch wieder freigegeben.

Die Anweisung try-except

Exceptions werden mit Hilfe der try-except-Anweisung behandelt. Beispiel:

```
try
  x := y/z;
except
  on EZeroDivide do ...;
end;
```

Tritt keine Exception auf, wird der Exception-Block ignoriert. Tritt eine Exception auf wird diese versucht zu behandlen:

- Stimmt die Exception mit einer Exception überein, wird die Steuerung an diese Rotzine übergeben.
- Wird keine Übereinstimmung gefunden, wird die Steuerung an den else-Abschnitt (wenn vorhanden) übergeben.
- Existiert im Exception-Block keine Exception-Behandlungsroutine, wird die Steuerung an die erste Anweisung im Exception-Block übergeben.

Trifft keine dieser Bedingungen zu wird die Exception noch oben in den nächsten try-excpet-Blok hoch gereicht. Findet sich auch im höchsten try-except-Block keine Anweisungsfolge oder eine Exception-Behandlungsroutine, wird das Programm mit der Exception-Meldung abgebrochen.

Beispiele:

```
try
  ...;
  ...;
  ...;
except
  on EZeroDivide do ...;
  on EOverflow do ...;
  on EMathError do ...;
end;
```

EMathError ist ein Vorfahre der beiden anderen Exception. Würde EMathError an erster Stelle stehen, würden die zwei anderen Exception-Behandlungen nie aufgerufen werden. Die übergeordnete Exception muss also immer am Schluss stehen.

```
try
  ...;
  ...;
  ...;
except
  on E: Exception do ShowMessage(E.Message);
end;
```

Will man auf das Exception-Objekt zugreifen, muss man vor dem namen der Exception-Klasse einen Bezeichner angeben über den man dann auf das Exception-Objekt zugreifen kann.

```
try
  ...;
  ...;
  ...;
except
  on EMathError do ...;
else
  ...;
end;
```

Hier werden im else-Zweig alle Exceptions behandelt, die kein EMathError sind.

```
try
  ...;
  ...;
  ...;
except
  ...;
end;
```

Hier werden alle Exceptions mit den Anweisungen im Exception-Block behandelt.

Exceptions erneut auslösen

Wird das reservierte Wort `raise` ohne nachfolgende Objektreferenz in einem Except-Block angegeben, wird die aktuell behandelte Exception erneut ausgelöst. Auf diese Weise kann in einer Behandlungsroutine begrenzt auf ein Fehler reagiert werden. Beispiel:

```
function getFileList(const Path: string): TStringList;
var
  i: Integer;
  SearchRect: TSearchRect;
begin
  Result := TStringList.Create;
  try
    i := FindFirst(Path, 0, SearchRect);
    while i = 0 do
    begin
      Result.Add(SearchRect.Name);
      i := FindNext(SearchRect);
    end;
```

```
  except
    Result.Free;
    raise;
  end;
end;
```

Hier wird das Objekt in der Behandlungsroutine freigegeben, weil es der aufrufenden Routine nicht bekannt ist. danach wird die Exception erneut ausgelöst, damit die aufrufende Routine auf den Fehler reagieren kann.

Verschachtelte Exceptions

In einer Exception-Behandlungsroutine können wiederum Exceptions ausgelöst werden.Man beachte allerdings, dass, wenn die zweite Exception die Routine verlässt, dass dann die original Exception verloren geht. beispiel:

```
type
  ETrigError = class(EMathError);

function Tan(x: Extended): Extended;
begin
  try
    Result := Sin(x) / Cos(x);
  except
    on EMathError do
      raise ETrigError.Create('Ungültiger Parameter');
  end;
end;
```

Tritt während der Ausführung ein EMathError auf, wird ein ETrigError ausgelöst. Da in *Tan* keine Routine für ETrigError definiert ist, verlässt die Exception die Behandlungsroutine und die ursprüngliche EMathError-Exception wird freigegeben. Für die aufrufende Routine stellt sich der Vorgang so dar, als ob in der Funktion *Tan* ein ETrigError ausgelöst wurde.

7.2 Abschlussbehandlung mit try-finally

In manchen Situationen will man sicher stellen, das bestimmte Anweisungen auch ausgeführt werden, wenn eine Exception auftritt. Wenn zum Beispiel in einer Routine eine Ressource zugewiesen wird, kann es wichtig sein, diese auch im Fehlerfall wieder freizugeben. In diesem fall wird die try-finally-Anweisung verwendet. Beispiel:

```
Reset(F);
try
  ..;
  ...;
  ...;
finally
  CloseFile(F);
end;
```

Der finally-Block wird in jedem Fall ausgeführt egal ob im try-Blocvk eine Exception auftritt oder nicht. Dabei ist es unabhängig davon, wie der try-Block verlassen wird. Das heißt auch bei `Break Exit` oder `Continue` werden die Anweisungen im finally-Block ausgeführt. Solch ein Konstrukt mit try-finally bezeichnet man auch als Ressourcenschutzblock.

8 Threads

8.1 Einführung

8.1.1 Geschichtliches

Am Anfang, als die Threads laufen lernten, gab es kein Multitasking. Ja, man konnte es noch nicht mal Singletasking nennen. Man hat sein Programm in Lochkarten gestanzt, hat sie im Rechenzentrum abgegeben, und Tage später bekam man einen Stapel Lochkarten wieder zurück, mit oder auch oft ohne die gewünschten Resultate. War ein Fehler drin, musste die entsprechende Lochkarte ganz neu gestanzt werden.

Aber die Dinge haben sich weiterentwickelt. Das erste Konzept, bei dem mehrere Threads parallel ausgeführt wurden, tauchte bei den so genannten ttime sharing systemsäuf. Es gab einen großen zentralen Computer der mehrere Client Computer mit Rechenleistung versorgte. Man spricht von Mainframes und Workstations. Hier war es wichtig, dass die Rechenleistung bzw. die CPU-Zeit gerecht zwischen den Workstations aufgeteilt wurde. In diesem Zusammenhang erschien auch zum ersten Mal das Konzept von Prozessen und Threads. Desktop-Computer haben eine ähnliche Entwicklung durchgemacht. Frühe DOS- und Windows-Rechner waren Singletasking-Systeme. Das heißt, ein Programm lief exklusiv und allein für sich auf einem Rechner. Selbst das Betriebssystem bekam während der Ausführung eines Programms keine Rechenzeit zugeteilt. Aber die Anforderungen stiegen und die Rechner stellten immer mehr Leistung zur Verfügung. Leistung die auch heute noch die meiste Zeit, die der Rechner in Betrieb ist, brachliegt. Warum sie also nicht so effizient wie möglich nutzen, und den Rechner mehrere Dinge gleichzeitig machen lassen, wenn er kann? Das Multitasking war geboren.

8.1.2 Begriffsdefinition

Was ist ein Prozess?

Ganz allgemein kann man sagen, dass ein Prozess die laufende Instanz einer Anwendung, oder wenn man so will, eines Programms, ist. Ein Prozess besteht immer aus zwei Teilen:

1. Dem Kernel-Objekt[1], welches dem Betriebssystem zur Verwaltung des Prozesses dient, und dem
2. Adressraum, der den ausführbaren Code, die Daten und die dynamischen Speicherzuweisungen (Stack- und Heap-Zuweisungen) enthält.

[1] Ein Kernel-Objekt ist im Grunde genommen nichts weiter als ein Speicherblock, den der Kernel belegt hat. Dieser Speicherblock stellt eine Datenstruktur da, deren Elemente Informationen über das Objekt verwalten. Hinweis: Kernel-Objekte gehören dem Kernel, nicht dem Prozess.

Jeder Prozess muss mindestens einen Thread besitzen, den primären Thread. Diesen primären Thread braucht man nicht selber zu erzeugen, dies übernimmt der Loader, der den Prozess startet. Bei Windows Programmen, die ein Fenster besitzen, enthält der primäre Thread die Nachrichtenschleife und die Fenster-Prozedur der Anwendung. Tut er dies nicht, so gäbe es keine Existenzberechtigung für ihn und das Betriebssystem würde ihn mitsamt seines Adressraumes automatisch löschen. Dies wiederum bedeutet, wird der primäre Thread eines Prozesses beendet, wird auch der zugehörige Prozess beendet, das Kernel-Objekt zerstört und der reservierte Adressraum wieder freigegeben. Ein Prozess dient also quasi als Container für den primären Thread und weiteren Threads , die im Laufe der des Programms abgespalten werden können.

Ein Prozess, oder genauer, dessen Threads führen den Code in einer geordneten Folge aus. Dabei operieren sie alle streng getrennt voneinander. Das heißt, ein Prozess kann nicht auf den Adressraum eines anderen Prozesses zugreifen. Ein Prozess sieht also die anderen Threads gar nicht und hat den Eindruck, als ob er alle Systemressourcen (Speicher, Speichermedien, Ein- / Ausgabe, CPU) für sich alleine hätte. In Wirklichkeit ist dies natürlich nicht der Fall, so dass das Betriebssystem die Aufgabe übernehmen muss, dem Prozess, dem seine eigenen «virtuellen» Ressourcen zur Verfügung stehen, reale Ressourcen zuzuordnen. In diesem Zusammenhang spricht man auch beispielsweise von virtuellen Prozessoren im System. Es gibt so viele virtuelle Prozessoren, wie es Prozesse / Threads im System gibt. Auf der anderen Seite gibt es aber auch Ressourcen, die von den Prozessen geteilt werden können. Da wären zum Beispiel dynamische Bibliotheken, die DLL's, zu nennen. Eine DLL kann von mehreren Prozessen gleichzeitig genutzt werden. Dabei wird sie nur einmal geladen, ihr Code aber in den Adressraum des Prozesses eingeblendet, welcher die DLL nutzt. Dies kann man unter anderem zur Inter Process Communication nutzen. Dies hat aber des Weiteren noch den Vorteil, dass gleicher Code nur einmal geschrieben werden muss und gleichzeitig von mehreren Prozessen genutzt werden kann.

Das Besondere ist, dass nur allein der Kernel Zugriff auf diesen Speicherblock hat. Anwendungen können ihn weder lokalisieren, noch auf ihn zugreifen. Der Zugriff kann nur über API-Funktionen erfolgen. Dies gewährleistet die Konsistenz des Kernel-Objektes.

8.1.3 Threads, die arbeitende Schicht

Ein Thread beschreibt nun einen Äusführungspfadïnnerhalb eines Prozesses. Und der Thread ist es, der den eigentlichen Programmcode ausführt. Der Prozess dient, wie schon gesagt, lediglich als Container, der die Umgebung (den Adressraum) des Threads bereitstellt.

Threads werden immer im Kontext des ihnen übergeordneten Prozesses erzeugt. Das heißt, ein Thread führt seinen Code immer im Adressraum seines Prozesses aus. Werden beispielsweise zwei oder mehr Threads im Kontext eines Prozesses ausgeführt, teilen sie sich einen einzigen Adressraum, dem des Prozesses nämlich. Sie können also denselben Code ausführen und dieselben Daten bearbeiten. Den Zugriff auf den gleichen Adressraum kann man einerseits als Segen, andererseits auch als Fluch sehen. Denn es ist zwar einfach Daten zwischen den Threads auszutauschen, nur die Synchronisation des Datenaustausches kann mit unter recht schwierig werden.

Wann immer es möglich, wenn Multithreading gefragt ist, sollte man einen eigenen Thread starten, anstatt eines eigenen Prozesses, wie man es unter 16-Bit Windows noch tun musste. Dies kannte zwar Multitasking, aber kein Multithreading. Denn im Gegensatz zum Anlegen eines Prozesses, erfordert das Starten eines Threads weniger Systemressourcen, da nur das Thread-Kernel-Objekt angelegt und verwaltet werden muss. Dass Anlegen und Verwalten eines Adressraumes fällt ja weg, da der Thread ja den Adressraum des übergeordneten Prozesses mitbenutzt. Einzig und allein ein weiterer Stack muss für den Thread im Adressraum des Prozesses angelegt werden.

Multitasking und Zeitscheiben

Wie setzt das Betriebssystem nun die Technik des Multitasking / Multithreading um? Das Problem ist ja, dass in heutigen herkömmlichen Desktop Computern meist nur ein Prozessor zur Verfügung steht. Diesen Prozessor müssen sich also nun die im System laufenden Prozesse teilen. Dafür gibt es zwei Ansätze unter Windows:

1. Kooperatives Multitasking (16-Bit Windows)
2. Präemptives Multitasking (32-Bit Windows)

Beim kooperativen Multitasking ist jeder Prozess selbst dafür verantwortlich Rechenzeit abzugeben und so anderen Prozessen die Möglichkeit zu geben weiter zu arbeiten. Die Prozesse müssen also «kooperieren». Wer hingegen beim präemptiven Multitasking die Zuteilung von Rechenzeit ganz alleine dem Betriebssystem obliegt. Anders wäre es beispielsweise nicht möglich gewesen den Taskmanager in das System einzufügen, denn wie will man einen anderen Prozess starten, wenn ein anderer den Prozessor in Beschlag nimmt und ihn nicht wieder freigibt? Da zudem der Taskmanager mit einer höheren Priorität ausgeführt wird, als andere Prozesse, kann er immer noch dazu genutzt werden «hängengebliebene» Prozesse zu beenden.

Beim präemptiven Multitasking wird die Rechenzeit, die zur Verfügung steht, in so genannte Zeitscheiben eingeteilt. Das heißt, ein Prozess kann die CPU für eine bestimmte Zeit nutzen, bevor das Betriebssystem die CPU diesem Prozess entzieht und sie einem anderen Prozess zuteilt. Dies gilt genauso für Threads innerhalb eines Prozesses. Denn unter 32-Bit-Windows ist die kleinste ausführbare Einheit ein Thread und nicht wie unter 16-Bit-Windows die Instanz eines Prozesses. Wird nun ein Thread unterbrochen, speichert das Betriebssystem den momentanen Zustand des Threads, sprich die Werte der CPU Register usw. und wenn dem Thread Rechenzeit zugeteilt wird, wird der Zustand der Register vom Betriebssystem wiederhergestellt und der Thread kann an der Stelle, an der er unterbrochen wurde, weiter Code ausführen. Tatsächlich wird also nichts parallel ausgeführt, sondern nacheinander. Nur erscheint die Ausführung für den Benutzer parallel, da die Zeitscheiben so kurz sind, dass es nicht auffällt und der Anschein von Parallelität erzeugt wird.

8.1.4 Wann sollte man Threads einsetzen und wann nicht?

Um es vorweg zu nehmen, es gibt keine festen Regeln, wann man mehrere Threads zu verwenden hat und wann nicht. Dies hängt ganz davon ab, was das Programm tun soll und

wie wir das erreichen wollen. Man kann eigentlich nur vage Empfehlungen aussprechen. Was man allerdings jedem raten kann, der Multithread-Anwendungen entwickelt, ist, dass Threads umsichtig eingesetzt werden sollten. Denn: Die Implementation ist eigentlich trivial. Das eigentlich Schwere an der Sache ist, Threads interagieren zu lassen, ohne dass sie sich ins Gehege kommen.

Der Einsatz von Threads ist dann sinnvoll, wenn Code im Hintergrund ausgeführt werden soll, also Code, der keine Benutzereingaben erfordert und auch keine Ausgaben hat, die den Benutzer zum Zeitpunkt der Ausführung interessieren. Als Beispiele wären da zu nennen: Jegliche langen Berechnungen von irgendetwas, eine Folge von Primzahlen oder ähnlichem. Oder die automatische Rechtschreibkorrektur in Textverarbeitungen. Oder die Übertragung / das Empfangen von Daten über ein Netzwerk oder Port. Auch Code der asynchron ausgeführt wird, wie das Pollen eines Ports, ist besser in einem eigenen Thread aufgehoben, anstatt mit dem Vordergrund-Task zu kämpfen, der im gleichen Thread ausgeführt wird.

Ein Beispiel, wann Threads nicht sinnvoll sind: Nehmen wir an, wir haben eine Textverarbeitung und wir wollen eine Funktion zum Drucken implementieren. Schön wäre es, wenn selbige nicht die ganze Anwendung blockieren würde und man mit ihr weiter arbeiten könnte. Diese Situation scheint doch gerade zu prädestiniert für einen Thread zu sein, der das Dokument ausdruckt. Nur bei etwas genaueren Überlegungen stößt man auf Probleme. Wie druckt man ein Dokument aus, welches sich ständig ändert, weil daran weitergearbeitet wird? Eine Lösung wäre, das Dokument für die Dauer des Drucks zu sperren und dem Benutzer nur an Dokumenten arbeiten zu lassen, die sich nicht im Druck befinden. Unschön. Aber wie sieht es mit der Lösung aus? Man speichert das zu druckende Dokument in einer temporären Datei und druckt diese? Ein Thread wäre dazu nicht mehr nötig.

8.2 Grundlagen

8.2.1 Veranschaulichung

Veranschaulichen wir uns das ganze mal an Hand einer Grafik.

Legende:

- schräge Striche kennzeichnen den Beginn eines Threads
- durchgezogene Linien, wann der Thread Code ausführt
- gestrichelte Linien, wann sich der Thread im Wartezustand befindet
- Punkte bezeichnen das Ende eines Threads

Zuerst wird der Hauptthread gestartet. Dies übernimmt der Loader, wenn der Prozess erzeugt wird. Als nächstes befindet sich der Hauptthread im Wartezustand, er führt also keinen Code aus. Der Benutzer tätigt keine Eingaben und macht auch sonst nichts mit dem Fenster. Ein Klick auf eine Schaltfläche weckt den Hauptthread und veranlasst ihn Code auszuführen. In diesem Fall wird ein zweiter Thread abgespalten. Dieser wird aber nicht gleich gestartet, sondern verbringt auch erst eine Zeit im Wartezustand bevor er von Hauptthread gestartet wird. Der folgende Abschnitt der Grafik verdeutlicht wie Code von beiden Prozessen parallel

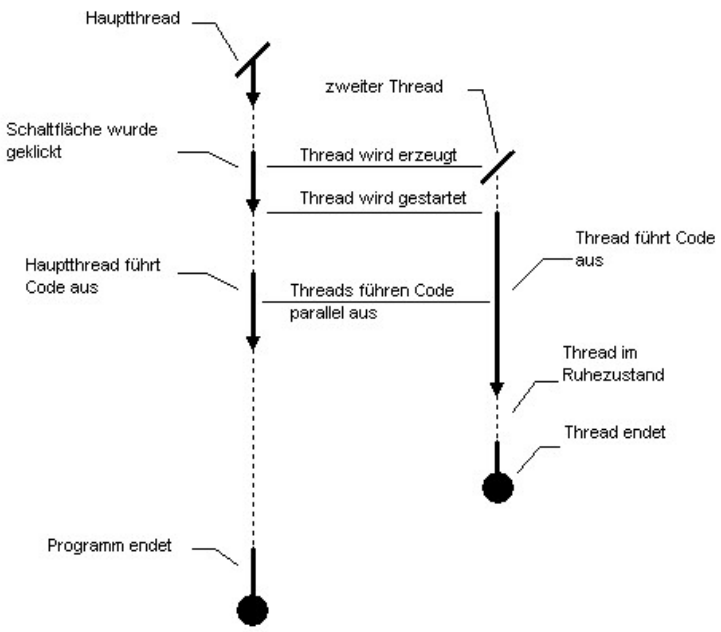

Abb. 8.1: Veranschaulichung von Threads

ausgeführt wird. Letztendlich enden beide Threads, wobei der zweite Thread vorher entweder automatisch endet, weil er seinen Code abgearbeitet hat oder sonst wie (dazu später) beendet wurde.

8.2.2 Die Thread-Funktion

Jeder Thread braucht eine Funktion, die den Code enthält, welchen der Thread ausführen soll. Diese bezeichnet man auch als Startfunktion des Threads. Die Delphi Deklaration sieht wie folgt aus:

```
type TThreadFunc = function(Parameter: Pointer): Integer;
```

Als Parameter kann optional ein Pointer übergeben werden, der auf eine Datenstruktur zeigt mit Werten, die dem Thread zum Arbeiten übergeben werden sollen. Der Rückgabewert der Thread-Funktion ist eine Ganzzahl, welche beim beenden dem Exitcode des Threads entspricht. Im Demo Programm *ThreadTimes* wird demonstriert, wie man das macht.

Zu beachten ist allerdings noch folgendes: Benutzt man *BeginThread* als Wrapper für die API-Funktion *CreateThread*, so darf man nicht die Aufrufkonvention *stdcall* für die Thread-Funktion benutzen. Grund: *BeginThread* benutzt eine TThread-WrapperFunktion anstatt des wirklichen Einsprungspunktes. Diese Funktion ist als *stdcall* definiert, sie verschiebt die Parameter auf dem Stack und ruft dann die eigene Thread-Funktion auf.

Des weiteren sollte man möglichst keine ungeschützten Zugriffe auf globale Variablen durchführen, da ein simultaner Zugriff mehrerer Threads den Variablen-Inhalt „beschädigen“ könn-

Parameter	Bedeutung
lpThreadAttributes	Zeiger auf eine Struktur, welche festlegt, ob das zurückgegeben Handle vererbt werden kann oder nicht.
dwStackSize	Legt die Größe des Stacks fest. 0 überlässt die Festlegung Windows.
lpStartAddress	Zeiger auf die Thread-Funktion.
lpParameter	Zeiger auf eine Variable, die dem Thread beim Start übergeben werden soll.
dwCreationFlags	Ein Flag, welches festlegt wie der Thread erzeugt werden soll. Dieser Flag kann entweder CREATE_SUSPENDED sein, der Thread wird im angehaltenen Zustand erzeugt oder 0, dann startet der Thread sofort nach seiner Erzeugung.
var lpThreadID	Eindeutiger Bezeichner des Threads.

Tab. 8.1: Parameter CreateThread

te. Parameter und lokale Variablen hingegen werden auf dem thread-eigenen Stack abgelegt und sind dort weniger anfällig für „Beschädigungen“. Wie man trotzdem einen sichern Zugriff auf globale Variablen realisiert wird im Kapitel sechs besprochen.

8.2.3 Einen Thread abspalten

Wie schon gesagt, wird der primäre Thread vom Loader automatisch angelegt. Will man nun in seinem Prozess einen weiteren Thread erzeugen, so muss man auf die Funktion CreateThread zurückgreifen:

```
function CreateThread(lpThreadAttributes: Pointer; dwStackSize: DWORD;
  lpStartAddress: TFNThreadStartRoutine; lpParameter: Pointer;
  dwCreationFlags: DWORD; var lpThreadId: DWORD): THandle; stdcall;
```

In der folgenden Tabelle (Tabelle 8.1, Seite 40) findet sich eine kurze Erklärung der Parameter.

Was passiert nun nach dem Aufruf von *CreateThread*? Das System legt ein Kernel-Objekt für den Thread an, um ihn verwalten zu können. Des weiteren wird im Adressraum des übergeordneten Prozesses ein separater Stack für den Thread angelegt.

Der neue Thread wird im selben Kontext ausgeführt wie der Thread, der ihn erzeugte. Folge: Der neue Thread kann auf alle prozess-eigenen Handles für Kernel-Objekte, den gesamten prozess-eigenen Speicher, sowie auf alle Stacks aller anderen Threads innerhalb des Prozesses zugreifen.

In einem Delphi-Programm sollten Sie nie die Funktion *CreateThread* direkt aufrufen. Benutzen Sie stattdessen die Funktion *BeginThread*. Grund: *BeginThread* kapselt zwar nur die API-Funktion *CreateThread*, setzt aber zusätzlich noch die globale Variable *IsMultiThreaded* und macht somit den Heap thread-sicher.

8.2.4 Parameter an einen Thread übergeben

Über den Parameter *lpParameter* kann man dem Thread beliebige Daten übergeben. Dazu muss man einen Zeiger auf die Variable mit den Daten übergeben. Einfache Daten (Integers, Strings) kann man ohne weiteres einfach so übergeben. Will man aber mehrere Daten an den Thread übergeben, wird es geringfügig aufwendiger. Dazu erstellt man eine Struktur und übergibt dann einen Zeiger auf diese Struktur der Funktion *BeginThread*.

```
type
  TThreadParams = packed record
    Number: Integer;
    Text: String;
  end;
  PThreadParams = ^TThreadParams;
```

Um einen Zeiger an die Funktion *BeginThread* zu übergeben, hat man jetzt zwei Möglichkeiten. Man kann den Parameter entweder über den Heap übergeben oder über den Stack. Übergibt man den Parameter über den Heap, muss man vor dem Aufruf von *BeginThread* Speicher anfordern und ihn natürlich am Ende vom Thread wieder freigeben:

```
function ThreadFunc(tp: PThreadParams): Integer;
var
  Number            : Integer;
  Text              : string;
  s                 : string;
begin
  // Parameter lokalen Variablen zuweisen.
  Number := PThreadParams(tp)^.Number;
  Text := PThreadParams(tp)^.Text;
  s := 'Zahl: ' + IntToStr(Number) + #13#10 + 'Text: ' + Text;
  // ExitCode enthält Rückgabewert der Messagebox.
  Result := MessageBox(0, PChar(s), 'Thread', MB_YESNO or MB_ICONINFORMATION);
  // Reservierten Speicher für Parameter wieder freigeben.
  Dispose(tp);
end;

procedure RunThread;
var
  tp                : PThreadParams;
  Thread            : THandle;
  ThreadID          : Cardinal;
  ExitCode          : Cardinal;
begin
  // Speicher für Struktur reservieren.
  New(tp);
  // Daten den feldern der Struktur zuweisen.
  tp.Number := 42;
  tp.Text := 'Die Antwort.';
  // Thread erzeugen.
  Thread := BeginThread(nil, 0, @ThreadFunc, tp, 0, ThreadID);
  // Auf Beendigung des Threads warten.
  WaitForSingleObject(Thread, INFINITE);
  // Rückgabewert ermitteln...
  GetExitCodeThread(Thread, ExitCode);
  // ...und auswerten.
```

```
  case ExitCode of
    IDYES: Writeln('Benutzer hat "Ja" angeklickt.');
    IDNO: Writeln('Benutzer hat "Nein" angeklickt.');
  end;
  // Thread-Handle schliessen udn somit das Thread-Objekt zerstören.
  CloseHandle(Thread);
end;
```

Alternativ kann man den Parameter auch über den Stack übergeben:

```
function ThreadFunc(tp: PThreadParams): Integer;
var
  Number            : Integer;
  Text              : string;
  s                 : string;
begin
  Number := PThreadParams(tp)^.Number;
  Text := PThreadParams(tp)^.Text;
  s := 'Zahl: ' + IntToStr(Number) + #13#10 + 'Text: ' + Text;
  Result := MessageBox(0, PChar(s), 'Thread', MB_YESNO or MB_ICONINFORMATION);
end;

procedure RunThread;
var
  tp                : TThreadParams;
  Thread            : THandle;
  ThreadID          : Cardinal;
  ExitCode          : Cardinal;
begin
  tp.Number := 42;
  tp.Text := 'Die Antwort.';
  Thread := BeginThread(nil, 0, @ThreadFunc, @tp, 0, ThreadID);
  WaitForSingleObject(Thread, INFINITE);
  GetExitCodeThread(Thread, ExitCode);
  case ExitCode of
    IDYES: Writeln('Benutzer hat "Ja" angeklickt.');
    IDNO: Writeln('Benutzer hat "Nein" angeklickt.');
  end;
  CloseHandle(Thread);
end;
```

Allerdings ist hier wichtig, dass die Funktion, die den Thread erzeugt, nicht verlassen wird, bevor der Thread beendet ist. Wird die aufrufende Prozedur verlassen, wird der Stack aufgeräumt und die übergeben Daten gehen verloren, weil sie gelöscht werden oder an ihrer Stelle schon andere Daten abgelegt wurden. Schlimmsten Falls kann dies zu einem Programmabsturz führen. Siehe dazu auch die Demo-Programme *ParameterHeap* und *ParameterStack* im Ordner *Parameter*.

8.2.5 Beenden eines Threads

Ein Thread kann auf vier Arten beendet werden:

1. Die Thread-Funktion endet.

2. Der Thread beendet sich selbst mit der Funktion ExitThread
3. Der Thread wird durch TerminateThread beendet.
4. Der Prozess des Threads endet.

Die Thread-Funktion endet

Threads sollten so entworfen werden, dass sie nach Ausführung ihres Codes selbstständig enden. Grundsätzlich ist es so, dass der Thread automatisch endet, wenn er seinen Code abgearbeitet hat. Handelt es sich um einen Thread, der die ganze Zeit etwas machen soll, also einen Port pollen oder so, dann wird dieser Code normalerweise in eine Schleife gesteckt. Diese Schleife kann man mit einem break oder sonst einer Abruchbedingung verlassen und so den Thread sich selber enden lassen:

```
while bRunning <> 0 do
begin
  ...;
  ...;
end;
```

Nutzt man das Thread-Objekt der VCL kann man alternativ die Eigenschaft Terminated abfragen, welche auch von außerhalb des Threads gesetzt werden kann:

```
for Loop := 0 to ... do
begin
  ...;
  if Terminated then
    breaek;
  ...;
end;
```

Ein Thread sollte sich deshalb selber beenden, weil dann sichergestellt ist, dass

- der thread-eigene Stack wieder freigegeben wird
- der ExitCode den Rückgabewert der Thread-Funktion enthält
- der Zugriffszähler des thread-bezogenen Kernel-Objektes decrementiert wird
- alle Referenzen auf geladene DLL's und ähnlichem decrementiert werden

ExitThread

Eine weitere Möglichkeit einen Thread sich selber beenden zu lassen, ist die Verwendung der API Funktion ExitThread.

```
procedure ExitThread(dwExitCode: DWORD); stdcall;
```

Parameter	Bedeutung
dwExitCode	Exitcode des Threads

Tab. 8.2: Parameter ExitThread

ExitThread hat keinen Rückgabewert, da nach Aufruf, der Thread keinen Code mehr ausführen kann. Auch hier werden alle Ressourcen wieder freigegeben.

Hinweis: In einer Delphi Anwendung sollte statt ExitThread das Äquivalent zu BeginThread EndThread aufgerufen werden.

TerminateThread

Im Gegensatz zu *ExitThread*, welches nur den aufrufenden Thread beenden kann (schon zu erkennen an der Tatsache, dass ExitThread kein Parameter übergeben werden kann, welcher den zu beendenden Thread näher identifiziert.), kann mit TerminateThread jeder Thread beendet werden, dessen Handle man hat.

```
function TerminateThread(hThread: THandle; dwExitCode: DWORD): BOOL; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads der beendet werden soll.
dwExitCode	Exitcode des Threads.

Tab. 8.3: Parameter TerminateThread

TerminateThread arbeitet asynchron. Das heißt, *TerminateThread* kehrt sofort zurück. Es teilt dem Betriebssystem nur mit, dass der betreffende Thread beendet werden soll. Wann letztendlich das Betriebssystem den Thread beendet, ob überhaupt (fehlende Rechte könnten dies verhindern), steht nicht fest. Kurz: Nach Rückkehr der Funktion ist nicht garantiert, dass der betreffende Thread auch beendet wurde.

Des weiteren ist zu beachten, dass alle vom Thread belegten Ressourcen erst dann wieder freigegeben werden, wenn der übergeordnete Prozess endet. Sprich der thread-eigene Stack wird zum Beispiel nicht abgebaut.

Abschließend kann man sagen, dass man vermeiden sollte einen Thread gewaltsam zu beenden. Und dies aus zwei Gründen:

1. Die belegten Ressourcen werden nicht freigegeben
2. Man weiß nicht, in welchem Zustand sich der Thread gerade befindet. Hat er beispielsweise gerade eine Datei exklusiv geöffnet, können auf diese Datei so lange keine anderen Prozesse zugreifen bis der Prozess des äbgeschossenenTThreads beendet wurde.

Vorgänge beim Beenden eines Threads

Wird ein Thread «sauber» beendet, werden alle zum Thread gehörenden Handles für User-Objekte freigegeben, der Exitcode wechselt von STIIL_ACTIVE auf den Rückgabewert der Thread-Funktion oder dem von *ExitThread* oder *TerminateThread* festgelegten Wert, wird der übergeordnete Prozess beendet, sofern es sich um den letzten Thread des Prozesses handelt und der Zugriffszähler des Thread–Objektes wird um eins verringert.

Mit der Funktion *GetExitCodeThread* lässt sich der Exitcode eines Threads ermitteln.

```
function GetExitCodeThread(hThread: THandle; var lpExitCode: DWORD): BOOL;
  stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads, dessen Exitcode ermittelt werden soll.
var lpExitCode	Out Parameter der den Exitcode nach Aufruf der Funktion enthält.

Tab. 8.4: Parameter GetExitCodeThread

So lange der Thread beim Aufruf noch nicht terminiert ist, liefert die Funktion den Wert STILL_ACTIVE ($103) zurück, ansonsten den Exitcode des betreffenden Threads.

8.3 Thread Ablaufsteuerung

Beim präemptiven Multitasking – im Gegensatz zum kooperativen Multitasking, wie es bei Windows der Fall ist, wird der Zeitraum in Scheiben definierter Länge unterteilt. Den einzelnen Prozessen werden durch den Scheduler Zeitscheiben zugewiesen. Läuft ihre Zeitscheibe ab, so werden sie unterbrochen und ein anderer Thread erhält eine Zeitscheibe.

Daher ist eine wesentliche Voraussetzung für echtes präemptives Multitasking eine CPU, die verschiedene Berechtigungs-Modi beherrscht und ein Betriebssystemkern, der in der Lage ist, die verschiedenen Modi dahingehend zu nutzen, Anwendungssoftware mit unterschiedlichen Berechtigungen auszuführen. Ein einfaches Beispiel:

- Anwendungssoftware ist „unterprivilegiert“ im Sinne von „sie darf nicht alles“. Sie muss nämlich von einem Prozess mit höher Berechtigungsstufe unterbrechbar sein. Das ist die Grundvoraussetzung dafür, dass der Kernel – wie oben beschrieben – ihr
  1. eine Zeitscheibe zuteilen und den Prozess aufrufen kann,
  2. bei Ablauf der Zeitscheibe den Prozess einfrieren und „schlafen“ legen kann und
  3. bei Zuteilung einer neuen Zeitscheibe den Prozess wieder so aufwecken kann, dass der Prozess nicht merkt, dass er unterbrochen wurde
- Der Systemkern selbst läuft in einem Modus mit maximaler Berechtigungsstufe.
  1. Bei Linux unterscheidet man hier zwischen Kernel-Mode und User–Mode,
  2. bei Windows NT und Nachfolgern spricht man von „Ring 0“ als Kernel-Modus und Ringen höherer Nummerierung für „unterprivilegierte“ Prozesse.

Die Kontextstruktur des Threads enthält den Zustand der Prozessorregister zum Zeitpunkt der letzten Unterbrechung der Thread Ausführung. Das System überprüft nun ca. alle 20 Millisekunden alle vorhandenen Thread-Objekte, dabei werden nur einige Objekte als zuteilungsfähig und damit rechenbereit befunden. Es wird nun ein zuteilungsfähiges Thread-Objekt gewählt und der Inhalt der Prozessorregister aus der Kontextstruktur des Threads in die Register der CPU geladen. Diesen Vorgang bezeichnet man als Kontextwechsel.

Im aktiven Zustand nun, führt ein Thread Code aus und bearbeitet Daten. Nach 20 Millisekunden werden die Registerinhalte wieder gespeichert, der Thread befindet sich nun nicht

mehr im aktiven Zustand der Ausführung und ein neues Thread-Objekt wird gewählt. Bei der Rechenzeitzuteilung werden nur zuteilungsfähige Threads berücksichtigt, das heißt Threads mit einem Unterbrechungszähler gleich 0. Besitzt ein Thread einen Unterbrechungszähler größer 0, bedeutet dies, dass er angehalten wurde und das ihm keine Prozessorzeit zugeteilt werden kann. Des weiteren bekommen auch Threads ohne Arbeit keine Rechenzeit, also Threads, die zum Beispiel auf eine Eingabe warten. Startet man zum Beispiel den Editor und gibt nichts ein, hat der Editor-Thread auch nichts zu tun und braucht auch keine Rechenzeit. Sobald aber etwas mit dem Editor geschieht, man nimmt eine Eingabe vor, oder man verschiebt das Fenster oder sollte das Fenster seine Neuzeichnung veranlassen müssen, so wird der Editor-Thread automatisch in den zuteilungsfähigen Zustand überführt. Das heißt nun aber nicht, dass er sofort Rechenzeit zugeteilt bekommt. Es heißt nur, dass er jetzt arbeitswillig ist und das System ihm bald Rechenzeit gewährt.

8.3.1 Anhalten und Fortsetzen von Threads

Fortsetzen

Ein Element im thread-bezogenen Kernel-Objekt stellt der Unterbrechungszähler dar. Beim Erzeugen eines Threads wird er mit 1 initialisiert, der Thread ist somit nicht zuteilungsfähig. Dies stellt sicher, dass er erst ausführungsbereit ist, wenn er vollständig initialisiert ist. Nach der Initialisierung wird der Flag CREATE_SUSPENDED (5. Parameter von *CreateThread*) überprüft. Ist er nicht gesetzt, wird der Unterbrechungszähler decrementiert und somit auf 0 gesetzt, der Thread ist zuteilungsfähig. Erzeugt man einen Thread im angehaltenen Zustand, so hat man die Möglichkeit, seine Umgebung, zum Beispiel die Priorität, zu ändern. Er kann dann mit *ResumeThread* in den zuteilungsfähigen Zustand versetzt werden.

```
function ResumeThread(hThread: THandle): DWORD; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads

Tab. 8.5: Parameter ResumeThread

ResumeThread gibt bei erfolgreicher Ausführung den bisherigen Wert des Unterbrechungszählers zurück oder $FFFFFFFF im Fehlerfall.

Das Anhalten eines Threads ist akkumulativ, das heißt, wird ein Thread dreimal angehalten, muss auch dreimal *ResumeThread* aufgerufen werden, um ihn wieder in den zuteilungsfähigen Zustand zu versetzten.

Anhalten

Das Gegenstück zu *ResumeThread* ist *SuspendThread*. Mit dieser Funktion kann ein Thread angehalten werden.

```
function SuspendThread(hThread: THandle): DWORD; stdcall;
```

SuspendThread gibt, wie auch *ResumeThread*, im Erfolgsfall den bisherigen Wert des Unterbrechungsszählers zurück. Beide Funktionen sind von jedem Thread aus aufrufbar, sofern man über das Handle des betreffenden Threads verfügt.

Analog lauten die Methoden des Thread-Objektes *Resume* und *Suspend*.

Hinweis: Ein Thread kann sich mit *SuspendThread* auch selber anhalten. Er ist aber nicht in der Lage seine Fortsetzung zu veranlassen.

Man sollte allerdings vorsichtig mit dem Aufruf von *SuspendThread* sein, da nicht bekannt ist was der Thread gerade tut. Reserviert er zum Beispiel gerade Speicher auf dem Heap und wird dabei unterbrochen, dann würde die Unterbrechung des Threads eine Sperrung des Heapzugriffes bewirken. Würden nun anschließend andere Threads versuchen auf den Heap zuzugreifen, würde ihre Ausführung so lange blockiert bis der andere Thread fortgesetzt wird und seinen Zugriff beenden kann.

Zeitlich begrenztes Unterbrechen

Ein Thread kann auch dem System mitteilen, dass er für eine bestimmte Zeitdauer nicht mehr zuteilungsfähig sein möchte. Dazu dient die Funktion Sleep.

```
procedure Sleep(dwMilliseconds: DWORD); stdcall;
```

Diese Prozedur veranlasst den Thread sich selbst, so lange anzuhalten wie in *dwMilliseconds* angegeben. Dabei gilt zu beachten: Wird *Sleep* mit 0 aufgerufen, verzichtet der Thread freiwillig auf den Rest seiner Zeitscheibe. Die Zuteilungsfähigkeit wird nur um ungefähr die angegebene Dauer in Millisekunden verzögert. Da es immer darauf ankommt, was sonst noch im System los ist. Ruft man *Sleep* mit dem Parameter INFINITE auf, wird Thread überhaupt nicht mehr zuteilungsfähig. Ist aber nicht sehr sinnvoll, da es besser wäre ihn zu beenden und somit die Ressourcen wieder freizugeben.

Temporärer Wechsel zu einem anderen Thread

Mit der Funktion SwitchToThread stellt das System eine Funktion bereit, um zu einem anderen Thread umzuschalten. Gibt es allerdings keinen zuteilungsfähigen Thread, kehrt SwitchToThread sofort zurück.

```
function SwitchToThread: BOOL; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads

Tab. 8.6: Parameter SuspendThread

SwitchToThread entspricht fast dem Aufruf von *Sleep(0)* mit dem Unterschied, dass *SwitchToThread* die Ausführung von Threads mit niedriger Priorität gestattet, während *Sleep(0)* den aufrufenden Thread sofort neu aktiviert, auch wenn dabei Threads mit niedriger Priorität verdrängt werden.

Hinweis: Windows98 besitzt keine sinnvolle Implementation von *SwitchToThread*.

Thread Ausführungszeiten, Beispielanwendung *ThreadTimes*

Es kann vorkommen, dass man ermitteln will wie viel Zeit ein Thread für die Durchführung einer bestimmten Aufgabe benötigt. Oft findet man zu diesem Zweck Code wie diesen:

```
var
  StartTime, TimeDiff: DWORD;
begin
  StartTime := GetTickCount();

  // zu untersuchender Code

  TimeDiff := GetTickCount() - Starttime;
```

Mit diesem Code geht man allerdings davon aus, dass der Thread nicht unterbrochen wird, was aber bei einem präemptiven Betriebssystem nicht der Fall sein wird. Wird einem Thread die CPU entzogen, ist es entsprechend schwierig zu ermitteln, wie lange der Thread für die Erledigung verschiedener Aufgaben benötigt. Windows stellt allerdings eine Funktion bereit, die den Gesamtbetrag an Rechenzeit zurück gibt, die dem Thread tatsächlich zugeteilt worden ist. Diese Funktion heißt: *GetThreadTimes*.

```
function GetThreadTimes(hThread: THandle; var lpCreationTime, lpExitTime,
  lpKernelTime, lpUserTime: TFileTime): BOOL; stdcall;
```

Die Funktion gibt vier verschiedene Zeitwerte zurück:

CreationTime enthält die Zeit seit dem 1. Januar 1601 in Nanosekunden–Intervallen, das gilt auch für *ExitTime*. *KernelTime* enthält den Wert in 100 Nanosekunden Intervallen. Gilt auch für *UserTime*.

Mit Hilfe dieser Funktion lässt sich nun genau ermitteln wie viel Zeit der Thread wirklich mit dem Ausführen von Code verbracht hat. Auszug aus dem Demo *ThreadTimes*:

```
function Thread(p: Pointer): Integer;
var
  lblCounter: TStaticText;
  lblTickCount, lblThreadTimes: TStaticText;
  Counter: Integer;
  TickCountStart, TickCountEnd: Cardinal;
```

Parameter	Bedeutung
dwMilliseconds	Dauer in Millisekunden der Unterbrechung.

Tab. 8.7: Parameter Sleep

```
  KernelTimeStart, KernelTimeEnd: FileTime;
  UserTimeStart, UserTimeEnd: FileTime;
  Dummy: TFileTime;
  KernelTimeElapsed, UserTimeElapsed, TotalTime: Int64;
begin
  // Variablen initialisieren
  result := 0;
  Counter := 1;
  lblCounter := PTThreadParameter(p)^.FLabel;
  lblTickCount := PTThreadParameter(p)^.FlblTickCount;
  lblThreadTimes := PTThreadParameter(p)^.FlblThreadTimes;
  // Anfangszeit stoppen
  TickCountStart := GetTickCount();
  GetThreadTimes(GetCurrentThread, Dummy, Dummy, KernelTimeStart, UserTimeStart);
  // Code ausführen
  while (Counter < MAXCOUNT + 1) and (bRunning = 0) do
  begin
    lblCounter.Caption := IntToStr(Counter);
    Inc(Counter);
    sleep(1);
  end;
  // Endzeit stoppen
  GetThreadTimes(GetCurrentThread, Dummy, Dummy, KernelTimeEnd, UserTimeEnd);
  TickCountEnd := GetTickCount();
  // mit FileTime soll man laut PSDK nicht rechnen,
  // Konvertierung nach Int64
  // Berechnung der Differenz
  KernelTimeElapsed := Int64(KernelTimeEnd) - Int64(KernelTimeStart);
  UserTimeElapsed := Int64(UserTimeEnd) - Int64(UserTimeStart);
  TotalTime := (KernelTimeElapsed + UserTimeElapsed) div 10000;
  // Ausgabe der Ergebnisse
  lblTickCount.Caption := IntToStr(TickCountEnd - TickCountStart);
  lblThreadTimes.Caption := IntToStr(TotalTime);
  // freigeben des Speichers für die Struktur auf der der zeiger zeigt
  FreeMem(p);
end;
```

Die Beispielanwendung *ThreadTimes* (siehe Abbildung 8.2, Seite 50) demonstriert dies. In einem Label wird ein Zähler hochgezählt. Betätigt man die Schaltfläche „Stop“, wird die Zählschleife verlassen und die Ausführungszeiten genommen. Ein *Sleep(0)* sorgt dafür, dass man auch bei relativ kurzen Zeitspannen etwas „sieht“.

Parameter	Bedeutung
CreationTime	Zeitpunkt, an dem der Thread erzeugt wurde.
ExitTime	Zeitpunkt, an dem der Thread beendet wurde. Wird der Thread noch ausgeführt, ist der Wert undefiniert.
KernelTime	... gibt an wie viel Zeit der Thread Code des Betriebssystems ausgeführt hat.
UserTime	... gibt an wie viel Zeit der Thread Anwendungscode ausgeführt hat.

Tab. 8.8: Parameter GetThreadTimes

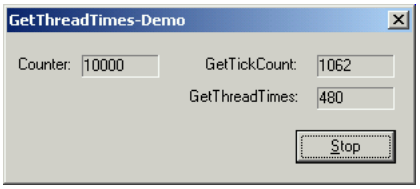

Abb. 8.2: Programmoberfläche *ThreadTimes*

8.4 Thread-Prioritäten

Wie schon gesagt, bekommt jeder Thread ungefähr 20 Millisekunden Rechenzeit der CPU. Dies gilt allerdings nur, wenn alle Threads die gleiche Priorität hätten, dem ist aber nicht so. Jeder Thread besitzt auch eine Priorität, das heißt „Wichtigkeit“. Diese Priorität bestimmt, welcher Thread als nächstes von der Ablaufsteuerung bei der Zuteilung von Rechenzeit berücksichtigt wird.

Windows kennt Prioritätsstufen zwischen 0 und 31, wobei 0 die niedrigste Priorität ist und 31 die höchste. Die Ablaufsteuerung berücksichtigt nun erst alle Threads mit der Prioritätsstufe 31. Findest es keine zuteilungsfähigen Threads mehr mit der Prioritätsstufe 31, kommen die anderen Threads dran. Jetzt könnte man davon ausgehen, dass Threads mit niedrigerer Priorität gar keine Rechenzeit mehr bekommen. Dies trifft nicht zu, da sich die meisten Threads im System im nichtzuteilungsfähigen Zustand befinden.

Beispiel: So lange in der Nachrichtenschlage keine Nachricht auftaucht, die von GetMessage übermittelt werden könnte, verbleibt der primäre Thread im angehaltenen Zustand. Wird nun eine Nachricht eingereiht, merkt das System dies und versetzt den Thread in den zuteilungsfähigen Zustand. Wird nun kein zuteilungsfähiger Thread mit einer höheren Priorität gefunden, wird dem Thread Rechenzeit zugewiesen.

8.4.1 Darstellung der Prioritäten

Die Windows-API definiert eine abstrakte Schicht oberhalb der Ablaufsteuerung, das heißt die Anwendung / der Programmierer kommuniziert niemals direkt mit der Ablaufsteuerung. Stattdessen werden Windows-Funktionen aufgerufen, die die übergebenen Parameter je nach verwendeten Betriebssystem interpretieren.

Bei der Wahl der Prioritätsklasse sollte man unter anderem berücksichtigen, welche anderen Anwendungen eventuell noch parallel mit der eigenen ausgeführt werden. Des weiteren hängt die Wahl der Prioritätsklasse auch von der Reaktionsfähigkeit des Threads ab. Diese Angaben sind sehr vage, aber sie hängen eben von der individuellen Aufgabe des Threads ab und müssen dementsprechend gewählt werden.

Prozess-Prioritäten

Windows kennt sechs Prioritätsklassen: „Leerlauf“, „Normal“, „Niedriger als normal“, „Normal“, „Höher als normal“, „Hoch“, „Echtzeit“. Siehe dazu Tabelle 8.9 auf Seite 51.

Prioritätsklasse	Beschreibung
Echtzeit	Die Threads in diesem Prozess müssen unmittelbar auf Ereignisse reagieren können, um zeitkritische Aufgaben durchführen zu können. Threads dieser Klasse, können sogar mit Threads des Betriebssystems konkurrieren. Nur mit äußerster Vorsicht benutzen.
Hoch	Es gilt das gleiche wie für die Klasse „Echtzeit“, nur dass Prozesse mit dieser Priorität unter der Priorität von Ëchtzeitlliegen. Der Taskmanger wird zum Beispiel mit dieser Priorität ausgeführt, um eventuell noch andere Prozesse beenden zu können.
Höher als normal	Die Ausführung erfolgt mit einer Priorität zwischen „Normal“ und „Hoch“. (neu in Windows2000)
Normal	Es werden keine besonderen Anforderungen an die Rechenzeitzuteilung gestellt.
Niedriger als normal	Die Ausführung erfolgt mit einer Priorität zwischen „Normal“ und „Leerlauf“. (neu in Windows2000)
Leerlauf	Threads in diesem Prozess werden nur ausgeführt, wenn es sonst nichts zu tun gibt. Beispielsweise Bildschirmschoner oder Hintergrundprogramme wie der Indexdienst.

Tab. 8.9: Beschreibung der Prozess–Prioritätsklassen

Die Prioritätsklasse „Hoch“ sollte nur dann Verwendung finden, wenn es unbedingt nötig ist, da auch Systemprozesse mit dieser Prioritätsklasse laufen. Die Threads des Explorers werden mit dieser Priorität ausgeführt. Dadurch wird gewährleistet, dass die Shell eine hohe Reaktionsfähigkeit besitzt und noch entsprechend reagiert, wenn andere Prozesse ansonsten das System belasten. Da die Explorer-Threads aber die meiste Zeit im angehaltenen Zustand verweilen belasten sie das System nicht, sind aber trotzdem „zur Stelle“ wenn der Benutzer sie braucht.

Die Prioritätsklasse „Echtzeit“ sollte hingegen sogar vermieden werden, da sie dann eine höhere Priorität als die System-Threads / –Prozesse hätten und sie so das Betriebssystem bei der Durchführung seiner Aufgaben behindern würden. Es könnten zum Beispiel E/A Operationen oder Netzwerkübertragungen ausgebremst werden. Auch Tastatur- und / oder Mauseingaben könnten unter Umständen nicht schnell genug verarbeitet werden, so dass für den Benutzer der Eindruck entstünde, dass das System blockiert sei. Die Prioritätsklasse „Echtzeit“ sollte nur dann Verwendung finden, wenn der Thread / Prozess die meiste Zeit nicht zuteilungsfähig ist und / oder der auszuführende Code schnell abgearbeitet ist.

Hinweis: Ein Prozess kann nur dann die Prioritätsklasse „Echtzeit“ zugewiesen werden, wenn der angemeldete Benutzer über die Berechtigung „Anheben der Zeitplanungspriorität“

verfügt. Standardmäßig haben diese Berechtigung Mitglieder der Gruppe Administratoren und Hauptbenutzer.

Die meisten Threads gehören der Prioritätsklasse „Normal“ an. Die Prioritätsklassen „Höher als normal“ und „Niedriger als normal“ wurden von Microsoft neu bei Windows2000 hinzugefügt, weil sich Firmen beklagt hatten, dass die bisher verfügbaren Prioritäten nicht flexibel genug wären.

Auf Abbildung (Abb. 8.3, Seite 52 sieht man den Taskmanagers von Windows 2000 mit Darstellung der Basisprioritätsklassen der Prozesse.

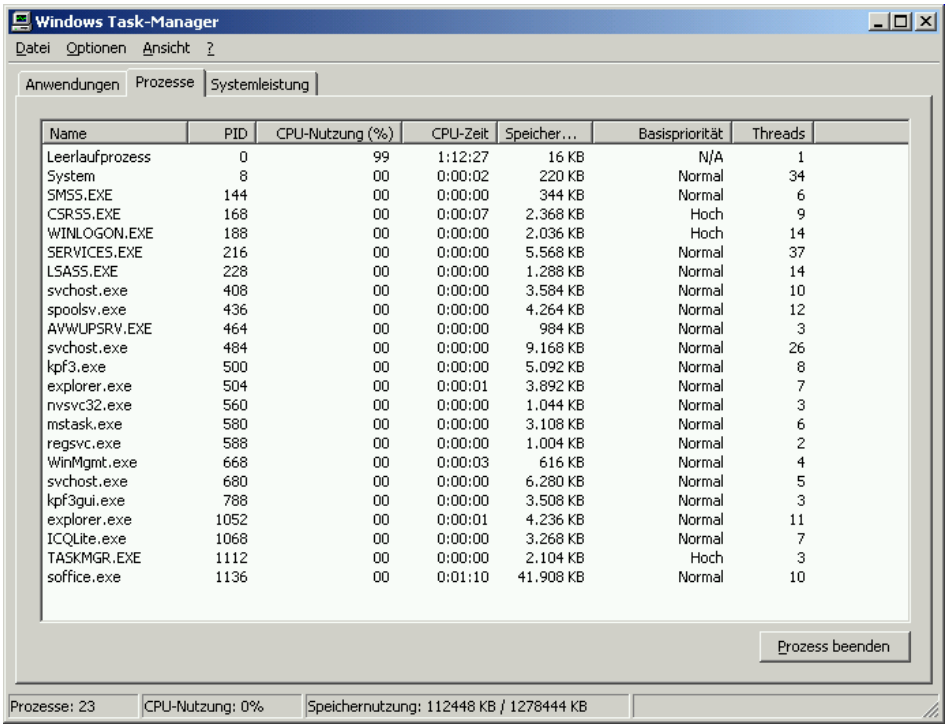

Abb. 8.3: Ansicht des Taskmanagers von Windows 2000 mit Prioritätsklassen

Wie man sieht, läuft der Taskmanager selber mit der Basisprioritätsklasse „Hoch“. Damit hat er die Möglichkeit immer noch zu reagieren, auch wenn andere Prozesse dies nicht mehr tun. Und hier besteht auch die Gefahr, wenn man einem Prozess die Basispriorität „Echtzeit“ zuweist. Dieser Prozess hätte dann eine höhere Priorität als der Taskmanager.

Thread-Prioritätsklassen

Windows unterstützt sieben Thread-Prioritätsklassen (siehe Tabelle 8.10, Seite 53): „Leerlauf“, „Minimum“, „Niedriger als normal“, „Normal“, „Höher als normal“, „Maximum“, „Zeitkritisch“. Diese Prioritäten sind relativ zur Prioritätsklasse des Prozesses definiert.

Die Basispriorität bezeichnet die durch die Prioritätsklasse des Prozesses festgelegte Prioritätsstufe.

Relative Thread-Priorität	Beschreibung
Zeitkritisch	Bei geltender Prioritätsklasse Ëchtzeitärbeitet der Thread mit der Priorität 31, bei allen anderen Prioritätsklassen mit Priorität 15.
Maximum	Der Thread arbeitet mit zwei Prioritätsstufen über der Basispriorität1.
Höher als Normal	Der Thread arbeitet mit einer Prioritätsstufe über der Basispriorität.
Normal	Der Thread arbeitet mit der Basispriorität.
Niedriger als normal	Der Thread arbeitet mit einer Prioritätsstufe unterhalb der Basispriorität.
Minimum	Der Thread arbeitet mit zwei Prioritätsstufen unterhalb der Basispriorität.
Leerlauf	Bei geltender Prioritätsklasse Ëchtzeit„' arbeitet der Thread mit Priorität 16, bei allen anderen Prioritätsklassen mit Priorität 1.

Tab. 8.10: Beschreibung der Thread-Prioritätsklassen

Die Zuordnung der prozess-bezogenen Prioritätsklassen und der relativen Thread-Prioritäten zu einer Prioritätsstufe übernimmt das Betriebssystem. Diese Zuordnung wird von Microsoft nicht festgeschrieben, damit Anwendungen unter Umständen auch lauffähig belieben, falls Microsoft an dieser Stelle etwas ändert. Tatsächlich hat sich diese Zuordnung beim Wechsel zwischen den verschiedenen Versionen des Betriebssystems geändert.

Relative Thread-Priorität	Prioritätsklasse des Prozesses					
	Leerlauf	Niedriger als normal	Normal	Höher als normal	Hoch	Echtzeit
Zeitkritisch	15	15	15	15	15	31
Maximum	6	8	10	12	15	26
Höher als normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Niedriger als normal	3	5	7	9	12	23
Maximum	2	4	6	8	11	22
Leerlauf	1	1	1	1	1	16

Abb. 8.4: Zuordnung der prozess-bezogenen Prioritätsklassen und den relativen Thread-Prioritäten

Beispiel für die Tabelle (Abb.: 8.4, Seite 53):
Ein normaler Thread in einem Prozess mit hoher Priorität besitzt die Prioritätsstufe 13. Wird nun die Prioritätsklasse des Prozesses von „Hoch“ in „Leerlauf“ geändert, erhält der Thread die Prioritätsstufe 4. Grund: Die Thread-Prioritäten sind relativ zur Prioritätsklasse des Prozesses definiert. Wird die Prioritätsklasse eines Prozesses geändert, hat dies keine Auswirkung auf die relative Thread-Priorität, sondern nur auf die Zuordnung zu einer Prioritäts-

stufe. Die prozess-bezogene Prioritätsklasse stellt eine Abstraktion von Microsoft dar, um den internen Mechanismus der Ablaufsteuerung von der Anwendungsprogrammierung zu trennen.

8.4.2 Programmieren von Prioritäten

festlegen der Prioritätsklasse

Beim Aufruf von *CreateProcess* kann die gewünschte Prioritätsklasse über den Parameter *dwCreationFlags* festgelegt werden. Zulässige Werte sind in der folgenden Tabelle (siehe 8.11, Seite 54) aufgelistet.

Prioritätsklasse	Konstante
Echtzeit	REAL_TIME_PRIORITY_CLASS
Hoch	HIGH_PRIORITY_CLASS
Höher als normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Niedriger als normal	BELOW_NORMAL_PRIORITY_CLASS
Leerlauf	IDLE_PRIORITY_CLASS

Tab. 8.11: Konstanten der Prozess-Prioritäten

Hinweis: Delphi 6 kennt die Konstanten für die Prioritätsklassen ABOVE_NORMAL_PRIORITY_CLASS und BELOW_NORMAL_PRIORITY_CLASS nicht. Grund ist der, dass diese erst ab Windows2000 verfügbar sind und von Borland in der Unit Windows.pas nicht berücksichtigt wurden. Abhilfe schafft eine eigene Deklaration dieser Konstanten:

```
BELOW_NORMAL_PRIORITY_CLASS = \$4000;
ABOVE_NORMAL_PRIORITY_CLASS= \$8000;
```

Erzeugt ein Prozess einen untergeordneten Prozess, so erbt dieser die Prioritätsklasse des Erzeuger Prozesses. Nachträglich kann die Prioritätsklasse mit der Funktion *SetPriorityClass* geändert werden.

```
function SetPriorityClass(hProcess: THandle; dwPriorityClass: DWORD): BOOL;
    stdcall;
```

Parameter	Bedeutung
hProcess	Handle für den Prozess
dwPriorityClass	Konstante für die Priorität. (Ein Wert aus Tabelle 8.11, Seite 54.)

Tab. 8.12: Parameter SetPriorityClass

Das Gegenstück dazu ist die Funktion *GetPriorityClass*, welchen die Prioritätsklasse von dem mit *hProcess* angegebenen Prozess zurück gibt.

```
function GetPriorityClass(hProcess: THandle): DWORD; stdcall;
```

Parameter	Bedeutung
hProcess	Handle für den Prozess

Tab. 8.13: Parameter GetPriorityClass

Rückgabewert ist die Priorität als eine Konstante aus Tabelle 8.9 (Seite 51).

Festlegen der Thread-Priorität

Da *CreateThread* keine Möglichkeit bietet die Thread-Priorität festzulegen, wird jeder neu erzeugte Thread mit der relativen Priorität „Normal“ erzeugt. Die relative Thread-Priorität kann aber mit der Funktion *SetThreadPriority* festgelegt werden.

```
function SetThreadPriority(hThread: THandle; nPriority: Integer): BOOL; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads
nPriority	Priorität (Ein Wert aus Tabelle 8.15, Seite 55.)

Tab. 8.14: Parameter SetThreadPriority

Parameter	Bedeutung
Zeitkritisch	THREAD_PRIORITY_TIME_CRITICAL
Maximum	THREAD_PRIORITY_HIGHEST
Höher als normal	THREAD_PRIORITY_ABOV_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Niedriger als normal	THREAD_PRIORITY_BELOW_NORMAL
Minimum	THREAD_PRIORITY_LOWEST
Leerlauf	THREAD_PRIORITY_IDLE

Tab. 8.15: Konstanten für die Thread-Prioritär

Will man nun einen Thread mit einer anderen Priorität erzeugen, so erzeugt man ihn mit *CreateThread* (oder dem Wrapper *BeginThread*) im angehaltenen Zustand CREATE_SUSPENDED), setzt mit *SetThreadPriority* die Priorität und ruft dann *ResumeThread* auf, um ihn zuteilungsfähig zu machen.

Mit dem Gegenstück zu *SetThreadPriority*, *GetThreadPriority*, lässt sich die aktuelle Thread-Priorität ermitteln.

```
function GetThreadPriority(hThread: THandle): Integer; stdcall;
```

Rückgabewert ist die Priorität als eine Konstante aus Tabelle 8.15, Seite 55.

Hinweis: Windows stellt keine Funktion zur Verfügung, mit der sich die Prioritätsstufe eines Threads ermitteln lässt. Microsoft behält es sich vor den Algorithmus zur Ermittlung der Prioritätsstufe zu ändern. Es sollte also keine Anwendung entworfen werden, die eine

Parameter	Bedeutung
hProcess	Handle für den Prozess

Tab. 8.16: Parameter GetThreadPriority

Kenntnis darüber erfordert. Eine Beschränkung auf die Prioritätsklasse und der relativen Thread-Priorität sollte sicherstellen, dass die Anwendung auch unter zukünftigen Windows-Versionen lauffähig ist.

Prioritätsanhebung durch das System

Wie schon gesagt, ermittelt das System die Prioritätsstufe eines Threads durch die Kombination der Prioritätsklasse des übergeordneten Prozesses mit der relativen Thread-Priorität. Es kann aber die Prioritätsstufe dynamisch anheben in Folge eines E/A Ereignisses, Einreihen einer Fensternachricht in die Warteschlage oder bei einem Zugriff auf die Festplatte.

Beispiel:
Ein Thread mit der Priorität „Normal“, der in einem Prozess mit der Basispriorität „Hoch“ läuft hat die Prioritätsstufe 13. In Folge eines Tastendrucks, welcher dazu führt, dass eine WM_KEYDOWN Nachricht in die Warteschlage eingereiht wird, wird der Thread nun zuteilungsfähig und der Tastaturtreiber kann das System veranlassen die Prioritätsstufe dieses Threads um zwei Stufen zu erhöhen. Für die Dauer der nächsten Zeitscheibe läuft dieser Thread also mit der Prioritätsstufe 15. Über die nächsten Zeitscheiben hinweg erfolgt dann eine stufenweise Absenkung der Prioritätsstufe um jeweils eins, bis der Thread wieder bei seiner Basisprioritätsstufe angekommen ist.

Es ist zu beachten, dass die Prioritätsstufe niemals unter der Basisprioritätsstufe abgesenkt werden kann. Des weiteren behält es sich Microsoft vor, dieses Konzept in Zukunft zu ändern / zu verbessern, um ein optimales Reaktionsverhalten des Gesamtsystems zu erzielen. Aus diesem Grund ist es von Microsoft auch nicht dokumentiert.

Eine dynamische Prioritätsanhebung erfolgt nur bei Threads mit einer Prioritätsstufe zwischen 1 und 15. Deshalb bezeichnet man diese Prioritätsstufen auch als dynamische Prioritäten. Es erfolgt auch nie eine dynamische Anhebung in den Echtzeitbereich, da viele Systemprozesse in diesem Bereich laufen und so verhindert wird, dass diese Threads mit dem Betriebssystem konkurrieren.

Diese dynamische Anhebung kann, laut einiger Entwickler, zu einem nachteiligen Leistungsverhalten der eigenen Anwendung führen. Microsoft hat daraufhin zwei Funktionen eingeführt, um die dynamische Prioritätsstufenanhebung zu deaktivieren.

```
function SetProcessPriorityBoost(hThread: THandle; DisablePriorityBoost: Bool):
    BOOL; stdcall;

function SetThreadPriorityBoost(hThread: THandle; DisablePriorityBoost: Bool):
    BOOL; stdcall;
```

Mit den Gegenstücken

Parameter	Bedeutung
hThread	Handle des Prozesses / Threads
DisablePriorityBoost	Aktivieren oder deaktivieren der dynamischen Prioritätsanhebung

Tab. 8.17: Parameter SetProcessPriorityBoost / SetThreadPriorityBoost

```
function GetProcessPriorityBoost(hThread: THandle; var DisablePriorityBoost: Bool
    ): BOOL; stdcall;

function GetThreadPriorityBoost(hThread: THandle; var DisablePriorityBoost: Bool)
    : BOOL; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Prozesses / Threads
var DisablePriorityBoost	Boolsche Variable die den Status aufnimmt

Tab. 8.18: Parameter GetProcessPriorityBoost / GetThreadPriorityBoost

lässt sich der Status (aktiv / inaktiv) der dynamischen Prioritätsanhebung abfragen.

Es gibt noch eine weitere Situation in der eine dynamische Prioritätsstufenanhebung erfolgen kann. Gegeben sei folgendes Szenario: Thread A läuft mit der Priorität von 4. Im System existiert noch ein Thread B mit der Prioritätsstufe 8, der ständig zuteilungsfähig ist. In dieser Konstellation würde Thread A nie Rechenzeit zugeteilt bekommen. Bemerkt das System, dass ein Thread über drei bis vier Sekunden derart blockiert wird, setzt es die Prioritätsstufe des Threads für zweit aufeinander folgende Zeitscheiben auf die Stufe 15 und senkt sie danach sofort wieder die Basisprioritätsstufe ab.

8.4.3 Beispielanwendung *Priority-Demo*

Anhand dieser Beispielanwendung kann mit den Prioritäten experimentiert werden. Es empfiehlt sich, zwei Instanzen des Programms zu starten und den Taskmanager nebenbei zu öffnen, um zu beobachten, wie sich die verschiedenen Kombinationen der Prioritäten auf das Verhalten der Anwendungen auswirken.

Ein Klick auf den Button „Suspend“ erzeugt einen zweiten Thread, welcher einfach nur eine Messagebox anzeigt und den primären Thread lahmlegt. Dadurch kann das Hauptfenster keine Fensternachrichten mehr verarbeiten, auch kein WM_PAINT, was durch Verschieben der Messagebox deutlich wird. Die macht hinreichend deutlich, dass sich der primäre Thread im angehaltenen Zustand befindet.

Anmerkung:
Seltsamerweise scheint die Messagebox nicht angezeigt zu werden, wenn das Programm aus der IDE gestartet wird. Startet man die Anwendung direkt aus dem Explorer, funktioniert sie wie gewünscht.

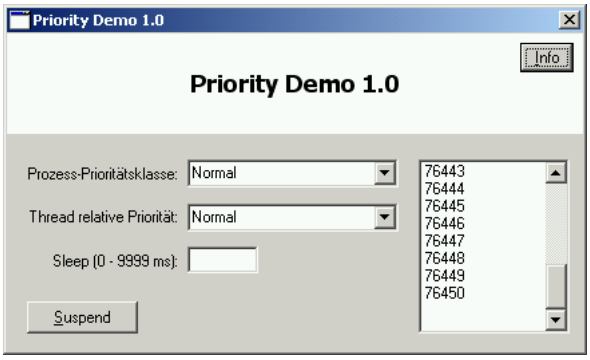

Abb. 8.5: Beispielanwendung “Priority-Demo“

8.5 Thread-Synchronisation

Im System haben alle Threads Zugang zu allen Systemressourcen, ansonsten könnten sie ihre Arbeit nicht verrichten. Da es aber nun nicht sein darf, dass jeder Thread, zu jeder Zeit, jede Ressource manipulieren kann – während ein Thread eine Ressource liest, darf kein anderer Thread im gleichen Moment schreibend auf die Ressource / Speicherblock zugreifen – muss der Zugriff „synchronisiert“ werden. Die Synchronisation im Benutzermodus kann entweder über einen atomaren Variablenzugriff mit den Interlocked–Funktionen erfolgen, wenn es sich um einen 32-Bit-Wert handelt oder über kritische Abschnitte, wenn es komplexe Datenstrukturen sind.

8.5.1 Atomarer Variablenzugriff

Unter atomaren Zugriffen versteht man, dass ein Thread mit der Gewähr auf eine Ressource zugreifen kann, ohne dass eine anderer Thread gleichzeitig auf sie zugreift. Ein Beispiel:

```
var
  g_Int: Integer = 0;
function ThreadOne(p: Pointer): Integer;
begin
  Inc(g_Int);
  result := 0;
end;

function ThreadTwo(p: Pointer): Integer;
begin
  Inc(g_Int);
  result := 0;
end;
```

Hier inkrementieren zwei Threads eine globale Integer-Variable um 1. Nun, was erwartet man, wenn beide Threads die Variable bearbeitet haben? Dass sie den Wert 2 hat, da sie ja in beiden Threads um 1 erhöht wurde. Aber muss das so sein? Nehmen wir an der Compiler generiert folgenden Code:

```
    mov eax, [g_Int]    // Wert von g_Int in Register kopieren
    inc eax             // Wert im Register inkrementieren
    mov [g_Int], eax    // Neuen Wert wieder in g_Int speichern
```

Führen nun beide Threads ihren Code nacheinander aus, passiert folgendes:

```
    mov eax, [g_Int]    // Thread 1: Kopiert 0 in Register
    inc eax             // Thread 1: Inkrementiert Registerinhalt um 1
    mov [g_Int], eax    // Thread 1: Speichert 1 in g_Int
    mov eax, [g_Int]    // Thread 2: Kopiert 1 in Register
    inc eax             // Thread 2: Inkrementiert Registerinhalt um 1
    mov [g_Int], eax    // Thread 2: Speichert 2 in g_Int
```

Das Ergebnis ist wie erwartet 2. Aber muss das so ablaufen? Windows ist ja ein präemptives Multitasking Betriebssystem, es ist also nicht gewährleistet, dass beide Threads schön nach einander durchlaufen. Wie könnte es aussehen, wenn ein Thread unterbrochen wird?

```
    mov eax, [g_Int]    // Thread 1: Kopiert 0 in Register
    inc eax             // Thread 1: Inkrementiert Registerinhalt um 1

    mov eax, [g_Int]   // Thread 2: Kopiert 0 in Register
    inc eax             // Thread 2: Inkrementiert Registerinhalt um 1
    mov [g_Int], eax   // Thread 2: Speichert 1 in g_Int

    mov [g_Int], eax    // Thread 1: Speichert 1 in g_Int
```

Und schon lautet unser Ergebnis nicht mehr 2 sondern 1! Das Ergebnis hängt von vielerlei Gegebenheiten ab: Wie wird der Code vom Compiler generiert? Welcher Prozessor führt den Code aus? Wie viele Prozessoren gibt es im System? Windows wäre als Multitasking Betriebssystem nicht zu gebrauchen, wenn es für unser Problem keine Lösung bereitstellen würde.

Und zwar stellt Windows die Interlocked-Funktionen bereit. Alle Funktionen dieser Familie haben eins gemeinsam, sie bearbeiten einen Wert auf atomare Weise. Nehmen wir als Beispiel mal die Funktion InterlockedExchangeAdd() (weitere Funktionen der Interlocked-Familie finden sich in den Demos: InterlockedExchangeAdd_Demo und SpinLock):

```
function InterlockedExchangeAdd(Addend: PLongint; Value: Longint): Longint
    stdcall; overload;

function InterlockedExchangeAdd(var Addend: Longint; Value: Longint): Longint
    stdcall; overload;
```

Parameter	Bedeutung
Addend (Plongint)	Zeiger auf ein Integer, der den neuen Wert aufnimmt.
Addend (Longint)	Variable, die den neuen Wert aufnimmt.
Value	Wert, um den inkrementiert werden soll.

Tab. 8.19: Parameter InterlockedExchangeAdd

InterlockedExchangeAdd inkrementiert einfach den ersten Parameter um den in Value angegebenen Wert und stellt sicher, dass der Zugriff atomar erfolgt. Unseren obigen Code können wir also wie folgt umschreiben:

```
function ThreadOne(p: Pointer): Integer;
begin
  InterlockedExchangeAdd(g_Int, 1);
  result := 0;
end;

function ThreadTwo(p: Pointer): Integer;
begin
  InterlockedExchangeAdd(g_Int, 1);
  result := 0;
end;
```

und so sicherstellen, dass unsere globale Variable *g_Int* korrekt und wie erwartet inkrementiert wird. Wichtig dabei ist, dass alle beteiligten Threads die Änderung an der Variablen nur atomar vornehmen.

Hinweis: Die Interlocked-Funktionen sind zu dem auch noch sehr schnell. Ein Aufruf einer Interlocked-Funktion verbraucht während ihrer Ausführung nur einige CPU-Befehlszyklen (in der Regel weniger als 50) und es findet kein Wechsel vom Benutzer- in den Kernel-Modus statt, was gewöhnlich mehr als 1000 Zyklen benötigt.

Werfen wir noch einen Blick auf die Funktion *InterlockedExchange()*:

```
function InterlockedExchange(var Target: Integer; Value: Integer): Integer;
    stdcall;
```

Parameter	Bedeutung
var Target	Variable, die den Ziel-Wert aufnimmt
Value	Wert für das Ziel

Tab. 8.20: Parameter InterlockedExchange

Diese Funktion ersetzt den Wert, der in *Target* steht, durch den Wert der in *Value* steht. Das besondere ist, dass *InterlockedExchange* den ursprünglichen Wert von *Target* als Funktions-Ergebnis zurückliefert. Damit lässt sich ein so genanntes SpinLock implementieren:

```
var
  g_IsInUse: Integer = 1;  // Status Flag; 1: in Gebrauch, 0: frei

function WatchDogThread(p: Pointer): Integer;
begin
  //  läuft so lange g_IsInUse 1 ist
  while InterlockedExchange(g_IsInUse, 1) <> 0 do
    Sleep(0);
  //  g_IsInUse ist 0 geworden, Schleife wurde verlassen
  writeln('frei.');
  result := 0;
end;
```

Die while-Schleife wird immer wieder durchlaufen, wobei sie den Wert von *g_IsInUse* auf 1 setzt und prüft, ob er vorher ungleich 0 war (1 bedeutet Ressource ist in Gebrauch, 0 auf Ressource kann zugegriffen werden). Ein anderer Thread bearbeitet die zu überwachende Ressource und setzt *g_IsInUse* auf 0, wenn er damit fertig ist. Damit liefert die while-Schleife irgendwann einmal *False*, die Ressource ist frei und wird verlassen. Dieses Vorgehen bezeichnet man als SpinLock.

Diese Technik ist allerdings nicht sehr effizient, da ein SpinLock viel Rechenzeit verbraucht, denn es muss ja ständig ein Wert vergliechen werden, der auf „wundersame“ Weise an anderer Stelle geändert wird. Außerdem setzt dieser Code voraus, dass alle Threads, die mit dem SpinLock arbeiten auf der gleichen Prioritätsstufe laufen.

Hinweis: Es gibt keine Interlocked-Funktion, die nur einen Wert liest, da eine solche Funktion überflüssig ist. Wenn ein Thread einen Wert liest, deren Inhalt mit einer Interlocked-Funktion geändert wurde, handelt es sich immer um einen brauchbaren Wert.

8.5.2 Kritische Abschnitte

Handelt es sich bei den Ressourcen nicht um reine 32-Bit Werte, dann kommt man nicht mehr mit den Interlocked-Funktionen aus - hier helfen jetzt kritische Abschnitte. Ein kritischer Abschnitt stellt einen Bereich innerhalb des Programmcodes dar, der bei seiner Ausführung exklusiven Zugriff auf eine oder mehrere Ressourcen benötigt. Hier mal ein Beispiel, wie kritische Abschnitte eingesetzt werden:

```
var
  g_Index: Integer = 0;
  g_cs: RTL_CRITICAL_SECTION;
  F: TextFile;
function FirstThread(p: Pointer): Integer;
begin
  result := 0;
  while g_Index < MAX_TIMES do
  begin
    // in CriticalSection eintreten, Ressource sperren
    EnterCriticalSection(g_cs);
    writeln(F, IntToStr(GetTickCount()));
    Inc(g_Index);
    // CriticalSection verlassen
    LeaveCriticalSection(g_cs);
    Sleep(0);
  end;
end;

function SecondThread(p: Pointer): Integer;
begin
  result := 0;
  while g_Index < MAX_TIMES do
  begin
    // in CriticalSection eintreten, Ressource sperren
    EnterCriticalSection(g_cs);
    Inc(g_Index);
    writeln(F, IntToStr(GetTickCount()));
    // CriticalSection verlassen
```

```
    LeaveCriticalSection(g_cs);
    Sleep(0);
  end;
end;
```

Wie man sieht greifen beide Threads auf die gleiche Datei zu (F) und schreiben einen Wert (*GetTickCount()*) in sie hinein, gleichzeitig wird eine globale Variable (*g_Index*) erhöht, welche für die Abbruchbedingung benötigt wird. Damit beim Schreiben nichts durcheinander geht, sind in den Threads die Zugriffe auf die Datei und die globale Variable mit kritischen Abschnitten geschützt.

Wie funktionieren kritische Abschnitte nun? Es wurde eine Struktur des Typs CRITICAL_SECTION mit dem Namen *g_cs* angelegt und jede Befehlsfolge, die auf gemeinsame Ressourcen zugreift wird in einen Aufruf von *EnterCriticalSection* und *LeaveCriticalSection* geklammert, denen man einen Zeiger auf die CRITICAL_SECTION-Struktur als Parameter übergibt. Das Prinzip ist nun folgendes: Tritt nun ein Thread in einen kritischen Abschnitt ein oder versucht es, wird mittels der CRITICAL_SECTION-Struktur überprüft, ob nicht schon auf die Ressource / den Code-Abschnitt zugegriffen wird. Ist die Ressource „frei“, betritt es den kritischen Abschnitt und hängt ein „Besetzt“-Schild an die Tür. Hat es die Ressourcen manipuliert und verlässt den kritischen Abschnitt wieder, wird das „Besetzt“-Schild praktisch wieder mit *LeaveCriticalSection* umgedreht, so dass es jetzt wieder „frei“ zeigt.

Wichtig ist, wie auch bei den Interlocked-Funktionen, dass jeder Thread, der auf gemeinsame Ressourcen zugreift, dies nur innerhalb eines kritischen Abschnittes tut. Ein Thread, der dies nicht tut, missachtet quasi das „Besetzt“-Schild – er prüft es gar nicht erst – und könnte so die Ressourcen kompromittieren. Andersherum, vergisst ein Thread nach Manipulation der Ressourcen ein *LeaveCriticalSection* aufzurufen, ähnelt dies dem Verhalten beim Verlassen das „Besetzt“-Schild nicht wieder umzudrehen und so zu signalisieren, dass die Ressourcen wieder «frei» sind.

Wie funktionieren kritische Abschnitte

Wie funktionieren jetzt aber kritische Abschnitte? Grundlage ist die Datenstruktur CRITICAL_SECTIONS. Sie ist nicht dokumentiert, da Microsoft der Meinung ist, dass sie vom Programmierer nicht verstanden werden müsste. Dies ist auch nicht nötig, da man API-Funktionen aufruft, um sie zu manipulieren. Diese Funktionen kennen die Details, die zur Manipulation nötig sind und garantieren so den konsistenten Zustand der Struktur.

In der Regel werden Variablen dieser Struktur global angelegt damit alle Threads auf sie zugreifen können. Des weiteren ist es nötig, dass die Elemente der Struktur initialisiert werden. Dies geschieht mit dem Aufruf von *InitializeCriticalSection()*:

```
procedure InitializeCriticalSection(var lpCriticalSection: TRTLCriticalSection);
    stdcall;
```

Sobald die kritischen Abschnitte nicht mehr gebraucht werden, sollte die CRITICAL_SECTION-Struktur zurückgesetzt werden mit dem Aufruf von *DeleteCriticalSection*:

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 8.21: Parameter InitializeCriticalSection

```
procedure DeleteCriticalSection(var lpCriticalSection: TRTLCriticalSection);
    stdcall;
```

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 8.22: Parameter DeleteCriticalSection

Wie schon im Beispiel zu sehen war, muss jedem Codestück, welches auf eine gemeinsame Ressource zugreift ein Aufruf von *EnterCriticalSection* vorangehen.

```
procedure EnterCriticalSection(var lpCriticalSection: TRTLCriticalSection);
    stdcall;
```

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 8.23: Parameter EnterCriticalSection

EnterCriticalSection untersucht dabei die Elementvariablen der CRITICAL_SECTION-Struktur und entnimmt ihnen den Zustand der Ressourcen. Dabei geschieht folgendes:

- Greift gerade kein Thread auf die Ressource zu, aktualisiert *EnterCriticalSection* die Elemente entsprechend, um anzuzeigen, dass dem aufrufenden Thread Zugriff gewährt wurde.
- Zeigt das Element an, dass der aufrufende Thread schon Zugang zu den Ressourcen hat, wird der Zähler erhöht, der festhält, wie oft dem aufrufenden Thread schon Zugriff gewährt wurde.
- Stellt die Funktion *EnterCriticalSection* fest, dass die Ressource sich im Zugriff eines anderen Threads befindet, überführt *EnterCriticalSection* den aufrufenden Thread in den Wartezustand. Das System merkt sich das und sorgt dafür dass der aufrufende Thread sofort zuteilungsfähig wird, sobald die Ressourcen wieder freigegeben werden.

Am Ende eines Codestücks, das auf eine gemeinsame Ressource zugreift, muss die Funktion *LeaveCriticalSection* aufgerufen werden.

```
procedure LeaveCriticalSection(var lpCriticalSection: TRTLCriticalSection);
    stdcall;
```

Die Funktion decrementiert einen Zähler, der anzeigt wie oft dem aufrufenden Thread der Zugriff auf die Ressource gewährt wurde. Fällt der Zähler auf 0, prüft die Funktion, ob weitere Threads auf die Freigabe dieses kritischen Abschnittes warten. Ist dies der Fall überführt es einen wartenden Thread in den zuteilungsfähigen Zustand. Gibt es keine wartenden

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 8.24: Parameter LeaveCriticalSection

Threads, aktualisiert *LeaveCriticalSection* das Element ebenfalls, um anzuzeigen, dass im Moment kein Thread auf die Ressourcen zugreift.

Kritische Abschnitte und SpinLocks

Wird ein Thread in den Wartezustand überführt, hat dies zur Folge, dass er von den Benutzer-Modus in den Kernel-Modus überwechseln muss. Dies nimmt etwa 1000 CPU-Befehlszyklen in Anspruch. Um nun die Effizienz von kritischen Abschnitten zu verbessern kann man den kritischen Abschnitt mit *InitializeCriticalSectionAndSpinCount* initialisieren:

```
function InitializeCriticalSectionAndSpinCount(var lpCriticalSection:
    TRTLCriticalSection;
  dwSpinCount: DWORD): BOOL; stdcall;
```

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.
dwSpinCount	Anzahl für die Schleifendurchläufe für die Abfrage des SpinLocks.

Tab. 8.25: Parameter InitializeCriticalSectionAndSpinCount

Der Parameter *dwSpinCount* gibt dabei die Anzahl der Schleifendurchläufe der Abfrage des SpinLocks an. Also wie oft der Thread versuchen soll Zugriff auf die Ressourcen zu bekommen bevor er in den Wartezustand geht und so in den Kernel-Modus wechselt. Erfolgt der Aufruf auf einem Einzelprozessorsystem wird dieser Parameter ignoriert und auf 0 gesetzt, da es nutzlos ist in diesem Fall einen SpinLockzähler zusetzen: Der Thread, der die Ressource kontrolliert, kann die Kontrolle nicht abgeben, solange ein anderer Thread aktiv wartet, also das SpinLock testet.

Der SpinCount eines kritischen Abschnittes kann mit *SetCriticalSectionSpinCount* nachträglich geändert werden:

```
function SetCriticalSectionSpinCount(var lpCriticalSection: TRTLCriticalSection;
  dwSpinCount: DWORD): DWORD; stdcall;
```

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.
dwSpinCount	Anzahl für die Schleifendurchläufe für die Abfrage des SpinLocks.

Tab. 8.26: Parameter SetCriticalSectionSpinCount

Siehe dazu auch die Demos: *InterLockedExchangeAdd*, *CriticalSection*, *SpinLock*.

8.5.3 Thread-Synchronisation mit Kernel-Objekten

Im vorherigen Kapitel habe ich die Thread-Synchronisation im Benutzermodus besprochen. Ihr großer Vorteil ist, dass sie sich positiv auf das Leistungsverhalten auswirken. Bei der Synchronisation mit Kernelobjekten ist es nötig, dass der aufrufende Thread vom Benutzermodus in den Kernelmodus wechselt. Dies ist jedoch mit nicht unerheblichen Rechenaufwand verbunden. So braucht auf der x86-Plattform die CPU ca. 1.000 Befehlszyklen für jede Richtung des Moduswechsels. Die Thread Synchronisation im Benutzermodus hat jedoch den Nachteil, dass sie gewissen Einschränkungen unterliegt und einfach eine Funktionalität nicht bietet, die man aber das eine oder andere mal braucht. Beispiele hierzu findet man in den Demos WaitForSingleObject, HandShake, WaitForMultipleObject und in den folgenden Unterkapiteln.

Hier soll es nun um die Thread-Synchronisation mit Kernelobjekten gehen. Was Kernelobjekte sind habe ich weiter oben schon mal erläutert. Im Prinzip geht es in diesem Kapitel darum eben diese Kernelobjekte, in diesem Fall beschränke ich mich des Themas willen hauptsächlich auf Thread- und Prozessobjekte, miteinander zu synchronisieren.

Hinsichtlich der Thread–Synchronisation wird bei jedem Kernelobjekt zwischen dem Zustand signalisiert und nicht–signalisiert unterschieden. Nach welchen Regeln dies geschieht ist in Windows für jeden Objekttypen eigens festgelegt. So werden zum Beispiel Prozessobjekte immer im Zustand nicht-signalisiert erzeugt und gehen in den Zustand signalisiert über, wenn der der Prozess endet. Diesen Zustandswechsel übernimmt das Betriebssystem automatisch. Innerhalb des Prozessobjektes gibt es einen bool'schen Wert, der beim Erzeugen des Objektes mit FALSE (nicht-signalisiert) initialisiert wird. Bei der Beendigung des Prozesses ändert das Betriebssystem automatisch diesen bool'schen Wert und setzt ihn auf TRUE, um den signalisierten Zustand des Objektes anzuzeigen. Soll also Code erstellt werden, der prüft, ob ein bestimmter Prozess noch aktiv ist, braucht nur, mit der dazu zur Verfügung gestellten Funktion, dieser Wert abgefragt zu werden.

Im speziellen werde ich in diesem Kapitel darauf eingehen, wie es einem Thread ermöglicht werden kann, darauf zu warten, dass ein bestimmtes Kernel-Objekt signalisiert wird und wie so etwas zur Thread-Synchronisation genutzt werden kann

WaitForxxx-Funktionen

WaitForSingleObject Die Wait-Funktionen veranlassen einen Thread, sich freiwillig so lange in den Ruhezustand zu begeben, bis ein bestimmtes Kernel-Objekt signalisiert wird. Während der Thread wartet braucht er keine Rechenzeit, das Warten ist also äußerst ressourcenschonend.

Die wohl am meisten gebrauchte Wait-Funktion ist wohl WaitForSingleObject:

```
function WaitForSingleObject(hHandle: THandle; dwMilliseconds: DWORD): DWORD;
    stdcall;
```

Parameter	Bedeutung
hHandle	Handle des Kernel-Objektes auf das gewartet werden soll
dwMilliseconds	Zeit in Millisekunden die gewartet werden soll (INFINITE = keine zeitliche Begrenzung)
Rückgabewerte	WAIT_OBJECT_0: Objekt wurde signalisierten, WAIT_TIMEOUT: Zeitüberschreitung, WAIT_FAILED: Fehler beim Aufruf der Funktion (ungültiges Handle oder ähnliches)

Tab. 8.27: Parameter WaitForSingleObject

Im Demo-Programm *wfso* im Unterordner *WaitForSingleObject* habe ich die Benutzung von *WaitForSingleObject* demonstriert: program wfso;

```
{\$APPTYPE CONSOLE}

uses
  Windows;

function Thread(p: Pointer): Integer;
resourcestring
  rsReturn = 'Druecken Sie die Eingabetaste - oder auch nicht...';
begin
  Writeln(rsReturn);
  Readln;
  result := 0;
end;

var
  hThread: THandle;
  ThreadID: Cardinal;
  wf: DWORD;
resourcestring
  rsThreadStart = 'Thread wird gestartet:';
  rsWO = 'Thread wurde vom Benutzer beendet';
  rsWT = 'Thread wurde nicht innerhalb von 10 Sekunden beendet';
  rsWF = 'Fehler}';
begin
  Writeln(rsThreadStart);
  Writeln('');
  hThread := BeginThread(nil, 0, @Thread, nil, 0, ThreadID);
  if hThread <> INVALID_HANDLE_VALUE then
  begin
    wf := WaitForSingleObject(hThread, 5000);
    case wf of
      WAIT_OBJECT_0: Writeln(rsWO);
      WAIT_TIMEOUT: Writeln(rsWT);
      WAIT_FAILED: Writeln(rsWT);
    end;
    CloseHandle(hThread);
  end;
end.
```

Im Hauptprogramm wird ein Thread abgespalten, den der Benutzer durch Drücken der Eingabetaste beenden kann. Das Hauptprogramm wartet mit WaitForSingleObject auf die Signalisierung des abgespaltenen Threads und zwar genau fünf Sekunden lang. Oder anders

ausgedrückt: Der Aufruf von WaitForSingleObject weist das Betriebssystem an, den Thread so lange nicht als zuteilungsfähig zu behandeln, bis entweder der Thread signalisiert oder die fünf Sekunden überschritten wurden. Dann wertet es den Rückgabewert von WaitForSingleObject aus und gibt eine entsprechende Meldung in der Konsole aus. Betätigt der Benutzer die Eingabetaste wird der Thread normal beendet und das Thread-Objekt wird signalisiert; Rückgabewert ist WAIT_OBJECT_0. Macht der Benutzer gar nichts, bricht nach fünf Sekunden die Funktion *WaitForSingleObject* das Warten ab und kehrt mit WAIT_TIMEOUT zurück. Sollte ein Fehler auftreten, weil zum Beispiel das Handle ungültig war, gibt die Funktion WAIT_FAILED zurück.

Eine praktische Anwendung wäre, wenn zum Beispiel in einem Thread etwas initialisiert wird und der Hauptthread warten muss bis der abgespaltene Thread seine Arbeit getan hat.

WaitForMultipleObjects Des weiteren stellt Windows noch eine Funktion bereit, um den Signalisierungssatus mehrerer Kernel-Objekte gleichzeitig abzufragen: *WaitForMultipleObjects*. Siehe Demo *WaitForMultipleObjects*.

MsgWaitForMultipleObject Überlegen wir uns mal folgenden Ablauf: Der Hauptthread startet einen weiteren Thread in dem eine längere Berechnung zum Beispiel durchgeführt wird. Natürlich soll irgendwann mit dem Ergebnis dieser Berechnung weitergearbeitet werden. Die Auslagerung in den Thread erfolgte aus dem Grund, damit das Fenster noch reagiert und der Vorgang eventuell abgebrochen werden kann. Um auf den abgespaltenen Thread zu warten benutzen wir *WaitForSingleObject*. Unser Code sähe dann ungefähr so aus:

```
hThread := BeginThread(...);
wf := WaitForSingleObject(hThread, INFINITE);
    case wf of
      WAIT_OBJECT_0: Writeln(rsWO);
      WAIT_TIMEOUT: Writeln(rsWT);
      WAIT_FAILED: Writeln(rsWT);
    end;
```

So weit so gut. Nur leider haben wir jetzt wieder genau das Problem, was wir versucht haben mit dem Thread zu umgehen: Unser Fenster reagiert nicht mehr, weil es darauf wartet, dass *WaitForSingleObject* zurückkehrt. Eine Lösung wäre jetzt mit zwei Threads zu arbeiten: Vom Hauptthread wird Thread A abgespalten, der Thread B startet und auf dessen Rückkehr wartet. Da das warten auch in einem Thread geschieht, reagiert unser Fenster noch und wir haben unser Problem gelöst. Es geht jedoch auch einfacher und ressourcenschonender mit der API Funktion *MsgWaitForMultipleObjects*. Diese Funktion bewirkt, dass der Thread sich selbst suspendiert, bis bestimmte Ereignisse auftreten oder die TimeOut-Zeit überschritten wird. Diese Ereignisse können Fensternachrichten oder Benutzereingaben sein. Dies demonstriert ganz gut folgendes kleines Programm:

```
function Thread(p: Pointer): Integer;
var
  i                     : Integer;
begin
```

```
  for i := 0 to 99 do
  begin
    Form1.ListBox1.Items.Add(IntToStr(i));
    sleep(50);
  end;
  result := 0;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  hThread                : THandle;
  ThreadID               : Cardinal;
  WaitResult             : DWORD;
  Msg                    : TMsg;
begin
  hThread := BeginThread(nil, 0, @Thread, nil, 0, ThreadID);
  repeat
    WaitResult := MsgWaitForMultipleObjects(1, hThread, False, INFINITE,
        QS_MOUSEMOVE);
    if WaitResult = WAIT_OBJECT_0 + 1 then
    begin
      while PeekMessage(Msg, Handle, 0, 0, PM_REMOVE) do
      begin
        TranslateMessage(Msg);
        DispatchMessage(Msg);
      end;
    end;
  until WaitResult = WAIT_OBJECT_0
end;
```

Führt man dieses Programm aus, wird sich die Listbox nur füllen, wenn man die Maus über dem Fenster bewegt. Benutzt man anstatt dem Flag QS_MOUSEMOVE zum Beispiel QS_KEY wird der Thread nur ausgeführt, wenn eine Taste gedrückt wird. Setzen wir nun an dieser Stelle QS_ALLINPUT ein, wird unser Thread gar nicht suspendiert, durch die Schleife und dem Aufruf von *PeekMessage* im Falle das WAIT_OBJECT_0 + 1 zurückgeliefert wird, kann der Hauptthread auf Fensternachrichten abarbeiten und unser Fenster reagiert weiterhin. Welche Flags es gibt und was sie bewirken kann man im MSDN oder PSDK unter dem entsprechenden Stichwort nachlesen.

8.5.4 Ereignisobjekte

Ereignisse finden meist dann Verwendung, wenn ein Thread signalisieren will, dass es mit etwas fertig ist oder mit etwas begonnen hat und andere Threads über seinen Zustand informieren will.

Mit der Funktion *CreateEvent* erzeugt man ein Ereignisobjekt:

```
function CreateEvent(lpEventAttributes: PSecurityAttributes;
  bManualReset, bInitialState: BOOL; lpName: PChar): THandle; stdcall;
```

Als Rückgabewert gibt *CreateEvent* ein prozessbezogenes Handle auf das Ereignisobjekt zurück.

Parameter	Bedeutung
lpEventAttributes	Zeiger auf eine Sicherheitsattributstruktur
bManuelReset	Manuel-Reset (True) oder Auto-Reset
bInitialState	Erzeugung des Objektes im signalisierten Zustand oder nicht
lpName	Dient zur Identifizierung des Objektes

Tab. 8.28: Parameter CreateEvent

Hat man einen Namen für das Ereignisobjekt vergeben, so kann man wieder Zugriff auf selbiges erlangen mit der Funktion *OpenEvent*:

```
function OpenEvent(dwDesiredAccess: DWORD; bInheritHandle: BOOL; lpName: PChar):
    THandle; stdcall;
```

Parameter	Bedeutung
dwDesiredAccess	Sicherheitsbeschreiber für das Objekt
bInheritHandle	Soll das Handle vererbt werden oder nicht
lpName	Name des Events

Tab. 8.29: Parameter OpenEvent

Wie üblich sollte die Funktion *CloseHandle* aufgerufen werden, wenn das Objekt nicht mehr länger gebraucht wird.

So bald das Ereignisobjekt erzeugt wurde, kann man dessen Zustand mit den Funktionen *SetEvent* und *ResetEvent* kontrollieren:

```
function SetEvent(hEvent: THandle): BOOL; stdcall;

function ResetEvent(hEvent: THandle): BOOL; stdcall;
```

Als Parameter erwarten beide Funktionen jeweils ein gültiges Handle auf ein Ereignisobjekt. *SetEvent* ändert den Zustand des Ereignisobjektes in signalisiert und *ResetEvent* ändert ihn wieder in nicht-signalisiert.

Zusätzlich hat Microsoft noch einen Nebeneffekt für erfolgreiches Warten für Auto-Reset-Ereeignisse definiert: Ein Auto-Reset-Ereignisobjekt wird automatisch in den Zustand nicht-signalisiert zurückgesetzt, wenn ein Thread erfolgreich auf das Ereignis wartet. Es ist deshalb nicht nötig, extra noch ResetEvent für ein Auto-Reset-Ereignis aufzurufen. Im Gegensatz dazu hat Microsoft für ein manuelle Ereignisse keinen solchen Nebeneffekt definiert.

8.6 Der Stack eines Threads

Wenn ein neuer Prozess erzeugt wird, wird von Windows der Adressraum des Prozesses eingerichtet. Am einem Ende des Adressbereiches wird der Heap eingerichtet, der so genannte standardmäßige Heap eines Prozesses und am anderen Ende des Adressraumes der Stack des Hauptthreads. Der Heap ist ein dynamischer Speicherbereich, aus dem zur

Laufzeit eines Programmes zusammenhängende Speicherabschnitte angefordert und in beliebiger Reihenfolge wieder freigegeben werden können. Die Freigabe kann sowohl manuell als auch mit Hilfe einer automatischen Speicherbereinigung erfolgen. Eine Speicheranforderung vom Heap bzw. Freispeicher wird auch dynamische Speicheranforderung genannt.

Der Unterschied zum Stack (Stapel- oder Kellerspeicher) besteht darin, dass beim Stack angeforderte Speicherabschnitte in der umgekehrten Reihenfolge wieder freigegeben werden müssen, in der sie angefordert wurden. Dies geschieht aber automatisch durch das Betriebssystem.

Globale Variablen und Instanzen von Objekten werden auf dem Heap abegelegt, während hingegen lokale Variablen und Funktionsparameter auf dem Stack abgelegt werden. So werden zum Beispiel bei einem Funktionaufruf die Parameter, je nach Aufrufkonvention auf dem Stack geschoben, um dann von der aufgerufenen Funktion von dort wieder abgeholt zu werden. Deutlich wird dass, wenn man sich den Aufruf einer einfachen MessageBox mal in Assemblercode anguckt. Die MesageBox-Funktion ist wie folgt deklariert:

```
int MessageBox(
  HWND hWnd,
  LPCTSTR lpText,
  LPCTSTR lpCaption,
  UINT uType
);
```

Als ersten Parameter erwartet die Funktion das Handle des aufrufenden Fensters, der zweite und dritte sind der Text und der Fenstertitel und der letzte definiert die Schaltflächen und das Icon, was angezeigt werden soll. Was für Code generiert nun der Compiler? Gucken wir mal:

```
push 0 ; uType
push offset Titel ; lpCaption
push offset Message ; lpText
push 0
call MessageBoxA
```

Mit `push` werden die erforderlichen Aufrufparameter auf den Stack geschoben. Und wie man sieht, geschieht das mit dem letzten zuerst, also wenn man sich die Liste der Parameter nebeneinander vorstellt, von rechts nach links. Dann wird mit `call` die Funktion selber aufgerufen. Sie findet ihre Parameter auf dem Stack, da wir sie genauso abgelegt haben, wie sie es erwartet. Kein Problem, funktioniert, alles prima. Funktion wird aufgerufen, die erwartete Messagebox erscheint, wir klicken sie weg und die Funktion kehrt zurück. So, danach kommt nichts mehr. Und warum kommt nichts mehr? Was ist mit dem Stack? Da liegen doch noch unsere Parameter rum. Nein tun sie nicht, denn die Funktion hat hinter uns hergeräumt und das Aufräumen für uns übernommen und den Stack wieder aufgeräumt. Das Ablegen von rechts nach links und dass die Funktion selber wieder aufräumt, wird mit der Aufrufkonvention stdcall vereinbart.

Des weiteren werden die Rücksprungadressen von Funktionen auf dem Stack abgelegt. Da der Stack wie ein Stapel funktioniert – bevor ich das unterste Elemente wegnehmen kann, muss ich zu vor alle nachträglich auf dem Stack abgelegten Elemente vom Stapel nehmen – lässt sich so die Verschachtelung von Funktionen sehr einfach zurückverfolgen.

8.6.1 Verwaltung des Stacks

Jedes mal, wenn innerhalb eines Prozesses ein neuer Thread erzeugt wird, reserviert das Betriebssystem den für den privaten Stack des Threads erforderlichen Bereich im Adressraum des Pozesses und stellt ausserdem für diesen Bereich sogleich eine bestimmte Menge physischen Speicher bereit. Standardmäßig werden dabei 1 MByte des Prozessadressraums reserviert und dafür zwei Seiten[2] physischen Speichers bereitgestellt. Diesen Standardwert kann man entweder mit einem Linker-Schalter oder mit einem Parameter der Funktion `CreateThread` beeinflussen. Siehe dazu Tabelle 8.1 auf Seite 40.

Tabelle 8.30 zeigt einen möglichen Aufbau des Stackbereichs auf eiem Rechner mit 4Kbyte Seitengröße, wobei die Reservierung ab der Adresse 0x0800000 erfolgte.

Speicheradresse	Zustand der Speicherseite
0x080FF000	Obergrenze des Stacks: Bereit gestellte Seite
0x080FE000	Bereit gestellte Seite mit dem Schutzattribut PAGE_GUARD
0x080FD000	Reservierte Seite
⋮	⋮
0x08003000	Reservierte Seite
0x08002000	Reservierte Seite
0x08001000	Reservierte Seite
0x08000000	Untergrenze des Stacks: Reservierte Seite

Tab. 8.30: Der Stackbereich eines Threads nach der Erstellung

Der Stack eines Threads wächst in Richtung der abnehmenden Adressen. So bald der Thread versucht, auf die mittels des Flag PAGE_GUARD geschützte „Wächterseite“ zu zugreifen, wird automatisch das Betriebssystem benachrichtigt, welches daraufhin, unmittelbar auf die „Wächterseite“ folgende Speicherseite bereitstellt, den Flag PAGE_GUARD von der aktuellen „Wächterseite“ entfernt und ihn stattdessen der neu bereit gestellten Seite zuweist. So wird für einen Thread immer nur so viel Speicher bereit gestellt, wie er aktuelle benötigt.

Nehmen wir an, dass der vom Thread erzeugte Aufrufbaum ziemlich umfangreich geworden ist und das Stackzeiger-Register verweist bereits auf die Adresse 0x08003004. Bei einem weiteren Funktionsaufruf muss das Betriebssystem noch mehr physischen Speicher bereitstellen. Bei der Bereitstellung physischen Speichers für die an der Adresse 0x08001000 beginnenden Seite verfährt das das Betriebssystem jedoch etwas anders. Der Unterschied besteht darin, das die neu bereit gestellte Speicherseite nicht mehr mit dem Flag PAGE_GUARD versehen wird. Siehe Tabelle 8.31.

Das bedeutet jedoch, dass damit dem für den Stack reservierten Adressbereich kein zusätzlicher physischer Speicher mehr bereit gestellt werden kann. Die unterste Speicherseite bleibt immer reserviert, und es kann kein physischer Speicher für diese Seite bereit gestellt werden. Bei der Bereitstellung physishcen Speichers an der Adresse 0x0801000 löst das Betriebssystem des weiteren eine Ausnahmebedingung (Exception) des Typs EXCEPTION_STACK_OVERFLOW aus. Diese kann man sich zunutze machen, um auf einen

[2] Eine Seite ist die Grundeinheit, die bei der Speicherverwaltung verwendet wird.

Speicheradresse	Zustand der Speicherseite
0x080FF000	Obergrenze des Stacks: Bereit gestellte Seite
0x080FE000	Bereit gestellte Seite
0x080FD000	Bereit gestellte Seite
⋮	⋮
0x08003000	Bereit gestellte Seite
0x08002000	Bereit gestellte Seite
0x08001000	Bereit gestellte Seite
0x08000000	Untergrenze des Stacks: Reservierte Seite

Tab. 8.31: Vollständig Stackbereich belegter Stackbereich eines Threads

Stacküberlauf zu reagieren, um ein Datenverlust im Programmablauf vorzubeugen. Wie dies konkret aussieht kann man in der Beispielanwendung *StackOverflow* am Ende des Kapitel sehen.

Falls der Thread trotz der den Stacküberlauf ignalisierten Ausnahmebedingung mit der Verwendung des Stacks fortfährt und versucht auf den Speicher, der sich an Adresse 0x08000000 befindet, zu zugreifen, generiert das Betriebssystem eine Speicherschutzverletzung, da dieser Speicherbereich lediglich reserviert, aber nicht bereit gestellt ist. Das Betriebssystem übernimmt dann die Kontrolle und beendet nicht nur den „schuldigen“ Thread, sondern mit diesem auch den gesamten Prozess. Dieser verschwindet dabei kommentarlos ohne dass der Benutzer davon unterrichtet wird.

Aber warum wird die unterste Seite eines jeden Stackbereichs überhaupt stets reserviert? Die Reservierung dieser niemals bereit gestellten Seite verhindert ungewolltes überschreiben anderer vom Prozess verwendeter Daten. Es wäre ja durchaus möglich, dass an der Adresse 0x07FFF000, also eine Seite unter der Adresse 0x08000000, für einen weiteren Adressbereich physischer Speicher bereit gestellt worden ist. Falls darüber hinaus auch für die an 0x08000000 beginnende Seite physischer Speicher bereitstünde, würden Zugriffe auf diese Seite vom Betriebssystem nicht mehr abgefangen. Ein Überlauf des Stacks bliebe demnach unentdeckt und der Thread könnte ungehindert andere im Adressraum des Prozesses befindliche Daten überschreiben. Derartige Programmfehler sind äußerst schwierig aufzuspüren.

Damit lässt sich auch die Frage beantworten, wie viele Threads man maximal in einem Prozess starten kann. Die Anzahl der Threads ist nur durch den Speicher begrenzt. Geht man davon aus, dass jeder Thread einen Stack der Größe 1 Megabyte zugewiesen bekommt, kann man theoretisch 2028 Threads erzeugen. Reduziert man die Stackgröße pro Thread, sind auch mehr möglich. Ob es sinnvoll ist so viele Threads zu erzeugen sei dahingestellt. Besser wäre es wahrscheinlich so gennante Threadpools zu verwenden, die man entweder selber implementiert oder in dem man die von Windows bereitgestellten API-Funktionen (`QueueUserWorkItem`) nutzt und Windows die Verwaltung überlässt.

8.6.2 Beispielanwendung *Stackoverflow*

Die Beispielanwendung *StackOverflow* berechnet die Summe aller Zahlen von 0 bis zum angegebenen Endwert. Dabei wird ein rekursiver Algorithmus benutzt, um eine Exception des Typs EXCEPTION_STACK_OVERFLOW (in Delphi EStackOverflow) zu provozieren.

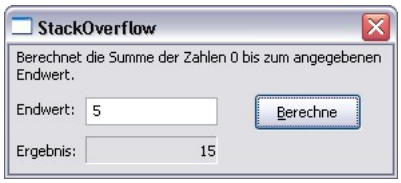

Abb. 8.6: Beispielanwendung *Stackoverflow*

Das Prinzip des Programms ist recht einfach. Nach dem der Benutzer einen Endwert eingegeben hat, wird ein neuer Thread gestartet.

```
MaxValue := GetDlgItemInt(Handle, IDC_EDT_VALUE, Translated, False);
hThread := BeginThread(nil, 0, @SumThread, Pointer(MaxValue), 0, ThreadID);
// Auf Thread warten.
WaitForSingleObject(hThread, INFINITE);
// ExitCode enthält die Summe.
GetExitCodeThread(hThread, Summe);
// Thread-Handle schliessen
CloseHandle(hThread);
// Wenn Rückgabewert = MaxInt dann ist ein Stacküberlauf aufgetreten.
if Summe = MaxInt then
  SetDlgItemText(Handle, IDC_STC_RESULT, PChar('Überlauf'))
else
  SetDlgItemInt(Handle, IDC_STC_RESULT, Summe, False);
```

Dieser Thread wiederum ruft rekursiv eine Funktion zum Berechnen der Summe auf:

```
function Sum(Num: Integer): Integer;
begin
  if Num = 0 then
    result := 0
  else
    result := Num + Sum(Num - 1);
end;

function SumThread(p: Pointer): Integer;
var
  Max               : Integer;
  Summe             : Integer;
begin
  // Parameter p enthält die Anzahl der zu addierenden Zahlen.
  Max := Integer(p);
  try
    // Um eine durch einen Stackueberlauf ausgelöste Ausnahme abzufangen,
    // muss die Funktion Sum innerhalb eines try-except-Blockes aufgerufen werden
    Summe := Sum(Max);
  except
    on E: EStackOverflow do
```

```
      // Bei einem Stackoverflow bekommt Sum den Wert MaxInt.
      Summe := MaxInt;
  end;
  Result := Summe;
end;
```

Der Thread gibt die berechnete Summe in seinem ExitCode zurück oder, wenn ein Stacküberlauf aufgetreten ist, den Wert der Konstante `MaxInt`.

8.7 Threadpools

In viele Situationen werden Threads erzeugt, die dann einen Großteil ihrer Zeit entweder damit verbringen zu warten und auf ein Ereignis warten. Andere Threads wiederum warten nur, um dann periodisch etwas abzufragen bzw. Statusinformationen zu aktualisieren. Mit Hilfe eines Thread Pools kann man nun mehrere Threads effizient verwalten bzw. vom System selber verwalten lassen. Man spricht dabei von so genannten *Worker Threads*, die von einem übergeordneten Thread verwaltet werden. So bald etwas anliegt, erzeugt der übergeordnete Thread einen Worker Thread bzw. aktiviert einen Worker Thread aus dem Thread Pool, der dann die zugehörige Callback-Funktion ausführt.

Die Implementierung ist relativ einfach, da das System die gesamte Verwaltung übernimmt. Um eine Funktion dem Threadpool hinzuzufügen reicht es die Funktion

```
function QueueUserWorkItem(LPTHREAD_START_ROUTINE: Pointer; Context: Pointer;
   Flags: DWORD): DWORD; stdcall; external
  'kernel32.dll';
```

aufzurufen. Die Methode wird ausgeführt, wenn ein Thread des Threadpools verfügbar wird.

Parameter	Bedeutung
LPTHREAD_START_ROUTINE	Zeiger auf die Thread-Funktion, die den auszuführenden Code enthält.
Context	Parameter, die an den Thread übergeben werden sollen.
Flags	Zusätzliche Flags, die das Verhalten steuern.

Tab. 8.32: Parameter QueueUserWorkItem

Die möglichen Flags können im SDK bzw. MSDN[3] nachgelesen werden. An dieser Stelle nur mal die wichtigsten.

Zu *I/O completion ports* siehe http://msdn2.microsoft.com/en-us/library/aa365198(VS.85).aspx und zu *Asynchronous Procedure Calls* (APC) und *Alertable I/O* siehe http://msdn2.microsoft.com/en-us/library/ms681951(VS.85).aspx und http://msdn2.microsoft.com/en-us/library/aa363772(VS.85).aspx.

Standardmäßig können im Threadpool 512 Threads erzeugt werden. Um die Anzahl der Threads zu erhöhen muss man das Makro

[3] http://msdn2.microsoft.com/en-us/library/ms684957.aspx

Flag	Bedeutung
WT_EXECUTEDEFAULT	Die Callback-Funktion wird an einen Thread übergeben, der I/O completion ports benutzt, was bedeutet, dass er nicht in einen signalisierten Zustand übergehen kann.
WT_EXECUTEINIOTHREAD	Die Callback-Funktion wird an einen I/O Worker Thread übergeben, welcher seinen Zustand signalisieren kann. Dies ist allerdings nicht sehr effizient, deshalb sollte dieser Flag nur benutzt werden, wenn die Callback-Funktion APCs generiert.
WT_EXECUTELONGFUNCTION	Signalisiert dem System, dass die Callback-Funktion länger benötigt, um zurückzukehren. Dies hilft dem System zu entscheiden, ob ein neuer Thread erzeugt werden oder auf bereits laufende Threads gewartet werden soll.

Tab. 8.33: Flags QueueUserWorkItem

```
#define WT_SET_MAX_THREADPOOL_THREADS(Flags,Limit) ((Flags)|=(Limit)<<16)
```

Daraus wird in Delphi folgende Funktion[4]:

```
function WT_SET_MAX_THREADPOOL_THREADS(AFlags, ALimit: ULONG): ULONG;
{$IFDEF SUPPORTS_INLINE} inline; {$ENDIF}
begin
  Result := AFlags or (ALimit shr 16);
end;
```

8.7.1 Beispielanwendung *Threadpool*

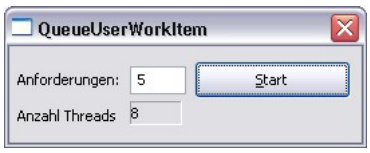

Abb. 8.7: Beispielanwendung *Threadpool*

Die Beispielanwendung *Threadpool* (siehe Abbildung 8.7) demonstriert die Verwendung der API-Funktion *QueueUserWorkItem*. Über das Texteingabefeld kann angegeben werden wie viele Anforderungen erstellt werden sollen. Nach dem man auf die Schaltfläche „Start“ geklickt hat, zeigt das Programm an, wie viele Threads im Threadpool erzeugt wurden, um die Anforderungen abzuarbeiten. Wie man sehr schön sehen kann, beeinflusst der

[4]Mein Dank geht an Nico Bendlin, für seine Hilfe bei der Übersetzung.

Flag WT_EXECUTELONGFUNCTION direkt, die Anzahl der Threads, die das System im Threadpool anlegt, um die Anforderungen abzuarbeiten. Ist das Flag gesetzt, geht das System davon aus, dass die Anforderungen länger brauchen um abgearbeitet zu werden und erstellt entsprechend mehr Threads, um den Anforderungen und auch zukünftigen Anforderungen gerecht zu werden.

8.8 Pseudo-Threads (Fibers)

Microsoft nahm Pseudo-Threads, so genannte *Fibers*, in Windows auf, um die Portierung vorhandener UNIX-Serveranwendungen nach Windows zu erleichtern.

Fibers kann man auch als Ligthweight-Threads bezeichnen, da sie einen geringeren Verwaltungsoverhead haben wie richtige Threads. Dies liegt daran, dass sie komplett im User-Mode implementiert sind, während hingegen Threads im Windows-Kernelmode implementiert sind. Der Kernel hat also eine gute Kenntnis „vom Wesen“ der Threads und teilt ihnen die Rechenzeit zu (Siehe dazu Kapitel 8.1.3 *Multitasking und Zeitascheiben* und Kapitel 8.4 *Thread-Prioritäten*). Der Kernel hat keinen Einblick in die Pseudo-Threads, so dass er ihnen auch keine Rechenzeit zuteilen kann. Für die Zuteilung der Rechenzeit ist daher der Programmierer selber verantwortlich. Fibers sind bezüglich des Kernels auch nicht präemptiv. Fibers existieren innerhalb eines Threads, haben aber einen eignen Stack, so dass sie sich mit dem Thread unter anderem dessen *Thread Local Storage* (TLS), den Rückgabewert von *GetLastError* und natürlich die Zeitscheibe des Threads teilen.

Fiber stellen also ein leichtgewichtiges, nebenläufiges Programmmodell bereit. Kontextumschaltungen (context switch) zwischen Fibern benötigen weniger CPU Zyklen, als das Wechseln zwischen zwei Threads, da nur relativ wenig Daten gesichert und kopiert werden müssen. Eine Kontextumschaltung beinhaltet nur folgende Aktionen:

- Die Register des aktuellen Fibers werden gespeichert. In der Regel sind das die selben Register die auch bei einem Funktionaufruf gesichert werden (Stackpointer, Instructionpointer).
- Die Register für den aufgerufenen Fiber werden geladen.
- Das instruction register wird auf den zu vor gesicherten Wert gesetzt. Der Fiber macht da weiter, wo er aufgehört hat.

Auch der Speicheroverhead ist sehr gering. Der gesamte Fiber Kontext ist gerade mal 200 Bytes groß und beinhaltet:

- Einen benutzerdefinierten Zeiger auf eine Datenstruktur, die als Parameter an den Fiber übergeben werden soll.
- Der Kopf einer für die strukturierte Ausnahmebehandlung zuständige Struktur.
- Einen Zeiger auf den Anfang und das Ende des Stacks des Fibers.
- Ein paar CPU Register.

Ein einzelner Thread kann ein oder mehrere Pseudo-Threads enthalten. Einem Thread wird durch den Kernel präemptiv Rechenzeit zugeteilt und dieser führt dementsprechend Code aus. Dabei kann er aber nur den Code eines Fibers ausführen. Wann dabei welcher Pseudo-Thread ausgeführt wird, liegt dabei allein in der Hand des Programmierers.

8.8.1 Erzeugen von Pseudo-Threads

Als aller erstes muss man einen bestehenden Thread, in einen Pseudo-Thread umwandeln, der dann weitere Pseudo-Threads beherbergen kann. Dies geschieht mit der API-Funktion *ConvertThreadToFiber*. *ConvertThreadToFiber* liefert als Ergebnis die Speicheradresse des Ausführungskontextes des Pseudo-Threads. Diese Umwandlung ist nötig, um in diesem Pseudo-Thread weitere Pseudo-Threads auszuführen. Um weitere Pseudo-Threads anzulegen, ruft man die Funktion *CreateFiber* auf.

```
function CreateFiber(dwStackSize: DWORD; lpStartAddress: TFNFiberStartRoutine;
    lpParameter: Pointer): Pointer; stdcall;
  external kernel32 name 'CreateFiber';
```

Parameter	Bedeutung
dwStackSize	Legt die Größe des Stacks fest. 0 überlässt die Festlegung Windows.
lpStartAddress	Zeiger auf die Fiber-Funktion.
lpParameter	Zeiger auf eine Variable, die dem Fiber beim Start übergeben werden soll.

Tab. 8.34: Parameter CreateFiber

Als Rückgabewert erhält man die Adresse des Ausführungskontextes des Fibers. Diese benötigt man noch, um gegebenenfalls mittels *SwitchToFiber* zwischen den Pseudo-Threads hin und her wechseln zu können. Die Fiber-Function muss dabei wie folgt definiert sein:

```
procedure FiberFunc(p: Pointer);
```

Im Gegensatz zu *ConvertThreadToFiber* wird der neue Thread aber nicht sofort ausgeführt, da der momentan laufende Fiber noch am Arbeiten ist. Und schließlich kann nur ein Fiber gleichzeitig ausgeführt werden. Die einzige Möglichkeit einem Fiber Rechenzeit zu zuteilen besteht darin die API-Funktion *SwitchToFiber* aufzurufen. Da *SwitchToFiber* explizit aufgerufen werden muss, um einem Fiber CPU-Zeit zu kommen zu lassen, hat man die uneingeschränkte Kontrolle über die Zeitzuteilung an die Fiber. Aber Achtung, dem Thread, in dem die Pseudo-Threads laufen, kann jederzeit durch das Betriebssystem unterbrochen werden. Wen nein Thread seine Zeitzuteilung erhält, läuft auch jeweils der aktive enthaltene Fiber. andere Pseudo-Threads kommen erst dann zum Zuge, wenn *SwitchToFiber* aufgerufen wird. *SwitchToFiber* erwartet als einziges Argument die Adresse des aufzurufenden Fiber-Kontextes und hat keinen Rückgabewert.

Zum Abbauen eines Pseudo-Threads wird die API-Funktion *DeleteFiber* aufgerufen. *DeleteFiber* erwartet als Parameter wieder, wie *SwitchToFiber* die Adresse des ausführungskontextes des betreffenden Fibers. Die Funktion gibt den Speicher für den Stack und den ausführungskontext des Pseudo-Threads wieder frei. *DeleteFiber* sollte nie für den Fiber aufgerufen werden der aktuell mit dem Thread verbunden ist, da sonst intern *ExitThread* aufgerufen würde, was das Ende des Threads und sämtlicher enthaltenen Pseudo-Threads zur Folge hätte. Pseudo-Threads weden also von aussen beendet, während richtige Threads sich selber mittesl *ExitThread* beenden sollten, da ein „gewaltsames“ Beenden von aussen

mittels *terminateThread* zur Folge hat, dass der Stack des betroffenen Threads nicht abgebaut wird.

Siehe dazu auch die Demo-Anwendung *Fibers*.

Anmerkungen zum Quellcode der Demo-Anwendung *Fibers*

Die Funktionen zum Erzeugen und Verwalten von Fibers sind zwar in der Unit Windows.pas deklariert, allerdings hat sich da ein Fehler eingeschlichen[5]. Laut Windows SDK ist *CreateFiber* wie folgt deklariert:

```
LPVOID WINAPI CreateFiber(
  SIZE_T dwStackSize,
  LPFIBER_START_ROUTINE lpStartAddress,
  LPVOID lpParameter
);
```

Bei Borland wurde dann daraus:

```
function CreateFiber(dwStackSize: DWORD; lpStartAddress: TFNFiberStartRoutine;
  lpParameter: Pointer): BOOL; stdcall;
```

Man beachte den Rückgabewert. Aus einem Zeiger wurde ein Wahrheitswert, obwohl *CreateFiber* im Erfolgsfall laut Windows SDK einen Zeiger auf den erzeugten Fiber zurückgeben soll:

> If the function succeeds, the return value is the address of the fiber.

BOOL entspricht LongBool, welches eine Größe von 4 Byte hat, also einen Zeiger aufnehmen könnte. Um ein Casten des Rückgabewertes zu vermeiden, habe ich mir die benötigten Funktionen noch einmal selber deklariert:

```
function CreateFiber(dwStackSize: DWORD; lpStartAddress: TFNFiberStartRoutine;
    lpParameter: Pointer): Pointer; stdcall;
  external kernel32 name 'CreateFiber';

procedure DeleteFiber(lpFiber: Pointer); stdcall; external kernel32 name '
    DeleteFiber';

function ConvertThreadToFiber(lpParameter: Pointer): Pointer; stdcall; external
    kernel32 name 'ConvertThreadToFiber';

procedure SwitchToFiber(lpFiber: Pointer); stdcall; external kernel32 name '
    SwitchToFiber';
```

[5] Siehe *Borland Developer Netwok - Quality Central*: „Incorrect return type for CreateFiber declaration“ (http://qc.borland.com/wc/qcmain.aspx?d=5760)

8.8.2 Vor- und Nachteile von Pseudo-Threads

Mittels Pseudo-Threads in einer Singlethread Anwendung kann man den Verwaltungsaufwand, der für die Synchronisation erforderlich, ist erheblich reduzieren. Das liegt daran, weil das Wechseln zwischen den Pseudo-Threads an anwendungsspezifischen Punkten erfolgt. Deswegen ist es auch nicht erforderlich mit CriticalSections zu arbeiten. Man muss nur darauf achten, dass der Ressourcenzugriff vor der nächsten Switch-Anweisung beendet ist.

Einige Vorteile:

- Schnelles Erzeugen, Abbauen und Wechseln zwischen Pseudo-Threads.
- Geringere Beanspruchung von Kernel-Speicher und anderer Kernel Ressourcen für die Thread Verwaltung.

Einige Nachteile:

- Keine Windows 95 und Windows CE Unterstützung.
- Keine Vorteile bei Mehrprozessorrechnern.
- Der Programmierer ist selber für die Zuteilung von Rechenzeit verantwortlich. Dass dies etwas komplexer ist zeigt die Demoanwendung *Fiber*.
- Es gibt keine Prioritäten (es sei denn man implementiert ein eigenes System).
- Man muss immer explizit zu einem bestimmten Fiber wechseln.

Abschliessend kann man sagen, dass Fibers dann sinnvoll eingesetzt werden können, wenn es auf die Performance ankommt und der Prozess ansonsten viel Zeit für Kontextwechsel aufwendet.

8.9 Das VCL Thread-Objekt

Die VCL bietet eine recht einfache Möglichkeit, mit Threads zu arbeiten und zwar die Klasse TThread. Sie übernimmt für einen eigentlich fast alles, was man zur Thread Programmierung braucht und kann sie dazu handhaben wie jedes beliebige Objekt in Delphi auch, zum Beispiel eine Stringliste oder ähnliches.

Die Klasse TThread stellt dazu ihre Eigenschaften und Methoden bereit. An dieser Stelle sollen nur die wichtigsten kurz angesprochen werden.

8.9.1 Erzeugen, Eigenschaften, Methoden

Erzeugen

Den Code für ein TThread-Objekt lässt sich mit Hilfe des Assistenten der IDE recht schnell erledigen: Daten | Neu | weitere | TThread-Objekt. Nach Eingabe des Klassennames, erstellt einem die IDE ein Grundgerüst:

```
unit Unit3;

interface

uses
  Classes;

type
  TMyFirstThread = class(TThread)
  private
    { Private-Deklarationen }
  protected
    procedure Execute; override;
  end;

implementation

{ Wichtig: Methoden und Eigenschaften von Objekten in VCL oder CLX können
   nur in Methoden verwendet werden, die Synchronize aufrufen, z.B.:

      Synchronize(UpdateCaption);

   wobei UpdateCaption so aussehen könnte:

    procedure TMyFirstThread.UpdateCaption;
    begin
      Form1.Caption := 'In einem Thread aktualisiert';
    end; }

{ TMyFirstThread }

procedure TMyFirstThread.Execute;
begin
  { Thread-Code hier einfügen }
end;

end.
```

Und eigentlich wird das wichtigste, was man nun über die Klasse wissen muss, um sie korrekt zu benutzen, schon im Kommentar erklärt:

1. Jeder Zugriff auf Elemente der VCL muss mit der Methode Synchronize synchronisiert werden.
2. Der Code welchen der Thread ausführen soll gehört in die Execute Methode.

Erzeugt wird eine Instanz des der Thread Klasse wie bei jedem Objekt mit der Methode *Create*. *Create* übernimmt einen Parameter *CreateSuspended*, so lange man sie nicht überschrieben hat, was auch möglich ist. Dieser eine Parameter gibt an, ob der Thread im angehaltenen (TRUE) oder im zuteilungsfähigen (FALSE) Zustand erzeugt werden soll. Will man noch den Feld-Variablen des Threads Werte zuweisen und oder seine Priorität setzen, ist es sinnvoll, ihn im angehaltenen Zustand zu erzeugen, alles zu initialisieren und ihn dann in den zuteilungsfähigen Zustand zu versetzten.

Parameter an den Thread zu übergeben ist auch etwas einfacher. Man muss nicht den Umweg über einen Zeiger auf eine eigene Datenstruktur gehen, sondern man legt einfach in der Thread-Klasse Felder im public-Abschnitt an:

```
type
  TMyThreads = class(TThread)
  private
    { Private-Deklarationen }
  protected
    procedure Execute; override;
  public
    // Zeilenindex des Listviews
    FIndex: Integer;
  end;
```

Und dann kann man, nachdem man das Thread-Objekt erzeugt hat, ihnen ganz einfach Werte zuweisen:

```
ThreadArray[Loop] := TMyThreads.Create(True);
ThreadArray[Loop].FIndex := Loop;
```

Eigenschaften

Eigenschaft	Bedeutung
FreeOnTerminate	Zerstört das Thread Objekt automatisch, wenn es seinen Code fertig ausgeführt hat.
Handle	Handle des Threads.
Priority	Thread Priorität (tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical).
Suspended	Zeigt an, ob ein Thread sich im angehaltenen Zustand befindet oder nicht.
Terminated	Zeigt an, ob der Thread terminiert werden soll oder nicht.
ThreadID	ID des Threads.

Tab. 8.35: Eigenschaften des Thread-Objektes

Methoden

8.9.2 Beispiel einer Execute–Methode

Hier geht es mir darum das Zusammenspiel der Eigenschaft *Terminated* und der Methode *Terminate* aufzuzeigen.

```
procedure TMyThreads.Execute;
begin
  // hier steht der Code, der vom Thread ausgeführt wird
  DoSomething();
end;
```

Eigenschaft	Bedeutung
Create	Erzeugt ein Thread-Objekt.
DoTerminate	Ruft den OnTerminate Eventhandler auf, beendet den Thread aber nicht.
Execute	Abstrakte Methode für den Code, den der Thread ausführt, wenn er gestartet wurde.
Resume	Versetzt den Thread in den zuteilungsfähigen Zustand.
Suspend	Versetzt den Thread in den angehaltenen Zustand.
Synchronize	Führt ein Methodeaufruf innerhalb des Haupt-VCL-Threads aus.
Terminate	Setzt die Eigenschaft Terminated auf True.

Tab. 8.36: Methoden des Thread-Objektes

```
procedure TMyThreads.DoSomething;
var
  Loop: Integer;
begin
  for Loop := 0 to 1000000 do
  begin
    Inc(FCount);
    // ist der Flag Terminated gesetzt, Schleife verlassen (* Unit1)
    if Terminated then
      break;
    Synchronize(UpdateLVCaption);
  end;
end;
```

Die Methode *Termintate* setzt die Eigenschaft *Terminated* auf TRUE. Damit sich der Thread nun beendet, ist es erforderlich, dass er in der Execute-Methode periodisch die Eigenschaft *Terminated* prüft, und dann entsprechend darauf reagiert. Im Code Beispiel ist das die if-Abfrage in der for-Schleife der Methode *DoSomething*. Ist *Terminated* TRUE, wird die Schleife verlassen, die Execute-Methode endet und damit auch der Thread.

8.9.3 Synchronisation des Thread-Objektes

Um einen Thread mit dem Haupt-VCL-Thread zu synchronisieren bietet die das Thread-Objekt die Methode *Synchronize* an. Nutzt man diese, erspart man sich das Einsetzten von kritischen Abschnitten, die natürlich trotzdem zur Verfügung stehen. Der Methode *Synchronize* übergibt man eine Methode der Thread Klasse, in der eine Methode aus dem Haupt-VCL-Thread aufgerufen wird, welche ihrerseits nun Objekte im Haupt-VCL-Thread aktualisiert. Diese Konstruktion sieht nun wie folgt aus. Auch hier etwas Beispiel-Code aus dem beiliegenden Thread-Objekt Demo:

Aufruf von Synchronize:

```
procedure TMyThreads.DoSomething;
var
  Loop: Integer;
```

```
begin
  for Loop := 0 to 1000000 do
  begin
    [..]
    Synchronize(UpdateLVCaption);
  end;
end;
```

Und die zugehörige Synchronize Methode der Thread Klasse:

```
procedure TMyThreads.UpDateLVCaption;
begin
  Form1.UpdateLVCaption(FIndex, FCount);
end;
```

Und zu guter letzt die Methode *UpdateLVCaption* aus dem Haupt-VCL-Thread, welcher die Ausgabe im Listview aktualisiert:

```
procedure TForm1.UpdateLVCaption(Index, Count: Integer);
begin
  Listview1.Items[Index].SubItems[0] := 'Position: ' + IntToStr(Count);
  if Count = 10000 then
    Listview1.Items[Index].SubItems[0] := 'fertig';
end;
```

Wichtig ist, so lange die zugehörige Synchronize Methode aus dem Haupt-VCL-Thread ausgeführt wird, steht der Thread. Um nun zu verhindern, dass es zu Performance-Einbußen kommt, sollten die Synchronisations-Methoden so kurz wie möglich gehalten werden. Rufen mehrere Threads gleichzeitig *Synchronize* auf, werden die Aufrufe serialisiert, das heißt, sie werden vom Haupt-VCL-Thread nach einander ausgeführt, was wiederrum bedeutet, dass alle Threads so lange warten, bis ihre Synchronize Methode aufgerufen und ausgeführt wurde. Als Beispiel siehe dazu Demo TThreadObject.

9 Listen

Listen gehören zu einer der elementarsten Datenstrukturen. Das liegt unter anderem daran, dass man Listen im richtigen Leben überall begegnet. Sei es in der einfachen Form einer Einkaufsliste oder sei es in Form einer Bundesligatabelle mit mehreren Spalten. In diesem Artikel soll es um die verschiedenen Möglichkeiten gehen solche Listen zu implementieren.

9.1 Arrays

Eine der verbreitesten Möglichkeiten eine Liste abzubilden ist wohl das *Array* bzw. Feld. Ein Array besteht aus einer Reihe von Elementen, die im Speicher direkt hintereinander liegen. Man kann also die einzelnen Elemente „anspringen“, indem man einfach einen Zeiger immer um die Größe eines Elementes im Speicher vorrückt oder zurückgeht. Dies ist ein recht einfaches und leicht verständliches Prinzip, da die einzelnen Elemente schön geordnet hintereinander im Speicher liegen.

Das hat nun allerdings auch ein paar Nachteile. Überlegen wir mal, welche Operationen erforderlich sind, um zum Beispiel ein neues Elemente in ein Array einzufügen. Als erstes muss man das Array um die Größe eines Elementes am Ende verlängern. Dann muss man alle Elemente, einschließlich des Elementes vor dem ein neues Element eingefügt werden soll, um einen Platz nach hinten kopieren. Dann kann man das neue Elemente an der gewünschten Stelle einfügen. Siehe dazu auch Skizze c) auf der Grafik am Ende der Seite. Alternativ kann man auch nur dasjenige Element nach hinten kopieren, an dessen Stelle das neue Element eingefügt werden soll. Dabei geht aber die eventuell vorhanden Sortierung verloren.

Das Löschen eines Elementes erfordert einen ähnlichen Aufwand. Entweder man kopiert das letzte Element auf den Platz des zu löschenden Elementes und verkürzt das Array um ein Elemente am Ende. Dann geht allerdings, wie beim Einfügen, die Sortierung verloren - falls vorhanden. Oder man kopiert alle nachfolgenden Elemente um einen Platz nach vorne. Beides ist aufwendig und wenn die Liste sortiert bleiben muss eventuell noch mit zusätzlicher Arbeit verbunden. Siehe Skizze c) und d).

Dazu etwas Beispielcode:

```
// Dynamische Arrays - Beispielprogramm
// Michael Puff [http://www.michael-puff.de]

program DynArrays;

{$APPTYPE CONSOLE}

type
```

```
  TDynArray = array of Integer;

var
  DynArray: TDynArray;

procedure AddElement(data: Integer);
begin
  SetLength(DynArray, Length(DynArray) + 1);
  DynArray[Length(DynArray) - 1] := data;
end;

procedure InsertElementAfter(Element: Integer; data: Integer);
var
  i: Integer;
begin
  SetLength(DynArray, Length(DynArray) + 1);
  for i := Length(dynArray) - 2 downto Element do
  begin
    DynArray[i+1] := DynArray[i];
  end;
  DynArray[Element] := data;
end;

procedure DeleteNextElement(Element: Integer);
begin
  DynArray[Element+1] := DynArray[Length(Dynarray) - 1];
  SetLength(DynArray, Length(DynArray) - 1);
end;

procedure WalkTheArray;
var
  i: Integer;
begin
  Writeln;
  for i := 0 to Length(DynArray) - 1 do
    Writeln(DynArray[i]);
end;

var
  i: Integer;
begin
  for i := 0 to 5 do
  begin
    AddElement(i);
  end;
  InsertElementAfter(4, 9);
  WalkTheArray;

  SetLength(DynArray, 0);
  for i := 0 to 5 do
  begin
    AddElement(i);
  end;
  DeleteNextElement(3);
  WalkTheArray;

  Readln;
```

```
end.
```

Noch ein paar Worte zum Speichermanagement. Wenn ein dynamisches Array vergrößert wird, passiert folgendes: Da alle Element hintereinander im Speicher liegen müssen, reserviert der Speichermanager neuen Speicherplatz, der um die Anzahl der Elemete, um die das Array vergrößert werden soll, größer ist. Dann werden alle Elemente von dem alten Speicherplatz für das Array in den neu reservierten Speicherplatz kopiert. Dann werden die neuen Elemente in den neu reservierten Speicherplätze abgelegt. Dies ist natürlich nicht sehr effizient. Will man also mehrere Elemente in einer Schleife hinzufügen, sollte man entweder, die Länge des Arrays vorher setzen oder, wenn das nicht möglich ist, die Länge des Arrays auf die ungefähr zu erwartende Länge setzen. Und dann entweder das Array auf die tatsächliche Länge verkürzen oder das Array noch mals entsprechend verlängern.

Eine Alternative bieten sogenannte *einfach verkettete Listen*.

9.2 Einfach verkettete Listen

Einfach verkettete Listen zeichnen sich dadurch aus, dass ihre Elemente, bei Listen spricht man meist von Knoten, nicht unbedingt hinter einander im Speicher liegen und zusätzlich zu den eigentlichen Daten noch zusätzlich gespeichert haben, welcher Knoten der nächste in der Liste ist. Sie besitzen also eine Art Zeiger der auf das nächste Element/Knoten in der Liste zeigt. Die einzelnen Knoten sind mit einander verkettet.

Damit vereinfachen sich nun bestimmte Operationen, die bei Arrays etwas umständlich waren. Will man einen neuen Knoten einfügen, lässt man den neune Knoten auf den Nachfolger zeigen von dem Knoten nach dem eingefügt werden soll und lässt den Knoten, nach dem eingefügt werden soll auf den neuen Knoten zeigen. Skizze 3).

Auch das Löschen ist einfach. Man lässt einfach den Knoten, nach dem gelöscht werden soll, auf den übernächsten Knoten zeigen. Es wird also einfach der Knoten, der gelöscht werden soll, übersprungen.

Man merkt schon an der Kürze der Beschreibung, dass dieses Vorgehen vom Prinzip einfacher ist als bei den Arrays. Allerdings bei der Implementierung ist etwas mehr Abstraktionsvermögen gefragt. Deswegen eine beispielhafte Implementierung einer einfach verketteten Liste in Delphi:

```
// Einfach verkettete Listen - Beispielprogramm
// Michael Puff [http://www.michael-puff.de]

program SingleLinkedList;

{$APPTYPE CONSOLE}

type
  PNode = ^TNode;
  TNode = record
    data: Integer;
    next: PNode;
  end;
```

```
var
  FirstNode: PNode; // Hilfsknoten
  LastNode: PNode; // Hilfsknoten
  CurrentNode: PNode; // aktueller Knoten

procedure InitList;
begin
  FirstNode := nil;
  LastNode := nil;
end;

procedure ClearList;
var
  TempNode: PNode;
begin
  CurrentNode := FirstNode;
  while (CurrentNode <> nil) do
  begin
    TempNode := CurrentNode.Next;
    Dispose (CurrentNode);
    CurrentNode := TempNode;
  end;
  FirstNode := nil;
  LastNode := nil;
end;

procedure AddNode(data: Integer);
begin
  New(CurrentNode);
  CurrentNode.data := data;
  CurrentNode.next := nil;
  // Liste leer
  // Ersten und letzten Knoten mit neuen Knoten gleichsetzen
  // Ein Knoten in der Liste
  if LastNode = nil then
  begin
    FirstNode := CurrentNode;
    LastNode := CurrentNode;
  end
  // Liste nicht leer
  // Letzten Knoten auf neuen Knoten zeigen lassen
  // Letzten Knoten zum aktuellen Knoten machen
  else
  begin
    LastNode.next := CurrentNode;
    LastNode := CurrentNode;
  end;
end;

procedure InsertNodeAfter(AfterNode: PNode; data: Integer);
var
  NewNode: PNode;
begin
  // neuen Knoten erzeugen
  New(NewNode);
  NewNode.data := data;
```

```
  // Neuer Knoten übernimmt Nachfolger
  // vom Knoten nach dem eingefügt werden soll
  NewNode.next := AfterNode.next;
  // Knoten nach dem eingefügt werden soll,
  // zeigt auf neuen Knoten
  AfterNode.next := NewNode;
end;

procedure DeleteNextNode(Node: PNode);
var
  TempNode: PNode;
begin
  if Node.next <> nil then
  begin
    TempNode := Node.next;
    // Vorheriger Knoten über nimmt übernächste Knoten als Nachfolger
    // (Überspringen des zu löschenden Knotens)
    Node.next := Node.next.next;
    Dispose(TempNode);
  end;
end;

procedure WalkTheList;
begin
  Writeln;
  CurrentNode := FirstNode;
  while CurrentNode <> nil do
  begin
    Writeln(CurrentNode.data);
    CurrentNode := CurrentNode.next;
  end;
end;

var
  i: Integer;
  TempNode: PNode = nil;

begin
  // Test AddNode und InsertNodeAfter
  InitList;
  for i := 0 to 5 do
  begin
    AddNode(i);
    if i mod 3 = 0 then
      TempNode := CurrentNode;
  end;
  WalkTheList;
  InsertNodeAfter(TempNode, 9);
  WalkTheList;
  // Test DeleteNextNode
  InitList;
  for i := 0 to 5 do
  begin
    AddNode(i);
    if i mod 3 = 0 then
      TempNode := CurrentNode;
  end;
```

```
  WalkTheList;
  DeleteNextNode(TempNode);
  WalkTheList;
  // Liste leeren
  ClearList;
  Readln;
end.
```

Am besten man veranschaulicht sich die einzelnen Prozeduren, in dem man auf einem Blatt-papier die Operationen einfach mal Schritt für Schritt nachvollzieht.

Bei genauerem Hinsehen stellt man allerdings fest, dass es ein Problem oder Nachteil gibt: Die Liste lässt sich nur in einer Richtung druchwandern. In unserem Beispiel nur von vorne nach hinten. Das liegt daran, dass ein Knoten nur seinen Nachfolger kennt, nicht aber seinen Vorgänger.

9.3 Doppelt verkettete Listen

Dies kann man beheben, indem man auf eine *doppelt verkettete Liste* zurückgreift. Bei einer doppelt verketteten Liste kennt ein Knoten nicht nur seinen Nachfolger, sonder auch seinen Vorgänger. Man kann also in der Liste vor und zurück gehen.

Bei dem Beispiel für die doppelt verkettete Liste habe ich mich nur auf das Hinzufügen von Knoten beschränkt:

```
// Doppelt verkettete Listen - Beispielprogramm
// Michael Puff  [http://www.michael-puff.de]>

program DoubleLinkedList;

{$APPTYPE CONSOLE}

type
  PNode = ^Tnode;
  TNode = record
    data: Integer;
    prev: PNode;
    next: Pnode;
  end;

var
  FirstNode: PNode;
  LastNode: PNode;
  CurrentNode: PNode;

procedure Init;
begin
  FirstNode := nil;
  LastNode := nil;
end;

procedure AddNode(data: Integer);
begin
```

```
  New (CurrentNode);
  CurrentNode.data := data;
  CurrentNode.prev := nil;
  CurrentNode.next := nil;
  if LastNode = nil then
  begin
    FirstNode := CurrentNode;
    LastNode := CurrentNode;
  end
  else
  begin
    LastNode.next := CurrentNode;
    CurrentNode.prev := LastNode;
    LastNode := CurrentNode;
  end;
end;

procedure WalkForward;
begin
  Writeln;
  CurrentNode := FirstNode;
  while CurrentNode <> nil do
  begin
    Writeln(CurrentNode.data);
    CurrentNode := CurrentNode.next;
  end;
end;

procedure WalkBackward;
begin
  Writeln;
  CurrentNode := LastNode;
  while (CurrentNode <> nil) do
  begin
    Writeln(CurrentNode.data);
    CurrentNode := CurrentNode.prev;
  end;
end;

var
  i: Integer;
begin
  Init;
  for i := 0 to 5 do
  begin
    AddNode(i);
  end;
  WalkForward;
  WalkBackward;
  Readln;
end.
```

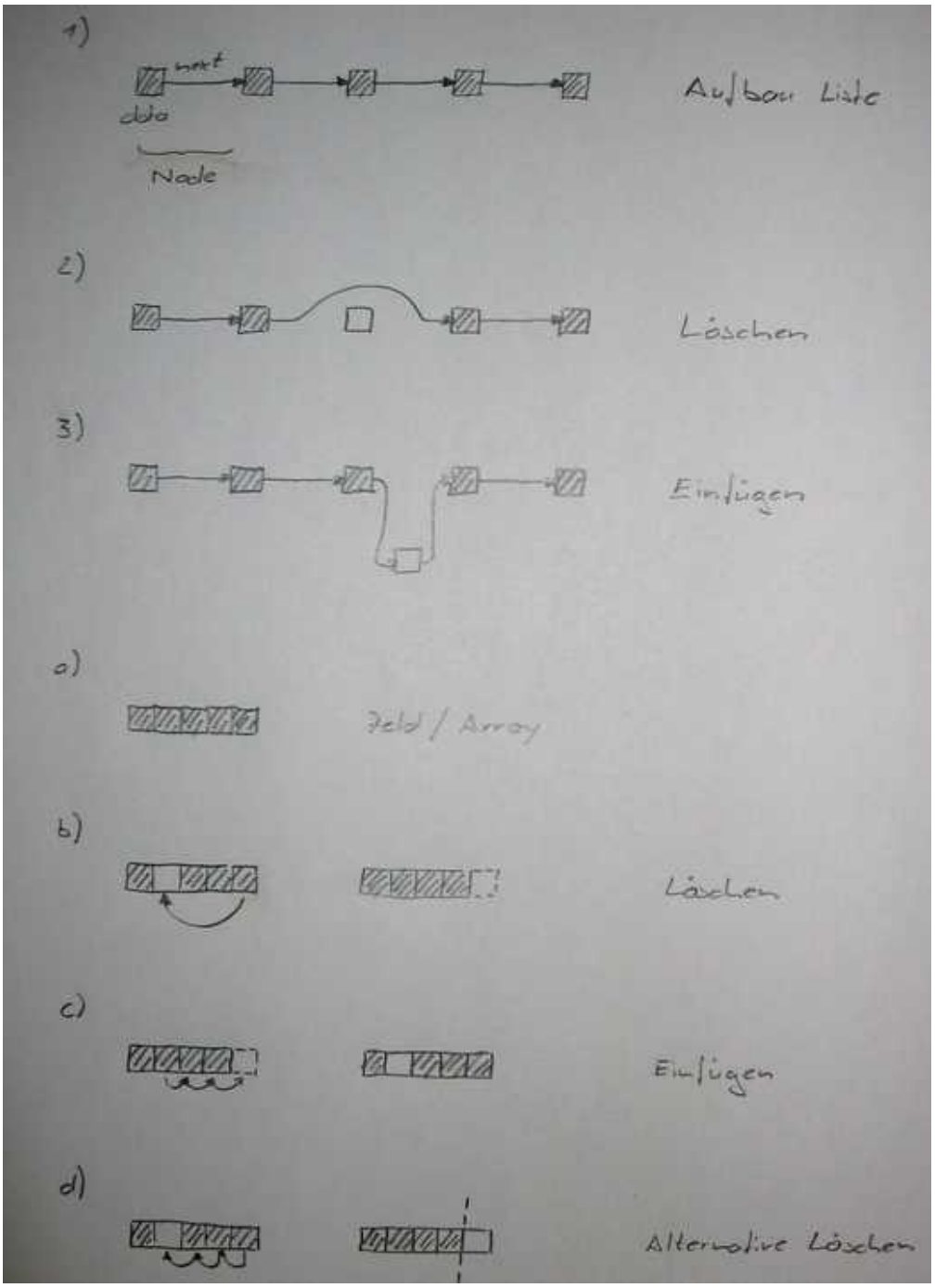

Abb. 9.1: Einfach verkettete Listen

10 Container-Klassen

Oftmals steht man vor dem Problem, dass man eine unbestimmte Anzahl von Objekten einer Klasse irgendwie verwalten muss, zum Beispiel Adressen für eine Adressdatenbank oder Spieler für ein Spiel. Nun kann man sich einen eigenen Datentyp (in Delphi: `record`) entwickeln und diesen dann mit einem dynamischen Array verwalten. Sobald es aber sinnvoll ist, dass die Elemente auch eigene Methoden und Eigenschaften mitbringen; ein Spieler-Element könnte zum Beispiel die Methode `Move` besitzen, die einen Spielzug macht oder ähnliches. In diesem Fall kommt man mit einem eigenen Datentyp nicht mehr weit und man muss auf Objekte zurückgreifen, wenn man OOP konform arbeiten will.

Und wenn man dann schon mit Objekten arbeitet, warum dann nicht gleich den ganzen Schritt machen und statt des dynamischen Arrays eine Liste implementieren, mit der man diese Objekte verwalten kann? Zum einem bleibt man damit OOP konform und zum anderen hat man dann noch die Möglichkeit Methoden in der Liste zu implementieren, um alle oder einzelne Objekte innerhalb der Liste zu manipulieren. Nehmen wir als konkretes Beispiel mal ein einfaches Spiel, was jeder kennt: *Memory*. Ein Memory-Spiel besteht aus Karten, von denen jede doppelt vorhanden ist und einem Spielfeld, meist ein Tisch oder eine beliebige andere ebene Fläche, auf der die Karten verdeckt, meist im Quadrat, angeordnet werden. Ziel ist es nun Kartenpaare zu finden und aufzudecken. Man deckt ein Kartenpaar auf und wenn es übereinstimmt, werden die Karten vom Spielfeld genommen. Stimmen sie nicht über ein, werden sie wieder umgedreht. Soweit kurz zu den Regeln. Gucken wir uns nun ein mögliches Lösungskonzept an. Wir haben die Spielkarten und wir haben das Spielfeld. Wollte man dies nun mit eigenen Datentypen und dynamischen Arrays lösen, wird die Implementierung recht umständlich, wie wir gleich sehen werden, wenn wir die OOP konforme Lösung betrachten.

10.1 OOP konforme Memory-Spiel Lösung mit Listen

An Hand des im Vorwort angeführten Beispiels eines einfachen Memory-Spiels, will ich hier eine mögliche objektorientierte Lösung aufzeigen.

10.1.1 Container-Klassen

Delphi stellt drei Klassen zur Verfügung, die als Container für Objekte dienen können:

TList: TList ist eine einfache Listenklasse, die eigentlich nur Array von Zeiger in einer Liste verwaltet und stellt Methoden zur Verfügung, um ein Zeiger der Liste hinzuzufügen, zu entfernen, zu verschieben usw.

TObjectList: TObjectList ist eine Erweiterung der TList-Klasse und besonders für Objekte geeignet, da sie auch den Speicherplatz der Objekte selber verwalten kann.

TCollection: TCollection ist eine spezielle Klasse, um Objekte vom Typ TCollectionItem zu verwalten. Näheres dazu kann m an in der Delphi-Hilfe nachlesen.

Für unsere Spielfeldklasse, die auch gleichzeitig die Container-Klasse für die Karten ist, würde sich die Klasse `TObjectList` anbieten, da sie uns einiges an Arbeit abnehmen kann, was die Speicherverwaltung betrifft. Ich habe mich jedoch für die einfachere Klasse `TList` entschieden, um zu zeigen, was man bei der Speicherverwaltung berücksichtigen muss.

Betrachtet man das Problem, die Implementierung eines Memory-Spiels, von der objektorientierten Seite, ergibt sich eine Lösung des Problems, wie man die Spielregeln umsetzen könnte, fast von alleine. Eine Karte wird in einem Karten-Objekt abgebildet und unser Spielfeld wird unser Container für die Karten. Unserer Karten-Klasse geben wir zwei Eigenschaften mit: *Wert* und *Status*. Der Wert entspricht dem aufgedruckten Symbol und der Status, ob die Karte auf- oder zugedeckt ist. Dies könnte man auch noch mit einem Record lösen. Aber es wäre bestimmt praktisch, wenn die Klasse, die die Karten verwaltet eine Benachrichtigung bekäme, wenn eine Karte umgedreht wird, um entsprechend darauf reagieren zu können. Also geben wir unserer Karten-Klasse noch ein Ereignis `OnFlip` mit, welches ausgelöst wird, wenn sich der Status der Karte ändert, sie also umgedreht wird. Die Container-Klasse, unser Spielfeld kann dann entscheiden, was passieren soll: Ist es die erste Karte, darf noch eine aufgedeckt werden, ist es die zweite, müssen die Werte der Karten verglichen werden um dann zu entscheiden, was weiter passieren muss. Daraus ergibt sich eigentlich schon welche Methoden, Ereignisse und Eigenschaften die Container-Klasse besitzen muss. Unser Spielfeld, muss im Prinzip nur eine Karte umdrehen können und ein Ereignis auslösen, wenn ein Zug zu ende ist, um den „Schiedsrichter“ benachrichtigen zu können, der dann entscheidet, was weiter passieren soll. Ich habe der Container-Klasse für unser Spielfeld noch ein paar zusätzliche Methoden gegönnt, um bestimmte Aufgaben etwas zu vereinfachen. Als da wären zum Beispiel alle Karten aufdecken oder wieder ein ganzes Kartenpaar umzudrehen, wenn sie nicht gleich waren.

10.2 Die Container-Klasse TMemoryField

Wir leiten unsere Spielfeldklasse nicht direkt von der Klasse `TList` ab, sondern einfach nur von der Basisklasse für alle Klassen `TObject`. Zum einem, weil wir noch ein paar zusätzliche Methoden implementieren wollen und zum anderen weil wir so die Möglichkeit haben selber zu bestimmen, welche Methoden mit welchen parameteren nach aussen hin sichtbar sind. So können wir sicher stellen, dass man in der Liste nur Objekte unserer Kartenklasse ablegen kann.

Dazu implementieren wir eine Klasse mit einer „inneren“ Liste
(`FCards: TList`), welche dann schließlich unsere Karten-Objekte aufnimmt. Da wir letztendlich nur Karten der Liste hinzufügen können müssen, reicht es wenn wir eine nicht öffentliche, nach aussen nicht sichtbare Methode `Add(Card: TCard)` implementieren, die eine Karte unserer inneren Liste `FCards` hinzufügt. Nicht sichtbar deswegen, weil das Spielfeld

und die Karten ja von der Klasse selber aufgebaut und erzeugt werden. Fehlt nur noch eine Eigenschaft, um auf die Objekte in unserer Liste zugreifen zu können. Die übliche Bezeichnung für diese Eigenschaft ist `items` mit der dazugehörigen Getter-und Setter-Methode. Somit wäre unser Container eigentlich schon fast fertig implementiert. Fehlt nur noch eins, das Freigeben der Objekte in der Liste, wenn unsere Spielfeld-Klasse selber freigegeben wird. Somit sähe unsere Spielfeld-Klasse bisher wie folgt aus:

```
  TMemoryField = class(TObject)
  private
    FCards: TList;
    // ...;
    // ...;
    function GetItem(Index: Integer): TCard;
    procedure SetItem(Index: Integer; const Value: TCard);
    procedure Add(Card: TCard);
    // ...;
    // ...;
  public
    constructor Create(CountCards: Integer; Parent: TWinControl; Width: Integer;
        Padding: Integer);
    destructor Destroy; override;
    property Items[Index: Integer]: TCard read GetItem write SetItem;
    // ...;
    // ...;
  end;
```

Und die zu gehörigen implementierten Methoden:

```
constructor TMemoryField.Create(CountCards: Integer; Parent: TWinControl; Width:
    Integer; Padding: Integer);
begin
  inherited Create;
  // ...;
  FCards := TList.Create;
  // ...;
end;

procedure TMemoryField.Add(Card: TCard);
begin
  FCards.Add(Card);
end;

function TMemoryField.GetItem(Index: Integer): TCard;
begin
  Result := FCards.Items[Index];
end;

procedure TMemoryField.SetItem(Index: Integer; const Value: TCard);
begin
  FCards.Items[Index] := Value;
end;

function TMemoryField.GetCount: Integer;
begin
  Result := FCards.Count;
end;
```

```
destructor TMemoryField.Destroy;
var
  i                 : Integer;
begin
  if FCards.Count > 0 then
  begin
    for i := FCards.Count - 1 downto 0 do
    begin
      TObject(FCards.Items[i]).Free;
    end;
  end;
  FCards.Free;
  inherited;
end;
```

Im Konstruktor wird unsere innere Klasse erzeugt:
`FCards := TList.Create;` und im Destruktor, nach Freigabe aller enthaltenen Objekte:

```
for i := FCards.Count - 1 downto 0 do
begin
  TObject(FCards.Items[i]).Free;
end;
```

wieder freigegeben: `FCards.Free;`.

Die Setter-Methode `SetItem` nimmt nur ein Objekt vom Typ `TCard` als Parameter an, so dass sicher gestellt ist, dass auch wirklich nur Objekte vom Typ `TCard` in die Liste aufgenommen werden können. Entsprechend gibt die Methode `GetItem` auch nur ein Objekt vom Typ `TCard` zurück.

10.3 Die Spiel-Methoden und zugehörigen Ereignisse

Das zentrale Ereignis ist die „OnClick“-Methode

```
procedure Click(Sender: TObject); reintroduce;
```

der Klasse `TCard`, welche die Klick-Methode der Vorfahrenklasse `TPanel` implementiert. In dieser Methode wird einfach nur der entsprechende Status gesetzt, nämlich „aufgedeckt“ und das Ereignis `OnFlip` ausgelöst, auf welches die Container-Klasse dann entsprechend reagiert:

```
procedure TMemoryField.OnFlip(Card: TCard);
begin
  Inc(FCountFlips);
  if FCountFlips = 1 then
    FFirstCard := Card
  else
    FSecondCard := Card;
  if FCountFlips = 2 then
  begin
    if Assigned(OnEndTurn) then
```

```
      OnEndTurn(FFirstCard.Value = FSecondCard.Value);
    FCountFlips := 0;
  end;
end;
```

Hier wird einfach die Häufigkeit des Ereignisses `OnFlip` gezählt und wenn es zwei mal aufgetreten ist, eine Runde ist zu ende, wird das Ereignis `OnEndTurn` der Klasse `TMemoryField` ausgelöst. Das Ereignis `OnEndTurn` liefert als Parameter auch gleich noch mit, ob die Karten den gleichen Wert haben, also ein Paar bilden, oder nicht:

```
TOnEndTurn = procedure(AreEquale: Boolean) of object;
```

Auf dieses Ereignis wiederum reagiert unsere Schiedsrichter-Klasse, die in diesem Beispiel auch gleich die Klasse unseres Formulares ist:

```
procedure TForm1.OnEndTurn(AreEqual: Boolean);
begin
  Panel1.Enabled := False;
  if AreEqual then
  begin
    if CheckBox1.Checked then
      Field.HideCouples
    else
      Field.FlipCouples(csFound);
  end
  else
  begin
    Delay(1000);
    Field.FlipCouples(csBlind);
  end;
  if IsGameOver then
    ShowMessage(rsGameOver);
  Panel1.Enabled := True;
end;
```

Mit diesen zwei, drei Methoden und Ereignissen hätten wir dann schon ein einfaches Memory-Spiel ohne großen Aufwand implemntiert. Alle anderen Methoden dienen eigentlich nur dazu den Umgang etwas zu vereinfache bzw. zu erleichtern und um den Spielkomfort zu erhöhen, deswegen will ich an dieser Stelle nicht weiter auf sie eingehen.

10.4 Ein Blick über den Tellerrand – Container-Klasse in C#

Abschliessend will ich noch mal etwas über den Tellerand schauen und zum Vergleich aufzeigen, wie man so etwas mit der .NET Sprache C# lösen könnte, wenn man keine vorgefertigte Klasse nutzen will. Im Prinzip kann unsere Container-Klasse identisch aufgebaut sein, nur eben mit der C# typischen Syntax für Eigenschaften:

```
class PersonenListe
    {
        private List<Person> innerList;
```

```
        public PersonenListe()
        {
            innerList = new List<Person>();
        }

        public void Add(Person person)
        {
            innerList.Add(person);
        }

        public int Count
        {
            get
            {
                return innerList.Count;
            }
        }

        public Person this[int index]
        {
            get
            {
                return (innerList[index]);
            }
            set
            {
                innerList[index] = value;
            }
        }

        public void RemoveAt(int index)
        {
            innerList.RemoveAt(index);
        }

        // benötigt für foreach
        public System.Collections.Generic.IEnumerator<Person> GetEnumerator()
        {
            return innerList.GetEnumerator();
        }

        public void Sort(IComparer<Person> comparer)
        {
            innerList.Sort(comparer);
        }
    }
```

Auffällig ist hier, wenn man mal von der C# spezifischen Syntax absieht, dass es keinen Destruktor gibt und nirgends etwas frei gegeben wird. Das liegt daran, dass C# eine Grabage collection besitzt, die automatisch dafür sorgt, dass nicht mehr benötigter Speicher wieder freigegeben wird. Insofern fällt unsere Klasse etwas einfacher aus, da man sich nicht mehr um das Freigeben des Speichers kümmern muss. Desweiteren wird eine generische Liste benutzt `innerList = new List<Person>();`, die eigentlich die
Wrapper-Klasse überflüssig macht, da die Typensicherheit schon durch die generische Liste gegeben ist. Erst wenn man eine erweiterte Funktionalität implementieren will, zum Beispiel

eine Prüfung vor dem Einfügen in der Liste oder ähnliches, würde eine Wrapper-Klasse wieder sinnvoll sein.

Die Klasse besitzt ausser dem noch eine zusätzliche Methoden, die mit der Verwaltung der Liste eigentlich nichts zu tun hat:
`GetEnumerator()`. Diese Methode ist nötig damit man mit Hilfe einer `foreach`-Schleife durch die Einträge in der Liste iterieren kann.

Allerdings bietet C# schon extra für solche Probleme vorgefertigte Klassen an. Als da wäre zum Beispiel die Klasse `Collection` für eine benutzerdefinierte Aufliste und darum handelt es sich ja schließlich.

10.5 Schlussbemerkung

Wie man sieht handelt es sich hier um ein Programmiersprachen unabhängiges Konzept. Man spricht auch von „Design Patter“ oder einem Entwurfsmuster bzw. Lösungsmuster. Siehe dazu den Wikipedia Artikel „Entwurfsmuster“[1]. Hat man dies erst ein paar mal durch exerziert, fällt es auch nicht schwer das Prinzip auf andere Sprachen zu übertragen.

Wie man auch gesehen hat, ergibt sich die Lösung mehr oder weniger schon von alleine beim Entwurf der Klassen. Die Implementierung ist dann meist nur noch Tipparbeit, da man das Problem, wie schon gesagt während des Entwurfs der Klassen und Benennung der Methoden und Ereignisse gelöst hat.

[1] http://de.wikipedia.org/wiki/Entwurfsmuster

11 MySQL

11.1 Was wir brauchen – Vorbereitungen

Um mit dem mySQL Server arbeiten zu können, brauchen wir erstmal den Server selber. Runterladen kann man ihn sich auf der Homepage des mySQL-Servers[1]. Ich empfehle die vollständige Windows Installation, die man hier[2] findet. Wird der Server nicht in das Verzeichnis `C:\mysql` installiert, ist es nötig noch eine Konfigurationsdatei im Windowsordner abzulegen, dies muss man eventuell von Hand machen, sollte nach dem Setup nicht gleich der Konfigurationsassitent starten. Bei mir befindet sich der Server und seine Daten im Verzeichnis `C:\Programme\MySQL\MySQL Server 4.1` entsprechend sieht die Konfigurationsdatei aus:

```
[mysqld]
basedir=C:/Programme/MySQL/MySQL Server 4.1/
datadir=C:/Programme/MySQL/MySQL Server 4.1/data/
[WinMySQLAdmin]
Server=C:/Programme/MySQL/MySQL Server 4.1/bin/mysqld-nt.exe!
```

Nach der Installation sollten sich im bin-Verzeichnis die Server-Anwendungsdateien befinden. Davon installieren wir die mysqld-nt.exe als Dienst für Windows. Wir geben dazu in der Konsole folgendes ein:

```
mysqld-nt --install
```

Danach sollte sich der mySQL Dienst in der Liste der lokalen Dienste wiederfinden. Den Server hätten wir also erstmal installiert und kongiguriert. Fehlen noch die für Delphi nötigen Header-Übersetzungen der C-API-Funktionen für den mySQL-Server.

Um nun unter Delphi den mySQL–Server einsetzten zu können brauchen wir noch die C-Header-Übersetzungen der C–Headerdateien. Diese bfindet sich mit in dem Archiv mit dem Delphi Demo–Programm: MySQL_mit_Delphi_Demos.zip. Diese Unit wird in unser Delphi-Projekt einfach eingebunden. Damit das ganze aber funktioniert, wird noch Client-Library libmysql.dll benötigt. Diese befindet sich im Verzeichnis
`C:\Programme\MySQL\MySQL Server 4.1\lib\opt` und muss sich im Suchpfad von Windows befinden. Also entweder im Anwendungsverzeichnis selber oder zum Beispiel im Windows-Ordner.

Damit wären unsere Vorbereitungen soweit abgeschlossen und wir können uns der eigentlichen Programmierung zu wenden.

[1] http://dev.mysql.com/
[2] http://dev.mysql.com/downloads/mysql/4.1.html

11.2 Mit dem Server verbinden

Als aller erstes muss man sich mit dem Datenbank-Server verbinden. Der MySQL-Server stellt dazu die Funktion `mysql_real_connect` zur Verfügung. Parameter siehe Tabelle 11.1, Seite 100.

Parameter	Datentyp	Bedeutung
Datenbankbeschreiber	PMYSQL	MYSQL-Struktur.
Host	PChar	Der Wert von Host kann entweder ein Hostname oder eine IP-Adresse sein. Wenn Host NULL oder die Zeichenkette «localhost» ist, wird eine Verbindung zum lokalen Host angenommen.
User	PChar	Benutzer der Datenbank.
Password	PChar	Passwort des Benutzers.
Datenbankname	PChar	Name der Datenbank, zu der verbunden werden soll. Wird ein leerstring angegeben, wird nur zum Server verbunden.
Port	Cardinal	Port-Nummer über den die TCP/IP Verbindung hergestellt werden soll.
Unix-Socket	PChar	Legt den Socket bzw. die named pipe fest, wenn dieser Parameter nicht leer ist.
Flag	Cardinal	Ist üblicherweise 0. Er kann aber auch eine Kombination von Flags wie unter anderem CLIENT_COMPRESS oder CLIENT_SSL enthalten.

Tab. 11.1: Parameter mysql_real_connect

Bevor die Funktion `mysql_real_connect` aufgerufen werden kann, muss der Datenbankbeschreiber erst mit `mysql_init` initialisiert werden.

Eine Funktion zum Verbinden mit dem Datenbankserver könnte nun so aussehen:

```
function Connect(Descriptor: PMYSQL; const Host, User, PW, DB: string; Port:
  Integer): PMYSQL;
begin
  result := mysql_real_connect(Descriptor, PChar(Host), PChar(User), PChar(PW),
    PChar(DB), PORT, nil, 0);
end;
```

Bzw. vollständig mit `mysql_init`:

```
Descriptor := mysql_init(nil);
Descriptor := Connect(Descriptor, HOST, USER, PW, '', PORT);
if Assigned(Descriptor) then
begin
...
...
```

Dieser Code befindet sich in der Demo Adress-Datenbank-Anwendung im OnPaint-Ereignis der Form, man kann ihn natürlich auch in das OnCreate-Ereigniss schreiben. Um entwaige Fehler mit zu loggen, habe ich mir eine Prozedur geschrieben, die immer den aktuellen Vorgang in ein Memo schreibt. Dies kann bei der Fehlersuche ganz nützlich sein, zeigt aber auch, was gerade passiert. Die Variable
`Descriptor` ist global, da wir sie auch bei anderen Funktionsaufrufen benötigen.

Habe wird erstmal eine Verbindung aufgebaut, können wir diverse Informationen abfragen (siehe Tabelle 11.2, Seite 101):

Funktion	Beschreibung
mysql_get_server_info	Gibt eine Zeichenkette zurück, die die Server-Versionsnummer bezeichnet.
mysql_get_host_info	Gibt eine Zeichenkette zurück, die den Typ der benutzten Verbindung beschreibt, inklusive des Server-Hostnamens.
mysql_get_client_info	Gibt eine Zeichenkette zurück, die die Version der Client-Bibliothek bezeichnet.
mysql_get_proto_info	Gibt die Protokollversion zurück, die von der aktuellen Verbindung benutzt wird.
mysql_character_set_name	Gibt den vorgabemäßigen Zeichensatz für die aktuelle Verbindung zurück.

Tab. 11.2: Informationen über eine MySQL-DB erhalten

Getrennt wird eine Verbindung mit dem Server mit der Funktion `mysql_close`. Als Parameter wird der Datenbankbeschreiber erwartet, der unsere geöffnete Verbindung zur Datenbank beschreibt.

11.3 Anlegen einer Datenbank

Die bisherigen Funktionen sind abhängig vom verwendeten Datenbank-Server. Benutzen wor einen anderen Server können sie entsprechend anders aussehen. Das Prinzip bleibt aber das gleiche. Eine Datenbank wird über ein SQL-Query, eine Abfrage, angelegt und ist weitgehend Server unabhängig. Allerdings haben einige Herstellen die herkömmliche SQL-Syntax erweitert bzw. eine für ihre Server spezifische Syntax implementiert, die nicht unbedingt kompatible sein muss. Die grundlegenden SQL-Befehle sollten allerdings kompatibel sein, so dass die hier vorkommenden SQL-Befehle entsprechend übertragen werden können. Alle weiteren direkten Datenbankoperationen werden mittels eines Querys ausgeführt.

Die SQL-Syntax zum Anlegen einer Datenbank lautet:

```
CREATE DATABASE <name>
```

Wobei `name` den Namen der Datenbank bezeichnet.

11.3.1 Ausführen eines Querys

Querys werden bei mySQL mittels der Funktion `mysql_real_query` ausgeführt (Tabelle 11.3, Seite 102).

Der Rückgabewert ist entweder 0, wenn kein Fehler oder ungleich 0, wenn ein Fehler aufgetreten ist. Der Rückgabewert sagt allerdings nichts über die Art des Fehlers aus, der aufgetreten ist. Um eine aussagekräftige Fehlermeldung zu erhalten, kann man folgende MySQL-Funktionen aufrufen (Tabelle 11.4, Seite 102):

Parameter	Datentyp	Bedeutung
Datenbankbeschreiber	PMYSQL	MYSQL-Struktur.
Query	PChar	Die Zeichenkette des Querys.
length	Cardinal	Länge der Zeichenkette des Querys.

Tab. 11.3: Parameter mysql_real_query

Funktion	Beschreibung
mysql_error	Gibt eine Zeichenkette zurück, die den zu letzt aufgetretenen Fehler beschreibt.
mysql_errno	Gibt den Fehlercode, der zu letzt aufgerufenen Funktion zurück.

Tab. 11.4: MySQL Error Funktionen

Beide Funktionen erwarten als Parameter den Datenbankbeschreiber, der gerade geöffneten Serververbindung. Konkret im Quellcode könnte das nun so aussehen:

```
var
  query: String;
  ErrorCode: Integer;
begin
  query := 'CREATE DATABASE' + ' ' + DBNAME;
  ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
DBNAME ist hier eine Konstante, die den Namen der Datenbank bezeichnet:
const
  DBNAME       = 'AdressDB';
```

Ich habe mir zum Ausführen von Querys eine extra Funktion geschrieben: ExecQuery:

```
function ExecQuery(const Datenbank, query: string; var Cols: TCols; var Rows:
  TRows): Boolean;
```

Im Moment ist für uns aber erst mal nur die Zeile von Interesse, in der der Query ausgeführt wird. Alles weiter werde ich im Verlauf des Tutorials erläutern.

11.4 Anlegen einer Tabelle

Eine Datenbank ist im eigentlichen Sinne nur ein Container, ein Container für Tabellen, die die eigentlichen Daten enthalten. Das heißt eine Datenbank kann mehrere Tabellen enthalten. Diese können untereinander verknüpft sein. So könnte man sich eine Datenbank vorstellen die in einer Tabelle Lieferanten und ihre Adressen verwaltet und in einer zweiten Tabelle werden die Produkte dieser Lieferanten aufgenommen. Dies macht in sofern Sinn, als dass ein Lieferant mehrere Produkte liefert und so zu jedem Produkt nicht noch die einzelnen Adressdaten des Lieferanten gespeichert werden müssen, sondern dass diese über eine gemeinsame Spalte in beiden Tabellen mit einander verknüpft werden. So wird vermieden, dass Daten mehrfach in einer Tabelle abgelegt werden. Man spricht dann auch von einer normalisierten Datenbank. Wir wollen es aber erstmal für den Einstieg einfach halten und nur mit einer Tabelle arbeiten.

Eine Tabelle besteht aus sogenannten Feldern oder auch Spalten genannt. Diese Felder definieren, was wir in unserer Tabelle ablegen können. Da wir eine Adressdatenbank erstellen wollen, habe ich folgende Felder gewählt (Siehe Tabelle 11.5, Seite 103):

Parameter	Datentyp	Bedeutung
id	eindeutige fortlaufende Datensatznummer	int NOT NULL AUTO_INCREMENT
name	Familienname	varchar(20)
vorname	Vorname	varchar(20)
strsse	Strasse	varchar(55)
plz	Postleitzahl	int
ort	Wohnort	varchar(50)
telefon1	Festnetznummer	varchar(17)
telefon2	Handynummer	varchar(17)
fax	Faxnummer	varchar(17)
email1	erste E–Mail Adresse	varchar(50)
email2	zweite E–Mail Adresse	varchar(50)
url	Homepage	varchar(50)
gebdat	Geburtsdatum	date
firma	Firma	varchar(25)
ts	Timestamp	timestamp

Tab. 11.5: Felder Tabelle Adress-Datenbank

Auf ein Feld der Tabelle möchte ich noch mal etwas genauer eingehen. Es handelt sich um das Feld `id`. Dieses hat noch zwei zusätzliche Attribute bekommen: `NOT NULL` und `AUTO_INCREMEMNT`. `NOT NULL` bedeutet, dass es keinen Datensatz geben darf, in dem dieses Feld leer ist. Der Grund ist der, dass mit Hilfe dieses Feldes jeder Datensatz eindeutig identifiziert werden soll. Wozu wir das brauchen, sehen wir, wenn wir Datensätze editieren oder löschen wollen, spätestens dann müssen wir nämlich einen Datensatz eindeutig identifizieren können. Wird nichts weiter angegeben, dann darf dieses Feld auch leer sein. Damit wir uns nicht selber darum kümmern müssen, habe ich dem Feld noch zusätzlich das Attribut `AUTO_INCREMENT` gegeben. Das heißt, die Datenbank sorgt selbständig dafür, dass dort immer ein Wert enthalten ist, der um eins höher ist, als der letzte vergebene Wert für dieses Feld, somit sollte eine Eindeutigkeit gewährleistet sein.

Bei der Wahl der Datentypen und der Größe der Felder habe ich erstmal nicht auf die Optimierung geachtet. Bei groß Datenbanken, die optimale Wahl der Datentypen und Größe der Felder sehr zur Performance beitragen.

11.4.1 Verfügbare SQL Datentypen

Die Datentypen und ihre Bedeutung kann man aus der folgenden Tabelle entnehmen (Siehe Tabelle 11.6, Seite 104):

11.4.2 Schlüssel und Indizes

Beschäftigt man sich mit Datenbanken wird man unweigerlich von Schlüssel, primär Schlüssel, sekundär Schlüssel und Indizes hören. Was ein Schlüssel bzw. Index ist, will ich hier kurz erläutern und versuchen zu erklären, wozu sie da sind und warum man sie benötigt.

Typ	Beschreibung
TINYINT	-128 .. 127
TINYINT UNSIGNED	0 .. 255
INT	-2.147.483.648 .. 2.147.483.647
INT UNSIGNED	0 .. 4.294.967.295
BIGINT	-3402823e+31 .. 3402823e+31
DECIMAL(length,dec)	Kommazahl der Länge length und mit dec Dezimalstellen; die Länge beträgt: Stellen vor dem Komma + 1 Stelle für Komma + Stellen nach dem Komma
VARCHAR(NUM) [BINARY]	Zeichenkette mit max NUM Stellen (1<= NUM <=255). Alle Leerstellen am Ende werden gelöscht. Solange nicht BINARY angegeben wurde, wird bei Vergleichen nicht auf Groß-/Kleinschreibung geachtet.
TEXT	Text mit einer max. Länge von 65535 Zeichen
MEDIUMTEXT	Text mit einer max. Länge von 16.777.216 Zeichen
TIME	Zeit; Format: HH:MM:SS, HHMMSS, HHMM oder HH
DATE	Datum; Format: YYYY-MM-DD, wobei - jedes nicht numerische Zeichen sein kann
TIMESTAMP	Setzt einen Datumswert beim Einfügen/Updaten einzelner Felder automatisch auf das Systemdatum. Format: YYYYMMDDHHMMSS. Wenn mehrere Felder den Typ TIMESTAMP haben, wird immer nur das erste automatisch geändert!

Tab. 11.6: MySQL Datentypen

Die Begriffe Schlüssel und Index sind eigentlich Synonyme und bezeichnen somit das gleiche. Ich werde im weiteren Verlauf den Begriff Index benutzen.

Indiziert man eine Spalte, wird diese separat, sortiert abgelegt und verwaltet. Das hat zur Folge das Abfragen schneller und effizienter berabeitet werden können. Ein kleines Beispiel:Nehmen wir an ich will alle Datensätze mit der ID zwischen 5 und 10 haben. Man kann davon ausgehen, dass diese in einer unsortierten Tabelle über die ganze Tabelle verstreut vorliegen. Stellt man sich jetzt vor dass jeder Datensatz als eine Karteikarte ais vorliegt, kann man sich leicht vorstellen, dass Aufgabe die Karteikarten mit den IDs zwischen 5 und 10 rauszusuchen recht mühsam ist. Sind die Karteikarten aber nach den IDs sortiert, erleichtert einem das die Arbeit ungemein. Man sollte also die Felder einer Tabelle indiziere, die am häufigsten sortiert bzw. abgefragt werden. Bei unserer Adressdatenbank wären das die Felder `name` und `vorname`. Also sollte auf genau diese Felder ein Index gesetzt werden. Hinzukommt, dass man über die ID eines Datensatzes auf den selbigen zugreift. Also sollte auch dieses Feld indiziert werden. Ein großer Nachteil von Indizes ist die Tatsache, dass sämtliche Datenänderungen langsamer werden (Die Sortierung muss ja neu aufgebaut werden). Man muss also immer abwiegen, ob ein Index auf Feld Sinn macht.

11.4.3 CREATE TABLE

Kommen wir nun zum eigentlichen anlegen der Tabelle. Dies geschieht auch wieder über einen Query. Wie man unter mySQL einen Query ausführt, habe ich schon unter Punkt 5.1 erläutert. Deswegen beschränke ich mich im Folgenden nur noch auf die SQL Syntax:

```
CREATE TABLE <tabellenname>(<feldname> <datentyp> <weitere Attribute>, <feldname>
    <datentyp> <weitere Attribute>, ...)
```

Indizes werden mit dem SQL-Schlüsselwort `KEY` oder `INDEX` definiert. Dann folgt der Name des Indexes und Klammern das zu indizierende Feld:

```
PRIMARY KEY(id), KEY idx_name (name), KEY idx_vorname (vorname)!.
```

Für unsere Datenbank würde der Query jetzt vollständig so aussehen:

```
CREATE TABLE Kontakte(id INT NOT NULL AUTO_INCREMENT, name varchar(20), vorname
    varchar(20), strasse varchar(55), plz int, ort varchar(50),
telefon1 varchar(17), telefon2 varchar(17), fax varchar(17), email1 varchar(50),
    email2 varchar(50), url varchar(50), gebdat date,
firma varchar(25), ts timestamp, PRIMARY KEY(id), KEY idx_name (name), KEY
    idx_vorname (vorname))
```

11.4.4 Löschen einer Tabelle

Gelöscht wird eine Tabelle mit dem SQL-Befehl
`DROP TABLE <tabellenname>`.

11.5 Datensätze einfügen, editieren und löschen

11.5.1 Einen Datensatz einfügen

Auch das Einfügen eines Datensatzes geschieht wieder über ein Query:

```
INSERT INTO <tabellenname>(<feldname>, <feldname>,...) VALUES('<wert>', '<wert>',
    ...)
```

Man gibt also die Tabelle an, in die man einen Datensatz einfügen will und dann in Klammern die betroffenen Felder als Parameter und dann als Parameter für das Schlüsselwort VALUES die entsprechenden Werte. das war eigentlich schon fast alles. Allerdings hat der Query keine Ahnung davon, mit welche Datenbank wir arbeiten wollen. Also müssen wir sie vorher auswählen. Dies geschieht mit der mySQL-Funktion `mysql_select_db`. Diese Funktion erwartet zwei Parameter. Als ersten wieder den Beschreiber vom Datentyp PSQL und als zweiten den Datenbanknamen. Der Rückgabewert ist wieder 0 oder ungleich 0 im Fehlerfall.

In dem Demo zu diesem Tutorial habe ich das wieder in eine separate Funktion ausgelagert:

```
function Insert(Kontakt: TKontakt): Boolean;
var
  query        : string;
  ErrorCode    : Integer;
begin
  ErrorCode := mysql_select_db(Descriptor, DBNAME);
  if ErrorCode = 0 then
  begin
    with Kontakt do
```

```
    begin
      query := 'INSERT INTO Kontakte(Name, Vorname, Strasse, Plz, Ort, ' +
        'Telefon1, Telefon2, EMail1, Gebdat) VALUES('
        + QuotedStr(Name)
        + SEP + QuotedStr(Vorname)
        + SEP + QuotedStr(Strasse)
        + SEP + IntToStr(PLZ)
        + SEP + QuotedStr(Ort)
        + SEP + QuotedStr(Telefon1)
        + SEP + QuotedStr(Telefon2)
        + SEP + QuotedStr(EMail1)
        + SEP + QuotedStr(GebDat)
        + ')';
    end;
    ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
  end;
  result := ErrorCode = 0;
end;
```

Diese Funktion erwartet als Parameter einen Record, der als Felder die Felder der Tabelle enthält:

```
type
  TKontakt = record
    ID: Integer;
    Name: string;
    Vorname: string;
    Strasse: string;
    Plz: Integer;
    Ort: string;
    Land: string;
    Telefon1: string;
    Telefon2: string;
    Fax: string;
    EMail1: string;
    EMail2: string;
    URL: string;
    Gebdat: string;
    Firma: string;
    Position: string;
  end;
```

Wie wir sehen, wird zu erst die Datenbank ausgewählt und dann der Query ausgeführt. Wichtig ist dabei noch, dass auch Datums-Werte als Zeichenketten übergeben werden, unabhängig davon dass dieses Feld in der Tabelle als date-Datentyp deklariert wurde.

11.5.2 Einen Datensatz editieren

Die allgemeine Syntax einen Datensatz zu editieren lautet:

```
UPDATE <tabellenname> SET <feldname>='<wert>' WHERE <feldname>='<wert>'
```

Wichtig hier bei ist die WHERE-Klausel mit der wir den SQL-Ausdruck nur auf einem bestimmten Datensatz anwenden. Würden wir dies weglassen, wären alle Datensätze betroffen. Günstiger weise nimmt man hier ein Feld, welches einen Datensatz eindeutig identifiziert. Man kann auch Felder der Tabelle kombinieren, um einen Datensatz eindeutig zu identifizieren. Um mir die Arbeit nicht unnötig schwer zu machen, habe ich deswegen in der Tabelle das Feld id definiert, welches einen Datensatz eindeutig identifiziert.

Im Quellcode könnte eine Routine, um einen Datensatz in eine Tabelle einzufügen nun so aussehen:

```
function UpdateRecord(ID: Integer; Kontakt: TKontakt): Boolean;
var
  query        : string;
  ErrorCode    : Integer;
begin
  log(Format('Datensatz %d ändern', [ID]));
  ErrorCode := mysql_select_db(Descriptor, DBNAME);
  if ErrorCode = 0 then
  begin
    with Kontakt do
      query := 'UPDATE Kontakte SET name=' + QuotedStr(Name) +
        SEP + 'vorname=' + QuotedStr(Vorname) +
        SEP + 'strasse=' + QuotedStr(Strasse) +
        SEP + 'plz=' + QuotedStr(IntToStr(PLZ)) +
        SEP + 'ort=' + QuotedStr(Ort) +
        SEP + 'telefon1=' + QuotedStr(Telefon1) +
        SEP + 'telefon2=' + QuotedStr(Telefon2) +
        SEP + 'email1=' + QuotedStr(EMail1) +
        ' WHERE ID =' + IntToStr(ID) + '';
    ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
  end;
  result := ErrorCode = 0;
end;
```

11.5.3 Einen Datensatz löschen

Das Löschen eines Datensatzes gestaltet sich entsprechend trivial:

```
DELETE FROM <tabellenname> WHERE <feldname>='<wert>'
```

Die WHERE-Klausel ist auch hier wieder von Bedeutung, sonst werden nämlich alle Datensatze gelöscht. Womit wir auch schon wissen, wie man alle Datensätze löscht.

11.6 Datensätze filtern

Jede noch so umfangreiche Datenbank wäre wertlos und überflüssig, wenn sie nicht die Möglichkeit bieten würde gezielt nach Informationen zu suchen und / oder die enthaltenen Informationen nach bestimmten Kriterien zu filtern. Denn was will man mit Informationen,

wenn kein gezielter Zugriff möglich ist? Um Datensätze auszuwählen und zu filtern, bietet SQL nun sehr umfangreiche Möglichkeiten.

`SELECT` tut eigentlich genau das, was es schon aussagt, es selektiert. Und zwar selektiert es Datensätze aus der angegebenen Tabelle. Ein SELECT-Statement hat folgende allgemeine Syntax:

```
SELECT <feldname>, <feldname>, <...> FROM <tabellenname> WEITERE_SQL_ANWEISUNGEN
```

Wollen wir uns zum Beispiel nur alle Nachname und die Vornamen ausgeben lassen, so sieht das passende SQL-Statement so aus:

```
SELECE name, vorname FROM kontakte
```

Will man sich alle Spalten einer Tabelle ausgeben lassen, kann man anstatt alle Spaltennamen hinzuschreiben auch ein «*» als allgemeinen Platzhalter angeben:

```
SELECT * FROM kontakte
```

Allerdings hilft uns das auch erstmal nicht viel weiter, wenn wir einen bestimmten Datensatz suchen, da sich die Datenmenge nicht reduziert hat. Ein erster Schritt wäre wohl erst mal die Datensätze zu sortieren.

11.6.1 ORDER BY

Mit `ORDER BY` wird festgelegt, nach welcher Spalte bzw. welchen Spalten sortiert werden soll. Mit `ASC` werden die Zeilen aufsteigend, mit `DESC` absteigend sortiert. Ist nichts angegeben, wird aufsteigend sortiert. Hier ein einfaches Beispiel, Datensätze nach dem Nachnamen sortieren:

```
SELECT name, vorname FROM kontakte ORDER BY name
```

Will man nach mehreren Spalten gleichzeitig sortieren, gibt man die weiteren Spalten einfach durch ein Komma getrennt mit an:

```
SELECT name, vorname FROM kontakte ORDER BY name, ort
```

Womit wir schon einen Schritt weiter wären. In einer sortierten Tabelle hat man wem falls schon mal eine Chance gezielt etwas zu finden. Aber so ganz das Wahre ist es noch nicht.

11.6.2 WHERE

Wenn wir gezielt Datensätze suchen bzw,. herausfiltern wollen, erweitern wir unser SQL-Statement mit dem Zusatz `WHERE`:

```
SELECT * FROM <tabellenname> WHERE <feldname>='<wert>'
```

Wollen wir zum Beispiel alle Meiers aus unserer Adress-Datenbank haben, könnte das entsprechende SQL-Statement wie folgt aussehen:

```
SELECT * FROM kontakte WHERE name='meier'
```

Ausdrücke können auch mit AND, OR und NOT miteinander verknüpft werden. Desweiteren ist es möglich Platzhalter zu verwenden: «_» steht für ein beliebiges Zeichen und «%» für eine beliebige Zeichenkette. Auch kann man natürlich `WHERE` noch mit `ORDER BY` und weiteren SQL-Ausdrücken kombinieren.

11.6.3 LIKE

Immer dann, wenn man in Textfeldern im Suchmuster Platzhalter oder Jokerzeichen verwenden will, können die Vergleichsoperatoren nicht verwendet werden. Statt dessen muss man in diesen Fällen auf den Operator `LIKE` zurückgreifen. Sollen zum Beispiel alle Personen mit der Vorwahl «0561» gefunden werden, sähe dies so aus:

```
SELECT name, vorname, telefon1 FROM kontakte WHERE telefon1 LIKE '%0561%'
```

11.6.4 BETWEEN

Ein weiterer Operator ist BETWEEN. BETWEEN wählt alle Spalten aus die zwischen den oberen und unteren Wert liegen:

```
SELECT name, vorname, gebdat FROM kontakte WHERE gebdat BETWEEN '1980-01-01' and
    '2005-01-01'
```

Diese Abfrage liefert uns alle Personen, die zwischen 1. Januar 1980 und 1. Januar 2005 geboren wurden. Man beachte die Angabe des Datums: yyyy-mm-dd. So und nicht anders muss es angegeben werden, damit es der mySQL Server versteht.

11.6.5 IN

Der letzte Operator aus dieser Gruppe ist der IN-Operator. Er wird benutzt, wenn man nicht mit einem einzelnen Wert, sondern mit einer Wertemenge vergleichen will. Beispiel: Wir wollen alle Personen die entweder «Schmidt» oder «Meier» heißen. Mit dem Vergleichsoperator «=» und einer Oder–Verknüpfung wird das bei vielen Werten, die zu vergleichen sind, schnell recht unübersichtlich. Einfacher geht es mit dem IN–Operator:

```
SELECT name, vorname FROM kontakte WHERE name IN ('schmidt', 'meier')
```

11.7 Die Demo-Anwendung „AdressDBSQL“

Wie schon in der Einleitung erwähnt, basiert dieses Tutorial auf einer kleinen Adress-Datenbank als Demo. Ich will an dieser Stell noch mal kurz die beiden zentralen Routinen der Anwendung, `ExecQuery` und `FillGrid`, erläutern.

11.7.1 ExecQuery

Die Funktion `ExecQuery` führt eine Abfrage aus:

```
function ExecQuery(const Datenbank, query: string; var Cols: TCols; var Rows:
    TRows):
  Boolean;
var
  MySQLRes     : PMYSQL_RES;
  MySQLRow     : PMYSQL_ROW;
  AffectedRows : Int64;
  ColCount     : Cardinal;
  Field        : PMYSQL_FIELD;
  i            : Integer;
  j            : Integer;
  ErrorCode    : Integer;
begin
  // Datenbank auswählen
  ErrorCode := mysql_select_db(Descriptor, PChar(Datenbank));
  if ErrorCode = 0 then
  begin
    // Query ausführen
    ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
    if ErrorCode = 0 then
    begin
      // Query speichern
      MySQLRes := mysql_store_result(Descriptor);
      if Assigned(MySQLRes) then
      begin
        // zurückgelieferte Anzahl der Spalten
        ColCount := mysql_num_fields(MySQLRes);
        SetLength(Cols, ColCount);
        // Spalten-Array füllen
        for i := 0 to ColCount - 1 do
        begin
          Field := mysql_fetch_field_direct(MySQLRes, i);
          Cols[i] := Field.Name;
        end;
        // Anzahl der betroffenen Zeilen ermitteln
        AffectedRows := mysql_affected_rows(Descriptor);
        SetLength(Rows, ColCount, AffectedRows);
        // Zeilen-array füllen
        for i := 0 to ColCount - 1 do
        begin
          for j := 0 to AffectedRows - 1 do
          begin
            MySQLRow := mysql_fetch_row(MySQLRes);
            Rows[i, j] := MySQLRow[i];
          end;
          mysql_real_query(Descriptor, PChar(query), length(query));
          MySQLRes := mysql_store_result(Descriptor);
        end;
        log(Format('Betroffene Zeile(n): %d',
          [mysql_affected_rows(Descriptor)]));
        // gespeicherte Abfrage wieder freigeben
        mysql_free_result(MySQLRes);
      end
```

```
    end
  end;
  result := ErrorCode = 0;
end;
```

Sie erwartet den Namen der Datenbank auf die die Abfrage ausgeführt werden soll, die Abfrage selber als String und dann die Spalten (Cols) und Zeilen (Rows), die die zurückgegeben Spalten bzw. die zurückgegebenen Zeilen beinhalten. Dabei ist der Parameter `Cols` vom Typ `TCols`, welcher ein eindimensionales dynamsiches String-Array ist und der Parameter `Rows` ist vom Typ `TRows`, welcher ein zweidimensionales dynamisches Array ist:

```
type
  TRows = array of array of string; // [Cols, Rows]
  TCols = array of string;
```

11.7.2 FillGrid

Die Prozedur FillGrid ist nun letztendlich dafür zuständig die Ergebnissmenge, die `ExecQuery` in den Parametern `Cols` und `Rows` zurückliefert, in einem StringGrid auszugeben.

```
procedure FillGrid(SG: TStringGrid; Cols: TCols; Rows: TRows);
var
  i, j         : Integer;
begin
  SG.ColCount := 0;
  SG.RowCount := 0;
  if Assigned(Rows) then
  begin
    // Wir brauchen eine Zeile mehr für die Spaltenüberschriften
    SG.RowCount := length(Rows[0]) + 1;
    SG.ColCount := length(Cols);
    SG.FixedRows := 0;
    // Spaltenüberschriften in die erste Zeile schreiben
    for i := 0 to length(Cols) - 1 do
    begin
      SG.Cols[i].Add(Cols[i]);
      SG.Cells[i, 0] := Cols[i];
    end;
    // zwei-dimensionales Zeilen-Array in den Zellen ausgeben
    for i := 0 to length(Cols) - 1 do
    begin
      for j := 0 to length(Rows[0]) - 1 do
      begin
        SG.Cells[i, j + 1] := Rows[i, j];
      end;
    end;
  end;
end;
```

12 COM

Das Component Object Model ist eine von Microsoft entwickelte Technologie, um unter Windows Interprozesskommunikation und dynamische Objekterzeugung zu ermöglichen. COM-fähige Objekte sind sprachunabhängig und können in jeder Sprache implementiert werden, deren Compiler entsprechenden Code erzeugen kann. Der Zugriff auf die Funktionalität eines COM-Objektes erfolgt über ein Interface, welches nach Instanzierung Zugriff auf die angebotenen Funktionen des COM-Objektes ermöglicht.

12.1 Die COM–Architektur

COM basiert auf dem Client/Server Prinzip. Ein Client instanziert eine COM-Komponente in einem COM-Server und nutzt dessen Funktionalität über Interfaces.

12.1.1 COM–Server

Ein COM-Server ist eine DLL oder ausführbare Datei, die eine COM–Komponente beinhaltet und bereitstellt. Es gibt drei Arten von COM-Servern:

1. In-process Server
2. Local Server und
3. Remote Server

12.1.2 In–process–Server

Handelt es sich um einen in–process–Server, ist die COM-Komponente in einer DLL implementiert (*.dll, *.ocx). Diese DLL muss die Funktionen

- `DllGetClassObject()`
- `DllCanUnloadNow()`
- `DllRegisterServer()` und
- `DllUnregisterServer()`

exportieren. Wird eine COM–Komponente aus einem COM–Server instanziert, wird der zugehörige Server in den Prozess der Anwendung (COM-Client) geladen. Dies hat den Vorteil, dass in–process–Server sehr schnell sind, da der Zugriff auf die COM-Komponente ohne Umwege erfolgen kann. Allerdings hat es den Nachteil, dass jeder Prozess, der eine COM-Komponente nutzen will, die in einem in–process–Server liegt, diesen in seinen Adressraum laden muss, was nicht besonders speicherschonend ist.

Local Server

Local Server sind ausführbare Programme, die eine COM–Komponente implementieren. Instanziert ein COM–Client eine COM–Komponente, so wird das Programm, welches die COM–Komponente beinhaltet, gestartet. Die Kommunikation erfolgt über ein vereinfachtes RPC-Protokoll (Remote Procedure Call). Local Server haben den Vorteil, dass sie nur einmal gestartet werden müssen, um mehrere Clients bedienen zu können. sie sind daher speicherschonender als die in–process Variante. Zudem lassen sich so mit Hilfe eines zentralen Servers leichter Zugriffe von den Clients auf gemeinsame Ressourcen synchronisieren. allerdings ist der Zugriff mittels RPC langsamer.

Remote Server

Befindet sich zwischen Server und Client ein Netzwerk, kommt DCOM (Distributed COM) zum Einsatz. Es unterscheidet sich von COM nur in sofern, als dass ein vollständiges RPC–Protokoll zum Einsatz kommt und ein Protokollstack vorgeschaltet wird. Aufgrund der Verwendung des vollständigen RPC–Protokolls werden die Aufrufe, auch bei geringer Netzwerklast, ziemlich verlangsamt.

12.1.3 Schnittstelle

Die Kommunikation zwischen COM–Server und COM–Client erfolgt über das COM–Interface. Jedes Interface wird über seinen Namen und eine GUID (Globally Unique Identifier) eindeutig identifiziert. Dadurch können mehrere Interfaces mit dem gleichen Namen existieren ohne dass es zu Verwechslungen kommt. Damit eine programmiersprachenübergreifende Kommunikation möglich ist, findet am Interface das so genannte Marshalling statt, welches die auszutauschenden Daten in eine vordefinierte Binärrepresentation umwandelt.

Ein Interface ist eine abstrakte Klasse, die nur virtuelle Methoden enthält, die wegen der Trennung von Definition und Implementation allesamt auf 0 im VTable gesetzt werden.

Wenn ein COM–Objekt ein Interface implementiert, muss es alle Methoden des Interfaces überschreiben. Dabei sind mindestens die drei Methoden von `IUnknown` zu implementieren, die für das Lifetimemanagement zuständig sind.

Ein Interface sieht in der für COM–Komponenten nutzbaren IDL (Interface Definition Language) unter Delphi wie folgt aus:

```
type
  IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;

  IUnknown = IInterface;
```

Jedes Interface muss über eine Interface–Vererbung die Funktionen des obigen Interfaces `IUnknown` definieren, da dies die grundlegenden Funktionen für COM implementiert.

Da Programmiersprachen wie Visual Basic Script keine Typen kennen, hat Microsoft eine weitere Möglichkeit entwickelt, Funktionen aus COM–Interfaces aufzurufen. Für diese Möglichkeit muss das Interface die Funktionen des Interfaces `IDispatch` definieren. Dies erlaubt es dann dem Programmierer, eine COM-Komponente über `IDispatch.Invoke()` anzusprechen, ohne dass der COM-Client die Typbibliothek des Servers kennen muss. Da der Zugriff über das Dispatch-Interface sehr viel langsamer als der Zugriff über ein typisiertes Interface ist, wird oft beides implementiert (Dual Interface), so dass sich der Programmierer bei Programmiersprachen, die Pointer beherrschen, den Weg des Zugriffs auf die COM-Komponente aussuchen kann.

12.1.4 COM–Komponente

Eine COM–Komponente bietet die aufrufbaren Funktionen des COM–Servers über ein Interface an. Die Instanzierung des Objektes erfolgt durch die Implementierung von `IClassFactory.CreateInstance()` im COM–Server. Zurückgeliefert wird dann eine Instanz der erzeugten Klasse. Und genau das ist der Punkt, den COM unter anderem auszeichnet: Das zur Verfügungstellen von Klassen in Programmbibliotheken (DLLs). DLLs können nämlich keine Objekte nach aussen zur Verfügung stellen, da eine DLL einen eigenen Speichermanager, gegenüber der Anwendung selber, welche die DLL nutzt, besitzt.

COM–Komponenten müssen, im Gegensatz zu gewöhnlichen Objekten, nicht wieder freigegeben werden. Dies übernimmt der COM–Server selbstständig. Wird ein Objekt instanziert, wird ein Referenzzähler hochgezählt, der bei einem Aufruf von `Release()` dekrementiert wird. So lange der Referenzzähler ungleich Null ist »lebt« das Objekt. Ein Aufruf von `Release()` erfolgt, wenn das Objekt im COM–Client auf einen Null–Zeiger (Delphi: `nil`, C++: `NULL`) gesetzt wird.

Eine COM–Komponente kann mehrere Interfaces zur Verfügung stellen. Dies ist insofern auch sinnvoll als das nur so eine Erweiterung der Funktionalität möglich ist ohne Gefahr zu laufen, bestehende Anwendungen, die diese Schnittstelle nutzen, neu kompilieren zu müssen. Da der Compiler die aus der VTable gelesenen Einsprungadressen der vom Client aufgerufenen Funktionen unter bestimmten Umständen fest kodiert. Wird nun das Interface einer Komponente geändert, kann sich die Einsprungsadresse ändern, was es den abhängigen Clients unmöglich machen würde, diese COM–Komponente zu nutzen ohne dass sie neu kompiliert werden müsste.

12.1.5 COM–Client

Als COM–Client bezeichnet man eine Anwendung, die die COM–Komponente eines COM–Servers letztendlich nutzt. Der Client kann die Funktionen der COM–Komponente nutzen, da diese in den entsprechenden COM–Interfaces deklariert sind. Die Veröffentlichung der

Interfaces erfolgt entweder über Typbibliotheken[1] oder Beschreibungen in der IDL (Interface Definition Language).

12.1.6 Apartments

COM-Objekte werden bei Instanzierung immer einem so genannten Apartment zugeordnet. Dabei handelt es sich um transparente Rahmen, welche zur Synchronisierung von Methodenaufrufen mehrerer Objekte dienen, die mit unterschiedlichen Anforderungen an die Threadsicherheit arbeiten. Wird COM nicht mitgeteilt, dass eine entsprechende Komponente threadsicher ist, wird COM nur einen Aufruf gleichzeitig an ein Objekt erlauben. Threadsichere Komponenten können auf jedem Objekt beliebig viele Aufrufe gleichzeitig ausführen.

Damit ein Thread COM benutzen kann, muss der Thread zuvor einem Apartment zugeordnet werden. Erst nach der Zuordnung ist eine Verwendung von COM möglich. Dabei kann ein Thread entweder einen schon bestehenden Apartment zugeordnet (MTA) werden oder es wird ein neues Apartment (STA) erstellt. Die Zuordnung geschieht mit der API-Funktion `CoInitialize()`. Programmiersprachen mit integrierter COM–Unterstützung führen diese Zuordnung meist automatisch. Erstellt man allerdings selber einen neuen Thread, so muss für diesen Thread die Zuordnung manuell mit einem Aufruf von `CoInitialize()` erfolgen und mit `CoUninitialize()` wieder aufgehoben werden.

Jede COM–Komponente wird beim Aufruf genauso einem Apartment zu geordnet. Falls die Apartment–Anforderungen der erzeugten Komponente zum Apartment Typ des erzeugenden Threads passen, wird das Objekt dem gleichen Apartment zugeordnet. Bei Aufrufen über Prozessgrenzen hinweg liegen die beiden Objekte immer in verschiedenen Apartments. Die Zuordnung zu einem Apartment kann während der Lebensdauer des Objektes nicht geändert werden.

Es gibt drei Arten von Apartments:

- Single Threaded Apartments (STA) besitzen genau einen Thread und beliebig viele Objekte. Aufrufe von verschiedenen Clients an das Objekt werden nacheinander abgearbeitet, wobei die Aufrufe der Clients in einer Warteschleife auf die Freigabe des Apartment–Threads warten. Dies ist vergleichbar mit dem Konzept der CriticalSections in der Windows API.
- Multi Threaded Apartments (MTA) besitzen beliebig viele Threads. In einem Multi Threaded Apartment können mehrere Clients gleichzeitig Aufrufe an das gleiche oder verschiedene Objekte machen. Dies erfordert allerdings auch eine entsprechend threadsichere Implementierung der Komponente.
- Neutral Thread Apartments (NTA) haben keine Threadaffinität. Jedes Objekt in einem NTA kann von einem STA/MTA Apartment ohne Threadübergang aufgerufen werden.

Näheres zu Apartments und deren Funktionalität auf der Seite http://www.codeguru.com in dem Beitrag „Understanding COM Apartments, Part I“[2].

[1] Eine Typbibliothek ist eine Binärdatei, die in einem MS spezifischen Format COM-Klassen und -Schnittstellen beschreibt.

[2] http://www.codeguru.com/Cpp/COM-Tech/activex/apts/article.php/c5529/

12.2 Typbibliotheken

Man muss für COM Routinen, die Out–of–process laufen – also einen eigenen Prozess besitzen – eine Schnittstellensprache verwenden, die IDL (Interface Definition Language). Sie wird in dem MS Format TLB gespeichert. In Delphi erstellt man dazu eine „Typbibliothek“ und fügt dort neue Schnittstellen ein. Diese wird in der IDL Sprache gespeichert.

Das muss so gemacht werden, weil jede andere Sprache diesen COM–Server ansteuern können soll – also nicht nur Delphi. Zudem müssen Parameter und Rückgabewert von einem Prozess in den COM-Prozess serialisiert (to marshall) werden. Es ist daher nicht möglich, beliebige Datentypen als Parameter zu verwenden. COM spezifiziert dazu seine eigenen Datentypen (BSTR) und andere müssen durch eine eigene Marshall–Implementation abgedeckt werden. Wenn man eine Typbibliothek erstellt hat, erstellt man darin ein COM–Objekt und implementiert es auch gleich. Also zuerst die abstrakten Methoden einfügen, dann das Interface ableiten und die Methoden implementieren.

12.3 Registration des COM–Servers

12.3.1 Einen COM–Server mit regsvr32 registrieren und de-registrieren

Aus Gründen, wie in Kapitel «12.4.3 Ortsunabhängigkeit» (Seite 119) erläutert, müssen COM–Server im System angemeldet oder anders ausgedrückt, im System bekannt gemacht werden, damit sie genutzt werden können. Dies geschieht mit dem Windows Kommandozeilenprogramm `regsvr32`. Um einen COM–Server zu registrieren wird das Programm wie folgt aufgerufen:

```
regsvr32 <Dateiname>
```

Wobei *Dateiname* der Dateiname der Datei ist, die den COM–Server enthält. Ein Beispielaufruf könnte demnach so aussehen:

```
regsvr32 COM_Test.dll
```

Um einen COM–Server wieder zu de–registrieren, wird das Programm mit dem zusätzlichen Schalter `\u` aufgerufen:

```
regsvr32 \u COM_Test.dll
```

Hinweis: Um einen COM–Server registrieren zu können müssen Schreibrechte im Registry–Zweig `HKEY_CLASSES_ROOT` vorhanden sein. In der Regel hat nur der Administrator diese Rechte. Eine Anwendung die einen COM–Server mitbringt, muss also von einem Administrator installiert oder eingerichtet werden. Im Idealfall, sollte entweder das Setup bei der Installation den COM–Server im System registrieren oder die Anwendung selber beim ersten Start, was aber wieder administrative Rechte erfordern würde.

12.3.2 COM–Server mittels API–Funktionen registrieren / de–registrieren

Jeder COM–Server implementiert und exportiert zwei Funktionen zum registrieren und de–registrieren seiner COM-Objekte: `DllRegisterServer` und `DllunregisterServer`. Will man nun einen Server im System registrieren und de–registrieren, muss man nur die entsprechende Funktion aus der COM–Server DLL aufrufen. Dazu müssen diese Funktionen jedoch dynamisch mit den API–Funktionen `LoadLibrary` und `GetProcAddress` geladen werden. Siehe dazu das folgendes Delphi–Beispiel:

```
type
  TDLLRegisterServer = function:DWORD;
  TDLLUnregisterServer = function:DWORD;

function RegisterServer(const Filename: String): Boolean;
var
  hLib: THandle;
  ProcAddress: TDLLRegisterServer;
begin
  Result := False;
  hLib := LoadLibrary(PChar(Filename));
  if hLib <> 0 then
  begin
    @ProcAddress := GetProcAddress(hLib, 'DllRegisterServer');
    if Assigned(ProcAddress) then
    begin
      Result := ProcAddress = S_OK;
    end
  end
end;

function UnregisterServer(const Filename: String): Boolean;
var
  hLib: THandle;
  ProcAddress: TDLLUnregisterServer;
begin
  Result := False;
  hLib := LoadLibrary(PChar(Filename));
  if hLib <> 0 then
  begin
    @ProcAddress := GetProcAddress(hLib, 'DllUnregisterServer');
    if Assigned(ProcAddress) then
    begin
      Result := ProcAddress = S_OK;
    end
  end
end;
```

12.3.3 Ist ein COM–Server / COM–Objekt im System bekannt?

Um zu testen, ob ein COM–Server im System registriert ist bzw. ob ein bestimmter COM-Objekt im System existiert, kann man entwder nach in der Registry im Zweig `HKEY\_CLASSES_ROOT\CLSID` nach der GUID suchen, wenn sie bekannt ist oder nach der

`ProgID`, welche sich aus dem Dateinamen des COM–Servers ohne Dateiendung und der Interface-Bezeichnung des COM–OBjektes zusammen setzt, verbunden mit einem Punkt: `COMServer.InterfaceBezeichnung` (siehe Abbildung 12.1 auf Seite 118).

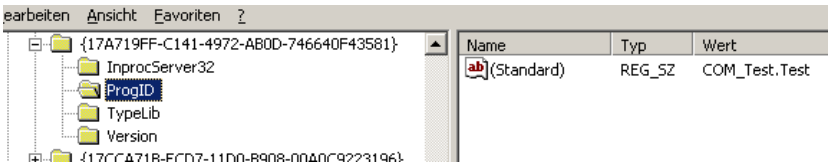

Abb. 12.1: Registry–Editor mit geöffneten Eintrag für die Demo–Anwendung

Siehe dazu auch die Abbildung 12.2 auf Seite 120, wo der Dateiname des zugehörigen COM–Servers zu sehen ist.

Die API–Funktion CLSIDFromProgID

Will man innerhalb eines Programmes überprüfen, ob ein COM–Objekt zur Verfügung steht, kann man die API–Funktion `CLSIDFromProgID` benutzen:

```
HRESULT CLSIDFromProgID(
  LPCOLESTR lpszProgID,
  LPCLSID pclsid
);
```

Parameter	Bedeutung
lpszProgID	[in] Pointer to the ProgID whose CLSID is requested.
pclsid	[out] Pointer to the retrieved CLSID on return.

Tab. 12.1: Parameter CLSIDFromProgID

Ein Beispiel für Delphi[3]:

```
function ProgIDExists(const ProgID:WideString):Boolean;
var
   tmp : TGUID;
begin
   Result := Succeeded(CLSIDFromProgID(PWideChar(ProgID), tmp));
end;
```

12.4 Vorteile von COM

COM bietet viele Vorteile gegenüber herkömmlichen Techniken, zum Beispiel DLLs, Funktionen Anwendungen zur Verfügung zu stellen.

- sprachunabhängig

[3]Quelle: Delphipraxis (http://www.delphipraxis.net/post797655.html#797655), Autor: shmia.

- versionsunabhängig
- plattformunabhängig
- objektorientiert
- ortsunabhängig
- automatisiert

Viele Windows Funktionen sind über COM zugänglich. Desweiteren ist COM die Basis für die OLE–Automation (Object Linking and Embedding) und ActiveX.

12.4.1 Sprachunabhängigkeit

Eine COM–Komponente kann in jeder beliebigen Programmiersprache implementiert werden. Ebenso kann der Zugriff über jede beliebige Programmiersprache erfolgen. Dies ist zwar bei DLLs auch der Fall, sollen allerdings DLLs Objekt–Instanzen zurückgeben, ist eine andere Technik nötig. Borland hat für Delphi, um dies zu ermöglichen, die BPL (Borland Package Library)–Technologie entwickelt. Dies ist natürlich nicht sprachunabhängig sondern eben nur mit Delphi bzw. dem Borland C++–Builder einsetzbar.

12.4.2 Versionsunabhängigkeit

Ein weiterer wichtiger Vorteil beim Einsatz von COM ist es, dass man die Verwaltung von neuen Softwarefeatures einfach in eine bestehende Anwendung integrieren kann. Oftmals kann es Probleme geben, wenn man herstellerneutrale oder herstellerübergreifende Softwarekomponenten mit weiteren Funktionen ausstattet. Dadurch kann zwar die eigene Software erweitert werden, jedoch besteht die Gefahr, dass andere Software, welche ebenfalls die herstellerübergreifenden Komponenten verwendet, nicht mehr funktionsfähig bleibt.

Mit COM hat man jetzt die Möglichkeit die Softwarekomponente mit weiteren Funktionen zu erweitern, in dem man weitere Interfaces hinzufügt. Dabei gehen die alten Interfaces nicht verloren. Interfaces werden also nicht erweitert oder verändert, sondern es werden weiter Interfaces hinzugefügt. Somit entsteht keine Inkonsistenz.

12.4.3 Plattformunabhängigkeit

x64-Applikationen können dank Marshalling auf 32bittige COM-Server zugreifen (und umgekehrt). Der COM-Server muss dann in einem eigenen Prozess laufen und seine Objekte können demnach nicht mit der INPROC_SERVER Variante instantiiert werden.

12.4.4 Ortsunabhängigkeit

COM ist ortsunabhängig, d. h. dass die einzelnen COM-Komponenten an einer zentralen Stelle (Registry) angemeldet werden und so der Zugriff auf die Komponenten unabhängig

von ihrem eigentlichen Ort erfolgen kann. Dies bezeichnet man auch als Ortstransparenz[4]. Dies ist auch der Grund, warum COM–Server registriert werden müssen. Da man COM–Server über deren GUID anspricht, muss an einem zentralen Ort hinterlegt werden, welche GUID zu welchem COM–Objekt gehört und in welcher DLL sich das COM–Objekt befindet – deswegen muss ein COM–Client nicht wissen, wo sich die DLL mit dem COM–Server befindet. Dies erledigt alles Windows für ihn. Diese Informationen werden in der Registry hinterlegt. Zusätzlich werden dort Informationen zu dem COM–Server abgelegt, wie zum Beispiel der Einsprungspunkt der DLL, ob es sich um einen in–process Server oder Local Server handelt und es wird der Typmarshaller festgelegt, wenn es sich um einen Local Server handelt.

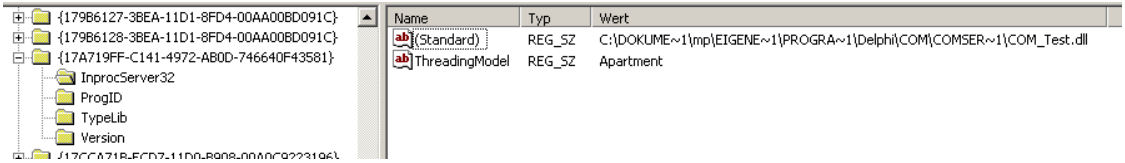

Abb. 12.2: Registry–Editor mit geöffneten Eintrag für die Demo–Anwendung

12.4.5 Automatisierung

Das Steuern von Anwendungen über COM-Interfaces wird als Automatisierung bezeichnet.

12.5 Erstellen eines COM-Servers mit Delphi

Das folgende Beispiel bezieht sich auf die Vorgehensweise mit Borland Delphi 6. Bei neueren Versionen sollte es aber ähnlich gehen.

Mit Delphi lässt sich ein COM–Server relativ schnell und unkompliziert selber erstellen. Dazu sind letztendlich nur fünf Schritte nötig:

1. Erstellen einer ActiveX–Bibliothek.
2. Erstellen eines COM–OBjektes in der ActiveX–Bibliothek mit Deklaration des Interfaces.
3. Hinzufügen von Methoden zu dem Interface über den Typbibliothekseditor.
4. Implementation der Methoden
5. Registration des COM–Servers.

Die einzelnen Schritte sollen im Folgenden noch mal etwas ausführlicher beschrieben und erläutert werden.

1. Erstellen der ActiveX–Bibliothek

[4]Ortstransparenz bedeutet in der EDV, dass der Benutzer einer verteilten Anwendung den tatsächlichen Ort des angefragten Objektes oder der angefragten Ressource nicht kennen muss.

Um eine ActiveX–Bibliothek anzulegen, wählt man beim Erstellen eines neuen Projektes in der Objektgalerie unter dem Seitenreiter ActiveX «ActiveX–Bibliothek» aus (siehe Abbildung 12.3, Seite 121, Rahmen eins).

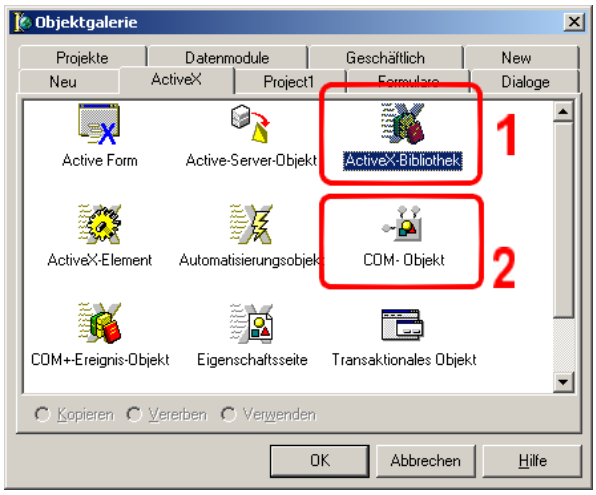

Abb. 12.3: Objektgalerie

Delphi erzeugt daraufhin das Code–Grundgerüst einer DLL in der Projektdatei (dpr) – unsere COM–Server DLL.

12.5.1 Erstellen des COM–Objektes

Das Erstellen eines COM–Objektes erfolgt auch über die Objektgalerie, wie man in Abbildung 12.3 auf Seite 121 im Rahmen 2 sehen kann. Mit Hilfe des COM–Objekt–Experten (siehe Abbildung 12.4, Seite 121)

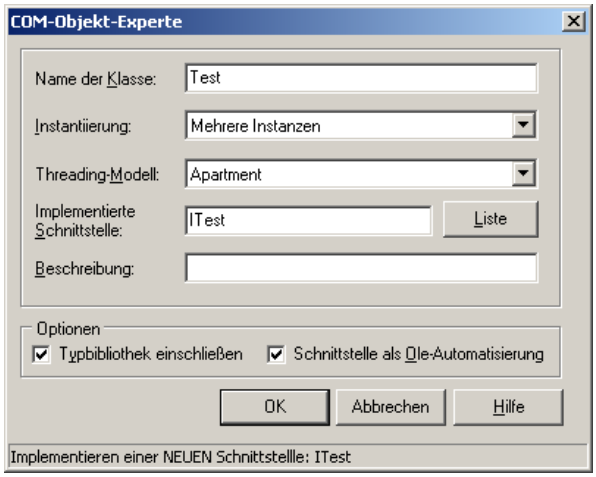

Abb. 12.4: COM–Objekt–Erxperte

kann man dann das COM–Objekt konfigurieren. Unter anderem kann man

- die Art der Instantiierung festlegen,
- das Threading-Modell und
- die implementierte Schnittstelle.

Delphi erstellt daraufhin eine Unit zur Implementation des Interfaces und öffnet den Typbibliotheken–Editor (siehe Abbildung 12.5, Seite 122, um das Interface zu implementieren):

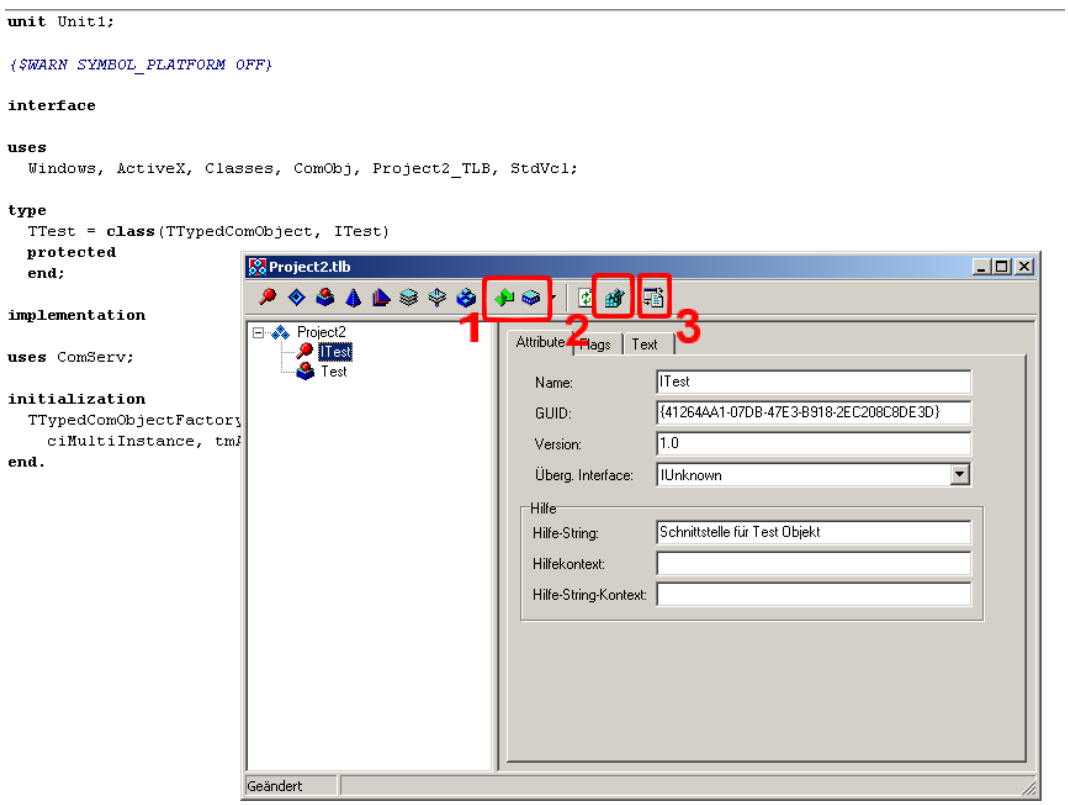

Abb. 12.5: Typbibliotheken–Editor

12.5.2 Hinzufügen und Implementation von Methoden

Über den Typbibliotheken–Editor kann man unter anderem neue Methoden, Eigenschaften (siehe Abbildung 12.5, Seite 122, Rahmen 1) anlegen, den COM–Server registrieren (siehe Abbildung 12.5, Seite 122, Rahmen 2) oder eine IDL-Datei erzeugen lassen. Die Implementation erfolgt dann direkt in der mit angelegten Unit.

Nach Hinzufügen einer Unit könnte dann der Quellcode von der Unit ungefähr so aussehen:

```
unit Unit1;

{\$WARN SYMBOL_PLATFORM OFF}

interface
```

```
uses
  Windows, ActiveX, Classes, ComObj, Project2_TLB, StdVcl;

type
  TTestCOM = class(TTypedComObject, ITestCOM)
  protected
    function Add(a, b: Integer): SYSINT; stdcall;
  end;

implementation

uses ComServ;

function TTestCOM.Add(a, b: Integer): SYSINT;
begin

end;

initialization
  TTypedComObjectFactory.Create(ComServer, TTestCOM, Class_TestCOM,
    ciMultiInstance, tmApartment);
end.
```

12.5.3 5. Registration des COM–Servers

Will man das Interface auch gleich nutzen, um es zum Beispiel zu testen kann man es auch über dem Typbibliotheken–Editor registrieren (siehe dazu Abbildung 12.5, Seite 122, Rahmen 2).

Mit Schaltfläche 3 kann man sich dann schliesslich noch eine IDL-Datei erzeugen lassen:

```
[
  uuid(F94B54FB-BF00-436D-89FC-14026CDE8A55),
  version(1.0),
  helpstring("Project2 Bibliothek")

]
library Project2
{

  importlib("STDOLE2.TLB");
  importlib("stdvcl40.dll");

  [
    uuid(EB80AEC8-5E7F-43B6-8B18-2D718ED653EA),
    version(1.0),
    helpstring("Schnittstelle für TestCOM Objekt"),
    oleautomation
  ]
   interface ITestCOM: IUnknown
  {
    [
    id(0x00000001)
    ]
```

```
    int _stdcall Add([in] long a, [in] long b );
  };

  [
    uuid(71027821-9B64-4D9C-9ED1-0D07B831C03F),
    version(1.0),
    helpstring("TestCOM")
  ]
  coclass TestCOM
  {
    [default] interface ITestCOM;
  };

};
```

Literaturverzeichnis

[1] Borland: *Sprachreferenz Delphi*. Borland Software Corporation, 2002

[2] Ray Lischner: *Delphi in a nutshell*. O'Reilly, 1. Auflage, März 2000, 1-56592-5

[3] Walter Doberenz, Thoams Kowalski: *Programmieren lernen in Borland Delphi 5*. Hanser, 1999, 3-446-21363-5

[4] Loos, Peter: *Go to COM*. Addison-Wesley, 1. Auflage, 2000), 3-8273-1678-2

[5] Kosch, Andreas: *COM/DCOM/COM+ mit Delphi*. Software & Support, 2. Auflage, 2000, 3-9350-4201-9