

# Einführung in DELPHI

## und die

# Programmiersprache ObjectPascal

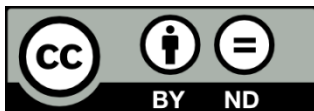
Basierend auf dem Lehrbuch des Delphi-Treff von den Autoren:  
FLORIAN HÄMMERLE, MARTIN STROHAL, CHRISTIAN REHN, ANDREAS HAUSLADEN

Angepasst an die DELPHI COMMUNITY EDITION 10.3 durch:

MATTHIAS EIBING

Version 4.0, 2020





Dieses Werk darf von jedem unter den Bedingungen der Creative-Commons-Lizenz CC-BY-ND 3.0 (de) genutzt werden.

Autoren: Florian Hämmerle, Martin Strohal, Christian Rehn, Andreas Hausladen, Matthias Eißing

Cartoon-Figur „Agi“: Karl Lux, [www.lux-cartoons.at](http://www.lux-cartoons.at)

Delphi-Treff-Logo: Yasmin Siebert

Embarcadero und Delphi sind Warenzeichen oder eingetragene Warenzeichen der Embarcadero Technologies, Inc.

Ursprüngliche Quelle dieses E-Books: [www.delphi-treff.de/downloads/e-book/](http://www.delphi-treff.de/downloads/e-book/)

## Inhalt

1	Erste Schritte .....	8
1.0	Worum geht es hier? .....	8
1.0.1	Was ist Delphi überhaupt? .....	8
1.0.2	Wo bekomme ich Delphi her? .....	8
1.0.3	Wer nutzt Delphi?.....	9
1.0.4	Weitere Informationen .....	9
1.1	Installation.....	10
1.2	Der erste Start .....	13
1.3	Hallo Welt!.....	13
1.3.1	„Hallo Welt“ für Konsole .....	14
1.3.2	„Hallo Welt“ mit GUI (VCL).....	16
1.3.3	„Hallo Welt“ mit GUI (FireMonkey).....	20
2	Schnellstart – Das Wichtigste .....	24
2.0	Allgemein.....	24
2.0.1	Über die Sprache .....	24
2.0.2	Grundlegendes zur Syntax.....	24
2.1	Bestandteile eines Delphi-Projekts .....	24
2.1.1	Projektdatei (*.dpr) .....	24
2.1.2	Quellcode (*.pas).....	25
2.1.3	Fensterdefinition (*.dfm) .....	26
2.1.4	Eine ausführbare Datei (EXE) erzeugen.....	26
2.1.5	Delphi-Programme weitergeben.....	27
2.2	Variablen .....	28
2.2.1	Was sind Variablen? .....	28
2.2.2	Variablen deklarieren .....	28
2.2.3	Werte zuweisen.....	29
2.2.4	Variablen umwandeln .....	29
2.2.5	Mit Variablen rechnen.....	30
2.3	Schleifen und Bedingungen.....	32
2.3.1	Die for-Schleife .....	32
2.3.2	Bedingungen.....	33
2.4	Unterprogramme: Prozeduren und Funktionen .....	35
2.4.1	Parameter .....	35
2.4.2	Prozeduren .....	35
2.4.3	Funktionen.....	36

2.5	Benutzereingaben .....	37
2.5.1	Eingaben in Konsolenanwendungen .....	37
2.5.2	Eingaben in VCL-Anwendungen .....	38
2.5.3	Wichtige Komponenten.....	41
2.6	Hilfe! .....	45
2.7	Delphi-Komponentenbibliotheken.....	45
3	Object Pascal im Detail .....	47
3.0	Variablen und Konstanten.....	47
3.0.1	Was sind Variablen? .....	47
3.0.2	Datentypen .....	47
3.0.3	Deklaration .....	48
3.0.4	Globale Variablen .....	48
3.0.5	Lokale Variablen .....	49
3.0.6	Zuweisungen.....	49
3.0.7	Initialisierung .....	49
3.0.8	Beispiel .....	50
3.0.9	Typumwandlung.....	51
3.0.10	Konstanten .....	52
3.1	Datentypen.....	53
3.1.1	Strings.....	54
3.1.2	Boolean.....	58
3.2	Verzweigungen .....	65
3.2.1	if-else .....	65
3.2.2	case-Verzweigung.....	66
3.3	Schleifen .....	68
3.3.1	Was sind Schleifen?.....	68
3.3.2	for-Schleife .....	68
3.3.3	while-Schleife .....	68
3.3.4	repeat-until-Schleife.....	69
3.3.5	for-in-Schleife .....	69
3.3.6	Schleifen abbrechen .....	70
3.4	Eigene Datentypen definieren.....	71
3.4.1	Typdefinition .....	71
3.4.2	Teilbereichstypen .....	71
3.4.3	Aufzählungstypen („Enumeration Types“) .....	72
3.4.4	Mengentypen („Set“) .....	72
3.4.5	Arrays.....	73

3.4.6	Records .....	75
3.4.7	Zeiger („Pointer“) .....	76
3.5	Prozeduren und Funktionen.....	78
3.5.1	Was sind Prozeduren und Funktionen? .....	78
3.5.2	Aufbau einer Prozedur .....	78
3.5.3	Aufbau einer Funktion.....	78
3.5.4	forward- und interface-Deklarationen .....	79
3.5.5	Parameter.....	80
3.5.6	Prozeduren und Funktionen überladen .....	83
3.5.7	Prozeduren und Funktionen abbrechen.....	84
3.6	Programmaufbau.....	85
3.6.1	Projektdatei .....	85
3.6.2	Units .....	86
3.6.3	Units verwenden .....	87
3.6.4	Positionen der uses-Klausel.....	88
3.6.5	interface und implementation .....	89
3.6.6	initialization und finalization .....	90
3.7	Objektorientierung.....	90
3.7.1	Klassen, Objekte und Instanzen .....	90
3.7.2	Schach! .....	91
3.7.3	Sichtbarkeiten.....	92
3.7.4	Instanz erzeugen.....	93
3.7.5	Elemente einer Klasse .....	94
3.7.6	Die Erben einer Figur .....	98
3.7.7	Überschreiben von Methoden .....	103
3.7.8	Polymorphie – alles nur Figuren.....	104
3.8	Funktions- und Methodenzeiger .....	106
3.9	Exceptions .....	107
3.9.1	Exceptions werfen .....	107
3.9.2	Eigene Exception .....	107
3.9.3	Exceptions fangen .....	107
3.9.4	Exceptions fangen und weiterwerfen .....	108
3.9.5	Ressourcenschutzblöcke: try – finally .....	109
3.10	Dateien .....	110
3.10.1	Binär- und Textdateien.....	110
3.10.2	Dateiname und -pfad.....	110
3.10.3	Relative und absolute Pfade.....	110
3.10.4	Die Dateitypen.....	111

3.10.5	Die Klasse TFileStream.....	118
3.10.6	Die Klasse TStringList.....	123
3.10.7	Arbeiten mit Dateien.....	125
3.11	Besondere Datentypen.....	127
3.11.1	Datum und Zeit.....	127
3.11.2	Listen mit TList.....	129
3.11.3	Dictionaries mit TDictionary.....	131
3.11.4	Stacks und Queues.....	132
3.12	Generische Datentypen („Generics“).....	133
4	Fehlerbehandlung.....	135
4.0	Die Vorbereitung.....	135
4.1	Die Elemente des Debuggers.....	140
4.1.1	Haltepunkte („Break points“).....	141
4.1.2	Durchschreiten des Quelltextes.....	143
4.1.3	Überwachte Ausdrücke.....	143
4.1.4	Aufruf-Stack.....	145
4.2	Verhalten bei Fehlern.....	146
4.2.1	Hinweise und Warnungen.....	146
4.2.2	Fehler zum Zeitpunkt der Kompilierung.....	149
4.2.3	Interpretieren von Laufzeitfehlern.....	153
4.3	Vermeiden von Fehlern.....	161
4.3.1	Lesbarkeit des Quelltextes.....	161
4.3.2	Speicherlecks.....	166
5	Grafische Benutzeroberflächen.....	171
5.0	Frameworks.....	171
5.1	Oberflächen-Stile.....	171
5.1.1	VCL-Stile.....	171
5.1.2	FireMonkey-Stile.....	173
5.1.3	Metropolis – der Windows 8-Stil.....	176
6	Beispielprojekte.....	179
6.0	Datenübertragung mit Indy-Komponenten.....	179
6.0.1	Server.....	180
6.0.2	Client.....	181
6.0.3	Der Test.....	182
6.1	Datenbankprogrammierung – SQLite mit Delphi.....	184
6.1.1	Was ist eine Datenbank?.....	184

6.1.2	Was ist SQL? .....	184
6.1.3	Was ist SQLite? .....	185
6.1.4	Relationale Datenbanksysteme .....	185
6.1.5	Wichtige SQL-Befehle .....	186
6.1.6	SQLite-Zugriff über FireDAC mit Delphi .....	188
7	Anhang .....	197
7.0	Lösungen zu den Übungsaufgaben .....	197
7.0.1	Kapitel 3 (Schnellstart) .....	197
7.0.2	Kapitel 4.1 (Variablen und Konstanten) .....	198

***Im Kapitel „Erste Schritte“ geht es darum, wofür man Delphi braucht, wie man es installiert und wie man in wenigen Minuten eine erste einfache Anwendung erstellt.***

## 1 Erste Schritte

### 1.0 Worum geht es hier?

Willkommen in dieser Delphi-Einführung! Wir freuen uns, dass du dich für die Programmierung mit Delphi interessierst. Dieses E-Book ist für alle gedacht, die noch nie etwas mit Delphi zu tun hatten, jetzt aber neugierig sind und wissen wollen, was man damit so alles anstellen kann. Ziel ist es also nicht, Delphi-Profis geheime Kniffe beizubringen, sondern Neulingen die Grundlagen zu erklären.

#### 1.0.1 Was ist Delphi überhaupt?

Delphi ist eine Entwicklungsumgebung der amerikanischen Firma Embarcadero, die auf der Programmiersprache Object Pascal basiert. Die erste Version von Delphi erschien schon 1995, damals noch hergestellt von der Firma Borland.

Und was kann man mit Delphi machen? Delphi ist für verschiedene Einsatzzwecke geeignet. Am wichtigsten ist jedoch das einfache und schnelle Erstellen von Windows-Anwendungen mit grafischer Benutzeroberfläche. Die Anwendungen, die Delphi erzeugt, sind nativ. Das bedeutet, es werden zum Ausführen keine Bibliotheken (DLLs oder Laufzeitumgebungen wie bei .NET oder Java) benötigt. Eine normale Windows-Installation als Grundlage für das Ausführen von Delphi-Anwendungen reicht aus.

Delphi ist außerdem darauf spezialisiert, Datenbankzugriffe einfach zu machen. Das heißt aber nicht, dass man es nur zur Entwicklung von Datenbankanwendungen gebrauchen könnte!

Seit Delphi XE2 ist es möglich, Anwendungen für verschiedene Plattformen zu bauen. Bis dahin war nur Win32 (32-Bit-Windows) möglich. Nun lassen sich auch native 64-Bit-Anwendungen für Windows erstellen. Und wer auf das neue Framework FireMonkey setzt, kann sogar Anwendungen für macOS, iOS und für Android erzeugen. Die Community-Edition, auf der dieses E-Book aufbaut, enthält auch das FireMonkey-Framework für iOS, Android, Windows und macOS.

#### 1.0.2 Wo bekomme ich Delphi her?

Delphi ist kostenlos als Community Edition erhältlich. Seit Juli 2018 gibt es die Delphi Community Edition, die für nicht-kommerzielle oder geringe kommerzielle Zwecke kostenfrei zum Download zur Verfügung steht. Diese basiert auf der „Professional“ Edition und bietet damit gegenüber der alten „Starter Edition“ einige wesentliche Vorteile:

- Kommandozeilen-Compiler
- Vollständiger Debugger



- 32- und 64-Bit Compiler
- Die VCL Quelltexte
- und einiges mehr

Für dieses Buch werden wir auf die Delphi Community Edition in der Version 10.3.3 zurückgreifen. Alles, was darin geht, ist natürlich auch in Professional-, Enterprise- und Architect-Editionen möglich. Vieles, wenn nicht das Meiste, kann auch mit älteren Versionen von Delphi nachvollzogen werden.

### 1.0.3 Wer nutzt Delphi?

Zugegeben – die Verbreitung von Delphi ist nicht so groß wie die von Java oder C#. Im Bereich der Windows-Desktopanwendungen ist Delphi jedoch unschlagbar. Und deshalb gibt es auch bekannte Anwendungen, die damit erstellt worden sind, z. B. Skype, PC Tools Spyware Doctor, The Bat!, Inno Setup – und natürlich Delphi selbst.

### 1.0.4 Weitere Informationen

Wer nach der Lektüre dieses E-Books mehr über Delphi wissen will, wird auf unserer Website Delphi-Treff fündig, auf der u.a. viele Tutorials und Quellcode-Schnipsel (Tipps & Tricks) zu finden sind.

Im deutschsprachigen Bereich sind außerdem die Foren Delphi-Praxis und Entwickler-Ecke zu empfehlen.

Falls du in diesem E-Book Fehler finden solltest (was wir nicht hoffen) oder Vorschläge für eine neue Version machen willst, schreibe uns doch bitte unter [ebook@delphi-treff.de](mailto:ebook@delphi-treff.de)! Wir freuen uns über jedes konstruktive Feedback.

Nun aber genug der einleitenden Worte – viel Spaß beim Programmieren mit Delphi!

### 1.1 Installation

Nach dem Download (<https://www.embarcadero.com/de/products/delphi/starter>) von der Webseite von Embarcadero muss das Delphi-Installationsprogramm mit Administratorrechten gestartet werden. Zu Beginn wird man nach einem gültigen Lizenzkey gefragt, den man im Verlauf des Downloads per eMail erhalten haben sollte. Während der Installation werden einige Software-Komponenten nachgeladen, weshalb eine schnelle Internetverbindung empfehlenswert ist.

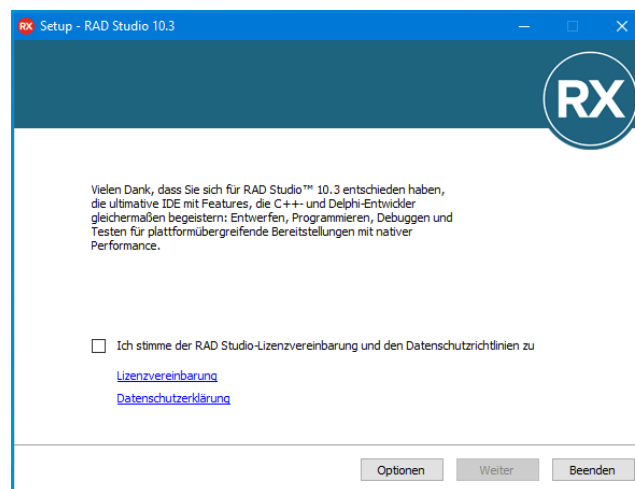


Abbildung 1 Installationsassistent von Delphi Community Edition

Im ersten Dialog hat man die Option (unter „Optionen“) das Zielverzeichnis zu ändern. Es werden aber dennoch viele Systembibliotheken auf dem Windows Laufwerk „C:“ installiert. Im nächsten Schritt wird nach einer Seriennummer gefragt, die man per eMail erhalten haben sollte:

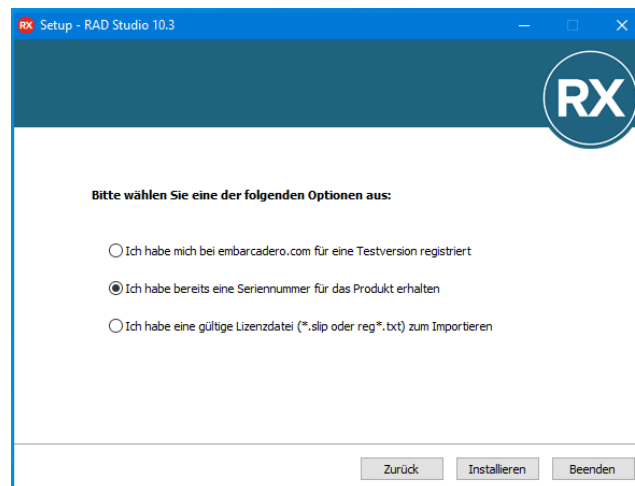


Abbildung 2a: Installationsassistent von Delphi Community Edition

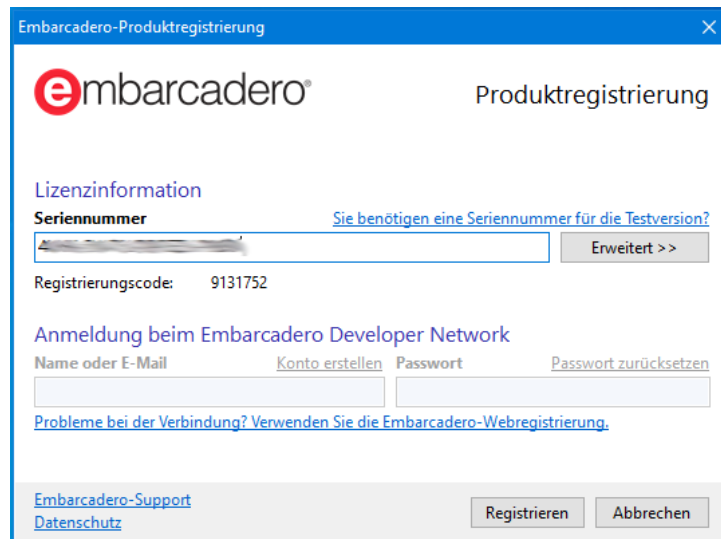


Abbildung 2b: Produktregistrierung beim ersten Start von Delphi

Während der Installation erscheint ein Registrierungsdialog. Delphi muss hier Online aktiviert werden. Dafür benötigt man einen Account im Embarcadero Developer Network (EDN). Wer Delphi als registrierter Benutzer heruntergeladen hat, wird nur nach der Seriennummer gefragt. Ansonsten muss man hier ein „Konto erstellen“.

Delphi Community Edition kann nicht gleichzeitig mit C++Builder Community Edition installiert werden.

Im Verlauf der Installation hat man die Möglichkeiten die verschiedenen Plattformen zu installieren. Für das Nachvollziehen in diesem Buch brauchen man nur die „Delphi Windows 32-Bit Community“ Plattform. Man kann natürlich alle auswählen:



Abbildung 3c: Plattformauswahl / Compilerauswahl

Man sollte im nächsten Schritt aber unbedingt folgendes auswählen:

- Deutsches Sprachpaket
- Samples (die Delphi Beispiele)
- Help (die kontextsensitive Hilfe, die lokal installiert wird)



Abbildung 4d: Auswahl der optionalen Funktionen

Ein nachträgliches Ändern der Plattformauswahl und der weiteren optionalen Funktionen ist jederzeit in der IDE möglich. Man kann also auch nachträglich die iOS oder Android Plattform installieren. Dazu wählt man bei installierter IDE im Menü: Tools | Plattformen verwalten... und man hat wieder diesen Dialog in der man weitere Plattformen oder Optionen installieren kann.

Die Installation lädt dann aus dem Internet die passenden Pakete

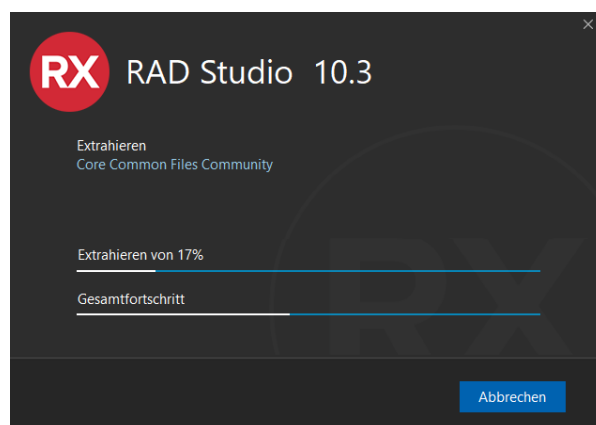


Abbildung 5e: Installationsausführung

### 1.2 Der erste Start

Nachdem die Installation erfolgreich abgeschlossen ist, wollen wir Delphi direkt starten. Nach der Konfiguration der IDE (Hell/Dukel-Modus und der Auswahl von Versionkontrollsystemen), die man erstmal mit „Weiter“ und „Los geht's!“ bestätigt, bekommen folgendes Fenster zu sehen:

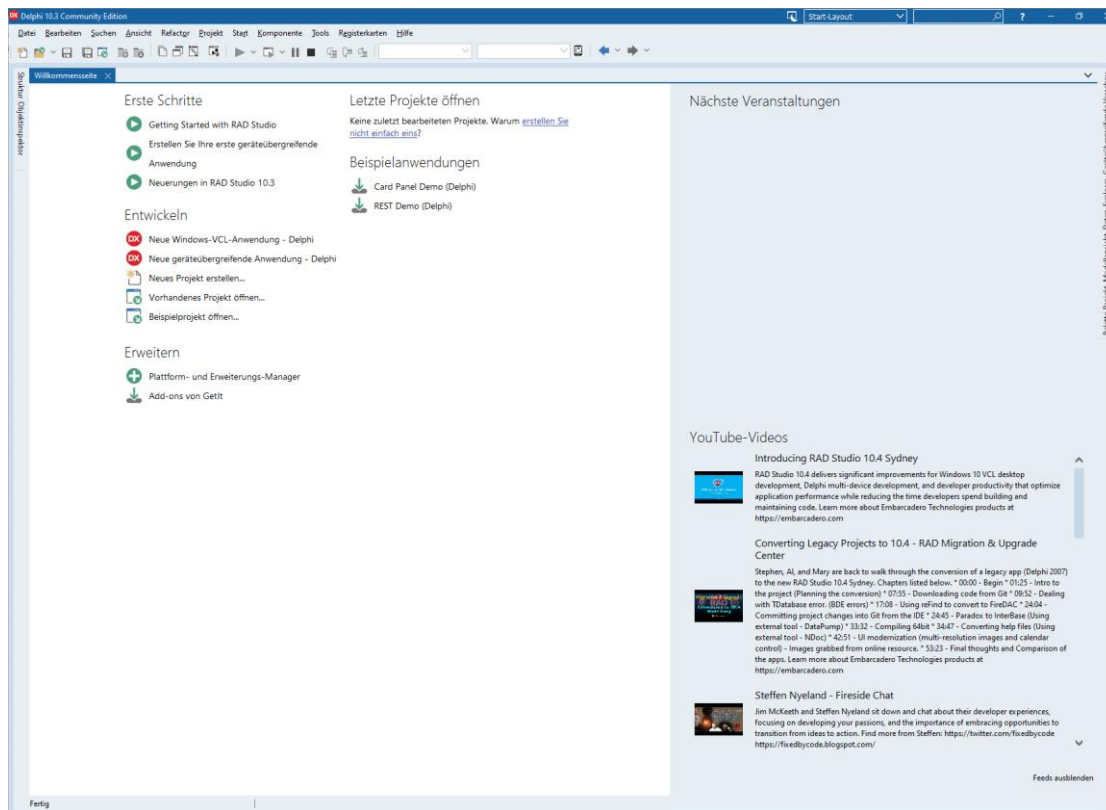


Abbildung 6: Entwicklungsumgebung mit Willkommenseite

Das also ist Delphi! In der Mitte sehen wir die Willkommenseite mit einem ausgewählten Newsfeed.

Die kleinen angedockten Fenster rings um die Willkommenseite sind momentan leer und ausgeblendet. Das ändert sich, sobald man ein neues Projekt anlegt oder ein bestehendes öffnet. Deshalb werden wir erst in den folgenden Kapiteln nach Bedarf auf die einzelnen Elemente der IDE (Integrated Development Environment) zu sprechen kommen.

### 1.3 Hallo Welt!

Bei Einführungen in eine Programmiersprache darf ein „Hallo Welt“-Beispiel nicht fehlen, damit der Leser sieht, wie schnell man etwas Kleines auf den Bildschirm bekommt. Darauf wollen wir hier nicht verzichten. Deshalb folgen Schritt-für-Schritt-Anleitungen für ein „Hallo Welt“ auf der Konsole und zwei mit grafischer Benutzeroberfläche (GUI – Graphical User Interface). In dem einen GUI-Beispiel kommt die seit Delphi 1 bekannte Komponenten-Bibliothek VCL (Visual Component Library) zum Einsatz, in dem anderen die mit Delphi XE2 eingeführte, plattformunabhängige Bibliothek

FireMonkey. In den später folgenden Kapiteln werden wir uns im Detail anschauen, was das alles zu bedeuten hat.

### 1.3.1 „Hallo Welt“ für Konsole

1. Starte Delphi.
2. Öffne das Menü „Datei“ und klicke auf „Neu“ „Weitere...“. Es öffnet sich die „Objektgalerie“.
3. Wähle im linken Baum „Delphi-Projekte“ aus und klicke dann auf „Konsolenanwendung“.

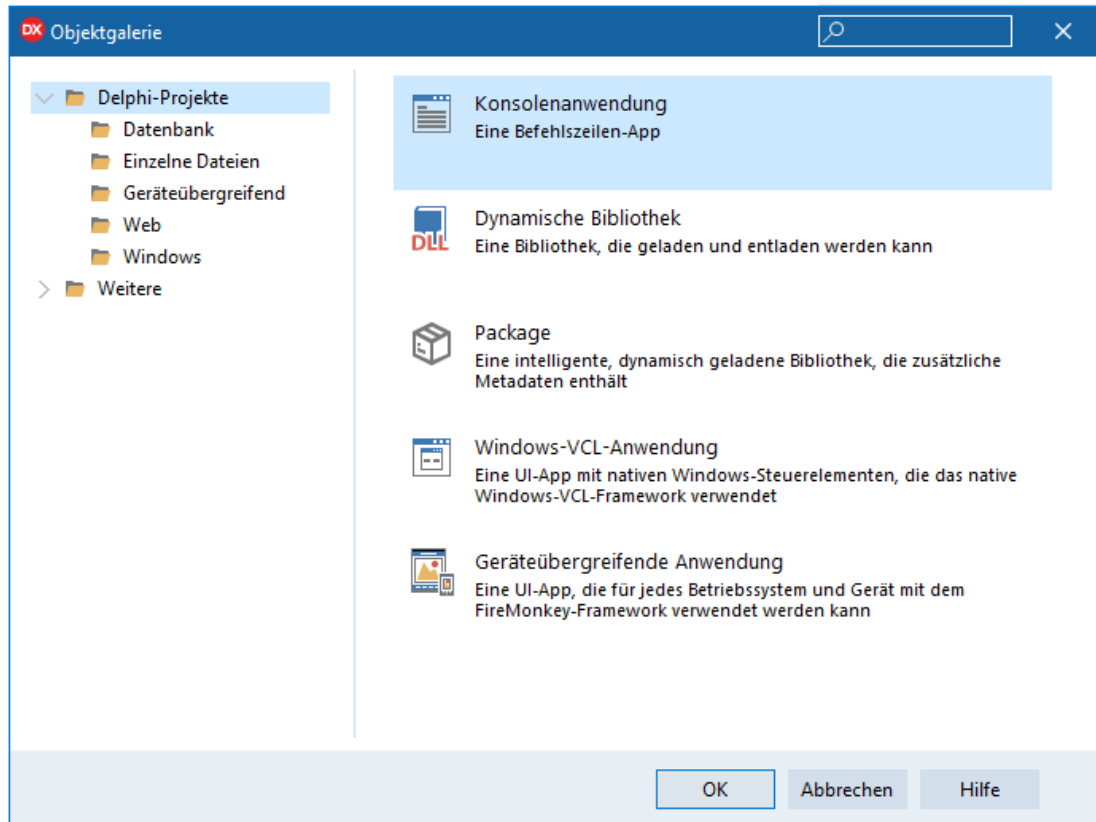


Abbildung 7: Objektgalerie

4. Ein Klick auf „OK“ erzeugt ein neues Delphi-Projekt mit ein paar generierten Zeilen Code:

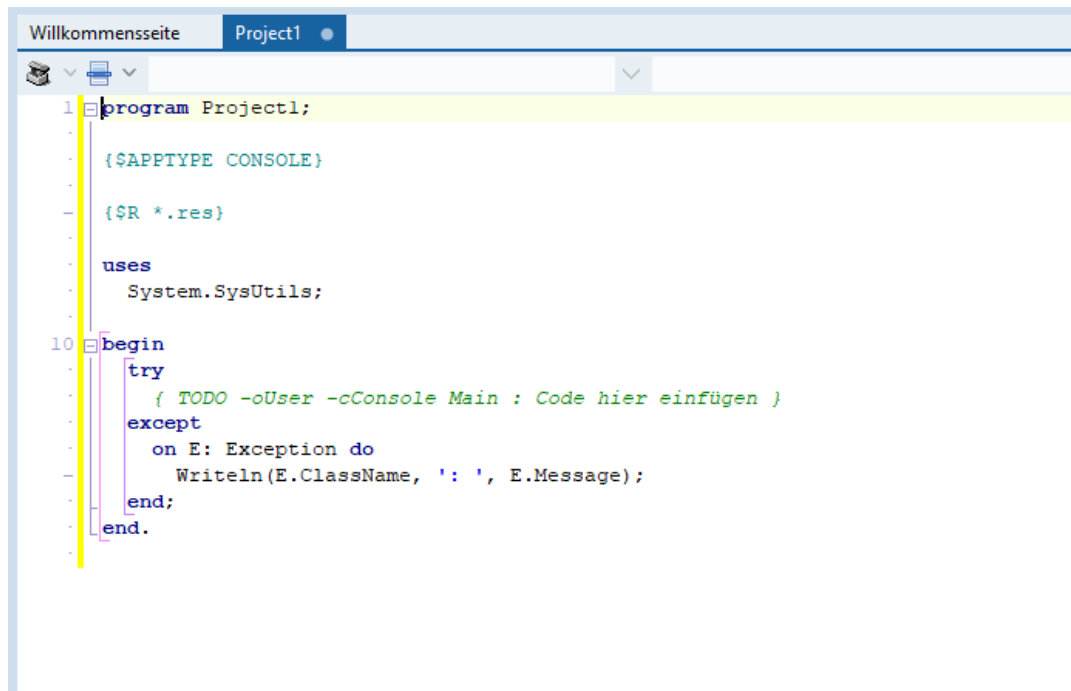


Abbildung 8: Rumpf einer Konsolenanwendung

5. Die erste Zeile definiert, wie unser Programm heißen soll (in diesem Beispiel „Project1“). Dann wird definiert, dass wir eine Konsolenanwendung programmieren (`{$APPTYPE CONSOLE}`), sowie unter „uses“ ein paar Standardfunktionen importiert. Zwischen „begin“ und „end.“ steht nun der eigentliche Code. Der Code, der bisher da steht, ist zur Ausgabe von eventuellen Fehlermeldungen gedacht und kann der Einfachheit halber vorerst ignoriert werden. In Zeile 12 befindet sich ein Kommentar in geschweiften Klammern, der anzeigt, dass unser Code an genau dieser Stelle eingefügt werden soll. Den Kommentar selbst können wir entfernen. Stattdessen fügen wir diese beiden Zeilen ein:

```
writeln('Hallo welt');
readln;
```

`Writeln` („write line“) gibt einen Text mit anschließendem Zeilenumbruch aus. `Readln` wartet auf eine Eingabe. Wir nutzen das an dieser Stelle, um das Programm nicht sofort wieder zu schließen, sondern auf Druck der Return-Taste zu warten. **Wichtige Regel in Delphi: Jede Anweisung endet mit einem Semikolon (Strichpunkt).** Dadurch ließen sich auch mehrere Anweisungen in eine einzige Zeile schreiben, wovon wir aus Gründen der Übersichtlichkeit aber dringend abraten.

6. Das Programm sieht nun so aus:



Abbildung 9: Konsolenanwendung mit "Hallo Welt"-Ausgabe

7. Ein Druck auf die Taste F9 startet das Programm:

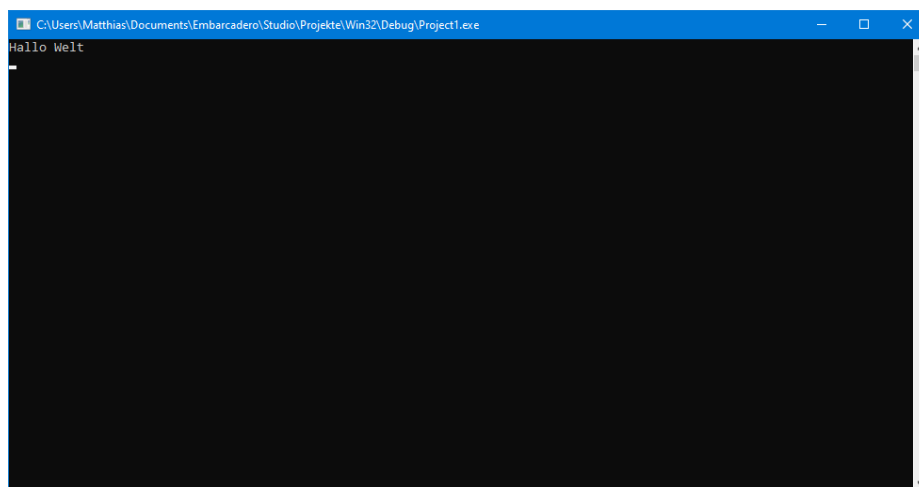


Abbildung 10: Laufende Konsolenanwendung

### 1.3.2 „Hallo Welt“ mit GUI (VCL)

1. Starte Delphi.
2. Öffne das Menü „Datei“ und klicke auf „Neu“ „Windows-VCL-Anwendung - Delphi“.



3. Es erscheint ein neues, leeres Fenster, das wir nach unseren Wünschen gestalten können.

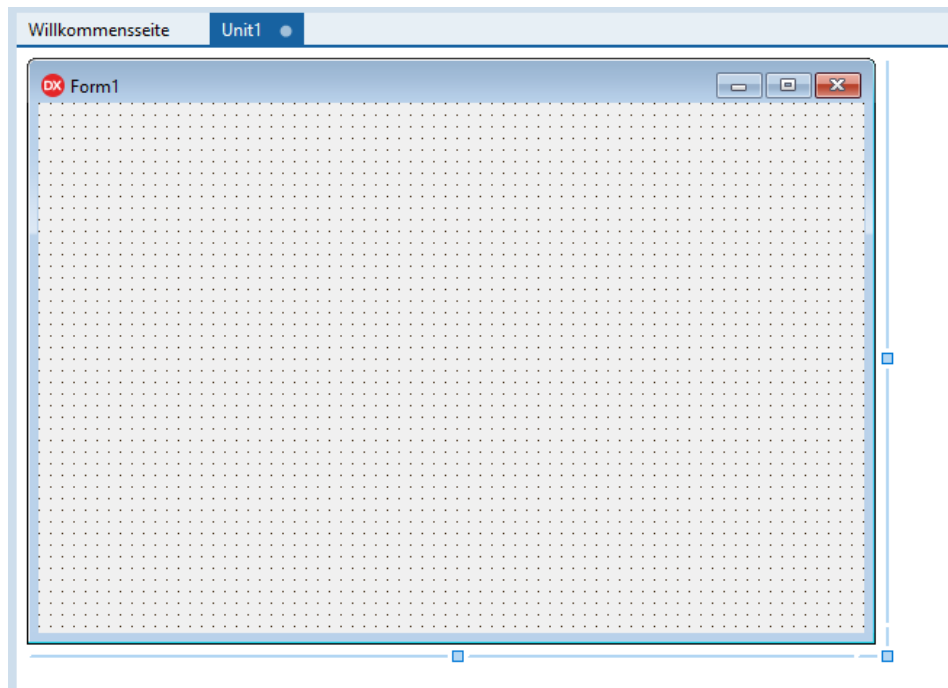


Abbildung 11: Leeres Fenster in der Entwicklungsumgebung

4. In der „Tool-Palette“ am rechten unteren Rand der Entwicklungsumgebung öffnen wir den Bereich „Standard“ und klicken auf „TButton“. (Bei „TButton“ handelt es sich um eine Komponente, die eine Schaltfläche darstellt, auf die der Anwender klicken kann.)

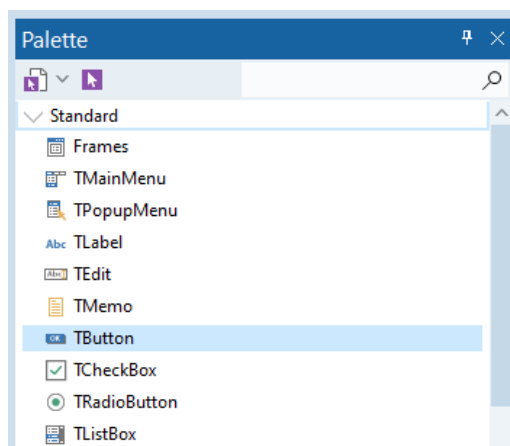


Abbildung 12: Tool-Palette

5. Anschließend klicken wir irgendwo auf das neue Fenster. An dieser Stelle erscheint nun der Button „Button1“ umgeben von acht blauen Markierungen, über die man das Label in seiner Größe verändern kann.

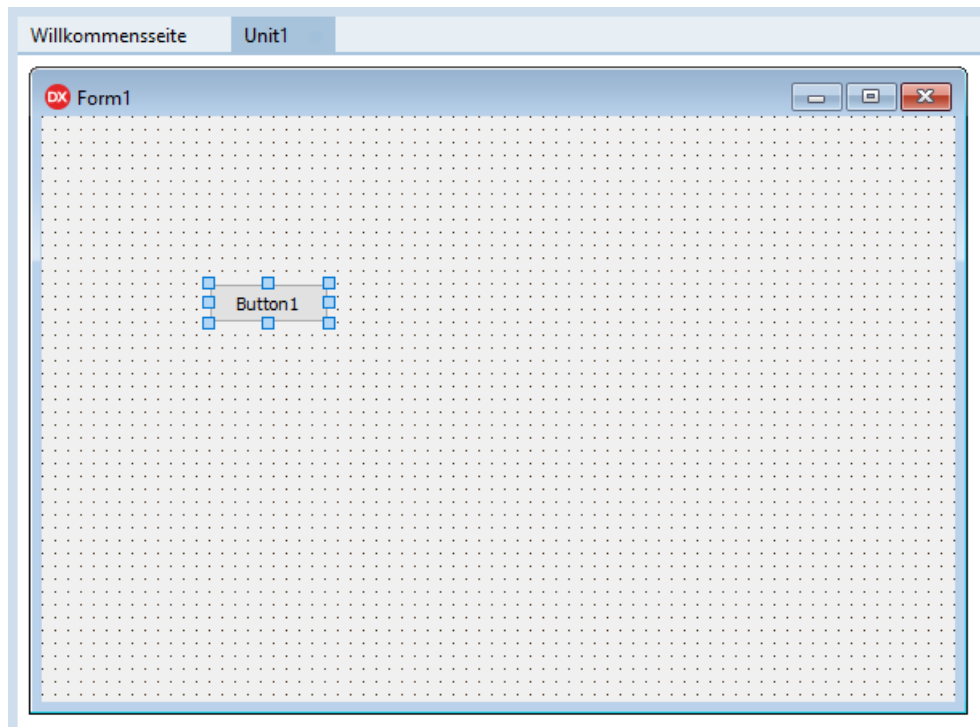


Abbildung 13: Fenster mit Label

6. Im „Objektinspektor“ auf der linken unteren Seite der Entwicklungsumgebung werden die Eigenschaften angezeigt. Die Eigenschaft „Caption“ steht für die sichtbare Beschriftung des Buttons. Als Wert ist momentan „Button1“ eingetragen. Diesen Wert verändern wir in „Hallo Welt!“.

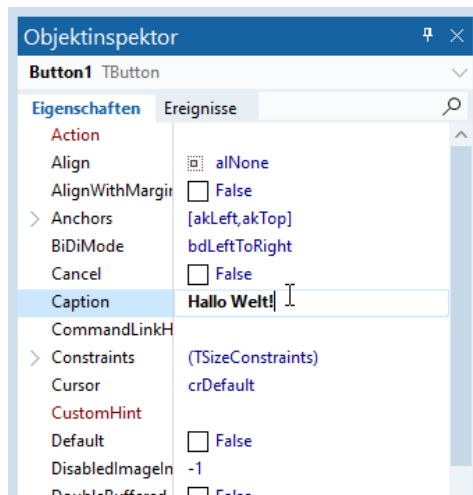
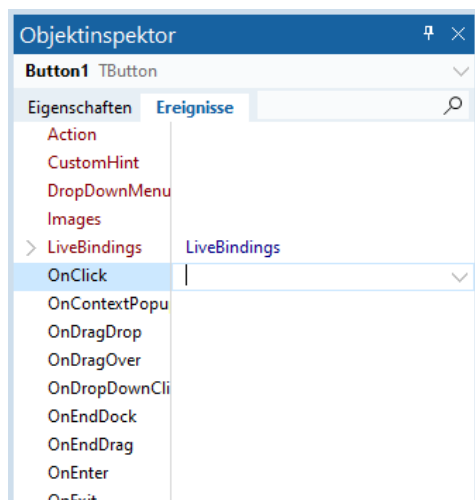


Abbildung 14: Objektinspektor

7. Bei einem Klick auf den Button soll auch etwas passieren. Dafür müssen wir eine Ereignisbehandlungsmethode schreiben. Wir wechseln im Objektinspektor auf die Seite

„Ereignisse“.



8. Unser Button soll auf Anklicken reagieren – dafür ist das Ereignis „OnClick“ zuständig. Wir führen in der rechten Spalte neben „OnClick“ einen Doppelklick aus. Daraufhin erzeugt Delphi automatisch eine leere Methode:

```
procedure TForm3.Button1Click(Sender: TObject);  
begin  
  
end;
```

9. Zwischen „begin“ und „end“ kommt der Code, der beim Anklicken des Buttons ausgeführt werden soll. Wir lassen einfach den Text „Hallo Welt!“ in einem kleinen Popup-Fenster anzeigen. Dafür gibt es in Delphi die Routine ShowMessage. Mit ihr kann man beliebigen Text anzeigen lassen.

```
procedure TForm3.Button1Click(Sender: TObject);  
begin  
    ShowMessage('Hallo Welt!');  
end;
```

10. Ein Druck auf die Taste F9 startet das Programm:

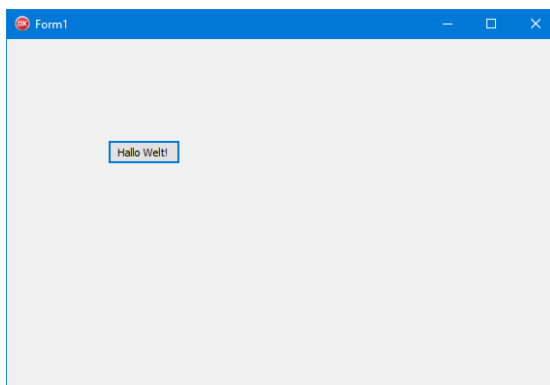


Abbildung 15: Laufende GUI-Anwendung

Und ein Klick auf den Button zeigt das kleine Popup-Fenster an:

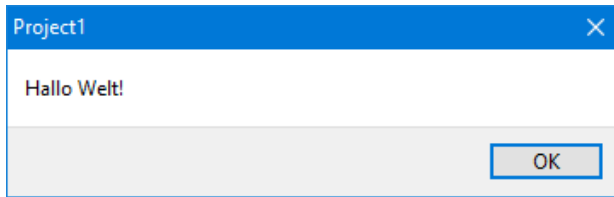


Abbildung 16: Popup-Fenster mit ShowMessage

### 1.3.3 „Hallo Welt“ mit GUI (FireMonkey)

1. Starte Delphi.
2. Öffne das Menü „Datei“ und klicke auf „Neu“ „Geräteübergreifende Anwendung - Delphi“.
3. Wähle „Leere Anwendung“
4. Es erscheint ein neues, leeres Fenster, das wir nach unseren Wünschen gestalten können.

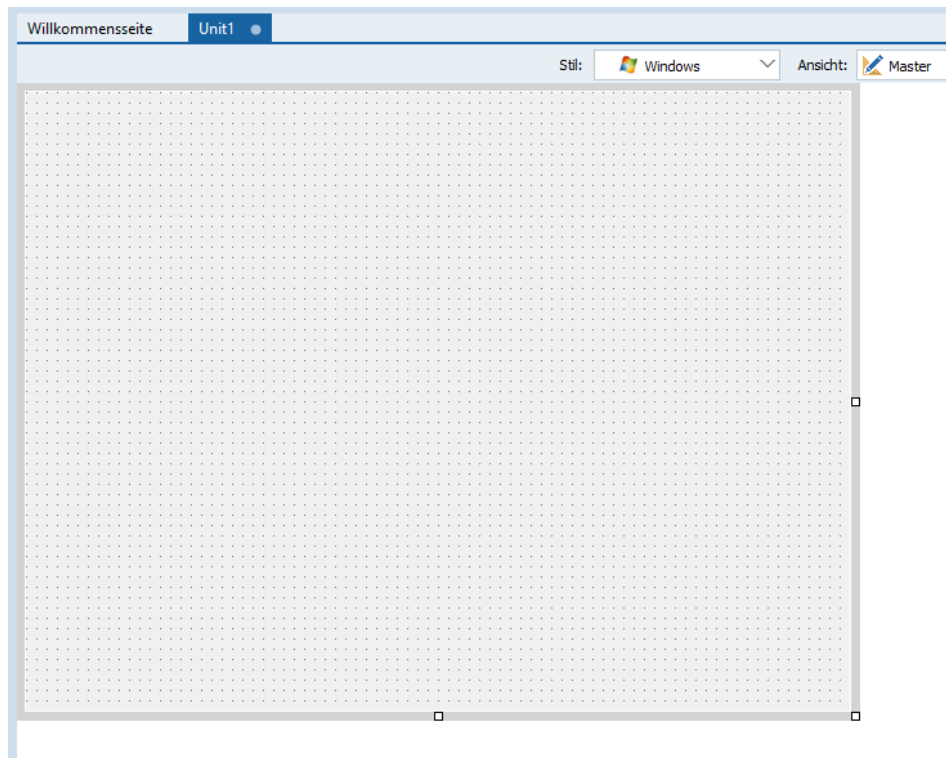


Abbildung 17: Leeres Fenster einer FireMonkey-Anwendung

5. In der „Tool-Palette“ am rechten unteren Rand der Entwicklungsumgebung öffnen wir den Bereich „Standard“ und klicken auf „TLabel“. (Bei „TLabel“ handelt es sich um eine Komponente zur Anzeige von Text in einer Anwendung.)

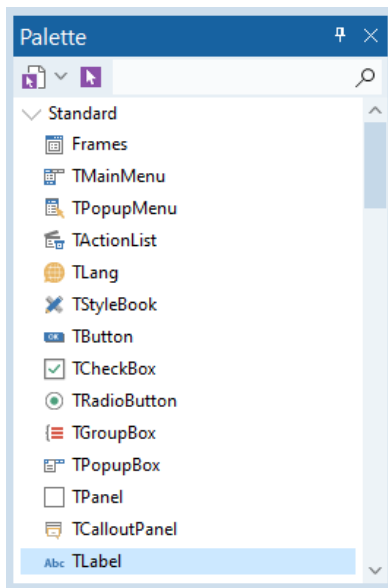


Abbildung 18: FireMonkey-Toolpalette

6. Anschließend klicken wir irgendwo auf das neue Fenster. An dieser Stelle erscheint nun der Text „Label1“ umgeben von acht grau-blauen Markierungen, über die man das Label in seiner Größe verändern kann.

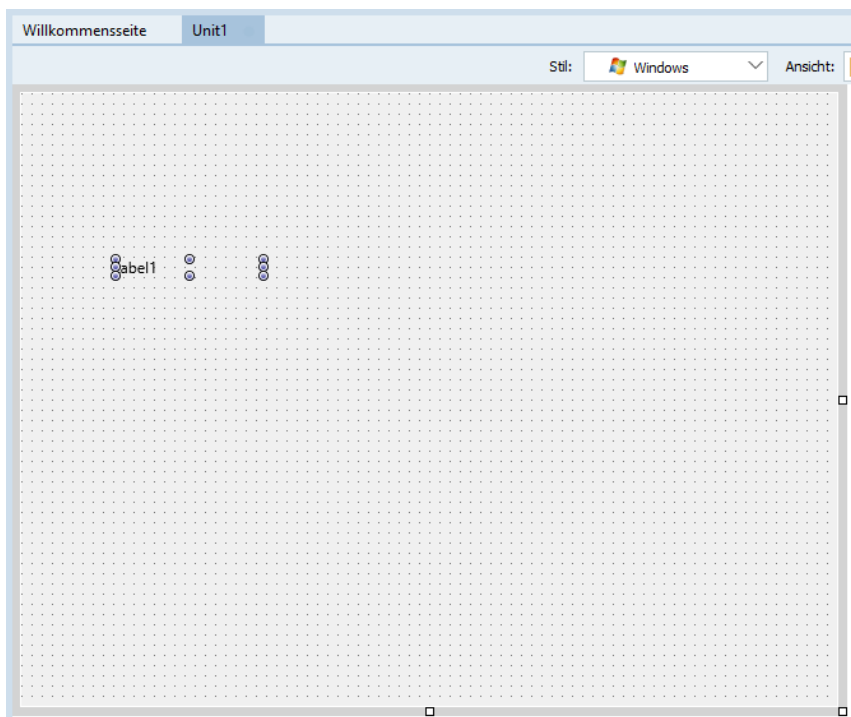


Abbildung 19: FireMonkey-Formular mit Label

7. Im „Objektinspektor“ auf der linken unteren Seite der Entwicklungsumgebung werden die Eigenschaften angezeigt. Die Eigenschaft „Text“ (nicht „Caption“ wie bei VCL-Anwendungen!) steht für die sichtbare Beschriftung. Als Wert ist momentan „Label1“ eingetragen. Diesen Wert verändern wir in „Hallo Welt!“.

8. Nun wollen wir noch einen Schritt weiter gehen und den Text animieren, was mit FireMonkey recht einfach geht. Wir gehen dazu in die Toolpalette und führen einen Doppelklick auf TFloatAnimation (Abschnitt „Animationen“) aus.

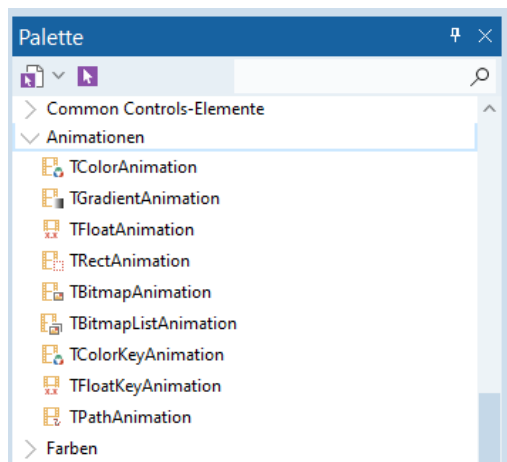


Abbildung 20: Animationskomponenten in der FireMonkey-Toolpalette

9. Die FloatAnimation-Komponente landet daraufhin in der Strukturansicht (oben links). Hier müssen wir sie allerdings noch per Drag&Drop auf Label1 ziehen, damit sie diesem untergeordnet ist. Das sollte dann so aussehen:

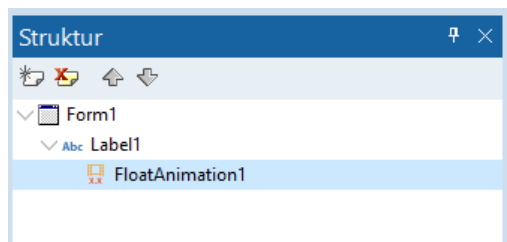
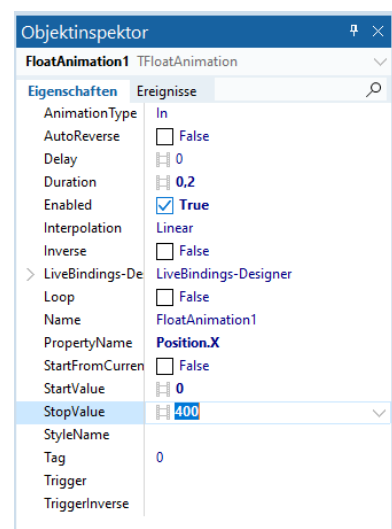


Abbildung 21: Strukturansicht

10. Im Objektinspektor wählen wir nun FloatAnimation1 aus und ändern ein paar Eigenschaften: Enabled auf True, Loop auf True, und bei „PropertyName“ wählen wir „Position.X“ aus. Hierbei handelt es sich um die Eigenschaft des Elternelements (also des Labels), die durch die Animation verändert werden soll. Der Wert dieser Eigenschaft bewegt sich zwischen StartValue und StopValue. Wir setzen diese Werte auf 0 und 400. Über die Eigenschaft „Duration“ lässt sich die Dauer der Animation festlegen, also wie viele Sekunden vergehen dürfen, bis das Label von X-Position 0 bis 400 gewandert ist. Bei der standardmäßig vorgegebenen Dauer von 0,2 Sekunden läuft das Ganze sehr schnell ab.



11. Ein Druck auf die Taste F9 startet das Programm. Wir sehen nun den Text „Halo Welt“ durch das Fenster flitzen:

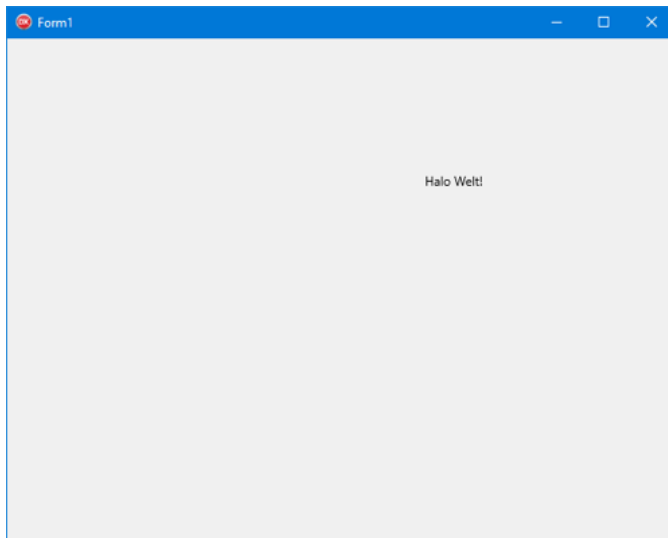


Abbildung 22: Laufende FireMonkey-Anwendung

***Nun geht es an die in Delphi verwendete Programmiersprache „Object Pascal“. Das Schnellstartkapitel beschreibt die wichtigsten Elemente mit Beispielen, ohne jedoch zu sehr in die Tiefe zu gehen.***

## 2 Schnellstart – Das Wichtigste

Bevor wir in die Details gehen, wollen wir an dieser Stelle die wichtigsten Dinge der Delphi-Programmierung kurz zusammenfassen. Dadurch lernst du die wichtigsten Begriffe und Konzepte schon einmal kennen.

### 2.0 Allgemein

#### 2.0.1 Über die Sprache

Bei Object Pascal handelt es sich um eine höhere, objektorientierte Programmiersprache. Die wichtigsten Dinge aus dem zugrunde liegenden Pascal sind natürlich noch enthalten, allerdings hat sich die Sprache, die mit Delphi 1 eingeführt wurde, seitdem stark weiterentwickelt.

#### 2.0.2 Grundlegendes zur Syntax

Wie in vielen anderen Programmiersprachen werden mehrere Befehle durch **Semikolon (;)** getrennt. Im Gegensatz zu C/C++ unterscheidet Object Pascal *nicht* zwischen **Groß- und Kleinschreibung**. Während `meine_variable` und `Meine_Variable` in C/C++ unterschiedlich sind, sind sie für die Delphi-Sprache gleich. In dieser Hinsicht braucht der Entwickler also nicht so viel Disziplin beim Programmieren. Leerzeichen (natürlich nicht innerhalb eines Bezeichners) und Leerzeilen können verwendet werden, um die Übersichtlichkeit im Code zu erhöhen.

Das heißt natürlich nicht, dass man als Programmierer machen kann, was man will. Object Pascal ist nämlich für ihre **Typstrenge** bekannt. Im Gegensatz zu Visual Basic muss eine Variable vor ihrer Verwendung in einem bestimmten Bereich deklariert werden. Ist ihr somit einmal ein Typ zugeordnet, kann sie keine Werte eines anderen Typs aufnehmen, nur Werte des gleichen Typs oder eines Untertyps. Auch Zuweisungen zwischen Variablen unterschiedlichen Typs lassen sich häufig nur über Konvertierfunktionen (wie z.B. `IntToStr`, um eine Ganzzahl in einen String umzuwandeln) bewerkstelligen. Doch dazu im Folgenden mehr.

### 2.1 Bestandteile eines Delphi-Projekts

#### 2.1.1 Projektdatei (\*.dpr)

In unserem Hallo-Welt-Beispiel haben wir bereits gesehen, dass ein Delphi-Projekt immer genau eine Projektdatei (Dateiendung `dpr`) enthält. Hierin steht, welche weiteren Bestandteile zu diesem Projekt gehören. Außerdem findet sich hier der Code, der beim Start der Anwendung als erstes ausgeführt wird. Bei einem Projekt mit grafischer Benutzeroberfläche werden hier die Fenster geladen und das



Hauptfenster angezeigt. Bei einer reinen Konsolenanwendung kann auch sämtlicher Code in der Projektdatei stehen.

Projektdatei einer Konsolenanwendung (Menü Datei/Neu/Weitere.../Konsolenanwendung):

```
program Project1;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

begin
  try
    { TODO -oUser -cConsole Main : Code hier einfügen }
  except
    on E: Exception do
      writeln(E.ClassName, ': ', E.Message);
    end;
  end.
end.
```

Projektdatei einer VCL-Anwendung (Menü Datei/Neu/Windows-VCL-Anwendung) über Projekt/Quelltext anzeigen:

```
program Project1;

uses
  Vcl.Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Bei VCL-Anwendungen muss an der Projektdatei in der Regel nichts verändert werden.

### 2.1.2 Quellcode (\*.pas)

In der Regel wird der Quellcode inhaltlich gegliedert auf mehrere Dateien verteilt, um die Übersicht zu behalten. Diese Dateien werden in Delphi „Unit“ genannt und haben die Dateierweiterung PAS. Eine Unit besteht mindestens aus den beiden Teilen „interface“ und „implementation“. Der Interface-Abschnitt enthält sozusagen das Inhaltsverzeichnis der Unit. Hier wird aufgelistet, welche Prozeduren, Funktionen, Klassen und Methoden die Unit enthält. Im Implementation-Teil folgt dann die eigentliche Programmlogik.

Eine neu angelegte Unit (Menü Datei/Neu/Unit) sieht so aus:

```
unit Unit1;  
  
interface  
  
implementation  
  
end.
```

Will man von einer Unit auf eine andere zugreifen, wird deren Name in der Uses-Klausel angegeben. Davon kann es zwei Stück geben: Die Uses-Klausel im Interface der Unit importiert Units, die innerhalb des Interface-Teils benötigt werden. Units, die nur im Implementierungsteil benötigt werden, können in der dortigen Uses-Klausel angegeben werden.

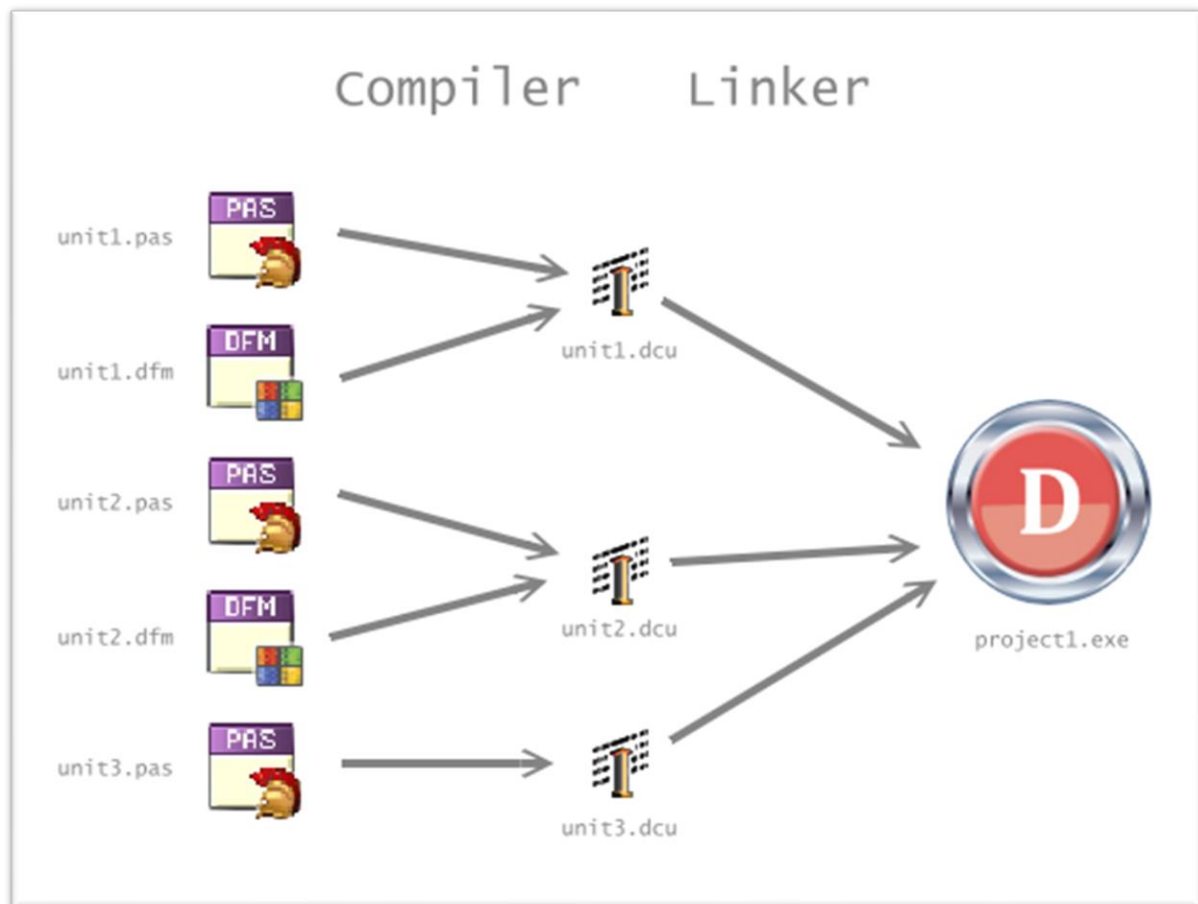
```
unit Unit1;  
  
interface  
  
uses Unit2;  
  
implementation  
  
uses Unit3;  
  
end.
```


### 2.1.3 Fensterdefinition (\*.dfm)

Zu einer Unit kann es noch Formulardateien (Dateiendung DFM (Delphi Form)) geben. Diese beschreiben das Aussehen und den Aufbau eines Fensters bzw. Formulars. Immer, wenn man ein neues Formular anlegt, werden sowohl eine PAS-, als auch eine DFM-Datei erzeugt.

### 2.1.4 Eine ausführbare Datei (EXE) erzeugen

Hat man ein eigenes Programm geschrieben, möchte man es natürlich auch ausführen. Dazu muss aber der geschriebene Code erst einmal in Maschinencode übersetzt werden. Das ist die Aufgabe des Compilers. Klickt man in der Delphi-Entwicklungsumgebung auf den Menüpunkt „Projekt“/„Projekt erzeugen“ (Tastenkombination Strg + F9), wird jede Unit einzeln übersetzt und als Datei mit der Endung dcu (Delphi Compiled Unit) gespeichert. Der Linker führt anschließend alles zu einer ausführbaren Datei (Endung exe) zusammen. Diese Datei kann ausgeführt werden wie jede andere Windows-Anwendung auch.



Dieser ganze Prozess, das Übersetzen in DCUs, das Zusammenbinden der einzelnen DCUs zu einer EXE, sowie das Starten derselben, wird durch den Button mit dem kleinen grünen Pfeil  (bzw. Taste F9) angestoßen.

Dabei werden nur die geänderten Units neu übersetzt und für alle anderen Units die bereits bestehenden DCUs verwendet. Möchte man wirklich alle Dateien neu erzeugen, so kann man das über den Menüpunkt Projekt – Projekt erzeugen (Strg+F9) bewerkstelligen.

### 2.1.5 Delphi-Programme weitergeben

Die Weitergabe eines selbstgeschriebenen Delphi-Programms ist im Normalfall ganz einfach: Die Exe-Datei reicht völlig aus. Auch jemand, der kein Delphi besitzt, kann diese ausführen. Anders verhält es sich, wenn du auf Datenbanken zugreifst oder Bibliotheken (DLLs) ansprichst, die nicht auf jedem Windows-PC vorhanden sind. Dann musst du diese (z.B. auch Datenbanktreiber o.ä.) mitliefern.

## 2.2 Variablen

### 2.2.1 Was sind Variablen?

Eine Variable ist ein Behälter für etwas, das zur Programmlaufzeit im Arbeitsspeicher gehalten werden soll, z.B. eine Zahl oder ein Text. Jede Variable hat in Delphi einen festen Datentyp. Die wichtigsten Datentypen sind Integer (ganze Zahl), Real (Kommazahl), String (Text) und Boolean (Wahrheitswert true oder false). In einer Integer-Variablen kann nur ein Integer-Wert abgelegt werden. Allerdings gibt es die Möglichkeit, Werte von einem Datentyp in einen anderen umzuwandeln, z.B. eine Zahl in einen String.

Beispiel: Der Integer-Variablen „zahl“ wird der Wert 42 zugewiesen.

```
var zahl: Integer;  
...  
zahl := 42;
```

### 2.2.2 Variablen deklarieren

Variablen können in Delphi zwar mittlerweile (Version 10.3 und höher) an beliebiger Stelle im Code deklariert werden. Wir verzichten aber darauf. Stattdessen gibt es Variablendeklarationsabschnitte, die mit „var“ eingeleitet werden und zu Beginn einer Unit oder einer Funktion stehen.

Beispiel einer Konsolenanwendung (Menü Datei/Neu/Weitere/Konsolenanwendung):

```
program Project2;  
  
{$APPTYPE CONSOLE}  
  
{$R *.res}  
  
uses  
    System.SysUtils, Vcl.Dialogs;  
  
var  
    zahl: Integer;  
  
begin  
    zahl := 42;  
    ...  
end.
```

Gelb hervorgehoben ist hier die Stelle, an der deklariert wird, dass es sich bei „zahl“ um eine ganze Zahl (Integer) handelt.

Wichtige Datentypen:

- Integer (ganze Zahlen)
- Real (Gleitkommazahlen)
- String (Text)

- Boolean (Wahrheitswerte true oder false)

### 2.2.3 Werte zuweisen

Im Code, der sich zwischen „begin“ und „end“ befindet, wird dieser Variablen die Zahl 42 zugewiesen. Zuweisungen von Werten zu einer Variablen werden in Object Pascal mit := vorgenommen. Dabei müssen wir natürlich darauf achten, dass der Wert rechts den gleichen Typ hat wie die Variable auf der linken Seite, in unserem Fall also „ganze Zahl“ (Integer).

Würden wir die Zuweisung weglassen, wäre die Variable `zahl` mit 0 vorbelegt. In anderen Fällen kann der Wert auch undefiniert (also quasi eine „Zufallszahl“) sein. Man sollte Variablen deshalb immer initialisieren, d.h. ihr einen Anfangswert zuweisen.

Verwendet man eine String-Variablen und möchte dieser einen Text zuweisen, so muss der Text in einfachen Anführungszeichen (Umschalt+““-Taste) stehen.

Man kann auch dieselbe Variable mehrmals in einer Zuweisung verwenden. Auf diese Weise kann man den zugewiesenen Wert vom momentanen Wert abhängig machen, also beispielsweise eine Variable um 1 erhöhen wie in folgendem Beispiel:

```
x := x + 1;
```

Der aktuelle Wert wird aus `x` ausgelesen, dann wird 1 addiert und das Ergebnis wieder in `x` gespeichert. Es handelt sich also nicht um eine Gleichung wie in Mathematik, sondern eben um eine Zuweisung.

### 2.2.4 Variablen umwandeln

Nun ergänzen wir den Code in obigem Beispiel und ersetzen die drei Punkte durch

```
ShowMessage('Meine Zahl: ' + IntToStr(zahl));
```

Dadurch wird die Variable `zahl` in einem Pop-up-Fenster ausgegeben. Zur Anzeige eines Pop-up-Fensters gibt es die Funktion `ShowMessage`. Diese ist in der Unit „`Vcl.Dialogs`“ definiert, weshalb wir diese in die Uses-Klausel aufnehmen müssen. Zudem ist zu beachten, dass im Pop-up-Dialog nur Text (Typ `String`) angezeigt werden kann. Wir müssen unsere Zahl also in einen String umwandeln. Das passiert über den Aufruf von `IntToStr` (= „Integer to String“). Über die Taste F9 kann diese Beispielanwendung direkt ausgeführt werden.

`ShowMessage` ist im Allgemeinen sehr hilfreich, um sich Zwischenergebnisse anzeigen zu lassen, wenn man auf Fehlersuche ist oder prüfen will, was sein Programm gerade rechnet.

Wichtige Funktionen zum Umwandeln von Datentypen:

Integer → String	<code>IntToStr</code>
String → Integer	<code>StrToInt</code>
Real → String	<code>FloatToStr</code>
String → Real	<code>StrToFloat</code>
Boolean → String	<code>BoolToStr</code>
String → Boolean	<code>StrToBool</code>

All diese Umwandlungsfunktionen befinden sich in der Unit SysUtils. Diese muss also in der Uses-Klausel einer Unit angegeben sein, wenn man eine der Funktionen verwenden möchte.

### 2.2.5 Mit Variablen rechnen

Normalerweise arbeitet man mit Variablen, um ihren Wert während des Programmablaufs auch verändern zu können. In Object Pascal verwendet man die Zeichen + - \* / für die bekannten Grundrechenarten. Für die Division gibt es noch die zwei Operatoren div (ganzzahlige Division) und mod (der Rest einer ganzzahligen Division. Beispiel:

- $9 / 2 = 4.5$
- $9 \text{ div } 2 = 4$  („4 Rest 1“)
- $9 \text{ mod } 2 = 1$  („4 Rest 1“)

Das Ergebnis einer Berechnung wird über den Zuweisungsoperator := einer Variablen zugewiesen:

```
var
  zahl1, zahl2, zahl3: Integer;
begin
  zahl1 := 20;
  zahl2 := 22;
  zahl3 := zahl1 + zahl2;
```

Ist eine Variable vom Typ String, also ein Text, kann man natürlich nicht so einfach mit ihr rechnen. Wenn man weiß (oder erwartet), dass der String eine Zahl enthält, kann man den String in eine Zahl umwandeln und anschließend wie oben beschrieben damit rechnen:

```
var
  eingabe: String;
  zahl1, zahl2: Integer;
begin
  eingabe := '42';
  zahl1 := StrToIntDef(eingabe, 0);
  zahl2 := zahl1 * 2;
```

In diesem Beispiel würde auch die Funktion StrToInt funktionieren. Allerdings steigt diese mit einem Fehler aus, wenn der umzuwandelnde String keine Zahl enthält. Würden wir direkt auf die Eingabe eines Benutzers zugreifen, müssten wir damit rechnen, auch ungültige Eingaben zu bekommen. Dabei hilft StrToIntDef. Lässt sich der String nämlich nicht in eine Zahl umwandeln, wird der angegebene Default-Wert (hier: 0) verwendet.

Aber auch mit einem beliebigen Text lässt sich das Plus-Zeichen verwenden, nämlich dann, wenn man zwei oder mehr Strings zu einem zusammenhängen möchte:

```
var
  ausgabe, eingabe: String;
begin
  eingabe := 'Hallo welt!';
  ausgabe := 'Das ist der wert von „eingabe“: ' + eingabe;
```

## 2.3 Schleifen und Bedingungen

### 2.3.1 Die for-Schleife

In der Programmierung ist es häufig erforderlich, dass bestimmte Code-Abschnitte mehrfach ausgeführt werden oder nur unter bestimmten Bedingungen. Zum Mehrfach-Ausführen gibt es sog. Schleifen. In diesem Schnellstarter-Kapitel werden wir uns die for-Schleife ansehen. Object Pascal kennt weitere Schleifenarten. Diese werden wir im Vertiefungsteil im späteren Verlauf dieses E-Books kennenlernen.

Beispiel: Der Text „Hallo Welt“ aus unserem Beispiel „Hallo Welt für Konsole“ soll 42-mal ausgegeben werden:

```
program Project1;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

var
  i: Integer;

begin
  for i := 1 to 42 do
    writeln('Hallo welt');
  end.
```

Bei der for-Schleife wird einer Schleifenvariablen (im Beispiel „i“) ein Startwert zugewiesen (im Beispiel 1). Der dann folgende Code wird so oft ausgeführt, bis die Schleifenvariable den Zielwert (im Beispiel 42) erreicht hat. Bei jedem Durchlauf wird der Wert der Schleifenvariablen automatisch um 1 erhöht. Die Schleifenvariable (auch Zählvariable genannt) definiert man wie jede andere Variable auch mit „var“. Sie wird nur in der Schleife genutzt und darf vom Programmierer nicht verändert werden. Ansonsten erhält man einen Fehler im Compile-Vorgang.

Möchte man mehr als eine Zeile Code innerhalb der Schleife ausführen, muss dieser in einem Begin-End-Block notiert werden:

```
begin
  for i := 1 to 42 do
    begin
      writeln('Hallo welt');
      // ... und noch mehr Code
    end;
  end.
```

Innerhalb der Schleife kann man auf die Schleifenvariable zugreifen und diesen z.B. ausgeben:



```
begin
  for i := 1 to 42 do
    begin
      writeln('Hallo welt im ' + IntToStr(i) + '. Durchlauf');
    end;
  end.
```

### 2.3.2 Bedingungen

Soll ein bestimmter Code nur unter bestimmten Bedingungen ausgeführt werden, verwendet man „if“ (engl. „falls“).

Beispiel: Der Text „Hallo Welt“ soll nur ausgegeben werden, wenn die Schleifenvariable „i“ größer als 10 ist:

```
begin
  for i := 1 to 42 do
    begin
      if i > 10 then
        writeln('Hallo welt ab dem 11. Durchlauf');
      end;
    end.
```

Ein wichtiger Hinweis: Bei „if“ handelt es sich um eine Bedingung, *nicht* um eine Schleife, da **if** ja nichts mehrfach ausführt. Der Begriff „if-Schleife“, von dem man ab und zu im Internet liest, ist also falsch.

Wie auch schon bei Schleifen gesehen, kann hinter **then** ein Begin-End-Block folgen, wenn mehrere Zeilen Code ausgeführt werden sollen, falls diese Bedingung wahr ist.

Möchte man etwas ausführen, falls diese Bedingung nicht zutrifft, kann man das Schlüsselwort **else** (engl. „sonst“) verwenden:

```
if i > 10 then
begin
  // wird ausgeführt, wenn i größer als 10 ist
end
else
begin
  // wird in allen anderen Fällen ausgeführt
end;
```

Zu beachten ist, dass die Zeile vor **else** nicht mit einem Semikolon beendet wird wie es sonst üblich ist.

#### 2.3.2.1 Logische Ausdrücke

Hinter „if“ steht also ein logischer Ausdruck, der mit den logischen Operatoren **>**, **<**, **=**, **<=**, **>=**, **<>** und **not** arbeitet. Es ist auch möglich, mehrere Ausdrücke mit **and** oder **or** zu verbinden. Dann ist es jedoch erforderlich, die einzelnen Ausdrücke mit einer Klammer zu versehen:

```
if (i > 10) and (i < 1000) then
```

Wie funktioniert das genau? Jeder einzelne logische Ausdruck hat ein Ergebnis, nämlich wahr (true) oder falsch (false). In obigem Beispiel würde das für  $i := 42$  so aussehen:

- $42 > 10 \rightarrow \text{wahr}$
- $42 < 1000 \rightarrow \text{wahr}$

Beide Ausdrücke sind mit dem logischen Und (**and**) verknüpft. Auch dieser Ausdruck hat als Ergebnis wahr oder falsch. Ein Ausdruck mit **and** ist genau dann wahr, wenn beide Ausdrücke links und rechts davon wahr sind. In jedem anderen Fall ist das Ergebnis falsch (false).

In obigem Beispiel würde der gesamte Ausdruck für  $i := 42$  also true ergeben, weil ja beide Einzelausdrücke true sind. Für  $i := 2$  sähe das ganz anders aus: Da wäre der zweite Ausdruck ( $2 < 100$ ) zwar immer noch wahr, der erste ( $2 > 10$ ) jedoch nicht. Dadurch wird das Gesamtergebnis zu false, und der Code nach der if-Bedingung wird nicht ausgeführt.

Anders sieht es bei einer Oder-Verknüpfung (**or**) aus. Hierbei reicht es, wenn eine der beiden Ausdrücke wahr ist. Es handelt sich allerdings nicht um ein entweder-oder. D.h. eine or-Verknüpfung ist auch wahr, wenn beide Ausdrücke wahr sind. Möchte man, dass nur genau einer der beiden Ausdrücke wahr sein darf, verwendet man den Operator **xor**.

#### Ein praktisches Beispiel:

Umgangssprache: „Samstags und sonntags darf ich ins Kino gehen, wenn meine Freunde mitkommen.“

Der logische Ausdruck: **Wenn** heute Samstag **oder** Sonntag ist **und** meine Freunde mitkommen, **dann** darf ich ins Kino gehen.

Und in Delphi-Syntax:

```
if ((Wochentag(heute) = 'Samstag') or (Wochentag(heute) = 'Sonntag'))  
  and (FreundeKommenMit > 0) then  
  InKinoGehen;
```

## 2.4 Unterprogramme: Prozeduren und Funktionen

Und noch ein grundlegender Bestandteil der Programmierung: Prozeduren und Funktionen. Code sollte in kleine Einheiten unterteilt werden, die einen sprechenden Namen haben, der besagt, was der enthaltene Code tut. Diese kleinen Einheiten werden Prozeduren und Funktionen genannt, wobei sich diese nur darin unterscheiden, ob sie nach ihrer Abarbeitung einen Ergebniswert zurückliefern (Funktionen) oder nicht (Prozeduren).

### 2.4.1 Parameter

Sowohl Funktionen als auch Prozeduren kann man beim Aufruf Parameter mitgeben, mit denen dann gearbeitet wird. Im Kopf der Prozedur wird in Klammern angegeben, welche Parameter beim Aufruf übergeben werden müssen. Beispiel:

```
procedure PrintName(name: String);
```

Das ist der Kopf (die Signatur) einer Prozedur. Sie erwartet beim Aufruf einen String-Wert. Innerhalb der Prozedur kann dieser String-Wert unter dem Variablennamen „name“ verwendet werden. Sobald die Prozedur zu Ende ist, ist die Variable „name“ nicht mehr bekannt. Der Wert an der Aufruf-Stelle ändert sich nicht.

Beispiel eines Aufrufs:

```
PrintName('Agi');
```

Erwartet eine Prozedur oder Funktion mehrere Parameter, werden sie im Kopf der Prozedur durch Semikolon getrennt angegeben. Beim Aufruf müssen die Werte in exakt der gleichen Reihenfolge angegeben werden. Die Typen müssen übereinstimmen.

Beispiel mit zwei Parametern:

```
procedure PrintName(name: String; alter: Integer);
```

Aufruf:

```
PrintName('Agi', 128);
```

### 2.4.2 Prozeduren

Beispiel: Die Schleife aus vorigem Beispiel soll in eine Prozedur verschoben werden, der die Anzahl der Schleifendurchläufe mitgegeben werden kann:

```
program Project1;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

procedure PrintHelloWorld(count: Integer);
var i: Integer;
begin
  for i := 1 to count do
    writeln('Hallo welt');
  end;

begin
  // Aufruf:
  PrintHelloWorld(42);
end.
```

In diesem Zusammenhang noch eine wichtige Ergänzung zum Thema Variablen: Bis zu diesem Kapitel haben wir Variablen auf oberster Ebene in unserem Programm definiert, direkt nach der Uses-Klausel. Damit waren sie überall in unserer Anwendung erreichbar. Man nennt sie „**globale Variablen**“. Das Gegenstück sehen wir in obigem Beispiel: Die Prozedur „PrintHelloWorld“ hat ihren eigenen var-Abschnitt vor dem „begin“ der Prozedur. Alles, was hier definiert wird, ist nur innerhalb dieser Prozedur bekannt. Deshalb nennt man diese Variablen „**lokale Variablen**“.

### 2.4.3 Funktionen

Funktionen werden genauso verwendet wie Prozeduren mit zwei Unterschieden: Sie werden mit dem Schlüsselwort „function“ begonnen (statt „procedure“) und haben einen Rückgabewert, dessen Typ in der ersten Zeile der Funktion (man spricht von der „Signatur“) steht:

```
function GetName(name: String): String;
begin
  result := 'Ich heie ' + name;
end;

// Aufruf:
writeln(GetName('Agi'));

// Ausgabe: Ich heie Agi
```

Die Funktion GetName erwartet also einen Namen als Aufrufparameter und liefert ihn auch wieder zurück, fügt dabei aber noch einen Text vorne an. result ist eine spezielle Variable, die automatisch in Funktionen zur Verfügung steht. Ihr wird der Rückgabewert der Funktion zugewiesen.

## 2.5 Benutzereingaben

Wie man Daten (z.B. Rechenergebnisse) eines Programms ausgibt, haben wir bereits gesehen, z.B. in den Hallo-Welt-Beispielen. Doch genauso wichtig ist es in der Regel, auch Daten eines Anwenders in das Programm zu bekommen. Wir werden uns das für die beiden Varianten Konsolenanwendung und VCL-Anwendung ansehen.

### 2.5.1 Eingaben in Konsolenanwendungen

Bei Konsolenanwendungen wird Text mit Write bzw. WriteLn ausgegeben – entsprechend erfolgt das Lesen von Benutzereingaben mit ReadLn. Die Eingabe ist dabei immer vom Typ String. Da ReadLn nur einen blinkenden Cursor anzeigt, sollte mit Write vorher ausgegeben werden, welche Eingabe denn von dem Anwender erwartet wird. Beispiel:

```
program Project1;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

var
  s: String;

procedure PrintHelloWorld(count: Integer);
var i: Integer;
begin
  for i := 1 to count do
    writeln('Hallo welt');
  end;
end;

begin
  write('Bitte eine Zahl eingeben: ');
  ReadLn(s);
  PrintHelloWorld(StrToInt(s));
end.
```

Der eingelesene Wert wird in der Variablen s gespeichert. Für den dann folgenden Aufruf von PrintHelloWorld wird dieser String in eine Integer-Zahl umgewandelt. Die Anwendung gibt also so oft „Hallo Welt“ aus, wie der Benutzer es wünscht, und beendet sich anschließend.

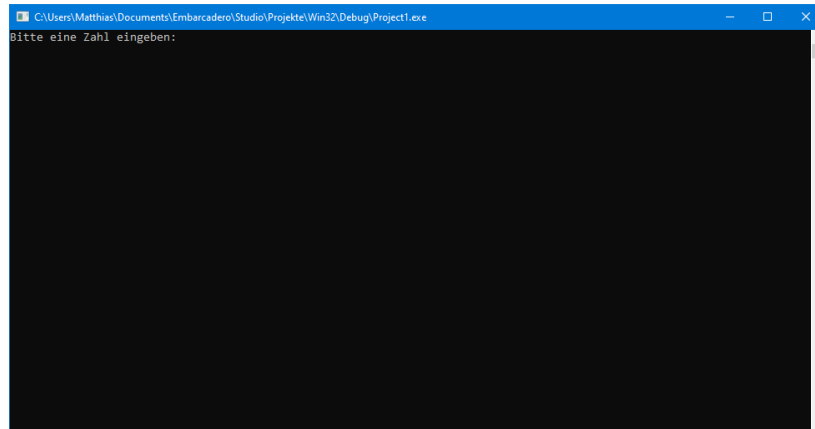


Abbildung 23: Konsolenanwendung wartet auf Eingabe

### 2.5.2 Eingaben in VCL-Anwendungen

In Anwendungen mit grafischer Benutzeroberfläche funktioniert das etwas anders. Hier erwartet der Anwender ein Eingabefeld, in das er seine Zahl eintragen kann, und einen Button, der beim Anklicken eine Aktion auslöst. Das kann der Benutzer so lange wiederholen, bis er keine Lust mehr hat und die Anwendung beendet.

#### 2.5.2.1 Delphis Entwicklungsumgebung (IDE)

Als erstes müssen wir uns etwas mit der Entwicklungsumgebung von Delphi beschäftigen.

Wir starten ein neues VCL-Projekt, indem wir den Menüpunkt Datei – Neu – Windows-VCL-Anwendung auswählen. Das Vorgehen entspricht unserem „Hallo Welt“-Beispiel.

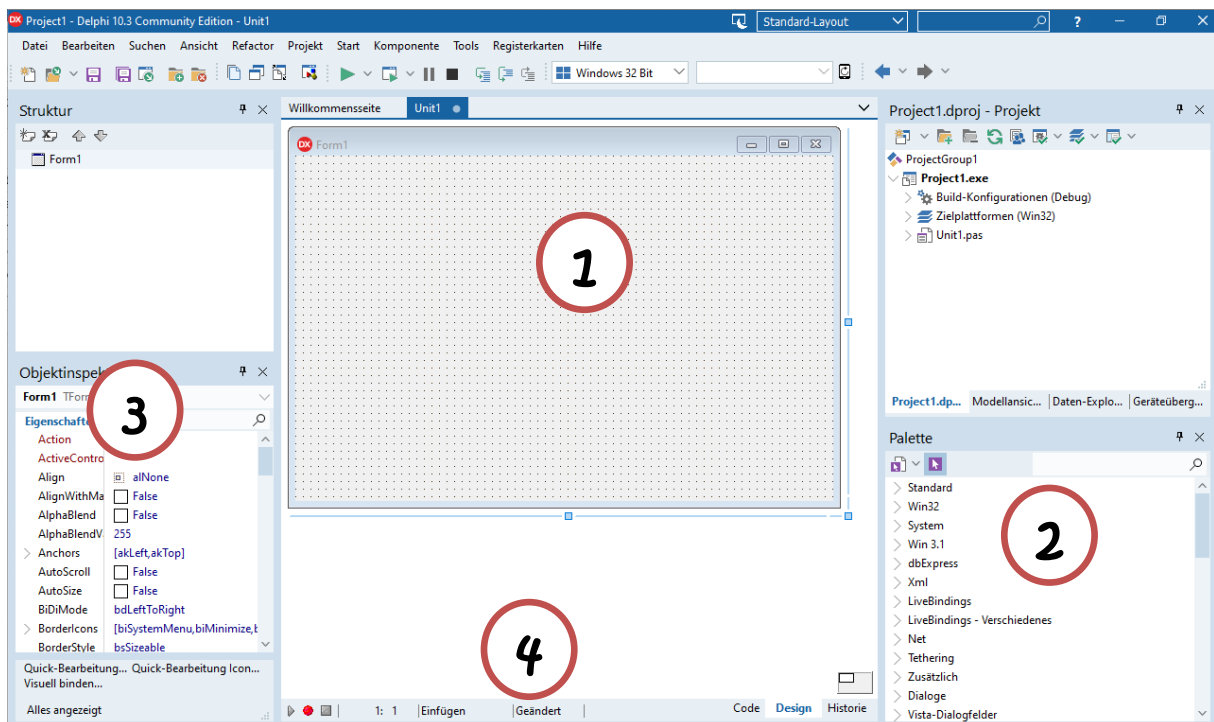


Abbildung 24: Die Delphi-IDE (Integrated Development Environment)

Im mittleren Fenster sehen wir nun den Formulardesigner (1). Hier können wir die Fenster unserer Anwendung gestalten. Der Werkzeugkasten („Tool-Palette“, 2) befindet sich standardmäßig am rechten unteren Rand von Delphi. Dort können wir Komponenten auswählen und auf dem neuen Fenster platzieren, wenn wir uns im Formulardesigner befinden. In der Code-Ansicht ist das natürlich nicht möglich.

**Komponenten** sind vorgefertigte Bausteine, die man in eigenen Anwendungen verwenden kann. Es gibt sichtbare Komponenten, die Steuerelemente (z.B. Buttons, Checkboxes, Menüs) darstellen. Und es gibt nicht- sichtbare Komponenten, die man zwar auch auf einem Formular platzieren kann, die aber zur Laufzeit nicht sichtbar sind.

Links befindet sich der Objektinspektor (3). Hier können wir die Eigenschaften der aktuell ausgewählten Komponente (auch das Fenster selbst ist eine Komponente) einstellen. Auf der hinteren Registerseite „Ereignisse“ lassen sich Ereignisse konfigurieren, so dass man auf Klick, Doppelklick oder anderes reagieren kann.

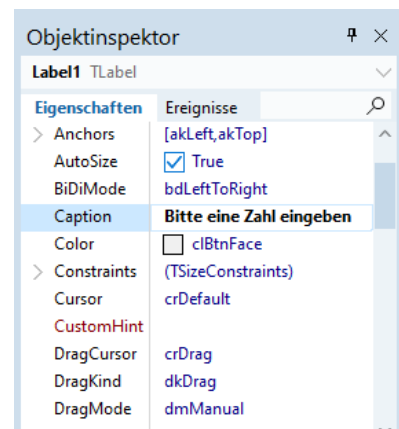
Natürlich wird aus dem reinen Zusammenklicken von Komponenten und ein paar Eigenschaftsänderungen noch keine fertige Anwendung. Irgendwo muss ja auch eigener Code platziert werden. In Delphi gehört zu jedem Fenster auch eine Unit, also eine Code-Datei, die mit der Endung PAS gespeichert wird. Diesen Code können wir sehen, wenn wir am unteren Ende des Formulardesigners (4) auf die Registerseite „Code“ klicken (oder die Taste F12 drücken). Was wir hier zu Gesicht bekommen, ist der bereits von Delphi generierte Code. Wenn wir Komponenten auf dem Formular platzieren, werden diese als Variablen im Code automatisch hinzugefügt. Und wenn wir neue Ereignisse anlegen, entstehen im Code neue Methoden.

Alle weiteren Daten, die sich aus dem Zusammenklicken eines Fensters ergeben (z.B. Größe des Fensters, Position der Komponenten), werden übrigens in einer Datei mit der Endung DFM gespeichert.

### 2.5.2.2 Fenster gestalten

Zur Erinnerung: In diesem Kapitel geht es einfach darum, Benutzereingaben entgegenzunehmen. Wir benötigen also ein Texteingabefeld und einen Button.

Wir klicken in der Tool-Palette auf den Eintrag „TEdit“ – dabei handelt es sich um Texteingabefelder. Anschließend klicken wir auf das Fenster im Formulardesigner. An dieser Stelle erscheint nun die Komponente. Damit der Anwender weiß, was er in die Eingabefelder einzutragen hat, sollte noch ein kurzer Text vor das Eingabefeld. Dafür verwendet man die Komponente TLabel. Wir setzen deshalb vor das Eingabefeld ein TLabel. Nun muss noch der Text für die Labels eingetragen werden. Dazu wählen wir das Label aus (einfacher Klick) und suchen dann im Objektinspektor die



Eigenschaft „Caption“ (engl. „Beschriftung“). Rechts daneben können wir nun den Text eintragen (z.B. „Bitte eine Zahl eingeben“).

Auf die gleiche Weise können wir auch die Beschriftung des Buttons verändern („Los geht's!“).

Nun können wir die Komponenten noch so zurecht rücken und in ihrer Größe ändern wie es uns gefällt. Dabei unterstützt uns Delphi, indem es farbige Hilfslinien einblendet, die uns zeigen, ob zwei Komponenten auf gleicher Höhe positioniert sind.

Anschließend wäre ein guter Zeitpunkt, das Projekt zu speichern (Menü Datei – Alles speichern). Wir werden nun zuerst nach einem Namen für die Unit gefragt. Dieser Name wird auch für die zugehörige dfm-Datei verwendet. Anschließend muss der Name des Projekts (Endung dproj) eingegeben werden. Dieser Name wird nach dem Kompilieren auch für die ausführbare Datei (exe) verwendet.

Unser Fenster sollte nun so aussehen:

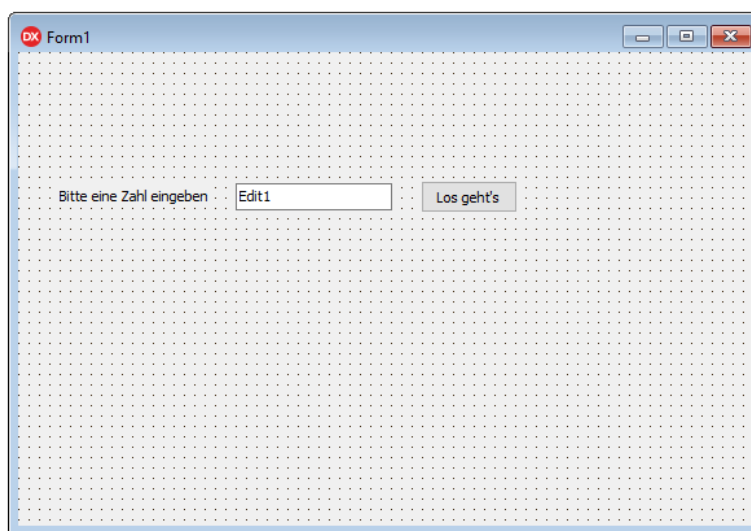



Abbildung 25: Fenster der Beispielanwendung

Um schon einmal das erste Erfolgserlebnis zu haben, klicken wir auf den Button mit dem grünen Pfeil in der Werkzeugleiste  oder drücken die Taste F9. Der Compiler übersetzt das Programm nun. Anschließend wird es gestartet.

Zugegeben, die Anwendung tut momentan nicht viel. Nur die Systembuttons in der Fensterkopfzeile funktionieren, so dass wir das Fenster auch wieder schließen können.

### 2.5.2.3 Auf Ereignisse reagieren

Nun wollen wir aber dem Los geht's-Button noch sagen, was er zu tun hat. Dazu gehen wir zurück in die Entwicklungsumgebung, wählen den Button aus und wechseln im Objektinspektor nun auf die Seite „Ereignisse“. Da der Button auf einen Mausklick reagieren soll, ist das „OnClick“-Ereignis für uns das richtige. Ein Doppelklick auf das freie Feld in der rechten Spalte neben „OnClick“ erzeugt eine neue Methode im Quellcode:



```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  
end;
```

Alternativ kann bei einem Button auch ein Doppelklick im Formular Designer auf den Button durchgeführt werden, um das „OnClick“-Ereignis im Code zu erzeugen oder zu einem bereits vorhandenen zu springen.

Zwischen ‚begin‘ und ‚end‘ kommt unser Code. Und dabei geht es uns ja darum, die Benutzereingabe auszulesen. Diese sollte im Eingabefeld zu finden sein. Alle Eingabefelder vom Typ TEdit haben die Eigenschaft „Text“. Sie enthält den eingegebenen Text als String. Der Name des Eingabefelds wird im Objektinspektor festgelegt (Eigenschaft „Name“) – standardmäßig lautet er „Edit“, gefolgt von einer Zahl. In unserem Fall ist es „Edit1“. Die Benutzereingabe erhalten wir also wie folgt:

```
procedure TForm1.Button1Click(Sender: TObject);  
var eingabe: String;  
begin  
    eingabe := Edit1.Text;  
end;
```

Nun können wir mit dem weitermachen, was wir in den vorigen Kapiteln gelernt haben: Wir versuchen die Eingabe mit StrToIntDef in eine Zahl umzuwandeln und prüfen dann ihre Größe. Das Ergebnis geben wir über ShowMessage aus:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    eingabe: String;  
    zahl: Integer;  
begin  
    eingabe := Edit1.Text;  
    zahl := StrToIntDef(eingabe, 0);  
    if zahl > 10 then  
        ShowMessage('Die eingegebene Zahl ist größer als 10.')    else  
        ShowMessage('Die eingegebene Zahl ist kleiner oder gleich 10.');end;
```

### 2.5.3 Wichtige Komponenten

Die drei wichtigsten Komponenten und ihre Verwendung haben wir gerade kennengelernt: TEdit, TLabel und TButton. Zwei weitere, häufig benötigte Komponenten sollen hier noch genannt werden: TMemo und TListBox.

#### 2.5.3.1 TMemo

Bei TMemo handelt es sich quasi um eine Notepad-Komponente. Sie kann beliebigen, unformatierten Text darstellen. Intern wird dieser Text zeilenweise in einer Liste von Strings abgelegt.

Wir setzen testweise ein TMemo auf das Fenster einer neuen Windows-VCL-Anwendung:

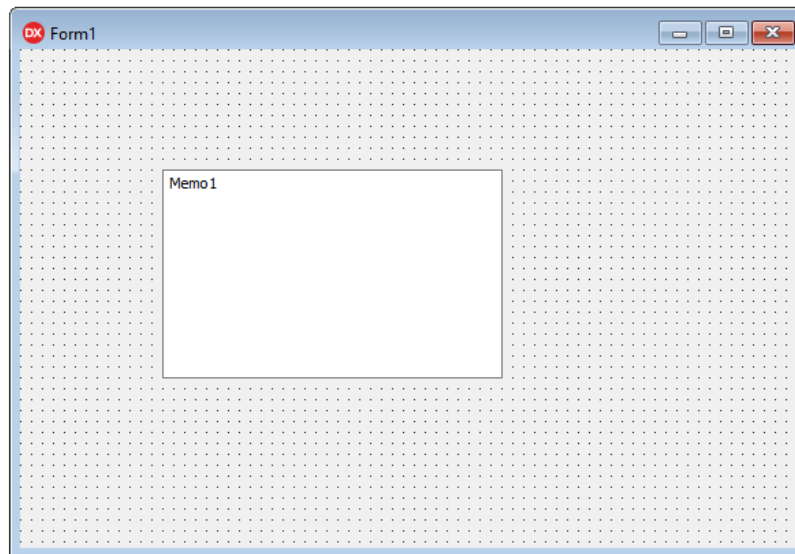


Abbildung 26: Fenster mit TMemo

Zudem können wir ein Eingabefeld und einen Button verwenden, um das Memo mit Inhalt zu füllen. Ein Klick auf den Button soll den eingegebenen Text ans Ende des Memos einfügen:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Memo1.Lines.Add(Edit1.Text);  
end;
```

Ein weiterer Button soll dafür da sein, den Inhalt wieder zu leeren:

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    Memo1.Clear;  
end;
```

Und jetzt kommt das Beste: Wir wollen den Inhalt des Memos in eine Textdatei speichern. Das soll ein dritter Button bewerkstelligen:

```
procedure TForm1.Button3Click(Sender: TObject);  
begin  
    Memo1.Lines.SaveToFile('C:\Users\Matthias\Desktop\test.txt');  
end;
```

Hier steht der Dateiname fest im Code. Aber dir fällt es nun sicher nicht mehr schwer, ein Eingabefeld auf dem Fenster zu platzieren, in das der Benutzer einen eigenen Dateinamen eingeben kann.

### 2.5.3.2 TListBox

Die Funktionsweise von TListBox ähnelt der eines Memos. Auch eine ListBox enthält eine Liste von Strings. Allerdings kann der Benutzer hier nicht frei Text eintragen. Vielmehr stellt eine ListBox eine Reihe von Einträgen zur Verfügung, aus denen der Benutzer einen oder mehrere auswählen kann. Im

Objektinspektor ist die Eigenschaft `MultiSelect` zu finden. Bei `false` kann immer nur ein Wert ausgewählt werden, bei `true` mehrere.

Das Befüllen einer `ListBox` funktioniert wie bei `TMemo`. Allerdings wird hier der Inhalt nicht in der Eigenschaft `Lines`, sondern in `Items` gehalten:

```
ListBox1.Items.Add(Edit1.Text);
```

Sinnvoll ist es aber, die Einträge schon zur Entwicklungszeit hinzuzufügen. Das ist über den Objektinspektor möglich und dort in der Eigenschaft „Items“.

Zur Laufzeit möchte man vielmehr wissen, wie viele und welche Einträge der Benutzer markiert hat. Bei Klick auf einen Button könnte also Folgendes passieren:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ListBox1.SelectCount = 0 then
    ShowMessage('Bitte mind. einen Eintrag auswählen!')
  else
    begin
      end;
    end;
```

**Bitte beachten:** Das in obigem Beispiel verwendete `SelectCount` funktioniert nur, wenn `MultiSelect` auf `true` steht. Ansonsten gibt es immer -1 zurück. Um herauszufinden, welche Werte im `MultiSelect`-Fall ausgewählt wurden, müssen wir eine Schleife über alle Einträge der `ListBox` machen und für jeden Wert prüfen, ob er ausgewählt wurde. Diese Prüfung erfolgt mit der Funktion `Selected`.

```
procedure TForm1.Button1Click(Sender: TObject);
var i: Integer;
begin
  for i := 0 to ListBox1.Items.Count - 1 do
    begin
      if ListBox1.Selected[i] then
        ShowMessage(ListBox1.Items[i] + ' ist ausgewählt');
      end;
    end;
```

Das Beispiel im Einzelnen:

Als erstes schreiben wir eine `for`-Schleife, wie wir sie in einem vorigen Abschnitt kennengelernt haben, über alle Einträge der `ListBox`. Diese befinden sich in der Eigenschaft „Items“. „Items“ wiederum hat eine Eigenschaft „Count“, die uns sagt, wie viele Einträge es überhaupt in der `ListBox` gibt. Da wir die Schleife bei 0 beginnen, müssen wir von `Count` 1 abziehen, damit wir nicht über das Ziel hinausschießen.

Für jeden Eintrag der `ListBox` wird der Code zwischen `Begin` und `End` ausgeführt. Hier prüfen wir mit `if`, ob der Wert an Position `i` ausgewählt ist. Falls ja, wird direkt ein `Popup`-Fenster angezeigt, das den

Text des Eintrags enthält. In eckigen Klammern hinter `Items` und `Selected` steht die Positionsnummer des Eintrags, wobei die Nummerierung immer mit 0 beginnt. D.h. Eintrag 1 hat den Index 0, Eintrag 2 den Index 1 usw.

Ist `MultiSelect` false, darf der Anwender also nur einen einzigen Eintrag auswählen, gestaltet sich die Sache wesentlich einfacher. Hier gibt uns die Eigenschaft `ItemIndex` die Position des Eintrags zurück, der ausgewählt ist bzw. -1, wenn kein Eintrag ausgewählt wurde:

```
procedure TForm1.Button1Click(Sender: TObject);
var i: Integer;
begin
  if ListBox1.ItemIndex = -1 then
    ShowMessage('Bitte einen Eintrag auswählen!')
  else
    ShowMessage(ListBox1.Items[ListBox1.ItemIndex] + ' ist ausgewählt');
end;
```

## 2.6 Hilfe!

Alles, was man von Haus aus mit Delphi machen kann (also alle Funktionen, Klassen, Komponenten usw.) sind natürlich beschrieben. Und zwar in der Hilfe, die man aus der Entwicklungsumgebung über das gleichnamige Menü erreichen kann. Oder man platziert den Cursor auf einer Komponente oder einem Befehl und drückt die Taste F1.

Das funktioniert auch, wenn beim Kompilieren oder zur Laufzeit des Programms Fehler auftreten. Die Delphi-Hilfe kann erläutern, was diese Fehlermeldungen bedeuten. Verwendet man z.B. in seinem Code etwas, das der Delphi-Compiler nicht versteht (oder man vergisst z.B. einfach ein Semikolon am Zeilenende), wird man direkt beim Compile-Vorgang darauf hingewiesen:

[dcc32 Fehler] Unit1.pas(30): E2066 Operator oder Semikolon fehlt

Ein Anklicken der Fehlermeldung und Drücken von F1 führt zu einer ausführlichen Erklärung des Problems, oftmals auch mit Beispielen.

Auch wer Delphi noch nicht installiert hat, kann diese Hilfe (teilweise mit Beispielen) anschauen. Dazu hat Embarcadero ein Wiki eingerichtet: <http://docwiki.embarcadero.com/RADStudio/de/Hauptseite>

## 2.7 Delphi-Komponentenbibliotheken

Wer mit Delphi eine GUI-Anwendung erstellt, muss sich gleich zu Beginn für eine Komponenten-Bibliothek entscheiden, wie wir in unserem „Hallo Welt“-Kapitel gesehen haben.



Zur Wahl stehen die **VCL** („**Visual Component Library**“) und FireMonkey („FMX“). Die VCL gibt es bereits seit Delphi 1. Alle VCL-Komponenten basieren auf den Windows-eigenen Bibliotheken und werden durch die VCL für den Delphi-Entwickler nur etwas einfacher benutzbar gemacht. Seit Delphi 1 ist die VCL ständig erweitert worden.



**FireMonkey** hingegen ist neu mit Delphi XE2 hinzugekommen. Zur Erstellung von Formularen wird hier nicht mehr auf die Windows-Funktionalitäten zurückgegriffen. Vielmehr handelt es sich um eine plattformunabhängige, vektororientierte Bibliothek, die alle Komponenten selbst zeichnet. Über verschiedene Stile kann man einstellen, dass sie z.B. aussehen wie Standard-Windows-Steuerelemente. Vorteil der Vektororientierung ist, dass man Fenster beliebig zoomen kann, ohne dass sie pixelig werden. FireMonkey-Anwendungen sind – wie auch VCL-Anwendungen – nativ, laufen also ohne Bibliotheken auf der Plattform, für die sie kompiliert wurden. Allerdings kann man in Delphi per Mausklick wählen, für welche Plattform kompiliert werden soll. FireMonkey-Anwendungen können für Win32, Win64, iOS, Android und macOS erstellt werden. Linux ist in der Enterprise Edition verfügbar.

In einer Anwendung kann man nicht VCL und FireMonkey kombinieren. Doch wie entscheidet man sich zwischen den Frameworks? Will man eine ganz normale Windows-Anwendung erstellen, bleibt man am besten bei der altbekannten und ausgereiften VCL. Zu FireMonkey sollte man greifen, wenn man eine Anwendung auch für andere Plattformen als Windows erstellen will oder wenn die Anwendung mit grafischem „Schnickschnack“ wie Animationen oder Effekten aufgepeppt werden soll.

Zu guter Letzt gib es übrigens auch noch die **RTL**, die Runtime Library. Unter diesem Begriff werden Units mit Standardfunktionalität ohne Komponenten (z.B. Datumsberechnungen) zusammengefasst. Die RTL kann sowohl in VCL- als auch in FireMonkey-Anwendungen eingesetzt werden.

#### Übungsaufgaben

1. Schreibe ein Programm, das die Zahlen von 12 bis 144 in einer TListBox ausgibt.
2. Schreibe ein Programm, das eine Integer-Zahl vom User erfragt, diese durch 2 teilt und das Ergebnis ausgibt. Beispiel:  $7 \text{ div } 2 = 3$  (Rest 1) ==> 3 soll ausgegeben werden. Benutze dazu ein Edit-Feld, einen Button und ein Label.
3. Schreibe ein Programm, das den Rest bei der Division durch 2 ausgibt.  $7 \text{ div } 2 = 3$  (Rest 1) ==> 2 soll ausgegeben werden. Verwende den Operator „mod“ (statt div) um den Rest zu berechnen.
4. Schreibe ein Programm, das prüft, ob eine Zahl durch 2 teilbar ist.
5. Schreibe ein Programm, das alle durch 2 teilbaren Zahlen zwischen 12 und 144 ausgibt.
6. Schreibe ein Programm, das die Zahlen von 12 bis 144 ausgibt, wobei alle durch 3 teilbaren Zahlen durch „Fizz“, alle durch 5 teilbaren Zahlen durch „Buzz“ und alle sowohl durch 3 als auch durch 5 teilbaren Zahlen durch „FizzBuzz“ ersetzt werden.

***In diesem Kapitel werden die Elemente von Object Pascal detaillierter betrachtet. Zudem geht es um objektorientierte Programmierung und das Arbeiten mit Dateien.***

### 3 Object Pascal im Detail

#### 3.0 Variablen und Konstanten

##### 3.0.1 Was sind Variablen?

Variablen sind einfach dafür da, irgendwelche Daten (Eingaben, Berechnungsergebnisse usw.) im Arbeitsspeicher abzulegen. Über den Variablennamen kann man direkt auf den Wert zugreifen.

In der Mathematik heißen Variablen meist  $x$  oder  $y$ . Ein Programmierer sollte solche Namen nicht verwenden, da sie die Lesbarkeit des Quellcodes erschweren. Diese Erfahrung macht jeder, der nach ein paar Monaten an einem Programm weiterarbeiten möchte. Variablen sollten deshalb selbsterklärende Namen haben. Die Groß- und Kleinschreibung ist nicht von Bedeutung, jedoch dürfen nur Buchstaben, Zahlen und der Unterstrich verwendet werden. Der Variablenname muss mit einem Buchstaben oder einem Unterstrich beginnen.

Seit Delphi 2006 können auch Umlaute benutzt werden, wenn der Quellcode im UTF8-Zeichensatz gespeichert wird. Man sollte aber trotzdem darauf verzichten, vor allem wenn man an Open Source Projekten arbeitet, die auch andere Personen aus anderen Nationen lesen. Zum Beispiel lassen sich chinesische Zeichen ohne entsprechende Kenntnisse sehr schlecht lesen.

##### 3.0.2 Datentypen

Bevor eine Variable verwendet werden kann, sollte man sich darüber im Klaren sein, welche Werte sie aufnehmen soll. Variablen sind Platzhalter oder „Container“ für einen Wert, der Platz im Arbeitsspeicher belegt; der Datentyp beschreibt, wie der Inhalt des Containers auszusehen hat und damit auch, wie groß der Container sein muss.

Folgendes sind die grundlegenden Datentypen in Delphi:

Typ	Wertebereich	Beispiel	
		Deklaration	Zuweisung
Integer (ganze Zahlen)	-2147483648 bis 2147483647	var Zahl: Integer;	Zahl:=14;
Real (Gleitkommazahlen)	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	var Zahl: Real;	Zahl:=3.4;

String (Zeichenketten)	ca. $2^{31}$ Zeichen	var Text: string;	Text:='Hallo Welt!';
Char (Zeichen)	1 Zeichen	var Zeichen: Char;	Zeichen:='a';
Boolean (Wahrheitswert)	true, false	var Richtig: Boolean;	Richtig:=true;

Dies sind die Standardtypen, die generell verwendet werden können. Vor allem für Zahlen gibt es jedoch noch weitere Typen. Wenn z. B. sicher ist, dass nur ganze Zahlen von 1 bis 50 gespeichert werden sollen, so kann statt Integer auch der Typ Byte verwendet werden, der nur die Zahlen von 0 bis 255 aufnehmen kann. Das Gleiche gilt für reelle Zahlen. Die weiteren Typen sind unter „Reelle Typen“ bzw. „Integer-Typen“ in der Hilfe aufgeführt.

### 3.0.3 Deklaration

Bevor man eine Variable verwenden kann, muss man sie dem Compiler bekannt machen, damit er beim Kompilieren entsprechende Typprüfungen durchführen kann. Das Bekanntmachen nennt man Deklaration. Vor der Deklaration einer Variablen wird das reservierte Wort **var** geschrieben. Als erstes kommt der Name der Variablen, hinter einem Doppelpunkt folgt der Typ. Mehrere Variablennamen vom gleichen Typ können in einer Zeile, durch Kommata getrennt, stehen.

Beispiel:

```
var Zahl1, Zahl2, Zahl3: Integer;
    Ergebnis: Real;
    Text, Eingabe: string;
```

Variablen können an mehreren fest definierten Orten im Code deklariert werden. Je nach Ort haben sie unterschiedliche Gültigkeitsbereiche: global oder lokal.

### 3.0.4 Globale Variablen

Bei manchen Programmierern sind sie verpönt, trotzdem sind sie möglich: globale Variablen. Ihr Wert ist in der gesamten Unit verfügbar und in allen Units, die diese einbinden. Und genau dieser freizügige Gültigkeitsbereich ist es, der es einem schwer macht, den aktuellen Wert abzuschätzen, da von überall der Wert geändert werden kann. Die Deklaration erfolgt am Anfang der Unit:



```
unit Unit1;

interface

uses
  System.SysUtils;

var
  Einezahl: Integer; // Diese Variable gilt in der ganzen Unit und
                    // in allen Units, die diese Unit einbinden

implementation

var
  Eine_andere_zahl: Real; // Diese Variable gilt nur in dieser Unit
```

Globale Variablen können bei ihrer Deklaration mit einem Startwert belegt werden:

```
var Einezahl: Integer = 42;
```

Diese Art der Zuweisung verwendet ein einfaches Gleichheitszeichen, nicht `:=`. Bei lokalen Variablen ist diese Initialisierung nicht möglich.

### 3.0.5 Lokale Variablen

Das Gegenstück zu globalen Variablen sind die lokalen. Hierbei wird eine Variable zu Beginn einer Prozedur, Funktion oder Methode deklariert. Sie kann somit nur innerhalb dieses Abschnitts verwendet werden. Wird die Prozedur/Funktion verlassen, dann wird der Speicher für die Variablen wieder freigegeben, d. h. auf die Werte kann nicht mehr zugegriffen werden. So könnte eine lokale Variablendeklaration aussehen:

```
procedure IchMacheIrgendwas;
var Text: string;
begin
  ... //Irgendwas Sinnvolles
end;
```

### 3.0.6 Zuweisungen

Die Zuweisung von Werten an eine Variable erfolgt in Pascal durch die Symbolfolge `:=` („ergibt sich aus“). Im Gegensatz zur Mathematik ist deshalb auch Folgendes möglich:

```
x := x + 1;
```

Das Laufzeitsystem berechnet hier die Summe von x und 1 und legt das Ergebnis dann wieder in x ab. x ergibt sich aus dem bisherigen x plus 1. Kurz: x wird um 1 erhöht.

### 3.0.7 Initialisierung

Wird globalen Variablen kein Startwert zugewiesen, so werden sie, je nach Typ, automatisch mit 0, nil oder Leerstring initialisiert. Bevor jedoch auf eine lokale Variable lesend zugegriffen werden kann,

muss sie explizit belegt werden, es muss ihr also ein Anfangswert zugewiesen werden. Strings enthalten zwar einen leeren String; alle anderen Variablentypen enthalten jedoch irgendwelche zufälligen Werte. Vergisst man den Anfangswert zu setzen, wird der Compiler beim Kompilieren eine Warnung ausgeben.

### 3.0.8 Beispiel

Nun wird eine kleine Beispielanwendung entwickelt, in der die weiter vorne im Kapitel genannten Wertebereiche überprüft werden. Um es möglichst einfach zu halten, wird diese Anwendung noch ohne Benutzeroberfläche programmiert. Über das Menü Datei -> Neu -> Weitere -> Konsolenanwendung wird das Grundgerüst einer Konsolenanwendung erstellt. Als erstes sollte man eine Anwendung immer speichern. Über Datei -> Projekt speichern werden alle Dateien, die zum Projekt gehören in den gewünschten Ordner gespeichert. Der Quelltext dieses Programmes sieht folgendermaßen aus.

```
program Ganzzahlen;  
  
{$APPTYPE CONSOLE}  
  
uses  
    System.SysUtils;  
  
var  
    i: Integer;  
  
begin  
    i := 2147483647;  
    writeln(i);  
    i := i + 1;  
    writeln(i);  
    ReadLn;  
end.
```

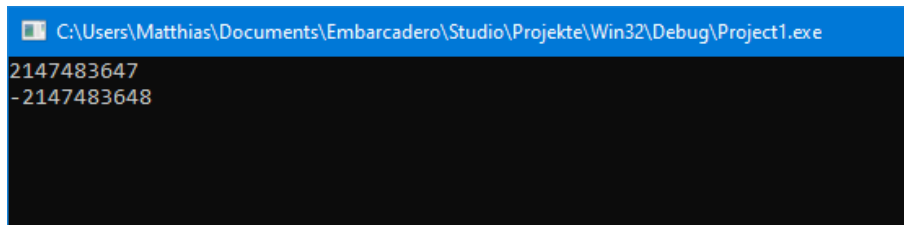
In der ersten Zeile wird dem Compiler mitgeteilt, dass die Anwendung den Namen *Ganzzahlen* haben soll. Die nächste Zeile enthält eine weitere Information für den Compiler. `APPTYPE CONSOLE` heißt, dass die Anwendung eine Konsolenanwendung ist. Durch das Schlüsselwort `uses` können Delphi-Units in die eigene Anwendung eingebunden werden, damit ihre Funktionalität verfügbar ist. In diesem Fall wird die Unit `SysUtils` eingebunden, die einige notwendige Funktionen zur Verfügung stellt.

Da in dieser Beispielanwendung der Wertebereich einer Integer-Variablen überprüft werden soll, muss eine Variable dieses Typs deklariert werden. Wie das gemacht wird, wurde weiter vorne bereits beschrieben.

Der eigentliche Programmablauf wird zwischen `begin` und `end.` geschrieben. Zuerst wird der Integervariablen ein Wert zugewiesen und über `writeln()` auf der Konsole ausgegeben. Dann wird der Wert um 1 erhöht und wiederum ausgegeben. Durch `ReadLn()` wird verhindert, dass sich das

Konsolenfenster sofort wieder schließt. `ReadLn()` wartet darauf, dass der Benutzer eine Eingabe durch Drücken der Enter-Taste bestätigt.

Die Anwendung kann jetzt über das Menü Start -> Ausführen oder den Shortcut F9 kompiliert und gestartet werden. Es erscheint das folgende Fenster:



Das ist jetzt aber merkwürdig. Die Variable wird im Programmcode um 1 erhöht, hat danach aber auf einmal einen negativen Wert. Wenn man sich den Wertebereich von Integer ansieht, fällt einem auf, dass der zuerst zugewiesene Wert die obere Grenze ist. Wenn man jetzt den Wert erhöht, kann die Variable den Wert nicht mehr speichern. Anstatt aber einen Fehler auszugeben, wird einfach von vorne begonnen zu zählen. Das hängt damit zusammen, wie Zahlen intern repräsentiert werden.

Ist dieses Verhalten nicht erwünscht, dann sollte man in den Compiler-Optionen die Überlaufprüfung aktivieren.

### 3.0.9 Typumwandlung

Im Gegensatz zu einigen anderen Sprachen ist Delphi bei Typen sehr streng. Es ist also nicht möglich, einer Integer-Variablen eine Gleitkommazahl-Variable zuzuweisen. Dafür steht eine große Auswahl an Konvertierungsfunktionen zur Verfügung:

von	nach	Funktion	Beispiel
Integer	Real	kein Problem, einfache Zuweisung	<b>var</b> intzahl: integer; realzahl: real; <b>begin</b> realzahl := intzahl;
Real	Integer	Möglichkeiten: - Nachkommastellen abschneiden (trunc) - kaufm. Runden (round) - aufrunden (ceil, Unit Math) - abrunden (floor, Unit Math)	<b>var</b> realzahl: real; intzahl: integer; <b>begin</b> intzahl := trunc(realzahl); intzahl := round(realzahl);
Integer	String	IntToStr	<b>var</b> textzahl: <b>string</b> ; intzahl: integer; <b>begin</b> textzahl := IntToStr(intzahl);

Real	String	FloatToStr FloatToStrF	<b>var</b> textzahl: <b>string</b> ; realzahl: real; <b>begin</b> textzahl := FloatToStr(realzahl);
String	Integer	StrToInt StrToIntDef	<b>var</b> textzahl: <b>string</b> ; intzahl: Integer; <b>begin</b> intzahl := StrToInt(textzahl);
String	Real	StrToFloat	<b>var</b> textzahl: <b>string</b> ; realzahl: real; <b>begin</b> realzahl := StrToFloat(textzahl);
String	Char	Zugriff über Index (1 ist erstes Zeichen)	<b>var</b> text: <b>string</b> ; zeichen: char; <b>begin</b> zeichen := text[1];
Char	String	kein Problem, einfache Zuweisung	<b>var</b> zeichen: char; text: <b>string</b> ; <b>begin</b> text := zeichen;

### 3.0.10 Konstanten

Konstanten sind letztendlich Variablen, die innerhalb des Programms jedoch nur ausgelesen, nicht überschrieben werden dürfen. Deklariert werden sie an den gleichen Stellen wie Variablen, allerdings mit dem Schlüsselwort **const** (anstelle von **var**), einem Gleichheitszeichen und ohne Angabe eines Datentyps:

```
const Version = '1.23';
```

Der Datentyp ergibt sich aus dem zugewiesenen Wert. Sind Anführungszeichen davor und dahinter, handelt es sich um einen String, sonst um eine Zahl. Enthält die Zahl kein Dezimaltrennzeichen, wird der Datentyp Integer verwendet, ansonsten Real.

### Übungsaufgaben

1. Welche dieser Variablendeklarationen sind in Delphi erlaubt (mehr als eine Antwort möglich):
  - `var _var1: Integer;`

- var \$i: String;
- var 2teZahl: Integer;
- var übung: Double;

2. An welcher Stelle in folgendem Code ist die Zeile „var x: Integer;“ *nicht* erlaubt?

```
unit 1;

interface

uses SysUtils;

(1)

implementation

(2)

procedure DoSomething;
(3)
begin
  (4)
  x := 1;
  WriteLn(IntToStr(x));
end;

end.
```

3. Was passiert bei folgender Zuweisung:

```
var x: Integer;
begin
  x := 100.123;
end;
```

- Beim Kompilieren tritt ein Fehler auf.
- Zur Laufzeit tritt ein Fehler auf.
- Es tritt gar kein Fehler auf.

### 3.1 Datentypen

Jeder Variablen liegt in Object Pascal ein Typ zugrunde. Es gibt verschiedene Arten von Datentypen. Die meisten einfachen Datentypen sind bereits im Kapitel über Variablen beschrieben worden. Doch es gibt weitere.

String ist der Datentyp, der Texte aufnimmt. Allerdings gibt es nicht nur einen String-Typ, sondern verschiedene.

### 3.1.1 Strings

#### 3.1.1.1 Unterschiedliche String-Typen

In Delphi gibt es verschiedene Möglichkeiten, Strings zu deklarieren:

- String
- ShortString
- AnsiString
- UnicodeString
- UTF8String
- WideString

#### 3.1.1.2 String

Der Datentyp „String“ ist kein eigener Datentyp, sondern nur ein Alias, dessen wirklicher Typ sich abhängig von der Delphi-Version unterscheidet. Sofern man keine besonderen Gründe hat, wird empfohlen, immer „String“ als Datentyp für Texte zu verwenden. Der Alias steht für:

- ShortString (in Delphi 1)
- AnsiString (Delphi 2 bis 2007)
- UnicodeString (seit Delphi 2009)

**var text: String;**

Die älteren String-Typen (ShortString, AnsiString) stehen nach wie vor auch in neuen Delphi-Versionen zur Verfügung. Nur reicht für ihre Verwendung nicht mehr die Angabe „String“, sondern es muss konkret z.B. „ShortString“ angegeben werden.

#### 3.1.1.3 ShortString

Verwendet man unter Delphi 1 den Typ „String“, meinte man den Datentyp „ShortString“. Ein ShortString besteht maximal aus 255 Zeichen, für die statisch Arbeitsspeicher reserviert wird. Das bedeutet, dass auch wenn nur 1 Zeichen in ShortString enthalten ist, trotzdem Speicher für 255 Zeichen belegt wird. Im ersten Byte (Index 0) des Strings befindet sich die Längenangabe. Somit liegt das erste Zeichen an Position 1. Die Länge eines ShortStrings kann man auch selbst angeben:

```
var text: String[10];
```

Wird keine Länge vorgegeben, wird 255 verwendet.

#### 3.1.1.4 AnsiString

Mit der Umstellung auf 32 Bit mit Delphi 2 fiel die 255-Zeichen-Grenze. Strings können nun bis zu 2 GB groß werden. Im Gegensatz zum ShortString wird deshalb nun nicht mehr die komplette Größe im Speicher reserviert, sondern nur so viel, wie momentan benötigt wird. Ist der String leer, so verwendet er keinen Speicher. Für jedes Zeichen des Strings steht 1 Byte zur Verfügung, es kann also nur der

ANSI-Zeichensatz verwendet werden. Der `AnsiString` ist ebenfalls 1-basiert, um zum `ShortString` kompatibel zu bleiben. Will man also eine Schleife über alle Zeichen des Strings implementieren, ist zu beachten, dass das erste Zeichen sich nicht an Position 0, sondern 1 befindet.

```
var text: AnsiString;
```

#### 3.1.1.5 *UnicodeString*

UnicodeStrings wurden mit Delphi 2009 eingeführt. Sie enthalten Unicode-Zeichen (UTF-16) und können somit wesentlich mehr Schriftzeichen abbilden als ANSI oder gar ASCII, z.B. arabische und chinesische Zeichen. Ansonsten gleicht ihr Verhalten dem Typ `AnsiString`. Ihre Länge ist nur vom zur Verfügung stehenden Arbeitsspeicher begrenzt. Wie bei allen String-Typen in Delphi beginnt die Zählung der Zeichen hier mit 1 und nicht wie sonst in der Programmierung üblich mit 0.

```
var text: UnicodeString;
```

#### 3.1.1.6 *UTF8String*

Der `UTF8String` ist ebenfalls neu in Delphi 2009. Wie der Name sagt, sind seine Zeichen UTF-8-kodiert, während der `UnicodeString` mit UTF-16 arbeitet. Die String-Typen sind zuweisungskompatibel.

#### **Exkurs: Zeichensätze – was ist der Unterschied zwischen ASCII, ANSI, UTF-8 und UTF-16?**

Die Entwicklung der Computer kommt aus den USA. Und da Speicher auf frühen Rechnern knapp war, enthielt der erste Zeichensatz nur die in den USA üblichen Zeichen und Ziffern, also auch keine Umlaute. Dafür benötigte man nur 7 Bit. Dieser Standard heißt **ASCII** (American Standard Code for Information Interchange) und kann 128 Zeichen abbilden.

Als sich Computer auch in Europa verbreiteten, wuchs der Bedarf nach weiteren Zeichen. So entstand der **ANSI**-Zeichensatz (benannt nach dem American National Standards Institute, offiziell ISO 8859). Er benötigt 1 Bit mehr, also 8 Bit (=1 Byte), und kann dafür 256 Zeichen darstellen. Die ersten 128 Zeichen entsprechen dem ASCII-Zeichensatz. Die restlichen 128 Plätze werden je nach Region unterschiedlich verwendet. Deshalb gibt es verschiedene Sets wie ISO 8859-1 („Latin-1“), das in Westeuropa zum Einsatz kommt.

Es gibt also verschiedene ANSI-Zeichensätze, aber keinen, der alle Zeichen enthält. Deshalb wurde seit den 1990er Jahren der **Unicode**-Zeichensatz entwickelt, der Ende 2010 bereits in Version 6 erschienen ist. Dieser enthält asiatische und arabische Zeichensätze, aber auch viele andere Zeichen wie Währungssymbole oder die Blindenschrift Braille. Der Unicode-Standard beinhaltet mittlerweile mehr als 100.000 Zeichen. Schriftdateiformate wie TrueType können jedoch nur maximal 65.536 Zeichen enthalten, so dass es momentan nicht möglich ist, alle Unicode-Zeichen auf einem PC darzustellen.

Zur Codierung von Unicode-Zeichen gibt es verschiedene Formate: Bei **UTF-16** besteht jedes Zeichen aus 2 oder 4 Bytes. **UTF-8** dagegen verwendet bei Zeichen, die sich auch mit nur 1 Byte darstellen lassen, nur 1 Byte. Deshalb sind ASCII-Strings kompatibel zu UTF-8. Wird der ASCII-Bereich

überschritten, verwendet UTF-8 automatisch mehr Bytes. Ein UTF-8-Zeichen kann maximal 4 Bytes lang werden.

#### 3.1.1.7 Nullterminierte Strings

Der folgende String-Typ wird nur verwendet, wenn man mit der „Außenwelt“ kommunizieren will – z.B. beim Aufrufen von Windows-Bibliotheken, die in C oder C++ geschrieben sind.

Bei nullterminierten Strings handelt es sich um Zeichenketten, die ihr Ende durch eine ASCII-Null (#0), also das erste Zeichen des ASCII-Zeichensatzes, kennzeichnen. Man deklariert sie so:

```
var Text: array [0..100] of char;
```

Da es sich hierbei um keine normalen Pascal-Strings handelt, müssen solche nullterminierten Strings mit speziellen Funktionen bearbeitet werden, z.B. StrPCopy, um einen Pascal-String in einen nullterminierten String zu kopieren.

Bei **PChar** handelt es sich um einen Zeiger auf ein C-kompatibles Zeichenarray. Dieser Typ wird von einigen API-Funktionen gefordert. Man erhält ihn ganz einfach, indem man einen String mittels PChar(↑langerText) umwandelt.

#### 3.1.1.8 Arbeiten mit Strings

Die Arbeit mit Strings (nicht nullterminierte Strings) ist recht einfach:

##### 3.1.1.8.1 Zeichenketten zusammenhängen

```
var Text1, Text2: String;
begin
  Text1 := 'toll!';
  Text2 := 'Ich finde Delphi ' + text1 + '!!!';
  // text2 enthält nun den Text 'Ich finde Delphi toll!!!'
```

##### 3.1.1.8.2 Zugreifen auf ein bestimmtes Zeichen eines Strings

Der Zugriff auf ein einzelnes String-Zeichen erfolgt über dessen Index:

```
var
  Text: String;
  Zeichen: Char;
begin
  Text := 'Ich finde Delphi toll!';
  Zeichen := Text[1];
  // zeichen enthält nun den Buchstaben 'I'
```

##### 3.1.1.8.3 Vergleich zweier Strings

Das Vergleichen von zwei Strings erfolgt mit dem Gleichheitszeichen. Auch wenn es sich bei Strings intern um Zeiger handelt, wird beim Vergleich der Inhalt der Strings verglichen, nicht die Speicheradresse, auf die die Zeiger zeigen (im Gegensatz zu Java). Beim Vergleich wird Groß- und Kleinschreibung beachtet.



```

var Text1, Text2: string;
begin
  ...
  if Text1 = Text2 then ...

```

Die Delphi-Laufzeitumgebung bietet noch einige weitere Funktionen, z.B. **AnsiCompareText** zum Vergleich zweier Strings ohne Berücksichtigung der Groß- und Kleinschreibung. **Pos** hilft beim Auffinden eines Teilstrings; **Copy** zum Kopieren eines Teilstrings und **Delete** zum Löschen eines Teilstrings sind ebenfalls wichtige Bearbeitungsmöglichkeiten.

Einige wichtige Funktionen für die Arbeit mit Strings sind in folgender Tabelle zusammengefasst.

Delete	<b>Löscht einen Teilstring</b>
Pos	Berechnet die Position eines Teilstrings
Copy	Erzeugt einen Teilstring aus einem String
Length	Gibt die Länge des Strings zurück
LowerCase	Gibt den String zurück nachdem er in Kleinbuchstaben umgewandelt wurde; um auch Umlaute umzuwandeln muss man AnsiLowerCase verwenden.
UpperCase	Gibt den String zurück nachdem er in Großbuchstaben umgewandelt wurde; um auch Umlaut umzuwandeln muss man AnsiUpperCase verwenden.

#### 3.1.1.8.4 StringBuilder

Zur effektiven Arbeit mit Strings bietet die Delphi Runtime Library seit Delphi 2009 die Klasse StringBuilder (Unit SysUtils).

```

var s1, s2: string;
    SB: TStringBuilder;
begin
  s1 := 'Hallo';
  s2 := 'Delphi';
  SB := TStringBuilder.Create(s1);
  try
    s1 := SB.Append(s2).ToString;
  finally
    SB.Free;
  end;

```

Im Gegensatz zu Java sind Strings in Delphi veränderbar. Der StringBuilder wurde in Delphi hauptsächlich für .NET eingeführt, weil dort Strings unveränderbar sind. Wenn zwei Strings zu einem verbunden werden sollen, bedeutet das, dass Speicher für den neuen String reserviert werden muss, dann werden die beiden alten Strings in den neuen kopiert und der Speicher der alten Strings

freigegeben. Im „normalen“ Win32-Delphi ist der StringBuilder nicht signifikant schneller als das Verbinden von Strings mit dem Plus-Zeichen.

### 3.1.2 Boolean

Variablen vom Typ „Boolean“ können einen Wahrheitswert aufnehmen: entweder `true` oder `false`. Weitere Zustände kennen Boolean-Variablen nicht. Hat man mehrere Variablen vom Typ Boolean, so kann man diese mit logischen Operatoren verbinden, man kann mit ihnen rechnen:

#### 3.1.2.1 Mit Wahrheiten rechnen

Es gibt verschiedene Rechenoperationen: Nullfunktion, Konjunktion (and), Inhibit, Identität, Antivalenz (xor), Disjunktion (or), nor, Äquivalenz, Negation (not), Implikation, nand, ...

Von diesen allen sollen und hier aber nur and, or, not, xor und die Äquivalenz interessieren.

##### 3.1.2.1.1 and

Der resultierende Wahrheitswert ist genau dann wahr, wenn beide Werte wahr sind:

```
var
  b1, b2: Boolean;
begin
  b1 := True;
  b2 := False;
  if b1 and b2 then
    ... // wird nicht ausgeführt
```

##### 3.1.2.1.2 or

Der resultierende Wahrheitswert ist genau dann wahr, wenn der eine oder der andere oder beide Werte wahr sind:

```
var
  b1, b2: Boolean;
begin
  b1 := True;
  b2 := True;
  if b1 or b2 then
    ... // wird ausgeführt
```

Das ist etwas anders als man es vom allgemeinen Sprachgebrauch her kennt. Mit or/oder ist das logische oder gemeint. Also nicht „entweder...oder“. Es können also auch beide Werte wahr sein.

##### 3.1.2.1.3 not

Not invertiert einen Wahrheitswert. Aus True wird False und aus False True:

```
var
  b: Boolean;
begin
  b := False;
  if not b then
    ... // wird ausgeführt
```

#### 3.1.2.1.4 Äquivalenz

Der resultierende Wahrheitswert ist genau dann wahr, beide Wahrheitswerte wahr oder beide falsch sind (also gleich).

```
var
  b1, b2: Boolean;
begin
  b1 := True;
  b2 := True;
  if b1 = b2 then
    ... // wird ausgeführt
```

#### 3.1.2.1.5 xor

xor (Antivalenz, exklusives oder) ist nun das aus dem allgemeinen Sprachgebrauch bekannte „entweder...oder“. Der resultierende Wahrheitswert ist genau dann wahr, wenn entweder der eine oder der andere, nicht aber beide Werte wahr sind (also unterschiedlich):

```
var
  b1, b2: Boolean;
begin
  b1 := True;
  b2 := True;
  if b1 xor b2 then
    ... // wird *nicht* ausgeführt
```

#### 3.1.2.2 Programmieren mit Boolean-Werten

Genug der grauen Theorie. Sehen wir uns nun einmal an, wo uns beim Programmieren überall Boolean-Werte begegnen:

##### 3.1.2.2.1 Boolean-Variablen

```
var
  b: Boolean;
```

Boolean-Variablen werden genauso deklariert, wie Integer und Stringvariablen z.B. auch. Rechnen kann man damit, wie oben beschrieben, mit den Operatoren and, or, not, xor und =. Dabei ist auch die Klammerung wichtig. Also im Zweifelsfall lieber mehr Klammern setzen...

##### 3.1.2.2.2 Funktionsrückgabewert

Viele Funktionen liefern als Ergebnis einen Boolean-Wert zurück. Dieser lässt sich dann auch in einer Variablen speichern:

```
var
  ok: Boolean;
begin
  ok := OpenDialog1.Execute; // Execute() gibt einen Boolean-Wert zurück
  // oder gleich auswerten:
  if SaveDialog1.Execute then
    SaveFile;
```

#### 3.1.2.2.3 Properties

```
if CheckBox1.Checked then
  ...
```

#### 3.1.2.2.4 Boolean-Ausdrücke

```
if a > b then
  DoSth;
```

$a > b$  ist hierbei ein Boolean-Ausdruck. Dieser ist genau dann True, wenn  $a$  größer als  $b$  ist... ;-)

#### 3.1.2.2.5 Kurzschluss?

Zur Auswertung von Boolean-Ausdrücken ist noch Folgendes zu sagen:

Bei

```
if a and b then
```

wird u.U.  $b$  gar nicht mehr geprüft, wenn  $a$  schon false ist. Ebenso ist es mit

```
if a or b
```

Hier wird die Auswertung unterbrochen, wenn  $a$  schon true ist. Dieses Verhalten wird manchmal Kurzschluss-Auswertung (short-circuit evaluation) genannt. In den Projektoptionen (Projekt->Optionen->Compiler->Vollst. Boolesche Auswertung) kann man dieses Verhalten abstellen. Dies ist jedoch nicht zu empfehlen, da die VCL diese teilweise recht ausgiebig nutzt. Über entsprechende Compiler-Direktiven lässt sich die Kurzschlussauswertung auch für bestimmte Code-Teile separat ausschalten:

```
{ $B } // vollständige Auswertung
{ $B- } // kurzschluss-Auswertung; default
```

Am einfachsten (und übersichtlichsten) ist es jedoch, wenn man entsprechende If-Anweisungen einfach schachtelt:

```
if Assigned(Objekt) then
begin
  if Objekt.Eigenschaft then
  begin
    ...
  end;
end;
```

statt

```
if (Assigned(Objekt)) and (Objekt.Eigenschaft) then  
begin  
    ...  
end;
```

So ist auf jeden Fall sichergestellt, dass auf das Objekt nur zugegriffen wird, wenn dieses auch wirklich existiert...

### 3.1.2.3 Typische Anfängerfehler

Viele Anfänger haben so ihre Probleme mit den Boolean-Werten. Die wichtigsten Anfänger-Fehler seien hier einmal genannt, damit man auch gleich lernen kann, sie zu vermeiden.

#### 3.1.2.3.1 Doppelt gemoppelt hält besser

```
if FileExists(FileName) = True then  
    ...  
// oder  
if FileExists(FileName) = False then  
    ...  
// oder  
var  
    b := Boolean;  
begin  
    b := True;  
    if b = True then  
        ...
```

Das alles ist ... falsch, wenn es auch in den meisten Fällen funktioniert.

Warum ist es nun falsch?

Das hat im Großen und Ganzen zwei Gründe. Der einleuchtendste zuerst:

Ein Boolean ist bereits Boolean. Man schreibt ja auch nicht:

```
if (a > b) = True then
```

Es ist einfach „doppelt gemoppelt“.

Für den zweiten Grund gibts erst einmal ein Codebeispiel:

```
// Diese Funktion soll das Verhalten mancher WinAPI-Funktionen imitieren:
```

```
function GaaanzFieseAPIFunktion: Boolean;  
asm  
    mov eax, -$01  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if GaaanzFieseAPIFunktion = True then  
        ShowMessage('Pech gehabt. Mich sieht man gar nicht!');  
    if GaaanzFieseAPIFunktion then  
        ShowMessage('Hallo');  
end;
```

Nanu, was passiert denn hier? Es wird ja wirklich nur die zweite Message angezeigt...

Das hängt mit der internen Darstellung zusammen. Ein Boolean wird als ganzes Byte gespeichert (einzelne Bits lassen sich nicht direkt speichern). Nun hat aber ein Byte 256 mögliche Werte und nicht nur 2. Es gibt nun ziemlich viele Möglichkeiten, wie man einen Boolean in einem Byte kodieren kann. Nicht alle sind sinnvoll, aber es gibt immerhin mehrere sinnvolle Möglichkeiten. In Delphi ist False 0 und 1 ist True, wobei alles andere erstmal nicht vorgesehen ist. Das ist aber nicht überall so. Andere Programmiersprachen, Frameworks, APIs & Co. könnten Boolean-Werte ganz anders kodieren (und tun dies auch). Sobald man also mit irgendetwas in Berührung kommt, was nicht 100% Delphi ist, muss man vorsichtig sein. Und das ist schneller passiert als man vielleicht vermutet. Z.B. dann, wenn man auf WinAPI-Funktionen zugreift.

Und genau das simuliert dieser Quelltext. Die Dummy-API-Funktion liefert -1 (also für die WinApi True) zurück. Da aber  $-1 \neq 1$  (True) ist, funktioniert das nicht ganz so, wie vielleicht gewünscht [4].

Wie macht mans also richtig?

```
// statt
  if FileExists(FileName) = True then
// ganz einfach
  if FileExists(FileName) then

// statt
  if FileExists(FileName) = False then
// richtig:
  if not FileExists(FileName) then

var
  b := Boolean;
begin
  b := True;
//statt
  if b = True then
// richtig:
  if b then
```

#### 3.1.2.3.2 für was denn not?

Manchmal, findet man auch solche abenteuerlichen Konstrukte:

```
if CheckBox1.Checked then
begin
end
else
  DoSth;
end;
```

Leere Anweisungsblöcke... Kurz und schmerzlos: So macht man's richtig:

```
if not CheckBox1.Checked then
begin
  DoSth;
end;
```

#### 3.1.2.3.3 Gehts nicht komplizierter?

```
if FileExists(FileName) then
  Result := False
else
  Result := True;
```

Auch dieser Code funktioniert. Allerdings liefert FileExists ja bereits einen Boolean-Wert zurück. Warum also nochmal abfragen? Man kann diese 4 Zeilen nämlich in einer einzigen schreiben:

```
Result := not FileExists(FileName);
```

Ganz ähnlich ist es hiermit:

```
if (a > 0) and (a <= 10) then
  ImBereich := True
else
  ImBereich := False;
```

Auch das kann man in eine einzige Zeile schreiben:

```
ImBereich := (a > 0) and (a <= 10);
```

```
if not (i = 5) then
  ...
```

Auch das lässt sich besser bzw. lesbarer schreiben:

```
if i <> 5
```

Einfach die richtigen Operatoren in der richtigen Situation benutzen. not ist zwar in vielen Fällen hilfreich und sinnvoll, aber nicht immer nötig...

#### 3.1.2.3.4 Klammern und warum man sie nicht vergessen sollte

Die Klammern im obigen Codebeispiel dienen nicht nur der besseren Lesbarkeit, sondern sind unabdingbar wichtig, damit sich das Programm korrekt übersetzen lässt. Auch dies ist ein beliebter Anfängerfehler. Lassen wir die Klammern weg, so erhalten wir einen netten Compilerfehler:

```
var
  a: Integer;
begin
  a := 5;
  if a = 5 or a = 9 then
    ShowMessage('hallo');
```

```
[dcc32 Fehler] Unit1.pas(32): E2008 Inkompatible Typen
```

Setzen wir die Klammern wieder, lässt sich der Code problemlos übersetzen...



## 3.2 Verzweigungen

### 3.2.1 if-else

Es gibt kaum ein Programm, bei dem immer alle Befehle hintereinander ausgeführt werden. Verzweigungen sind ein häufig eingesetztes Mittel. Es handelt sich hierbei um Fallunterscheidungen, die in der Regel mit `if` durchgeführt werden:

```
if <boolean-ausdruck> then  
    <anweisung>;
```

oder

```
if <boolean-ausdruck> then  
    <anweisung>  
else  
    <anweisung>;
```

Eine Anweisung kann dabei wiederum aus einer neuen if-Bedingung bestehen.

Beispiel:

```
if x > 0 then ...  
else if x < 0 then ...  
else ...;
```

Das Beispiel bedeutet Folgendes: Ist x größer als Null, wird das ausgeführt, was hinter dem ersten `then` steht (die drei Punkte). Handelt es sich dabei um mehr als einen Befehl, muss der Block mit `begin` und `end` umgeben werden.

`else`, zu Deutsch „sonst“, leitet eine Alternative ein. Wenn also die erste Bedingung nicht erfüllt ist, wird die zweite geprüft, hier, ob x vielleicht kleiner als Null ist. Trifft auch diese Bedingung nicht zu, bleibt noch ein `else` ohne Bedingung. Die letzten drei Punkte werden also immer dann ausgeführt, wenn die ersten beiden Bedingungen nicht zutreffen.

Wäre bereits die erste Bedingung erfüllt, so würden die folgenden `else`-Abschnitte gar nicht mehr geprüft.

Selbstverständlich muss so ein `if`-Block keine `else if`- oder `else`-Alternativen bieten. Das kommt immer auf die Situation an. Da sich das Ganze aber sehr stark an mathematische Logik anlehnt, dürfte die Notation nicht allzu schwerfallen.

Was allerdings zu beachten ist: In Delphi steht hinter dem letzten Befehl vor dem Wörtchen `else` kein Strichpunkt (Semikolon) wie sonst üblich – im Gegensatz zu C++.

Noch ein Beispiel, wobei jeweils mehrere Befehle ausgeführt werden:

```
var Eingabe: Integer;
...
if Eingabe = 1 then
begin
    Eingabe := 0;
    Ausgabe := 'Sie haben eine 1 eingegeben';
end //kein Strichpunkt!
else if Eingabe = 2 then
begin
    Eingabe := 0;
    Ausgabe := 'Sie haben eine 2 eingegeben';
end
else
begin
    Eingabe := 0;
    Ausgabe := 'Sie haben eine andere Zahl als 1 oder 2 eingegeben';
end;
```

Hier sieht man besonders den Sinn von `else`. Wären die drei Fallunterscheidungen durch drei getrennte `if`-Abfragen dargestellt worden, dann hätten wir folgendes Problem: Angenommen `Eingabe` ist 1, so ist die erste Bedingung erfüllt. Da hier `Eingabe` jedoch auf 0 gesetzt wird, träfe nun auch die dritte Bedingung zu. Im obigen Beispiel mit `else` stellt das kein Problem dar.

### 3.2.2 case-Verzweigung

Müssten wir in obigem Beispiel mehr als nur zwei Werte prüfen, hätten wir ganz schön Tipparbeit. Für solche abzählbaren (ordinalen) Typen wie `Integer` und `Char` gibt es in Delphi eine Abkürzung:

```
case Eingabe of
    1: Ausgabe := 'Sie haben 1 eingegeben';
    2: Ausgabe := 'Sie haben 2 eingegeben';
    3: Ausgabe := 'Sie haben 3 eingegeben';
    else Ausgabe := 'Sie haben nicht 1, 2 oder 3 eingegeben';
end;
```

Zugegeben, das Beispiel ist nicht besonders sinnvoll, da die Variable `Eingabe` direkt in einen String umgewandelt werden könnte. Allerdings stellt es gut die Funktionsweise von `case` dar. Zu beachten

ist, dass am Ende eines Case-Blocks ein **end** stehen muss. Gehören mehrere Anweisungen zusammen, können sie wie bei **if** durch **begin** und **end** als zusammengehörig gekennzeichnet werden.

Bei **case** steht (im Gegensatz zu **if**) vor dem Schlüsselwort **else** ein Strichpunkt!

Mit **case** ist auch Folgendes möglich:

```
case Eingabe of  
  1,3,5,7,9: Ausgabe := 'Sie haben eine ungerade Zahl kleiner als 10 eingegeben';  
  2,4,6,8,0: Ausgabe := 'Sie haben eine gerade Zahl kleiner als 10 eingegeben';  
  10..20:    Ausgabe := 'Sie haben eine Zahl zwischen 10 und 20 eingegeben';  
end;
```

### 3.3 Schleifen

#### 3.3.1 Was sind Schleifen?

Bisher liefen die Programme stur von oben nach unten ab. In einem Programmablauf kommt es öfters vor, dass eine bestimmte Befehlsfolge mehrmals hintereinander ausgeführt werden soll. So etwas nennt man „Schleife“. Nur mal als Beispiel: Wer möchte die Zahlen von 1 bis 10.000 einzeln mit `writeln()` ausgeben? Mit einer Schleife geht das ruck-zuck.

Schleifen gibt es in unterschiedlichen Arten: Sie unterscheiden sich darin, ob die Abbruchbedingung vor dem ersten Durchlauf geprüft werden soll oder erst danach und ob bereits feststeht, wie oft eine Schleife durchlaufen wird.

#### 3.3.2 for-Schleife

Die for-Schleife hat folgenden Aufbau:

```
for i := 1 to 10 do  
begin  
    ... // Befehlsfolge, die öfters ausgeführt werden soll  
end;
```

Die Beispielschleife wird von 1 bis 10, also zehnmal durchlaufen. Nach jeder „Runde“ wird die sog. Schleifenvariable automatisch um 1 erhöht. Die Schleifenvariable heißt im Beispiel `i`. Das muss natürlich nicht so sein. Allerdings ist es eine übliche Konvention. Auf jeden Fall ist es aber ein Ordinalwert. Die Grenzen (1 bis 10) sind hier direkt als Zahlen vorgegeben, es können jedoch auch Integer-Variablen, Aufzählungen oder Chars sein. Es sind alle Typen erlaubt, bei denen es eine feste Reihenfolge gibt.

Die Schleifenvariable darf grundsätzlich nicht innerhalb der For-Schleife verändert werden. Ist die Obergrenze kleiner als die Untergrenze, wird die Schleife nicht durchlaufen (z. B. `for i:=1 to 0`); sind die Grenzen identisch, wird sie einmal durchlaufen.

Alternativ zum Hochzählen der Schleifenvariable ist auch Folgendes möglich:

```
for i := 10 downto 1 do ...
```

#### 3.3.3 while-Schleife

Im Gegensatz zur for-Schleife verwendet die while-Schleife keine Schleifenvariable, die automatisch hochgezählt wird. Hier wird vor jedem Durchlauf geprüft, ob eine bestimmte Bedingung

erfüllt ist. Trifft diese nicht mehr zu, wird die Schleife nicht mehr durchlaufen und der Programmablauf danach fortgesetzt. Trifft die Bedingung bereits am Anfang nicht zu, wird die Schleife überhaupt nicht betreten. Man spricht hier von einer kopfgesteuerten Schleife.

Die while-Schleife hat folgende Struktur:

```
while x <> y do
begin
    ... // Befehlsfolge, die öfters ausgeführt werden soll
end;
```

Solange also x ungleich y ist, wird die Schleife durchlaufen. Es ist also ratsam, x und/oder y innerhalb der Schleife zu verändern; andernfalls wäre das Durchlaufkriterium immer erfüllt, die Schleife würde nie zu einem Ende kommen. Ein Programmierer spricht hier von einer Endlosschleife, die dazu führt, dass die Anwendung nicht mehr reagiert. Ist x bereits zu Beginn gleich y wird die Schleife überhaupt nicht durchlaufen.

Die Bedingung hinter `while` kann ein beliebiger Ausdruck sein, der einen Wahrheitswert (Boolean) ergibt.

### 3.3.4 repeat-until-Schleife

War bei der `while`-Schleife das Durchlaufkriterium anzugeben, ist es bei der `repeat-until`-Schleife das Abbruchkriterium. Außerdem wird dieses erst am Ende eines Schleifendurchlaufs geprüft. Ein Durchlauf findet also auf jeden Fall statt. Man spricht hier von einer fußgesteuerten Schleife.

```
repeat
    ... // Befehlsfolge, die öfters ausgeführt werden soll
until x = y;
```

Die Beispielschleife wird solange durchlaufen, bis x gleich y ist. Auch hier ist wieder darauf zu achten, dass keine Endlosschleife entsteht. Auch wenn x schon zu Beginn gleich y ist, wird die Schleife dennoch einmal durchlaufen.

### 3.3.5 for-in-Schleife

Seit der Version 2005 gibt es eine neue Schleife in Delphi – die `for-in`-Schleife. Dies erleichtert besonders Schleifen über Arrays, wenn kein Indexwert benötigt wird. Die allgemeine Syntax lautet:

```
for element in collection do statement;
```

Beispiel:

```
var StringArr: array of String;  
    s: String;  
begin  
    ...  
    for s in StringArr do  
        ShowMessage(s);  
end;
```

### 3.3.6 Schleifen abbrechen

Schleifen lassen sich natürlich auch vor dem regulären Ende verlassen. Dazu gibt es **break**. **break** kann nur innerhalb von Schleifen verwendet werden und setzt den Programmablauf mit der ersten Anweisung nach der Schleife fort. **break** sollte sparsam verwendet werden, da dadurch die Verständlichkeit des Quellcodes leidet.

Außerdem gibt es Situationen, in denen man schon zu Beginn eines Schleifendurchlaufs weiß, dass man gleich mit der nächsten „Runde“ fortfahren kann. Hier kann man **continue** verwenden. Dadurch wird die Durchführung eines Schleifendurchlaufs abgebrochen und mit dem nächsten Durchlauf begonnen. Bei **for**-Schleifen wird der Index erhöht.

### 3.4 Eigene Datentypen definieren

Wie wir in den folgenden Abschnitten sehen werden, gibt es in Delphi die Möglichkeit, Datentypen auf die eigenen Bedürfnisse anzupassen, z.B. einen Wertebereich einzuschränken wie beim Teilbereichstyp oder aus mehreren Variablen zusammenzusetzen wie bei Records. In diesen Fall ist es ratsam, dem neuen Typ erst einmal einen Namen zu verpassen, unter dem man ihn später verwenden kann. Bei „einfachen“ Datentypen wie String, Real, Integer, Boolean usw. ist das natürlich nicht erforderlich.

#### 3.4.1 Typdefinition

Eine Typdefinition wird in einem type-Block hinterlegt. Das Konstrukt ähnelt einer Variablendeklaration. Da aber statt einer Variablen ein Typ deklariert wird, ist das Folgende auch kein Variablen- sondern ein Typname. Typnamen beginnen in Delphi mit einem großen T, wenn sich der Programmierer an den Styleguide gehalten hat. Statt eines Doppelpunkts wird ein Gleichheitszeichen verwendet.

In den folgenden Beispielen wird zuerst ein neuer Typ definiert („type“) und anschließend eine Variable von diesem Typ angelegt („var“). Innerhalb einer Unit sähe das z.B. so aus:

```
program MeinProgramm;  
  
{$APPTYPE CONSOLE}  
  
uses  
    System.SysUtils;  
  
type  
    TKleineZahl = 0..200;  
  
var  
    kleineZahl: TKleineZahl;  
  
begin  
    ...  
end
```

#### 3.4.2 Teilbereichstypen

Einen weiteren Datentyp gibt es noch, nämlich den Teilbereichstyp, auch Unterbereichstyp genannt. Hierüber kann man Variablen z. B. zwar den Typ Integer zuordnen, aber nicht den kompletten Definitionsbereich, sondern nur einen Teilbereich davon:

```
type TKleineZahl = 0..200;  
var kleineZahl: TKleineZahl;
```

Wir definieren hier zuerst einen eigenen Typ mit dem Namen TKleineZahl. Von diesem Typ können wir anschließend beliebig viele Variablen deklarieren und ihn natürlich auch in Methoden-Signaturen für Parametertypen verwenden.

Der Variablen `KleineZahl` lassen sich jetzt nur ganze Zahlen zwischen 0 und 200 zuweisen.

### 3.4.3 Aufzählungstypen („Enumeration Types“)

Will man nicht mit Zahlen, aber auch nicht mit Freitext arbeiten, kann man einen Aufzählungstyp definieren. Das macht den Code besser lesbar. Die einzelnen Werte werden fest definiert und sind zur Laufzeit nicht erweiterbar.

```
type TFarbe = (blau, gelb, gruen, rot);  
var Farbe: TFarbe;
```

Intern werden die Werte bei Null beginnend durchnummeriert und haben deshalb eine feste Reihenfolge. Über die Funktion `Ord` lässt sich die Position bestimmen.

Wichtige Funktionen zum Arbeiten mit Aufzählungstypen sind:

<code>ord</code>	gibt die Position des Bezeichners zurück
<code>pred</code>	gibt den Vorgänger zurück
<code>succ</code>	gibt den Nachfolger zurück
<code>low</code>	gibt den niedrigsten Wert zurück
<code>high</code>	gibt den höchsten Wert zurück

Die Typen `Integer` und `Char` gehören ebenfalls zu den **ordinalen (abzählbaren) Typen**, d.h. die Werte lassen sich in einer Reihenfolge anordnen, weshalb o.g. Funktionen auf sie ebenfalls angewandt werden können.

### 3.4.4 Mengentypen („Set“)

Um in einer einzigen Variablen eine unterschiedliche Menge an Werten des gleichen Typs zu speichern, gibt es Mengentypen. Es ist eine Menge an möglichen Werten vorgegeben, aus der eine beliebige Anzahl (keiner bis alle) in der Variablen abgelegt werden kann. Folgendermaßen wird eine solche Mengenvariable deklariert:

```
type TZahlen = set of 1..10;  
var Zahlen: TZahlen;
```

Damit können der Variablen `zahlen` Werte aus der Menge der Zahlen von 1 bis 10 zugewiesen werden - mehrere gleichzeitig oder auch gar keiner:

```
Zahlen := [5, 9];           // zahlen enthält die Zahlen 5 und 9  
Zahlen := [];              // zahlen enthält überhaupt keine werte  
Zahlen := [1..3];          // zahlen enthält die Zahlen von 1 bis 3  
Zahlen := Zahlen + [5];     // zahlen enthält die Zahlen 1, 2, 3, 5  
Zahlen := Zahlen - [3..10]; // die Zahlen von 3 bis 10 werden aus der Menge  
                           // entfernt, es bleiben 1 und 2
```

Um nun zu prüfen, ob ein bestimmter Wert in der Menge enthalten ist, wird der Operator `in` verwendet:



```
if 7 in Zahlen then ...
```

In einem Set sind nur Werte mit der Ordnungsposition von 0 bis 255 möglich.

### 3.4.5 Arrays

Müssen mehrere Werte des gleichen Typs gespeichert werden, ist ein `Array` (zu deutsch Feld) eine praktische Lösung. Will man auf ein einzelnes Element eines Arrays zugreifen, verwendet man einen Index, den man in eckigen Klammern hinter den Variablennamen schreibt. Am einfachsten lässt sich das mit einer Straße vergleichen, in der sich lauter gleiche Häuser befinden, die sich jedoch durch ihre Hausnummer (den Index) unterscheiden.

Eine Deklaration der Art

```
type TTestwert = array [0..10] of Integer;  
var Testwert: TTestwert;
```

bewirkt also, dass wir in `Testwert` quasi elf verschiedene Integer-Variablen bekommen. Man kann auf sie über den Indexwert zugreifen:

```
Testwert[0] := 15;  
Testwert[1] := 234;
```

usw.

Vorteil dieses Indexes ist, dass man ihn durch eine weitere Variable ersetzen kann, die dann in einer Schleife hochgezählt wird. Folgendes Beispiel belegt alle Elemente mit dem Wert 1:

```
for i := 0 to 10 do  
  Testwert[i] := 1;
```

Auf Schleifen wird jedoch in einem gesonderten Kapitel eingegangen.

Bei dem vorgestellten `Array` handelt es sich genauer gesagt um ein eindimensionales, statisches `Array`. „Eindimensional“, weil die Elemente über nur einen Index identifiziert werden, und „statisch“, weil Speicher für alle Elemente reserviert wird. Legt man also ein `Array` für Indexwerte von 1 bis 10000 an, so wird für 10000 Werte Speicher reserviert, auch wenn während des Programmlaufs nur auf zwei Elemente zugegriffen wird. Außerdem kann die `Array`-Größe zur Programmlaufzeit nicht verändert werden.

#### 3.4.5.1 Dynamische Arrays

Wenn schon so viel Wert auf die Eigenschaft statisch gelegt wird, muss es ja eigentlich auch etwas Dynamisches geben. Und das gibt es auch, zumindest seit Delphi 4: die dynamischen Arrays.

Der erste Unterschied findet sich in der Deklaration: Es werden keine Grenzen angegeben.

```
type TDynArray = array of Integer;  
var DynArray: TDynArray;
```

`DynArray` ist nun prinzipiell ein Feld von Integer-Werten, die bei Index Null beginnt.

Bevor man Werte in das Array stecken kann, muss man Speicher für die Elemente reservieren. Dabei gibt man an, wie groß das Array sein soll:

```
SetLength(DynArray, 5);
```

Nun kann das Array fünf Elemente (hier Integer-Zahlen) aufnehmen.

Man beachte: Da die Zählung bei Null beginnt, befindet sich das fünfte Element bei Indexposition 4!

Der Zugriff erfolgt ganz normal:

```
DynArray[0] := 321;
```

Damit es mit der Unter- und vor allem der Obergrenze, die ja jederzeit verändert werden kann, keine Probleme gibt (Zugriffe auf nicht (mehr) reservierten Speicher), lassen sich Schleifen am einfachsten so realisieren:

```
for i := 0 to high(DynArray) do  
  DynArray[i] := 0;
```

Dadurch werden alle Elemente auf 0 gesetzt. `high(dynArray)` entspricht dem höchstmöglichen Index. Über `length(a)` lässt sich die Länge des Arrays ermitteln, welche immer `high(dynArray)+1` ist. Dabei handelt es sich um den Wert, den man mit `SetLength` gesetzt hat.

Da die Länge des Arrays jederzeit verändert werden kann, könnten wir sie jetzt mit

```
SetLength(DynArray, 2);
```

auf zwei verkleinern. Die drei hinteren Werte fallen dadurch weg. Würden wir das Array dagegen vergrößern, würden sich am Ende Elemente mit undefiniertem Wert befinden.

### 3.4.5.2 Mehrdimensionale Arrays

Wenn es eindimensionale Arrays gibt, muss es auch mehrdimensionale geben. Am häufigsten sind hier wohl die zweidimensionalen. Man kann mit ihnen z. B. ein Koordinatensystem oder Schachbrett abbilden. Die Deklaration ist wie folgt:

```
type TKoordinate = array [1..10, 1..10] of Integer;  
var Koordinate: TKoordinate;
```

Es werden also 10×10=100 Elemente angelegt, die jeweils einen Integer-Wert aufnehmen können. Für den Zugriff auf einzelne Werte sind zwei Schreibweisen möglich:

```
Koordinate[1, 6] := 34;  
Koordinate[7][3] := 42;
```

Und es gibt auch mehrdimensionale dynamische Arrays:

```
type TKoordinate = array of array of Integer;  
var Koordinate: TKoordinate;
```

Die Längenzuweisung erfolgt dann durch Angabe der Länge für jede Dimension:

```
SetLength(koordinate, 10, 10);
```

Zudem ist es möglich, ungleichförmige Arrays anzulegen.

```
SetLength(koordinate, 2);  
SetLength(koordinate[0], 3);  
SetLength(koordinate[1], 4);
```

Dies beschreibt ein Array mit 2 Spalten, wobei die erste Spalte 3 Zeilen und die zweite Spalte 4 Zeilen beinhaltet.

### 3.4.6 Records

Records entsprechen von der Struktur her einem Datensatz einer Datenbank – nur dass sie, wie alle bisher aufgeführten Variablen, nur zur Laufzeit vorhanden sind. Wenn wir unterschiedliche Daten haben, die logisch zusammengehören, können wir sie sinnvollerweise zu einem Record zusammenfassen.

Beispiel: Personen.

```
type TPerson = record  
    Name: String;  
    Alter: Integer;  
    Gewicht: Real;  
end;  
var Person: TPerson;
```

Alle Variablen vom Typ TPerson werden also in drei „Untervariablen“ aufgegliedert.

Folgendermaßen greift man auf die einzelnen Felder zu:

```
Person.Name := 'Hans Müller';  
Person.Alter := 59;  
Person.Gewicht := 77.5;
```

#### 3.4.6.1 Beispiel

Im Folgenden wird eine kleine Personenverwaltung basierend auf einem Array und Records erstellt. Im Array werden 10 Personen Platz finden. Für jede Person werden der Name, das Geschlecht und die Adresse gespeichert. Das Beispiel hat nicht sehr viel gemein mit einem echten Adressbuch, da alle Daten bereits in den Code geschrieben werden, aber es verdeutlicht sehr schön den Nutzen von Records. Stellen wir uns einmal vor, wir müssten für 10 Personen jeweils den Namen, das Geschlecht und die Adresse in einer eigenen Variablen speichern. Da ist eine Zusammenfassung zu einem Record viel angenehmer und übersichtlicher.

```
program Personen;  
  
{$APPTYPE CONSOLE}  
  
uses  
    System.SysUtils;  
  
type  
    TGeschlecht = (geFrau, geMann);  
  
    TPerson = record  
        Name: string;  
        Geschlecht: TGeschlecht;  
        Adresse: string;  
    end;  
  
var  
    Personen: array[0..9] of TPerson;  
  
begin  
    Personen[0].Name := 'Hans Müller';  
    Personen[0].Geschlecht := geMann;  
    Personen[0].Adresse := 'Musterstraße 3';  
    // Weitere Personen einfügen  
    WriteLn(Personen[0].Name + ', ' + Personen[0].Adresse);  
    ReadLn;  
end.
```

### 3.4.7 Zeiger („Pointer“)

Bei Zeigern handelt es sich um Variablen, die eine Speicheradresse enthalten. Eine Zeigervariable zeigt also auf eine bestimmte Stelle im Arbeitsspeicher. Zum einen gibt es den Typ Pointer, der auf beliebige Daten zeigt und zum anderen spezialisierte Zeigertypen.

In der Delphi-Sprache kommen Zeiger (auch Referenzen genannt) häufig vor, so z.B. bei Objektreferenzen, bei langen Strings und bei dynamischen Arrays. An der Syntax fällt das an diesen Stellen nicht auf. Will man Zeiger auf selbstdefinierte Typen erstellen, muss dagegen die noch aus Pascal-Zeiten stammende Zeigersyntax mit @ und ^ verwendet werden.

Doch zunächst ein Beispiel zur Deklaration von Zeigern:

```
type
  PAdressRecord = ^TAdressRecord;
  TAdressRecord = record
    name: string;
    plz: integer;
    ort: string;
end;

var adresse: TAdressRecord;
    adresszeiger: PAdressRecord;
```

#### 3.4.7.1 ^ zur Zeigertypdefinition

Steht das Symbol ^ vor einem Typbezeichner, wird daraus ein Typ, der einen Zeiger auf den ursprünglichen Typ darstellt. ^TAdressRecord ist ein Zeigertyp auf einen Speicherbereich, an dem sich etwas vom Typ TAdressRecord befinden muss.

#### 3.4.7.2 @ zur Ermittlung einer Speicheradresse

Anfänglich zeigt unsere Variable adresszeiger ins Leere - auf nil (not in list). Wir wollen nun, dass er auf die Variable adresse zeigt. Da ein Zeiger sich bekanntlich nur Speicheradressen merken kann, benötigen wir die Speicheradresse unserer Variable adresse. Diese ermitteln wir mit dem Operator @:

```
adresszeiger := @adresse;
```

#### 3.4.7.3 ^ zur Dereferenzierung

Nun können wir auch über den Zeiger auf den Inhalt von adresse zugreifen. Wir wollen also, dass uns der Zeiger nicht die Adresse bekannt gibt, auf die er zeigt, sondern den Wert an dieser Speicherstelle. Diesen Vorgang nennt man Dereferenzieren. Verwendet wird dafür wieder das Symbol ^ - allerdings diesmal hinter einer Zeigervariablen:

```
var gesuchterName: string;
begin
  gesuchterName := adresszeiger^.name;
```

Eine Dereferenzierung ist nur mit spezialisierten Zeigern möglich, nicht mit dem Allround-Talent Pointer. Um auch mit dem Typ Pointer entsprechend arbeiten zu können, muss dieser in einen anderen Zeigertyp umgewandelt werden.

## 3.5 Prozeduren und Funktionen

### 3.5.1 Was sind Prozeduren und Funktionen?

Prozeduren und Funktionen, auch „Unterprogramme“ oder Routinen genannt, haben die Aufgabe, öfter wiederkehrenden Programmcode sozusagen als Baustein zusammenzufassen. Dieser Baustein erhält einen eindeutigen Namen, über den er ausgeführt werden kann.

### 3.5.2 Aufbau einer Prozedur

Jede Prozedur besteht aus dem Schlüsselwort **procedure**, gefolgt von einem gültigen Namen und evtl. einer Parameterliste in runden Klammern. Sind keine Parameter vorhanden, können die Klammern sowohl bei der Deklaration als auch beim Aufruf weggelassen werden. Diesen Teil nennt man Kopf der Prozedur. Es folgen Variablen- und Konstantendeklarationen und anschließend zwischen **begin** und **end** die Anweisungen, die die Prozedur durchführen soll:

```
procedure <Name>(<Parameter>);  
<Variablen- und Konstanten>  
begin  
    <Anweisungen>  
end;
```

Beispiel: Die folgende Prozedur gibt so viele Töne über den PC-Lautsprecher aus, wie über den Parameter Anzahl angegeben.

```
procedure Toene(Anzahl: integer);  
var i: Integer;  
begin  
    for i := 1 to Anzahl do  
        beep;  
end;
```

Der Aufruf für fünf Töne geschieht so:

```
Toene(5);
```

### 3.5.3 Aufbau einer Funktion

Eine Funktion unterscheidet sich nur geringfügig von einer Prozedur. Sie besitzt einen Rückgabewert und wird mit dem Schlüsselwort **function** deklariert anstelle von **procedure**.

```
function <Name>(<Parameter>): <Rückgabety>;  
<Variablen- und Konstanten>  
begin  
    <Anweisungen>  
end;
```

Beispiel: Eine Funktion, die drei Zahlen addiert und das Ergebnis zurückliefert.

```
function SummeAusDrei(Zahl1, Zahl2, Zahl3: Integer): Integer;  
begin  
    Result := Zahl1 + Zahl2 + Zahl3;  
end;
```

Bei `result` handelt es sich um eine vordefinierte Variable, der der Rückgabewert zugewiesen wird, den die Funktion haben soll. Es ist möglich, `result` öfters einen Wert zuzuweisen. Letztlich bildet der Wert den Rückgabewert der Funktion, der der Variablen `result` als letztes zugewiesen wurde.

Der Rückgabewert kann dann an der Aufrufstelle ausgewertet werden (`ergebnis` sei eine Integer-Variable):

```
Ergebnis := SummeAusDrei(3, 5, 9);
```

### 3.5.4 forward- und interface-Deklarationen

Prozeduren und Funktionen können nur aufgerufen werden, wenn ihr Name im Code bekannt ist. Und dieser Gültigkeitsbereich beginnt erst an der Stelle der Deklaration. Soll eine Routine, die nicht im Kopf der Unit deklariert ist, vor ihrer Definition bekannt sein, wird eine forward-Deklaration eingesetzt.

Dabei wird lediglich der Kopf einer Routine, versehen mit der Direktive `forward`, an eine frühere Stelle im Code gesetzt. Für die Funktion `SummeAusDrei` sähe das so aus:

```
function SummeAusDrei(Zahl1, Zahl2, Zahl3: Integer): Integer; forward;
```

Die eigentliche Funktion (die Definition), wie sie im letzten Abschnitt dargestellt ist, muss dann später im Code folgen.

Soll eine Prozedur oder Funktion auch aus einer anderen Unit aufrufbar sein, muss ihr Kopf im Interface-Teil der Unit stehen. Die Definition folgt im Implementation-Teil. Das Verhalten entspricht einer forward-Deklaration, die Direktive `forward` darf hierbei aber nicht verwendet werden.

Folgendes Beispiel zeigt eine interface-Deklaration, Implementierung sowie Aufruf einer Funktion:

```
unit Unit1;

interface

uses
    System.SysUtils;

    function SummeAusDrei(Zahl1, Zahl2, Zahl3: Integer): Integer;
    //Deklaration nur mit dem Kopf der Funktion

implementation

{$R *.DFM}

function SummeAusDrei(Zahl1, Zahl2, Zahl3: Integer): Integer;
begin
    Result := Zahl1 + Zahl2 + Zahl3;
end;
```

### 3.5.5 Parameter

In den meisten Fällen wird eine Routine zwar öfters gebraucht, allerdings – z. B. bei Berechnungen – nicht immer mit den gleichen Werten. Deshalb gibt es, wie oben bereits gesehen, die Möglichkeit, Routinen Werte beim Aufruf zu übergeben.

Beim Aufruf einer Prozedur/Funktion mit Parametern muss beachtet werden, dass Anzahl und Typ der Werte übereinstimmen.

Anhand der Reihenfolge der Werte steht in obigem Beispiel fest, dass die Variable `zahl1` den Wert 3, `zahl2` den Wert 5 und `zahl3` den Wert 9 erhält. Diese Variablen werden nicht wie üblich über `var` deklariert. Ihre Deklaration erfolgt durch die Nennung im Funktionskopf. Außerdem gelten sie nur innerhalb der Funktion. Von außerhalb (z. B. nach Beendigung der Funktion) kann nicht mehr auf sie zugegriffen werden.

Um das Ganze etwas komplizierter zu machen, gibt es verschiedene Arten von Parametern, die durch `var`, `const` oder `out` gekennzeichnet werden.

#### 3.5.5.1 Wert- und Variablenparameter

In obigen Beispielen wird immer eine Kopie eines Wertes an die Prozedur/Funktion übergeben. Wenn dieser Wert also innerhalb der Prozedur/Funktion verändert wird, ändert sich nicht die Variable, die beim Aufruf verwendet wurde:



```
procedure MachWas(Zahl: Integer);  
begin  
    Zahl := Zahl + 5;  
end;  
  
procedure Aufruf;  
var EineZahl: Integer;  
begin  
    EineZahl := 5;  
    MachWas(EineZahl);  
end;
```

Im Beispiel ruft also die Prozedur **Aufruf** die Prozedur **MachWas** mit dem Wert der Variablen **EineZahl** auf. In **MachWas** wird dieser Wert über **Zahl** angesprochen. Und obwohl **Zahl** nun verändert wird, ändert sich der Wert in **EineZahl** nicht. Er ist am Ende immer noch 5. Man spricht von einem Wertparameter, es wird nur der Inhalt der Variablen übergeben.

Im Fall des Variablenparameters wird das „Original“ übergeben. Ein solcher Parameter wird mit dem Schlüsselwort **var** gekennzeichnet.

```
procedure MachWas(var Zahl: Integer);  
begin  
    Zahl := Zahl + 5;  
    ...  
end;  
  
procedure Aufruf;  
var EineZahl: Integer;  
begin  
    EineZahl := 5;  
    MachWas(EineZahl);  
end;
```

Hier wird keine Kopie des Variableninhalts übergeben, sondern eine Referenz (also die Speicheradresse) der Variablen **EineZahl**. Wird der Wert in **MachWas** nun um 5 erhöht, geschieht dies auch mit der Variablen **EineZahl**, weil es sich um dieselbe Variable im Speicher handelt. Sie wird nur von den beiden Prozeduren mit anderen Namen angesprochen.

Über den Umweg des **var**-Parameters kann man sogar Prozeduren dazu bewegen, Werte zurückzugeben:

```

procedure MachWas2(var wert1, wert2: Integer);
begin
    wert1 := 2;
    wert2 := 3;
end;

procedure Aufrufen;
var Zahl1, Zahl2: Integer;
begin
    MachWas2(Zahl1, Zahl2);
    ...
end;

```

Dass die Variablen Zahl1 und Zahl2 vor der Übergabe an MachWas2 nicht initialisiert wurden, macht nichts, da sie dort sowieso nicht ausgelesen werden. In MachWas2 werden Wert1 und Wert2 Werte zugewiesen - und da es sich dabei um Referenzparameter handelt, automatisch auch Zahl1 und Zahl2. Wenn MachWas2 abgearbeitet wurde, enthält Zahl1 also den Wert 2 und Zahl2 den Wert 3.

#### 3.5.5.2 Konstantenparameter

Wird ein übergebener Wert in der Funktion/Prozedur nicht verändert und auch nicht als var-Parameter zum Aufruf einer weiteren Routine verwendet, kann man ihn als Konstantenparameter (**const**) deklarieren:

```

procedure MachWas(const Zahl: Integer);

```

Das ermöglicht dem Compiler eine bessere Optimierung, außerdem wird nun nicht mehr zugelassen, dass der Wert innerhalb der Prozedur verändert wird.

#### 3.5.5.3 Ausgabeparameter

Ausgabeparameter werden mit dem Schlüsselwort **out** deklariert. Wie der Name bereits sagt, können solche Parameter nur zur Zuweisung eines Ausgabewerts verwendet werden. Eine Übergabe von Werten an eine Routine ist damit nicht möglich. Ansonsten entspricht der Ausgabeparameter einem Variablenparameter.

#### 3.5.5.4 Array-Parameter

Es ist natürlich auch möglich, Arrays als Parameter einer Routine zu verwenden. Jedoch nicht auf die Art, die man intuitiv wählen würde:

```

procedure MachWasAnderes(Feld: array [1..20] of Integer); //falsch!

```

Stattdessen muss zunächst ein eigener Typ definiert werden. Dieser kann dann in Prozedur- oder Funktionsköpfen verwendet werden:

```
type TMeinFeld = array [1..20] of Integer;  
procedure MachWasAnderes(Feld: TMeinFeld);
```

#### 3.5.5.5 Default-Parameter

In Delphi können Parameter optional gemacht werden, so dass beim Aufruf auf sie verzichtet werden kann. Dafür müssen zwei Bedingungen erfüllt sein:

- Der optionale Parameter benötigt einen Default-Wert, der dann verwendet wird, wenn der Parameter beim Aufruf nicht angegeben wird.
- Die optionalen Parameter müssen am Ende der Parameterliste stehen. Das Weglassen beim Aufruf ist nur von hinten her möglich.

Beispiel:

```
function MyTest(a: Integer; b: Integer = 42);
```

In diesem Beispiel muss der zweite Parameter (b) nicht übergeben werden. Dann wird der Wert 42 verwendet. Es sind also folgende Aufrufe möglich:

```
MyTest(100); // a = 100, b = 42  
MyTest(100, 100); // a = 100, b = 100
```

#### 3.5.6 Prozeduren und Funktionen überladen

In Delphi ist es möglich, im selben Gültigkeitsbereich mehrere Routinen mit identischem Namen zu deklarieren. Dieses Verfahren wird „Überladen“ genannt.

Überladene Routinen müssen mit der Direktiven `overload` deklariert werden und unterschiedliche Parameterlisten haben.

```
function Divide(X, Y: Real): Real; overload;  
begin  
    Result := X / Y;  
end;  
  
function Divide(X, Y: Integer): Integer; overload;  
begin  
    Result := X div Y;  
end;
```

Diese Deklarationen definieren zwei Funktionen namens `Divide`, die Parameter unterschiedlicher Typen entgegennehmen. Wenn `Divide` aufgerufen wird, ermittelt der Compiler die zu verwendende Funktion durch Prüfung des übergebenen Parametertyps.

`Divide(6.0, 3.0)` ruft beispielsweise die erste `Divide`-Funktion auf, da es sich bei den Argumenten um reelle Zahlen handelt, auch wenn der Nachkommateil Null ist.

Überladene Methoden müssen sich deshalb entweder in der Anzahl der Parameter oder in den Typen dieser Parameter signifikant unterscheiden.

### 3.5.7 Prozeduren und Funktionen abbrechen

Nach dem Ausführen einer Prozedur bzw. Funktion wird die Programmausführung an der aufrufenden Stelle fortgesetzt. Wenn man dort aber weitermachen will, bevor die Prozedur/Funktion vollständig ausgeführt wurde, kann man `exit` verwenden. `exit` bricht eine Prozedur/Funktion ab und setzt das Programm an der aufrufenden Stelle fort. Bei Funktionen ist darauf zu achten, dass bereits ein Rückgabewert definiert wurde. Rückgabewerte werden der automatisch vorhandenen Variablen `result` zugewiesen.

Im Gegensatz zu `return` in anderen Sprachen bricht `result` den Ablauf in einer Routine nicht an dieser Stelle ab. Allerdings ist es seit Delphi 2009 möglich, den Befehl `exit`, der normalerweise nur die Ausführung einer Routine abbricht, mit einem Rückgabewert zu versehen:

```
function MyFunction(x: Integer): Integer;  
begin  
  if (x < 0) then  
    Exit(0);  
  ...  
end;
```

In diesem Beispiel gibt die Funktion den Wert 0 zurück, wenn der Aufrufparameter `x` kleiner als 0 ist. Der Code, der anstelle der drei Punkte folgt, wird dann nicht mehr ausgeführt. Er wird nur durchlaufen, wenn `x` größer oder gleich 0 ist.

### 3.6 Programmaufbau

#### 3.6.1 Projektdatei

Eine in Delphi erstellte Anwendung (auch Projekt genannt) besteht aus einer Projektdatei mit der Endung **.dpr**, die das Hauptprogramm enthält, und evtl. einer oder mehreren Units. Unter Units versteht man die Quelltextdateien, die später den eigentlichen Quellcode enthalten werden. Sie tragen die Endung **.pas**.

Der Aufbau der Projektdatei sieht folgendermaßen aus, wenn es sich um eine Anwendung mit grafischer Benutzeroberfläche (GUI) handelt:

```
program Project1;

uses
  Vcl.Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Das Schlüsselwort **program** legt fest, dass aus dem Projekt nach dem Kompilieren eine ausführbare Anwendung wird. In der Projektdatei wird definiert, aus welchen Teilen (Units) unser Projekt besteht (das sind die beiden Zeilen unter „**uses**“). Anschließend werden die Formulare erzeugt (**CreateForm**) und schließlich die Anwendung gestartet (**Run**).

Über Datei/Neu/Weitere.../Konsolenanwendung ist es jedoch auch möglich ein Programm mit einer textbasierten Oberfläche (Konsole) zu erstellen. Dabei hat die dpr-Datei prinzipiell folgenden Aufbau:

```
program Project2;
{$APPTYPE CONSOLE}
{$R *.RES}
uses
  System.SysUtils;
begin
  // Hier Anwender-Code
end.
```

Jedes Delphi-Projekt besteht also aus einem Hauptprogramm, das sich in der Projektdatei befindet. Erstellt man Programme mit einer grafischen Oberfläche, muss diese Projektdatei normalerweise nicht bearbeitet werden. Ansehen kann man sie sich über das Menü „Projekt“ / „Quelltext anzeigen“.

### 3.6.2 Units

Um nicht allen Code in eine einzige Datei schreiben zu müssen und dadurch schnell die Übersicht zu verlieren, gibt es das Konzept der Units. Der Entwickler kann seinen Code, z.B. nach Aufgaben geordnet, in verschiedene Programmmodule aufteilen. Diese Programmmodule (Units) haben die Dateierweiterung `.pas` und folgenden Aufbau:

```
unit <name>;

interface

uses <liste>;

<interface-abschnitt>

implementation

uses <liste>

<implementation-abschnitt>

initialization
<programmcode>

finalization
<programmcode>

end.
```

Solch ein Grundgerüst erhält man, wenn man in Delphi über das Menü Datei Neu/Unit auswählt. Die Abschnitte `initialization` und `finalization` sind optional.

Jede Unit beginnt in der ersten Zeile mit dem Schlüsselwort `unit`. Dahinter folgt der Name der Unit, der nicht von Hand bearbeitet werden darf. Er entspricht dem Dateinamen (ohne die Endung `.pas`) und wird von Delphi beim Speichern der Unit automatisch angepasst.

Nun folgen zwei Schlüsselwörter, die jeweils einen neuen Abschnitt einleiten: der `interface`- und der `implementation`-Teil. Eine Unit endet mit dem Schlüsselwort `end` gefolgt von einem Punkt.

Der Interface-Teil enthält verschiedene Vereinbarungen (so genannte Deklarationen). Diese Deklarationen beschreiben, „was die Unit kann“ bzw. was sie enthält. Der Interface-Teil ist also quasi eine Art Inhaltsverzeichnis für die Unit. Im Implementation-Abschnitt steht dann der eigentliche Inhalt, der Code, der später auch wirklich ausgeführt wird.

Als Beispiel schreiben wir nun eine Unit, die lediglich eine Funktion zur Mehrwertsteuerberechnung enthält:

```
unit Unit1;  
  
interface  
    function Brutto(netto: real): real;  
  
implementation  
  
    function Brutto(netto: real): real;  
    begin  
        result := netto * 1.19;  
    end;  
  
end.
```

Wird einer Anwendung ein neues Fenster hinzugefügt, so gehört dazu immer auch eine Unit. Dagegen kann man zur besseren Strukturierung seines Codes beliebig viele Units einsetzen, die nicht mit einem Fenster in Verbindung stehen.

Beim Kompilieren wird aus jeder .pas-Datei eine .dcu-Datei (Delphi Compiled Unit) erzeugt.

Die Unit System wird automatisch in jede Unit und jedes Hauptprogramm eingebunden, ohne dass sie extra erwähnt wird. Auf alle dort definierten Routinen kann also jederzeit zugegriffen werden.

### 3.6.3 Units verwenden

Mit solch einer Unit alleine kann man nicht viel anfangen. Wir müssen sie in eine Anwendung einbinden. Um die Funktion Brutto jedoch aus einer anderen Unit oder dem Hauptprogramm aufrufen zu können, müssen wir sie dort bekannt machen. Das geschieht über das Schlüsselwort **uses**, das bereits im ersten Beispiel zur Projektdatei oben zu sehen ist. Wichtig ist hierbei, dass jede Unit innerhalb eines Projekts einen eindeutigen Namen haben muss. Man kann nicht in Unit1 eine weitere Unit1 einbinden. Deshalb speichern wir unsere Beispiel-Unit unter dem Namen „beispiel.pas“. Die erste Zeile ändert sich nun automatisch in **unit beispiel**;

Nun legen wir über das Datei-Menü eine neue Anwendung an. In die Unit1 binden wir unsere Unit ein:

```

unit Unit1;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Beispiel;

type
  TForm1 = class(TForm)
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

  {$R *.DFM}

end.

```

Nun können wir in der gesamten Unit unsere Funktion `Brutto` verwenden, als wäre die Funktion in der gleichen Unit implementiert. Gibt es in einer anderen eingebunden Unit eine Funktion/Prozedur gleichen Namens, muss zuerst der Unit-Namen genannt werden, um klarzustellen, welche Funktion gemeint ist, z.B. `ergebnis := Beispiel.Brutto(1500);`

### 3.6.4 Positionen der uses-Klausel

Werden Units eingebunden, die bereits im Interface benötigt werden (z.B. weil in ihnen Typen definiert sind), so werden sie der uses-Klausel im interface-Teil eingefügt. In allen anderen Fällen sollte eine Unit der uses-Klausel des implementation-Abschnitts hinzugefügt werden.

Haben wir z. B. ein Projekt mit zwei Formularen und den dazugehörigen Units `unit1` und `unit2`, und soll `Form2` (in `unit2`) aus `Form1` aufgerufen werden, so braucht die `Unit1` Zugriff auf die `Unit2`. Dazu wechseln wir in die Anzeige von `Unit1` und klicken im Datei-Menü von Delphi auf „Unit verwenden...“. In dem erscheinenden Fenster sind alle Units des Projekts aufgelistet, die von der aktuellen Unit noch nicht verwendet werden. Hier wählen wir „Unit2“ aus und schließen das Fenster. Delphi hat nun automatisch eine Zeile direkt am Anfang des implementation-Abschnitts eingefügt. Folgendes Beispiel zeigt diesen Fall, wobei `Form1` einen Button (`Button1`) enthält, auf dessen Klick `Form2` geöffnet wird:



```

unit Unit1;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms,
  Vcl.Dialogs, Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

uses Unit2;

  {$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;

end.

```

Die Verwendung der zweiten uses-Klausel hat ihren Grund darin, dass zwei Units sich nicht gegenseitig im uses-Abschnitt des Interface einbinden können (zirkuläre Unit-Referenz). Im Implementation-Teil ist dies jedoch möglich.

Allgemein gilt, dass Units, deren Routinen nur für den implementation-Abschnitt benötigt werden, auch im implementation-Abschnitt eingebunden werden. Wird dagegen ein Teil einer anderen Unit (z.B. ein selbstdefinierter Datentyp) bereits im interface-Abschnitt benötigt, muss die Unit natürlich schon dort eingebunden werden.

### 3.6.5 interface und implementation

Der interface-Abschnitt (zu deutsch „Schnittstelle“) dient dazu, Funktionen, Prozeduren, Typen usw. dieser Unit anderen Units zur Verfügung zu stellen. Alles, was hier steht, kann von außen verwendet

werden. Bei Prozeduren und Funktionen steht hier nur der Kopf der Routine (Name, Parameter und evtl. Rückgabewert). Die Definition folgt dann im implementation-Abschnitt.

Der interface-Abschnitt endet mit Beginn des implementation-Abschnitts.

### 3.6.6 initialization und finalization

Bei Bedarf können am Ende einer Unit noch zwei Abschnitte stehen: `initialization` und `finalization`.

Initialization muss dabei als erstes aufgeführt werden. Hier werden alle Befehle aufgeführt, die bei Programmstart der Reihe nach ausgeführt werden sollen. Danach folgt das Ende der Unit (`end.`) oder der `finalization`-Abschnitt. Dieser ist das Gegenstück zu `initialization`. Hier können z. B. vor Programmende Objekte freigegeben werden.

Der Aufbau einer Units sieht dann so aus:

```
unit Unit1;

interface

implementation

initialization

finalization

end.
```

`finalization` kann nur verwendet werden, wenn es auch einen `initialization`-Abschnitt gibt; `initialization` kann jedoch auch ohne `finalization` vorkommen. Beide Abschnitte werden eher selten verwendet. Units funktionieren auch ohne sie. Bei Verwendung von Klassen (siehe folgendes Kapitel) kommen Konstruktor und Destruktor zum Einsatz, um Code zum Initialisieren und Freigeben ausgeführt werden soll.

## 3.7 Objektorientierung

In diesem Kapitel wollen wir uns anschauen, was objektorientierte Programmierung ist. Ohne es zu wissen, hatten wir bereits in obigem Beispiel damit zu tun, denn jede Komponente, die in der Toolbar zu finden ist, ist ein Objekt. Auch das Fenster selbst.

### 3.7.1 Klassen, Objekte und Instanzen

Eine *Klasse* ist der Bauplan eines Objekts, sozusagen ein komplexer Datentyp, der auch eigene Funktionalität enthält. Nehmen wir als Beispiel uns Menschen. Eine Klasse „Mensch“ würde beschreiben, welche Eigenschaften ein Mensch haben kann (Größe, Gewicht, Haarfarbe usw.) und welche Aktionen er ausführen kann (z.B. schlafen, sprechen, gehen).

Ein *Objekt*, auch *Instanz* genannt, ist ein konkretes „Ding“. In unserem Beispiel bist du, deine Eltern und jeder, der dir sonst so auf der Straße begegnet, eine Instanz der Klasse „Mensch“. Alle folgen dem

in der Klasse definierten Bauplan. Alle haben sie eine Größe, ein Gewicht und eine Haarfarbe. Jede Instanz kann hier natürlich unterschiedliche Werte haben, schließlich sind nicht alle Menschen gleich groß und schwer.

Weiteres Beispiel sind die Eingabefelder in unserer Anwendung: Sie folgen alle dem Bauplan der Klasse „TEdit“. D.h. sie haben gleiche Eigenschaften und gleiches Verhalten. Von diesem Bauplan kann es beliebig viele Instanzen geben, die alle unterschiedliche Ausprägungen haben können. Jedes Eingabefeld hat einen anderen Namen, einen anderen Inhalt, eine andere Position innerhalb des Fensters usw. Das alles sind Eigenschaften eines TEdit (und noch viel mehr).

Wenn man etwas programmieren will, überlegt man sich vorher, welche Dinge aus der Realität in dem Programm abgebildet werden sollen, die jeweils eigene Eigenschaften und eigenes Verhalten haben sollen. Diese implementiert man dann als Klassen. Würde man eine Anwendung für ein Einwohnermeldeamt schreiben, so wäre „Mensch“ tatsächlich eine Klasse, die man gebrauchen könnte.

Das klingt etwas abstrakt, deshalb schauen wir uns das an einem konkreten Beispiel an.

### 3.7.2 Schach!

Du kennst sicher das Spiel Schach. Wir wollen hier kein komplettes Schachspiel programmieren, weil das nicht ganz einfach ist, besonders wenn ein Spieler durch künstliche Intelligenz gesteuert werden soll. Unser Augenmerk liegt auf der objektorientierten Modellierung.

Früher, zu Zeiten der prozeduralen Programmierung z.B. mit Turbo Pascal, hätte man die Position der Figuren auf dem Schachbrett z.B. in einem Array gehalten. Eine lange Liste an Prozeduren hätte dann bei jedem Zug geprüft, ob eine Schachfigur von ihrer aktuellen Position auf die neue verschoben werden darf und ob dort bereits jemand steht.

Jetzt stehen uns Klassen zur Verfügung. Dadurch können wir das Ganze viel anschaulicher programmieren. Welche Klassen brauchen wir denn für das Schachspiel?

Als erstes fällt einem vermutlich das Schachbrett ein. Wofür soll die Klasse „Schachbrett“ denn zuständig sein? Sie muss wissen, welche Figur gerade auf welchem Feld steht. Außerdem muss man ihr sagen können, dass Figuren auf neue Positionen verschoben werden sollen.

Und dann gibt es natürlich die Schachfiguren. Eine Schachfigur muss wissen, wer sie ist, sie muss sich auf dem Schachbrett bewegen können, und dafür muss sie die Regel kennen, in welche Richtung sie überhaupt gehen darf. Diese Dinge muss jede Figur können, egal, ob es sich um einen Bauer, einen Läufer oder einen Turm handelt.

Wir erstellen also eine Klasse „Figur“, die all das kann und weiß, was wir gerade festgestellt haben. In Delphi legt man dafür eine neue Unit an (Datei – Neu – Unit). Anschließend schreibt man die Definition der Klasse in das Interface der Unit:

```

type
  XKoordinate = 'a'..'h';
  YKoordinate = 1..8;
  TFigur = class
    private
      FX: XKoordinate;
      FY: YKoordinate;
    public
      procedure Ziehe(x: XKoordinate; y: YKoordinate);
      function IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean;
  end;

```

Hier gibt es nun einiges zu sehen: Eine Klassen-Deklaration besteht aus dem Namen der Klasse (hier TFigur, in Delphi gibt es die Konvention, dass Klassennamen mit einem großen T für type beginnen) und dem Schlüsselwort „class“. Es folgen verschiedene Sichtbarkeitsbereiche, die optional sind (hier „private“ und „public“). Wenn man sie nicht braucht, kann man sie weglassen.

Bei „XKoordinate“ und „YKoordinate“ handelt es sich um Teilbereichstypen, wie in Kapitel 3.4.2 beschrieben.

### 3.7.3 Sichtbarkeiten

Was hat es nun mit diesen Sichtbarkeitsbereichen auf sich? Prinzipiell geht es darum, festzulegen, welche Methoden oder Felder von wem gesehen werden können. Im Sinne eines guten Programmierstils sollte eine Klasse nach außen (also anderen Klassen desselben Programms) nur das zeigen, was diese aufrufen können sollen. Enthält die Klasse weitere Methoden, die nur für interne Zwecke benötigt werden, muss das niemand außerhalb wissen. Felder, also die Variablen einer Klasse, sollten prinzipiell nicht von außen verändert werden können, sondern nur über den Aufruf von Methoden. In unserem obigen Beispiel werden X und Y nur dadurch verändert, dass jemand von außen die Methode „Ziehe“ aufruft, welche die Zielkoordinaten als Parameter mitbekommt.

Die folgenden Sichtbarkeitsattribute stehen zur Auswahl:

Sichtbarkeitsattribut	Beschreibung
private	Ein private-Element kann nur <i>innerhalb der gleichen Unit</i> verwendet werden. Aus anderen Units ist ein Zugriff nicht möglich.
protected	Ein protected-Element ist wie ein private-Element <i>innerhalb der gleichen Unit</i> verwendbar. Darüber hinaus haben alle abgeleiteten Klassen darauf Zugriff, unabhängig davon, in welcher Unit sie sich befinden.
public	public-Elemente unterliegen keinen Zugriffsbeschränkungen.
published	published-Elemente haben dieselbe Sichtbarkeit wie public-Elemente. Zusätzlich werden diese Element im Objektinspektor angezeigt, weshalb nicht alle Typen als published-Element eingesetzt werden können.

<code>strict private</code>	<code>strict private</code> -Elemente sind nur <i>innerhalb der Klasse</i> sichtbar. Andere Klassen können nicht darauf zugreifen, auch wenn sie sich in derselben Unit befinden.
<code>strict protected</code>	<code>strict protected</code> -Elemente sind <i>innerhalb der Klasse</i> und in allen davon abgeleiteten Klassen sichtbar (egal, in welcher Unit sie sich befinden), jedoch nicht in anderen (nicht verwandten) Klassen derselben Unit.

### 3.7.4 Instanz erzeugen

Instanzen sind nun Objekte, die nach dem Bauplan einer Klasse erstellt wurden. Sie belegen bei der Programmausführung Arbeitsspeicher und können Daten aufnehmen. Von jeder Klasse kann es – wie oben bereits erwähnt – beliebig viele Instanzen geben, die alle gleich aufgebaut sind, aber unterschiedliche Daten enthalten können.

Um nun eine Instanz von oben beschriebener Klasse `TFigur` zu erstellen, muss zunächst eine Variable deklariert werden:

```
var  
  Bauer: TFigur;
```

Variablen, deren Typ eine Klasse ist (wie oben „Bauer“) heißen Objektreferenz. Die Werte von Objektreferenzen sind Zeigerwerte (Adressen im Hauptspeicher). Das Deklarieren einer Objektreferenz wie oben reicht jedoch nicht aus, um eine Instanz zu erzeugen. Denn durch die reine Deklaration enthält „Bauer“ nun den Wert `nil` – einen Zeiger ins Nirgendwo. Es ist also noch kein Bereich im Hauptspeicher für unsere Figurinstanz reserviert worden. Wir haben lediglich mit dem Compiler vereinbart, dass es sich um etwas, das dem Bauplan von `TFigur` entspricht, handelt, wenn wir die Variable `Bauer` verwenden.

Die Erzeugung der Instanz geschieht über den Aufruf des Konstruktors „Create“:

```
Bauer := TFigur.Create;
```

Wird ein Objekt nicht mehr benötigt, muss es freigegeben werden, da es ansonsten bis zum Programmende Speicher belegt. Für die Freigabe wird „Free“ aufgerufen. Damit die Freigabe auf jeden Fall erfolgt, auch wenn es zu Fehlern kommt, wird häufig folgendes Konstrukt eingesetzt:

```
Bauer := TFigur.Create;  
try  
  ...  
finally  
  Bauer.Free;  
end;
```

Was das genau bedeutet, werden wir uns noch im Kapitel über Exceptions anschauen. Das „try“ steht übrigens immer erst *nach* dem Aufruf von „Create“. Denn wenn bei „Create“ etwas schief geht, enthält die Variable „Bauer“ keine Objektreferenz, auf der man die Methode „Free“ aufrufen könnte.

### 3.7.5 Elemente einer Klasse

In den Sichtbarkeitsabschnitten können verschiedene Deklarationen stehen. Die verschiedenen Möglichkeiten wollen wir uns hier ansehen.

#### 3.7.5.1 Methoden

Methode ist der Überbegriff für Prozeduren und Funktionen, die Teil einer Klasse sind. Die Methoden stellen das Verhalten eines Objekts dar.

Methoden werden mit ihrem Kopf innerhalb der Klasse deklariert. An einer späteren Stelle im Code folgt die Implementierung der Methode. Diese erfolgt wie bei einer normalen Prozedur oder Funktion, außer dass vor den Methodennamen der Name der Klasse – getrennt durch einen Punkt – geschrieben wird.

In obigem Beispiel haben wir die Methoden-Signaturen „Ziehe“ und „IstZugErlaubt“ gesehen.

Innerhalb der Unit folgt im Implementation-Abschnitt die Implementierung der Methoden. In unserem Beispiel hätten diese folgenden Aufbau:

```
procedure TFigur.Ziehe(x: XKoordinate; y: YKoordinate);  
begin  
    ...  
end;  
  
function TFigur.IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean;  
begin  
    ...  
end;
```

Im Prinzip funktioniert das also wie bei Funktionen und Prozeduren. Allerdings steht vor dem Methodennamen noch der Name der Klasse, zu der die Methode gehört, damit es im Code nicht zu Missverständnissen kommt.

Der Aufruf einer Methode erfolgt dann am Namen einer Instanz der Klasse (also nach dem Aufruf von Create):

```
var Bauer: TFigur;  
...  
Bauer := TFigur.Create;  
Bauer.Ziehe('a', 5);
```

#### 3.7.5.2 Attribute oder Felder

Attribute (auch Felder genannt) sind Variablen, die zu einem Objekt gehören und damit seinen Zustand beschreiben. Attributnamen beginnen in Delphi mit einem großen F (Field). Wie auch beim T vor Klassennamen handelt es sich hierbei um eine Vereinbarung.

Es gehört zum „guten Ton“, Felder immer im private-Teil einer Klasse zu deklarieren, weil sie den internen Zustand eines Objekts enthalten, der von außen nur über definierte Schnittstellen

(Methoden) verändert werden können sollte. Prinzipiell ist aber auch eine andere Position innerhalb der Klasse möglich. Jedoch müssen die Felddeklarationen vor den Methodendeklarationen stehen.

Felder sind in unserem Beispiel von TFigur „FX“ und „FY“ – das F steht für Field, und X und Y sind die Koordinaten des Schachspielfelds.

### 3.7.5.3 *Eigenschaften oder Properties*

Eigenschaften sind keine eigenständigen Variablen, sie belegen zur Laufzeit keinen Speicherplatz. Über sie lassen sich Lese- und Schreibzugriffe auf Attribute regeln. Die Eigenschaften sind es auch, die im Objektinspektor angezeigt werden.

Eigenschaften werden sinnvollerweise im public- oder published-Abschnitt einer Klasse definiert:

```
type
  TFigur = class
    private
      FX: Integer;
      FY: Integer;
    public
      procedure Ziehe(x: XKoordinate; y: YKoordinate);
      function IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean;
      property X: XKoordinate read FX;
  end;
```

Eine Eigenschaft verfügt über eine read- oder eine write-Angabe oder über beide. Dahinter folgt dann entweder direkt der Name des Feldes, auf das lesend oder schreibend zugegriffen werden soll, oder der Name einer Methode, die den Zugriff steuern soll.

Wird für read eine Methode verwendet, darf diese keinen Parameter haben und muss einen Rückgabewert liefern, der dem Typ der Eigenschaft entspricht. Bei einem Feld muss dieses natürlich auch den Typ der Eigenschaft haben. Beispiel:

```
type
  TFigur = class
    private
      FX: Integer;
      FY: Integer;
    public
      function LiesXKoordinate: XKoordinate;
      property X: XKoordinate read LiesXKoordinate;
  end;
```

Eine Methode für write muss genau einen Aufrufparameter besitzen, der ebenfalls dem Typ der Eigenschaft entspricht. Beispiel:

```
type
  TFigur = class
    private
      FX: Integer;
      FY: Integer;
    public
      procedure SetzeXKoordinate(x: XKoordinate);
      property X: XKoordinate write SetzeXKoordinate;
  end;
```

Hinter read und write selbst wird immer nur der reine Methodenname ohne Parameter oder Rückgabewert angegeben.

Enthält eine Eigenschaft nur eine read-Angabe, kann sie nur gelesen werden; enthält sie nur eine write-Angabe, kann sie nur geschrieben werden.

Properties können nicht als var-Parameter eingesetzt werden, da es sich bei ihnen ja nicht um direkte Zugriffe auf Felder handeln muss; sondern es können ja „nur“ Methoden dahinterstecken.

#### 3.7.5.4 Objekte erzeugen: Der Konstruktor

Instanzen einer Klasse entstehen erst dadurch, dass sie ausdrücklich erzeugt werden. Dies geschieht mit dem Konstruktor „Create“. „Konstruktor“ bezeichnet jemanden, der etwas konstruiert, also erzeugt. Man könnte „Create“ auch eine Methode nennen; es gibt jedoch einen großen Unterschied: Methoden können nur aufgerufen werden, wenn die Instanz bereits existiert. Ausnahme davon sind Klassenmethoden. Der Konstruktor dagegen erzeugt die Instanz aus dem Nichts.

Der Konstruktor muss nicht von uns programmiert werden. Der Standard-Konstruktor „Create“ ist automatisch immer vorhanden.

So wird eine Instanz erzeugt:

Objektreferenz := Klasse.Create;

Damit wird Speicherplatz für alle Attribute der Klasse im Hauptspeicher reserviert und die zugehörige Adresse in der Variablen „Objektreferenz“ gespeichert.

Beim Erzeugen einer Klasse werden die jeweils vorhandenen Attribute mit folgenden Startwerten belegt:

- Alle Datenfelder mit einem ganzzahligen Datentyp (z.B. Integer) werden mit 0 initialisiert.
- Alle Datenfelder mit einem String-Typ werden durch eine leere Zeichenkette initialisiert.
- Alle Datenfelder mit einem Zeigertyp werden mit dem Wert nil initialisiert.
- Alle Datenfelder mit einem Gleitkommazahlentyp (z.B. Real) werden mit 0.0 initialisiert.

„Create“ kann übrigens auch an einer Objektreferenz aufgerufen werden. Dadurch wird dann keine neue Instanz erzeugt, sondern die bereits initialisierten Felder des Objekts mit neuen Werten



überschrieben, was zu Speicherlecks führen kann. Will man ein Objekt neu initialisieren, sollte man eine eigene Methode für diese Aktion bereitstellen.

### 3.7.5.5 Eigener Konstruktor

Es ist möglich, für jede Klasse einen eigenen Konstruktor sowie einen eigenen Destruktor, der das Gegenstück zum Konstruktor ist und dessen reservierten Speicher wieder freigibt, zu schreiben. Besonders bei Konstruktoren kann das sinnvoll sein, weil der von TObject geerbte Konstruktor „Create“ nicht unbedingt das ausführt, was man sich wünscht.

Eigene Konstruktoren können ebenfalls „Create“ heißen, wodurch das ursprüngliche „Create“ verdeckt wird – sie können aber auch ganz andere Namen haben, wobei man sich an den Namen „Create“ halten sollte, um den Code einheitlicher und somit lesbarer zu machen.

Damit trotzdem auch der Code des verdeckten Konstruktors ausgeführt wird, muss zu *Beginn* des eigenen Konstruktors „**inherited**“ aufgerufen werden:

```
constructor TMyClass.Create;  
begin  
    inherited; // Ruft den Konstruktor der Oberklasse auf  
               // Hier Werte initialisieren o.ä.  
end;
```

Kleiner Einschub zu „inherited“: Heißt die Methode der Oberklasse genauso und erwartet keine Parameter, so genügt das Schlüsselwort inherited. Unterscheidet sich die Methode der Oberklasse im Namen oder den erwarteten Parametern, so muss hinter inherited ihr Name und ihre Parameter angegeben werden, z.B.:

```
inherited MyCreate(1);
```

Was es genau mit Oberklassen auf sich hat, wird im folgenden Kapitel zum Thema Vererbung erklärt.

Das Kennzeichen eines Konstruktors ist, dass er an einer Klasse (nicht an einer Instanz) aufgerufen wird, wodurch eine Instanz erzeugt wird (Reservierung von Speicher usw.). Der Aufruf normaler Methoden, die mit procedure oder function beginnen, ist erst möglich, wenn eine Instanz existiert. Um dies zu unterscheiden, beginnt die Deklaration eines Konstruktors mit dem Schlüsselwort „constructor“:

```

type
  TFigur = class
    private
      FX: Integer;
      FY: Integer;
    public
      procedure Ziehe(x: XKoordinate; y: YKoordinate);
      function IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean;
      constructor Create(x: XKoordinate; y: YKoordinate);
  end;

```

### 3.7.5.6 Objekte freigeben: Der Destruktor

Wird eine Instanz einer Klasse nicht mehr benötigt, sollte der dafür verwendete Hauptspeicher wieder freigegeben werden. Dies geschieht mit dem Gegenstück zum Konstruktor, dem Destruktor („Zerstörer“). Dieser heißt „Destroy“. Um ein Objekt freizugeben, sollte jedoch die Methode Free verwendet werden. Free prüft, ob ein Verweis auf nil vorliegt, anschließend wird Destroy aufgerufen. Die Objektreferenz verweist jetzt allerdings nicht unbedingt auf nil. Soll das der Fall sein, kann auf die Prozedur FreeAndNil (Unit SysUtils) zurückgegriffen werden, die das freizugebende Objekt als Parameter erwartet – es handelt sich hier nicht um eine Methode.

Destroy verdeckt (wie Create beim Konstruktor) die gleichnamige Methode der Oberklasse. Entsprechend sollte auch hier mit Hilfe von **inherited** deren Destruktor aufgerufen werden, allerdings *erst am Ende* des Destruktors, nachdem man z.B. Ressourcen der eigenen Klasse freigegeben hat:

```

destructor TMyClass.Destroy; override;
begin
  // Hier Ressourcen freigeben
  inherited; // Ruft den Destruktor der Oberklasse auf
end;

```

Da die Methode Destroy in der Basisklasse TObject als virtual deklariert ist, muss in abgeleiteten Klassen das Schlüsselwort „override“ zum Einsatz kommen. Details zu virtual und override in den folgenden Kapiteln.

## 3.7.6 Die Erben einer Figur

### 3.7.6.1 Vererbung - Alles erbt von TObject

Zurück zu unserem Beispiel der Schachfiguren. Wir haben jetzt eine Klasse TFigur, die u.a. eine Methode IstZugErlaubt enthält. Doch wie sollen wir diese implementieren? Wir wissen ja gar nicht, ob es sich um einen Bauer, einen Springer oder einen Turm handelt! Diese Methode muss also für jede Art von Figur unterschiedlich implementiert werden, während der Rest für alle gleich ist.

Zu früheren Zeiten hätte man das über eine lange if-Unterscheidung abgehandelt. Wir wollen aber objektorientiert an die Sache herangehen und nutzen deshalb die sog. „Vererbung“.

In der Programmierung ist der Begriff „Vererbung“ mehr im Sinne der Biologie zu verstehen als im Sinne der Justiz. Schließlich wird hier nicht der Nachlass einer verstorbenen Klasse verteilt. Sondern

es geht darum, dass Unterklassen die gleichen Eigenschaften und Fähigkeiten wie ihre Eltern (Oberklasse) haben. Einschränkung hier: Klassen können nur genau einen Vorfahren haben, nicht mehrere, da es sonst zu Konflikten kommen könnte. Deshalb ist die Vererbung in der Programmierung auch wesentlich einfacher als in der Biologie. Man muss z.B. nicht wissen, wer Gregor Mendel ist. Der Stammvater aller Klassen heißt in Delphi „TObject“. Das erklärt auch, weshalb jede Klasse z.B. „Create“ und „Free“ kennt, auch wenn wir es gar nicht implementiert haben. Der Code dafür steckt in TObject.

Nun erstellen wir neue Klassen, für jeden Figurentyp eine, also Bauer, Springer, Läufer, Turm, Dame und König. Alle sollen von der Klasse TFigur erben. Das erreicht man, indem man hinter „class“ in Klammern den Namen der Oberklasse angibt. Wird bei einer Klassendefinition kein Name einer Oberklasse angegeben, verwendet Delphi automatisch TObject als Vorfahr.

### 3.7.6.2 Abstrakte Methoden

Weil zwar alle Unterklassen von TFigur (also z.B. TBauer oder TSpringer) die Methode IstZugErlaubt haben sollen, wir sie aber in TFigur nicht implementieren können, kennzeichnen wir die Methode mit Hilfe der Schlüsselwörter „**virtual**“ und „**abstract**“ als abstrakt. „virtual“ bedeutet, dass die Methode überschrieben werden darf, „abstract“, dass in der aktuellen Klasse keine Implementierung der Methode vorhanden ist. Die Methode ist also nur noch ein Platzhalter, um sicherzustellen, dass Unterklassen die Methode implementieren. Delphi hat nun nichts mehr dagegen, dass die Methode keine Implementierung hat. Allerdings dürfen wir nun keine Instanz der Klasse TFigur erzeugen, weil es ja nicht wüsste, wie es sich beim Aufruf von IstZugErlaubt verhalten sollte. Damit ist die Klasse TFigur abstrakt. Sie enthält alle Gemeinsamkeiten ihrer Unterklassen, kann aber selbst nicht über „Create“ instanziiert werden.

Wir erstellen nun selbst eine Unterklasse. Hier am Beispiel des Turms der Code für TFigur und TTurm:  
figur.pas:

```
unit figur;  
  
interface  
  
type  
  TFigur = class  
    private  
      FX: XKoordinate;  
      FY: YKoordinate;  
    public  
      procedure Ziehe(x: XKoordinate; y: YKoordinate);  
      function IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean; virtual;  
abstract;  
      constructor Create(x: XKoordinate; y: YKoordinate);  
    end;  
  
  implementation  
  
  constructor TFigur.Create(x: XKoordinate; y: YKoordinate);  
  begin  
    inherited;  
    FX := x;  
    FY := y;  
  end;  
  
  procedure TFigur.Ziehe(x: XKoordinate; y: YKoordinate);  
  begin  
    if IstZugErlaubt(x, y) then  
    begin  
      FX := x;  
      FY := y;  
    end;  
  end;  
  
end.
```

turm.pas:

```

unit turm;

interface

uses figur;

type
  TTurm = class(TFigur)
  end;

implementation

end.

```

Im Code der Unit turm sind zwei Dinge zu sehen: Wir erben von der Klasse „TFigur“ (deshalb die Angabe von „TFigur“ in Klammern hinter dem Schlüsselwort „class“). Und damit die Bezeichnung „TFigur“ an dieser Stelle überhaupt bekannt ist, müssen wir die Unit „figur“, in der die Klasse „TFigur“ implementiert ist, zuvor mit „uses“ einbinden.

Nun haben wir also eine Unterklasse zu TFigur erstellt. Diese Klasse kann alles, was ihre Oberklasse auch kann. Nun müssten wir noch die abstrakte Methode „IstZugErlaubt“ konkretisieren, denn die Sprungregeln für einen Turm sind ja bekannt. Aber vorher fragen wir den Delphi-Compiler, was er von unserem Werk hält. Ein Druck auf Strg+F9 (oder Menü Projekt – Compilieren) zeigt: Der Compiler ist zufrieden. Kann das sein, wo wir doch die abstrakte Methode nirgends implementiert haben?

Wir setzen testweise einen Button auf das Hauptfenster der Anwendung und lassen uns im OnClick-Ereignis eine Turm-Instanz erzeugen, an der wir die abstrakte Methode aufrufen:

```

procedure TForm1.Button1Click(Sender: TObject);
var turm: TTurm;
begin
  turm := TTurm.Create('a', 1);
  try
    if turm.IstZugErlaubt('e', 5) then
      ShowMessage('Sprung ist erlaubt');
  finally
    turm.Free;
  end;
end;

```

Ein Blick in das Meldungsfenster am unteren Rand der Entwicklungsumgebung zeigt uns jetzt nach dem Kompilieren folgende Warnung:

[dcc32 warnung] Unit1.pas(58): w1020 Instanz von 'TTurm' mit der abstrakten Methode 'TFigur.IstZugErlaubt' wird angelegt

Delphi ist also so großzügig, uns auch Instanzen von Klassen mit abstrakten Methoden erzeugen zu lassen. Solange wir diese Methoden nicht aufrufen, ist das auch kein Problem. Aber was passiert, wenn wir diese doch aufrufen?

Wir probieren es einfach aus, indem wir obigen Code ausführen.

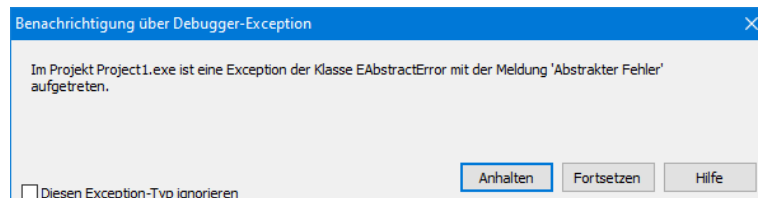


Abbildung 27: Anzeige einer Exception in der IDE

Eigentlich ist hier nicht der Fehler abstrakt, sondern der aufgerufene Code. Denn der Fehler ist ganz klar: Wir sollten „IstZugErlaubt“ endlich implementieren.

turm.pas würde dann z.B. so aussehen:

```
unit turm;

interface

uses figur;

type
  TTurm = class(TFigur)
  public
    function IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean;
  end;

implementation

function TTurm.IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean;
begin
  result := (x = FX) or (y = FY);
end;

end.
```

Was ist hier zusehen? Wir deklarieren die Methode „IstZugErlaubt“ exakt so wie in der Klasse TFigur – allerdings verzichten wir auf die Schlüsselwörter „virtual“ und „abstract“. Im „implementation“-Teil folgt dann die konkrete Implementierung. Zur Erinnerung: FX enthält in der Klasse TFigur die X-Koordinate der aktuellen Position der Figur, FY die Y-Koordinate. Wenn die neue Position (übergeben als Parameter „x“ und „y“) in der gleichen Zeile oder Spalte liegt, dann darf der Turm sich bewegen; der Rückgabewert der Funktion ist also „true“.

Allerdings funktioniert das so nicht ganz. Warum? FX und FY hatten wir in „TFigur“ private gemacht. Nun kommen die Sichtbarkeitsregeln zum Tragen. Private-Felder sind in Unterklassen nicht bekannt. Deshalb ändern wir das Wörtchen „private“ in der Klasse „TFigur“ in „protected“ – und schon haben auch alle Unterklassen Zugriff darauf.

### 3.7.7 Überschreiben von Methoden

Der Compiler sollte nun mit unserem Code zufrieden sein. Ein Blick in das Meldungsfenster zeigt allerdings eine neue Warnung:

Methode „IstZugErlaubt“ verbirgt virtuelle Methode vom Basistyp „TFigur“

Was hat das denn zu bedeuten? Nun, Delphi weist uns darauf hin, dass wir in TTurm die Methode „IstZugErlaubt“ implementiert haben, die exakt der virtuell-abstrakten Methode in TFigur entspricht. Der Compiler will nun wissen, ob das Absicht von uns war oder ob wir zufällig den gleichen Namen gewählt haben. Natürlich war das in unserem Fall Absicht. Das müssen wir dem Compiler mitteilen, indem wir das Schlüsselwort „override“ hinter die Methodensignatur schreiben:

```
type
  TTurm = class(TFigur)
  public
    function IstZugErlaubt(x: XKoordinate; y: YKoordinate): Boolean; override;
  end;
```

Mit „override“ werden Methoden einer Oberklasse überschrieben, wenn sie dieselbe Signatur haben (also gleicher Name, gleiche Parameter und gleicher Rückgabotyp).

#### 3.7.7.1 Zugriff auf Methoden der Oberklassen - *inherited*

Möchte man die gleichnamige Methode der Oberklasse aufrufen, so kommt **inherited** zum Einsatz:

```
procedure TTurm.Example;
begin
  inherited;
  DoSomething;
end;
```

In diesem Beispiel wird durch inherited die gleichnamige Methode einer Oberklasse aufgerufen. Soll eine Methode anderen Namens oder mit Parametern aufgerufen werden, muss hinter inherited der Name der Methode gefolgt von den Parametern in Klammern angegeben werden.

Selbstreferenz – self

Möchte man explizit auf Felder oder Methoden der aktuellen Klasse zugreifen, kann das Schlüsselwort „self“ vorangestellt werden. Das ist z.B. dann erforderlich, wenn man eine lokale Variable von einem Feld gleichen Namens unterscheiden möchte:

```

type
  TTurm = class
    x: Integer;
    procedure DoSomething;
  end;

procedure TTurm.DoSomething;
var x: Integer;
begin
  x := self.x;
end;

```

Im Beispiel wird der Wert des Feldes x der lokalen Variablen x zugewiesen. Das funktioniert, macht den Code aber unnötigerweise schwerer verständlich. Am besten sorgt man für eindeutige Namen.

### 3.7.8 Polymorphie – alles nur Figuren

Wir haben nun also eine erste Unterklasse von „TFigur“ erstellt, nämlich „TTurm“. Auf die gleiche Weise können wir für die anderen Spielfiguren des Schachspiels vorgehen.

Zu guter Letzt kommen wir an den Punkt, an dem wir etwas brauchen, das alle Figuren zusammenhält: das Schachbrett. Dafür implementieren wir eine neue Klasse „TSchachbrett“. Als Datenstruktur für die Position der Figuren eignet sich am besten ein zweidimensionales Array. Aber von welchem Typ? Wir können ja nicht einige Felder vom Typ „TTurm“ und andere vom Typ „TBauer“ deklarieren, schließlich kann im Verlauf eines Spiels jeder Figurtyp auf jedem Feld stehen. Aber wir haben ja zum Glück unsere Oberklasse TFigur, von der all unsere Figuren erben. Aus diesem Grund können wir Folgendes schreiben:

```

type
  TSchachbrett = class
  private
    FFelder: array[1..8, 1..8] of TFigur;
  end;

```

Auf dem Schachbrett stehen jedoch keine abstrakten Figuren, sondern konkrete Bauern, Türme usw. Wir schreiben unseren eigenen Konstruktor, in dem wir das Brett füllen. Dazu erzeugen wir Instanzen der jeweiligen Figurtypen und weisen sie einer Stelle in dem zweidimensionalen Array zu. Da wir in TFigur festgelegt haben, dass jeder Figur beim Erzeugen ihre X- und Y-Koordinate übergeben werden muss, werden die gleichen Werte auch im Konstruktor verwendet:

```

constructor TSchachbrett.Create;
begin
  FFelder[1, 1] := TTurm.Create('a', 1);
  FFelder[1, 2] := TBauer.Create('a', 2);
  ...
end;

```

Unser Schachbrett ist nun mit Figuren gefüllt, das Spiel kann also beginnen.



Angenommen, von außen (ein Teil unseres Programms, den wir für dieses Beispiel einfach ignorieren) käme der Befehl, die Figur von Position A1 nach A5 zu verschieben. Unser TSchachbrett kann in seinem Array nachschauen und findet auf Position A1 tatsächlich eine Figur. Da das Array aber vom Typ TFigur ist, weiß es erst einmal nicht, ob es sich um einen Turm oder einen Bauern handelt. Das ist aber auch egal. Wichtig ist, dass alle Figuren die Methode „Ziehe“ kennen. Diese haben wir nämlich in „TFigur“ implementiert und damit auch an die Unterklassen vererbt. Folgender Aufruf ist also möglich:

```
if (FFelder[1, 1] <> nil) then  
  FFelder[1, 1].Ziehe('a', 5);
```

Sicherheitshalber prüfen wir, ob auf dieser Position tatsächlich eine Figur steht oder nicht. „nil“ ist ein Zeiger ins Leere. Alle Felder des Schachbretts, an denen keine Figur steht, wurden mit „nil“ initialisiert.

Die Methode „Ziehe“ selbst ruft „IstZugErlaubt“ auf, was zwar in TFigur nicht deklariert ist, aber von jeder Unterklasse zur Verfügung gestellt wird, was die abstrakte Methode in TFigur verlangt.

Dieses Konzept nennt man Polymorphie, zu Deutsch „Vielgestaltigkeit“. Eine Variable (hier unser Array) wird mit einem bestimmten Typ deklariert (TFigur), enthält zur Laufzeit aber einen anderen Typ (TTurm, TBauer...). Das funktioniert übrigens nur mit Unterklassen des deklarierten Typs.

### 3.7.8.1 Typbestimmung und -umwandlung mit *is* und *as*

Der Aufruf von Methoden, die alle Nachfahren von TFigur kennen, ist also kein Problem. Aber was wäre, wenn der Turm eine zusätzliche Methode anbieten würde, die nur er kennt? In der Realität wäre das z.B. Rochade. Dabei handelt es sich um einen besonderen Schachzug, bei dem König und Turm getauscht werden. (Für alle, die es genau wissen wollen: <http://de.wikipedia.org/wiki/Rochade>) Dann würde folgender Aufruf zum Fehler führen:

```
FFelder[1, 1].Rochade; // Funktioniert nicht!
```

Denn alle Elemente des Arrays sind ja vom Typ TFigur, und TFigur kennt die Methode „Rochade“ nicht.

Wir müssen also prüfen, ob es sich bei der Figur um einen Turm handelt. Das funktioniert in Delphi mit dem Schlüsselwort „is“:

```
if (FFelder[1, 1] is TTurm) then ...
```

Wenn wir uns nun sicher sind, einen Turm in der Hand zu haben, müssen wir dem Compiler noch mitteilen, dass er das Objekt in einen Turm umwandeln soll, damit die Methode „Rochade“ aufgerufen werden kann. Das macht man mit dem Schlüsselwort „as“:

```
(FFelder[1, 1] as TTurm).Rochade;
```

Üblicherweise werden „is“ und „as“ nicht gemeinsam verwendet, sondern mit „is“ geprüft und dann hart gecastet:

```
if (FFelder[1, 1] is TTurm) then  
  (FFelder[1, 1] as TTurm).Rochade;
```

Oder nur mit „as“ gearbeitet, wenn man sich bereits sicher ist, eine bestimmte Klasse zu haben.

### 3.8 Funktions- und Methodenzeiger

Funktionen und Prozeduren können selbst in Variablen gespeichert werden (bzw. ein Verweis auf sie). Somit ist es möglich, einer Funktion eine andere Funktion als Parameter zu übergeben.

```
type  
  TMeineFunktion = function: Integer;
```

In diesem Fall haben die Funktionen oder Prozeduren keinen Namen, sondern bestehen aus dem Schlüsselwort `function` oder `procedure`, bei Bedarf gefolgt von Parametern in runden Klammern und einem Rückgabewert. Handelt es sich um Methoden, folgt noch „of object“:

```
TMeineMethode = procedure of object;
```

Praktischer Nutzen der Sache: Man kann somit eine Ereignisbehandlungsroutine erstellen und erst zur Laufzeit einem Objekt (einer Komponente) zuordnen.

Beispiel:

```
type  
  TMyOnClick = procedure(Sender: TObject) of object;  
  TMyForm = class(TForm)  
    procedure MyButtonClickEvent(Sender: TObject);  
  end;  
  
var  
  MyOnClick: TMyOnClick;  
  MyForm: TMyForm;  
  ...  
  MyOnClick := MyForm.MyButtonClickEvent;
```

Die OnClick-Events der Komponenten haben ebenfalls einen Methodenzeiger als Typ, den gleichen wie TMyOnClick im Beispiel. Deshalb ist es möglich, z.B. einem Button zur Laufzeit eine Methode mit dem gleichen Parameter zuzuweisen. Variablen vom Typ einer Funktion/Prozedur beinhalten die Speicheradresse, also einen Zeiger auf die Routine. Bei Methoden werden zwei Zeiger gespeichert: die Adresse der Methode und eine Referenz auf die Instanz.

### 3.9 Exceptions

Bei der Programmierung kommt man immer wieder in Situationen, dass z.B. eine eigene Methode mit Parametern aufgerufen wird, die nicht sinnvoll sind. In unserem Schachbeispiel wäre das der Fall, wenn eine Figur an eine Position weit außerhalb des Schachbretts verschoben werden sollte. Andere Fälle wären z.B. das Kopieren einer Datei, die überhaupt nicht existiert oder das Dividieren durch Null. Bei all diesen Beispielen handelt es sich um Ausnahmen im Programmablauf.

In der objektorientierten Programmierung können solche Ausnahmen (engl. „Exceptions“) behandelt werden. Dazu werden von der Methode, die mitteilen möchte, dass etwas nicht nach Plan läuft, ein Exception-Objekt erzeugt und an den Aufrufer übermittelt. Dieser kann entweder damit rechnen, dass dieser Fall eintritt, und die Exception fangen – oder er wirft sie weiter an seinen eigenen Aufrufer usw. Wenn letztlich niemand die Exception fängt, wird sie dem Anwender angezeigt.

#### 3.9.1 Exceptions werfen

Im Delphi-Sprachumfang gibt es bereits einige Exception-Klassen, die von der VCL verwendet werden, z.B. `EConvertError`, wenn ein Problem bei Typumwandlungen auftritt, `EDivByZero`, wenn durch Null geteilt wird, `ERegistryException`, wenn der Zugriff auf die Registry nicht wie gewünscht funktioniert. Hieran erkennt man schon die Konvention: Exception-Namen beginnen immer mit einem E.

Als Entwickler sollte man jedoch eher eigene Exception-Klassen verwenden, was im folgenden Abschnitt beschrieben wird.

Um nun in einer Methode eine Exception auszulösen (zu „werfen“), kommt folgender Code zum Einsatz:

```
if zahl2 = 0 then
  raise EInvalidArgument.Create('zahl2 ist 0!');
```

#### 3.9.2 Eigene Exception

Natürlich muss man sich nicht auf die Exceptions beschränken, die es in der VCL bereits gibt. Man kann auch seine eigenen definieren. Dabei ist eine Exception eine ganz normale Klasse, die Exception als Oberklasse hat:

```
type
  EMyException = class(Exception);
```

#### 3.9.3 Exceptions fangen

Was machen wir nun mit einer geworfenen Exception? Natürlich: fangen! Dafür muss der Aufruf einer Methode, von der wir wissen, dass sie bei Problemen eine (oder mehrere) Exceptions werfen kann, mit `try ... except` umgeben. Wird tatsächlich eine Exception geworfen, wird die Ausführung direkt im `except`-Block fortgesetzt. Auch, wenn innerhalb des `try`-Blocks noch weiterer Code gefolgt wäre.

```
try
  ergebnis := teileZahl(zahl1, zahl2);
  zeigeErgebnis(ergebnis);
except
  ShowMessage('Es ist ein Fehler aufgetreten!?');
  ergebnis := 0;
end;
machWas;
```

Was passiert in diesem Beispiel? Es wird versucht, die Funktion „teileZahl“ aufzurufen. Es wird damit gerechnet, dass hier eine Exception geworfen wird. Tritt keine Exception auf, wird „zeigeErgebnis“ aufgerufen und anschließend „machWas“. Der except-Block wird ignoriert. Tritt in „teileZahl“ hingegen eine Exception auf, geht es direkt im except-Block weiter. „zeigeErgebnis“ wird übersprungen und dafür „ShowMessage“ aufgerufen und die Variable „ergebnis“ auf 0 gesetzt. Anschließend geht es auch hier mit „machWas“ weiter.

Eine Methode oder Funktion kann natürlich auch verschiedene Arten von Exceptions werfen. Wenn man unterschiedlich auf diese reagieren will, muss im except-Block der Typ geprüft werden:

```
try
  irgendeineMethode;
except
  on EZeroDivide do value := MAXINT;
  on EMyException do value := 0;
end;
```

### 3.9.4 Exceptions fangen und weiterwerfen

Grundsätzlich sollte man Exceptions nur fangen, wenn man auch sinnvoll darauf reagieren kann. Wird eine Exception nicht mittels try-except gefangen, wird sie an den Aufrufer des Aufrufers weitergegeben usw., bis die Exception auf der obersten Ebene des Programms angekommen ist. Wenn sie auch hier nicht gefangen wird, bekommt sie der Anwender angezeigt.

Manchmal ist es sinnvoll, eine Exception zu fangen, um z.B. bestimmte Variablen zu setzen, trotzdem will man sie an den Aufrufer weiterwerfen. Dies passiert mit dem einfachen Aufruf von „raise“:

```
try
  machwas;
except
  on EMyException do
  begin
    // Setze Variablen
    raise; // wirf die Exception weiter
  end;
end;
```

### 3.9.5 Ressourcenschutzblöcke: try – finally

Häufig kommt es vor, dass man Objekte oder Speicher freigeben möchte, auch wenn eine Exception auftritt. Für diesen Fall gibt es try-finally-Blöcke. Der finally-Block wird auf jeden Fall ausgeführt, auch wenn zwischen try und finally eine Exception geworfen wird. Falls keine Exception aufgetreten ist, wird die Ausführung direkt nach dem finally-Block fortgesetzt; ansonsten geht es mit der Exception-Behandlung in der Aufrufer-Methode weiter. Der finally-Block selbst wird aber in jedem Fall zuerst ausgeführt:

```
var sl: TStringList;
...
sl := TStringList.Create;
try
  // Mache irgendwas mit sl
finally
  sl.Free;
end;
```

Dieses Konstrukt wird auch „Ressourcenschutzblock“ genannt, weil sichergestellt wird, dass Ressourcen (im Beispiel der für die StringList reservierte Speicher) auf jeden Fall wieder freigegeben wird.

Das Erzeugen der StringList bzw. eines beliebigen Objekts steht übrigens vor dem „try“. Denn wenn hier etwas schief geht, ist die Variable „sl“ noch gar nicht initialisiert, so dass sie auch noch nicht wieder freigegeben werden muss. Das würde vielmehr zu einem weiteren Fehler führen.

### 3.10 Dateien

Alle bislang verarbeiteten Daten waren flüchtig. Sie lagen lediglich im Arbeitsspeicher vor und waren spätestens mit dem Ausschalten des Rechners verloren. Wollen wir aber diese Daten sitzungsübergreifend verwenden, so müssen wir sie in Dateiform, auf einem nicht-flüchtigen Speichermedium ablegen, z.B. auf einer Festplatte.

#### 3.10.1 Binär- und Textdateien

Dateien sind eine begrenzte Folge von Bytes, abgelegt auf einem Speichermedium. Betriebssystem und Anwendungssoftware interpretieren diese Folge in einer Form, dass wir zwischen Binär- und Textdateien unterscheiden können.

Textdateien können wir mit Hilfe eines Text-Editors lesen und auch verstehen, da sie lediglich aus darstellbaren Zeichen und einigen wenigen Steuercodes, z.B. dem Zeilenumbruch bestehen.

In Binärdateien kann jeder beliebige Wert eines Bytes auftauchen und somit auch Zeichen, für deren Darstellung wir keine Interpretation haben.

#### 3.10.2 Dateiname und -pfad

Um eine Datei auf der Festplatte anzusprechen, benötigen wir ihren Dateinamen und den Ort, an dem sie in der Verzeichnisstruktur zu finden ist. Vergeben wir den Dateinamen selbst, dann wählen wir einen Namen, welcher Aufschluss über Art und Inhalt der Datei gibt.

Wollen wir nun auf eine solche Datei zugreifen, so muss sie an einem Ort liegen, an dem wir die benötigten Zugriffsrechte haben. Diese Schreib- und Leserechte sind für normale Benutzer in einem Windows-Dateisystem eingeschränkt.

Um eine Datei selbst abzuspeichern bietet sich deshalb das persönliche Profilverzeichnis des angemeldeten Benutzers an, denn dort hat er volle Zugriffsrechte. Je nach Windows-Version und Installation beschreibt ein angepasstes FN, in der Folge, eine solche Datei.

```
const
  FN = 'C:\Dokumente und Einstellungen\<Benutzername>\<Dateiname>';
  //bzw. FN = 'C:\Users\<Benutzername>\<Dateiname>';
```

#### 3.10.3 Relative und absolute Pfade

Wir sehen, dass der Dateiname hier im Bezug zum Ursprung der Partition angegeben wird. Da diese absolute Pfadangabe keine Veränderliche aufweist, ist die Datei immer an dieser Stelle zu finden. Die Alternative wäre ein Dateiname ohne Pfadangabe:

```
const
  FN = '<Dateiname>';
```

Dies stellt die Datei in Relation zum gerade aktuellen Arbeitsverzeichnis. Da sich dieses aber durch verschiedene Aktionen ändern und auch von Programmlauf zu Programmlauf unterschiedlich sein kann, verzichten wir hier ganz bewusst auf eine relative Pfadangabe.

### 3.10.4 Die Dateitypen

Wir unterscheiden drei Dateitypen: typisierte und untypisierte Dateien, sowie Textdateien:

```
var
  UntypisierteDatei: file;
  TypisierteDatei: file of "Datentyp";
  TextDatei: TextFile;
```

Der untypisierten Datei liegt, wie der Name schon sagt, kein konsistenter Datentyp zugrunde oder er ist uns unbekannt. Deklariert werden solche Dateien durch das Schlüsselwort `file`.

Bei typisierten Dateien folgt hinter `file` noch die Angabe des Datentyps, getrennt durch das Schlüsselwort `of`. Dieser Datentyp entspricht den Standardtypen oder einer Zusammensetzung davon, in Form eines Records. Hier dürfen wir nichts verwenden, was intern durch einen Zeiger angesprochen wird. Somit keine dynamischen Arrays, Strings oder Instanzen von Klassen beispielsweise. Die Größe eines Datensatzes steht bereits zum Zeitpunkt der Kompilierung fest.

#### 3.10.4.1 Öffnen und Schließen der Datei

Egal welchen Dateityp wir ansprechen wollen, die folgenden Mechanismen sind für alle, in leicht abgewandelter Form, identisch:

Zunächst verbinden wir mittels `AssignFile` unsere Dateivariablen mit dem voll qualifizierten Dateinamen. Über die jetzt initialisierte Dateivariablen laufen alle Zugriffe auf die Datei; der Dateiname wird fortan nicht mehr benötigt.

Der Gültigkeitsbereich für Öffnen und Schließen einer Datei sollte bei beiden übereinstimmen. Spätestens jedoch am Ende des Programms muss die Datei mit `CloseFile` wieder geschlossen werden, da sonst Datenverlust droht. Schreibvorgänge werden mitunter gepuffert, so dass erst dann auf die Festplatte geschrieben wird wenn genügend Daten vorhanden sind, das Entpuffern angestoßen wird oder die Datei durch Schließen in einen konsistenten Zustand versetzt wird.

```

program LeereUntypisierteDatei;

{$APPTYPE CONSOLE}

uses
  System.SysUtils;

const
  FN = 'C:\Users\user\Desktop\UntypisierteDatei.dat';
  //hier „user“ mit dem gültigen Windows-User-Namen ersetzen
var
  UntypisierteDatei: file;
begin
  AssignFile(UntypisierteDatei, FN);
  Rewrite(UntypisierteDatei);
  CloseFile(UntypisierteDatei);
end.

```

Passen wir den Dateinamen in dieser Konsolenanwendung an und lassen das Programm laufen, so sehen wir nur ein kurzes Aufflackern der Konsole. Es wird lediglich eine leere Datei durch **Rewrite** erzeugt und danach beendet sich das Programm. Existiert die Datei bereits vorher, dann wird sie gelöscht und neu erstellt.

Die beiden anderen Möglichkeiten eine Datei zu öffnen, sind **Reset** und **Append**. **Reset** öffnet eine vorhandene Datei, mit dem in der globalen Variable **FileMode** angegebenen Zugriffsmodus:

Modus	Wert	Bedeutung
fmOpenRead	0	Öffnet eine Datei zum Lesen
fmOpenWrite	1	Öffnet eine Datei zum Schreiben
fmOpenReadWrite	2	Öffnet eine Datei zum Lesen und Schreiben

Voreingestellt, aber veränderbar, ist hier 2.

Für Textdateien hat **FileMode** keine Bewandnis, denn **Reset** öffnet Textdateien immer zum Lesen. Um aber auch bestehenden Textdateien Text hinzuzufügen, gibt es **Append**. Damit öffnen wir eine Datei, um Text anzuhängen.

Bei untypisierten Dateien besitzen **Rewrite** und **Reset** einen weiteren, optionalen Parameter, der die Blockgröße angibt. Voreingestellt ist hier 128.

#### 3.10.4.2 Schreiben und Lesen

Zum Schreiben in typisierten Dateien bietet Delphi uns **Write** an. Parameter sind neben der Dateivariablen mindestens eine Variable des Datentyps. Da Textdateien zeilenorientiert sind, gibt es hier zusätzlich **WriteLn**, das nach dem Schreiben durch **Write** einen Zeilenumbruch der Datei



anfügt. Anders als in typisierten Dateien kann hier als zweiter Parameter auch eine Konstante übergeben werden. Das Einlesen der Daten mit `Read` und `ReadLn` verhält sich analog zum Schreiben - bis auf die Konstante.

```
var
  TypisierteDatei: file of Byte;

procedure SchreibeDatei;
var
  i: Integer;
begin
  AssignFile(TypisierteDatei, FN);
  Rewrite(TypisierteDatei);
  for i := 1 to 10 do
    write(TypisierteDatei, i); //implizite Typumwandlung
  CloseFile(TypisierteDatei);
end;

procedure LeseDatei;
var
  wert: Byte;
begin
  AssignFile(TypisierteDatei, FN);
  Reset(TypisierteDatei);
  while not Eof(TypisierteDatei) do //solange nicht Dateiende erreicht ist
  begin
    Read(TypisierteDatei, wert);
    writeln(wert); //ohne Dateiparameter=Ausgabe auf Konsole
  end;
  CloseFile(TypisierteDatei);
end;

begin
  SchreibeDatei;
  LeseDatei;
  ReadLn; //wartet auf Eingabe
end.
```

Das Programm ist in zwei Prozeduren aufgeteilt, die das Schreiben und Lesen der 10 Werte erledigt. Interessant in der Write-Zeile sind die unterschiedlichen Typen. Vorhanden ist ein Integer, die Datei ist aber vom Typ Byte. Hier findet also eine implizite Typumwandlung statt, denn maßgebend ist immer der Datentyp der Datei. Vor solchen Umwandlungen - von groß nach klein - sollte man sich hüten! Falls der Wertebereich überschritten wird, ist die Umwandlung verlustbehaftet oder fehlerhaft. Neu ist die boolsche Funktion `Eof`, die dann True zurückgibt, wenn das Dateiende erreicht ist.

Bei untypisierten Dateien ist die Handhabung etwas komplizierter, da der Inhalt der Datei nicht strukturiert sein muss. Diese Dateien werden mit `BlockWrite` geschrieben und durch `BlockRead`

gelesen. Zusätzliche Parameter sind hier eine Variable, aus der gelesen bzw. in die geschrieben wird und die Anzahl der Datensätze pro Lese- und Schreibvorgang. Diese Anzahl steht in direkter Verbindung zur optionalen Blockgröße in **Reset** bzw. **Rewrite** und bestimmt die Menge an Daten, die pro Lese- oder Schreibzugriff verarbeitet werden. Wir nehmen die gerade erzeugte typisierte Datei und lesen sie in Blöcken zu einem Byte aus:

```
procedure LeseDatei;  
var  
    untypisierteDatei: file;  
    buffer: Byte;  
begin  
    AssignFile(untypisierteDatei, FN);  
    Reset(untypisierteDatei, 1); //Blockgröße = 1  
    while not Eof(untypisierteDatei) do  
        begin  
            BlockRead(untypisierteDatei, buffer, SizeOf(Byte)); //SizeOf(Byte) = 1  
            writeln(buffer);  
        end;  
end;
```

#### 3.10.4.3 Der Dateizeiger

Die momentane Schreib- bzw. Leseposition innerhalb einer Datei wird durch den Dateizeiger beschrieben und nur dort erfolgt der Zugriff auf die Datei. Nach dem Öffnen bzw. Erstellen einer Datei durch **Reset** bzw. **Rewrite**, steht der Dateizeiger am Anfang der Datei. **Append** platziert den Dateizeiger am Ende der Datei, da Text angefügt werden soll. Wir können in typisierten und untypisierten Dateien die Position dieses Dateizeigers mit **Seek** verändern, weil dort die Datensatzgröße bekannt ist bzw. bei untypisierten Dateien eine Blockgröße angegeben wird.

```

var
  TypisierteDatei: file of Longint;
  IntArray: array [1..5] of Longint = (1, 2, 3, 4, 5);
  i: Integer;
  Buffer: Longint;
begin
  AssignFile(TypisierteDatei, FN);
  Rewrite(TypisierteDatei);
  for i := Low(IntArray) to High(IntArray) do
    Write(TypisierteDatei, IntArray[i]);
  i := High(IntArray) - 1;
  while i >= 0 do
    begin
      Seek(TypisierteDatei, i);
      Read(TypisierteDatei, Buffer);
      WriteLn(Buffer);
      Dec(i);
    end;
  CloseFile(TypisierteDatei);
  ReadLn;
end.

```

Obiges schreibt die Zahlen 1 bis 5 in eine Datei und liest sie rückwärts wieder aus. Die 5 Datensätze, also die Positionen 0 bis 4, werden einzeln durch `Seek` angesprungen, in dem der Dateizeiger jeweils vor dem Datensatz platziert wird. Zudem erkennen wir hier, dass `Rewrite` die Datei zum Lesen und Schreiben öffnet.

Da wir die Position des Dateizeigers also steuern können, können wir auch einzelne Datensätze überschreiben bzw. der Datei anhängen. Die Position, die dazu angestrebt werden muss, ist die Stelle, an der `Eof` `True` zurückgibt. Würden wir die Datei von vorne auslesen wollen, so müssten wir `Seek` mit der Position 0 aufrufen oder die Datei erneut mit `Reset` öffnen.

#### 3.10.4.4 Fehlerquellen

Bisher haben wir mögliche Fehler weitestgehend ignoriert. Delphi bietet hier aber einige Möglichkeiten an, das Programm sehr viel robuster zu gestalten.

Zunächst können wir die Fehlerbehandlung per Compiler-Schalter `{ $i+ }` und `{ $i- }` steuern. Dieser aktiviert bzw. deaktiviert die Überprüfung der letzten Eingabe/Ausgabe-Routine. Ist der Schalter aktiviert - was die Vorgabe ist - so führt ein E/A-Fehler zu einer Exception. Bei deaktiviertem Schalter wird keine Exception ausgelöst, der Fehlerstatus muss mit der Funktion `IOResult` überprüft werden. Diese liefert im Fehlerfall einen Wert ungleich 0 zurück.

Liegt ein Fehler vor, dann werden weitere E/A-Operationen blockiert. Eine Abfrage von `IOResult` setzt diesen Fehlerstatus wieder auf 0. Dies zeigt, dass `IOResult` in allen Fällen abgefragt und ausgewertet werden muss!

```

const
  FN = 'C:\Users\user\Desktop\textdatei.txt';
  //hier „user“ mit dem gültigen windows-User-Namen ersetzen
var
  bytearray: array [1..5] of Byte = (1, 2, 3, 4, 5);

procedure SchreibeDatei;
var
  textdatei: TextFile;
  i: Integer;
begin
  AssignFile(textdatei, FN);
  {$i-} //Fehlerbehandlung ausschalten
  Rewrite(textdatei);
  {$i+} //Fehlerbehandlung einschalten
  if IOResult = 0 then //Abfragen eines möglichen Fehlers
  begin
    for i := Low(bytearray) to High(bytearray) do
      WriteLn(textdatei, bytearray[i]);
    CloseFile(textdatei);
  end
  else
    WriteLn('Fehler beim Erstellen der Datei');
  end;

```

Das Beispiel zeigt, dass wir nicht nur Text, sondern auch Zahlen mit Write(Ln) schreiben können. Die Fehlerbehandlung wird genau für eine E/A-Operation ausgeschaltet. Danach fragen wir mit IOResult sofort den Fehlerstatus ab.

Wichtig ist hier noch, dass CloseFile nur aufgerufen werden darf, wenn Rewrite, Reset und Append fehlerfrei durchlaufen. Ansonsten wäre die Datei nicht geöffnet und kann somit auch nicht geschlossen werden, was einen Folgefehler hervorrufen würde.

Dieses Konzept der Fehlerbehandlung war bereits in Turbo Pascal bekannt. Delphi bietet uns hier aber eine angenehmere Methode die Datei sicher zu schließen - den bereits bekannten Ressourcenschutzblock:

```

procedure LeseDatei;
var
    textdatei: TextFile;
    buffer: string;
begin
    if FileExists(FN) then //Ist Datei vorhanden
    begin
        AssignFile(textdatei, FN);
        Reset(textdatei);
        try
            while not Eof(textdatei) do
            begin
                ReadLn(textdatei, buffer);
                WriteLn(buffer);
            end;
            finally
                CloseFile(textdatei);
            end;
        end
    else
        WriteLn('Keine Datei vorhanden');
    end;

begin
    SchreibeDatei;
    LeseDatei;
    ReadLn;
end.

```

Wir überprüfen mittels `FileExists`, ob die Datei überhaupt existiert und lesen sie wieder ein. Mit einfachsten Mitteln begegnen wir hier der wohl häufigsten Fehlerquelle in diesem Bereich. Zudem wissen wir, dass `CloseFile` nur nach einem fehlerlos durchlaufenen `Reset` ausgeführt werden darf. Somit erklärt sich die Position des `Try-Finally`-Blocks von selbst.

Auffällig an allen Beispielen ist, dass hier durchgehend fundamentale Typen verwendet wurden und ganz gezielt auf generische Typen verzichtet wurde. Das hat den einfachen Grund, dass die Breite eines generischen Typs nur in der aktuellen Implementierung feststeht. Ein fundamentaler Typ belegt auch in Zukunft so viele Bytes wie heute.

Beim Schreiben eines Records müssen wir auf die Ausrichtung innerhalb des Records achten. Wenn wir 1 Byte schreiben wollen und die Ausrichtung 4 Bytes beträgt, dann werden 1 Daten-Byte und 3 Leer-Bytes geschrieben. Das führt dazu, dass solch ein Record mehr Platz beansprucht, als die Summe seiner einzelnen Elemente.

Je nachdem, wie man die Daten wieder einliest, sind die Grenzen eines Datensatzes verschoben und man erhält Datenmüll. Diese Problematik können wir aber leicht umgehen, indem wir solch einen Verbund als „packed record“ deklarieren. Die Ausrichtung erfolgt dann an Byte-Grenzen, der kleinsten

adressierbaren Einheit. Alternativ könnte man hier auch den Compiler-Schalter {\$A} verwenden. Hier, wie auch bei den fundamentalen Typen, gilt nur bedingt das Abwägen von Schnelligkeit und Sicherheit.

### 3.10.5 Die Klasse TFileStream

Filestreams sind die objektorientierte Alternative zu den zuvor behandelten Dateitypen. Sie gleichen den untypisierten Dateien, da sie ebenfalls keine vordefinierte Datensatz-Struktur besitzen müssen. Nahezu alles, was wir bisher gelernt haben, können wir in leicht modifizierter Form hier anwenden. Um mit einer Instanz von TFileStream zu arbeiten, müssen wir diese erzeugen. Der Konstruktor erwartet 2 Parameter, den Dateinamen und den Zugriffsmodus. Zu den uns bereits bekannten Modi gesellt sich `fmCreate`, welches sich wie `Rewrite` verhält. Nach dem Aufruf gibt es eine leere geöffnete Datei. Am Ende muss die Instanz mit `Free` freigegeben werden.

#### 3.10.5.1 Orientierung im FileStream

`FileStream.Size` gibt die Größe des Streams in Bytes an. Die aktuelle Position innerhalb des Streams liefert `FileStream.Position` und wird interpretiert in Bytes, als Abstand zum Anfang des Streams. Mit `Seek` können wir wieder die Position im Stream festlegen. Im Gegensatz zur File-Variante erfordert `FileStream.Seek` einen weiteren Parameter, welcher in Relation zu einer festen Position im Stream steht. Entweder zum Anfang, zum Ende oder zur aktuellen Stellung. Es gibt folgende Möglichkeiten:

Wert	Bedeutung
<code>soFromBeginning</code>	Position ist danach übergebener Wert (Wert $\geq 0$ )
<code>soFromCurrent</code>	Position ist danach <code>FileStream.Position</code> + übergebener Wert
<code>soFromEnd</code>	Position ist danach <code>FileStream.Size</code> + übergebener Wert (Wert $\leq 0$ )

Bezieht man die Position auf das Ende des Streams, muss man also einen Wert  $\leq 0$  angeben, da der übergebene Wert aufaddiert wird.

#### 3.10.5.2 Schreiben und Lesen

Zum Lesen und Schreiben aus einem und in einen Stream stehen uns `Read` und `ReadBuffer` bzw. `Write` und `WriteBuffer` zur Verfügung. Als erster Parameter wird eine Variable erwartet, welche die im zweiten Parameter übergebenen Anzahl an Bytes übergibt bzw. aufnimmt. Die Parameter der Buffer-Varianten unterscheiden sich nicht zu ihrem Pendant. Intern rufen `ReadBuffer/WriteBuffer` sogar `Read/Write` auf. Im Gegensatz zu `Read` und `Write` gibt es bei den Buffer-Varianten aber eine Exception, falls es ein Problem beim Übertragen der Daten gibt. Bei `Read` und `Write` muss der Rückgabewert ausgewertet werden, da er die tatsächliche Anzahl gelesener bzw. geschriebener Bytes enthält.

```

program file_stream;

{$APPTYPE CONSOLE}

uses
    System.SysUtils, System.Classes;

const
    FN = 'C:\users\user\Desktop\filestream.dat';
    //hier „user“ mit dem gültigen windows-User-Namen ersetzen
procedure SchreibeDatei;
var
    filestream: TFileStream;
    datum: TDateTime;
begin
    if FileExists(FN) then
        filestream := TFileStream.Create(FN, fmOpenWrite)
    else
        filestream := TFileStream.Create(FN, fmCreate);
    try
        datum := Now;
        filestream.Seek(0, soFromEnd);
        filestream.WriteBuffer(Datum, SizeOf(datum));
    finally
        filestream.Free;
    end;
end;

```

Falls die Datei bereits existiert, wird sie im Schreibmodus geöffnet, ansonsten neu erstellt. In Datum wird der aktuelle DateTime-Wert geschrieben und danach wird der Satzzeiger am Ende des Streams positioniert. Wurde die Datei gerade neu erstellt, so ist diese Position auch der Anfang des Streams. Dort wird das Datum angehängt. Bei jedem Programmlauf erscheint somit ein DateTime mehr in der Datei. Am Ende wird der Stream freigegeben, abgesichert durch einem Try-Finally-Block.

```
procedure LeseDatei;  
var  
    filestream: TFileStream;  
    datum: TDateTime;  
begin  
    if FileExists(FN) then  
        begin  
            filestream := TFileStream.Create(FN, fmOpenRead);  
            try  
                while filestream.Position < filestream.Size do  
                    begin  
                        filestream.ReadBuffer(datum, SizeOf(datum));  
                        writeln(DateTimeToStr(datum));  
                    end;  
                finally  
                    filestream.Free;  
                end;  
            end  
        else  
            writeln('Keine Datei vorhanden');  
        end;  
  
    begin  
        SchreibeDatei;  
        LeseDatei;  
        ReadLn;  
    end.
```

Auch das Einlesen gestaltet sich nach bekannten Mustern. Es wird so lange gelesen, wie die aktuelle Position innerhalb des Streams kleiner ist, als die Größe der Datei.

Wollen wir Datenstrukturen speichern, deren Größe beim Einlesen nicht anhand der Datentypen abgelesen werden können (wie z.B. dynamische Arrays und Strings), dann behilft man sich mit einem Trick: Beim Speichern schreibt man zuerst die Größe des Typs in die Datei und erst dann den Wert. Beim Lesen können wir somit durch Auslesen der Größe, die Dimension des Typs festlegen.



```

type
  TDatensatz = record
    Zahl: Longint;
    Wort: AnsiString;
  end;

var
  Schreiksaetze, Lesesaetze: array of TDatensatz;

procedure SchreibeDatei;
var
  filestream: TFileStream;
  arraylaenge, stringlaenge: Longint;
  i: Integer;
begin
  filestream := TFileStream.Create(FN, fmCreate);
  try
    // Arraylänge ermitteln
    arraylaenge := Length(Schreiksaetze);
    // Schreiben der Arraylänge
    filestream.WriteBuffer(arraylaenge, SizeOf(arraylaenge));
    for i := 0 to arraylaenge - 1 do
      begin
        filestream.WriteBuffer(Schreiksaetze[i].Zahl, SizeOf(Schreiksaetze[i].Zahl));
        // Stringlänge ermitteln
        stringlaenge := Length(Schreiksaetze[i].Wort);
        // Schreiben der Stringlänge
        filestream.WriteBuffer(stringlaenge, SizeOf(stringlaenge));
        // Schreiben des Strings
        filestream.WriteBuffer(Schreiksaetze[i].Wort[1], stringlaenge);
      end;
    finally
      filestream.Free;
    end;
  end;
end;

```

Es handelt sich also hier um ein dynamisches Array eines Records, welcher unter anderem einen AnsiString enthält. Beim Einlesen gehen wir nun den umgekehrten Weg:

```

procedure LeseDatei;
var
    filestream: TFileStream;
    arraylaenge, stringlaenge: Longint;
    i: Integer;
begin
    if FileExists(FN) then
        begin
            filestream := TFileStream.Create(FN, fmOpenRead);
            try
                // Arraylänge einlesen
                filestream.ReadBuffer(arraylaenge, SizeOf(arraylaenge));
                // Zuweisen der Array-Länge
                SetLength(Lesesaeetze, arraylaenge);
                for i := 0 to arraylaenge - 1 do
                    begin
                        filestream.ReadBuffer(Lesesaeetze[i].Zahl, SizeOf(Lesesaeetze[i].Zahl));
                        // Stringlänge einlesen
                        filestream.ReadBuffer(stringlaenge, SizeOf(stringlaenge));
                        // Zuweisen der String-Länge
                        SetLength(Lesesaeetze[i].Wort, stringlaenge);
                        // Lesen des Strings
                        filestream.ReadBuffer(Lesesaeetze[i].Wort[1], stringlaenge);
                    end;
                finally
                    filestream.Free;
                end;
            end
        else
            WriteLn('Keine Datei vorhanden');
        end;

```

Anhand der vorab gelesenen Werte können wir nicht nur die Länge des dynamischen Arrays und der Strings zuweisen, sondern wissen auch, wie viele Bytes wir im nachfolgenden Schritt einlesen müssen.

Initialisierung und Darstellung findet im Hauptprogramm statt:

```

var
  i: Integer;
begin
  SetLength(Schreibsaetze, 10);
  for i := 0 to High(Schreibsaetze) do
  begin
    Schreibsaetze[i].Zahl := i;
    Schreibsaetze[i].Wort := IntToStr(i) + ' . Datensatz';
  end;
  SchreibeDatei;
  LeseDatei;
  for i := 0 to High(Lesesaetze) do
  begin
    WriteLn(Lesesaetze[i].Zahl);
    WriteLn(Lesesaetze[i].Wort);
    WriteLn;
  end;
  ReadLn;
end.

```

### 3.10.6 Die Klasse TStringList

Der Grund, warum wir Textdateien bisher eher stiefmütterlich behandelt haben, ist, dass es komfortablere Methoden der Bearbeitung dafür gibt. Hier bieten sich Stringlisten an, da das Verwalten der Strings in einer Liste dem Benutzer zusätzliche Möglichkeiten bietet.

#### 3.10.6.1 Die Handhabung

Um eine Stringliste zu erstellen und wieder freizugeben, gehen wir den üblichen Weg:

```

uses
  Classes;

var
  stringliste: TStringList;
begin
  stringliste := TStringList.Create;
  try
    //Mache was mit stringliste
  finally
    stringliste.Free;
  end;
end;

```

Soll die Lebensdauer einer Stringliste an die Anwendung oder das Vorhandensein eines Formulars gekoppelt werden, so bieten sich zum Erzeugen und Freigeben die Ereignisse OnCreate und OnDestroy an. Für die Anwendung benutzt man hierbei die Ereignisse des Hauptformulars.

Wir können natürlich vorhandene Textdateien in eine Stringliste laden und auch wieder speichern:

```
stringliste.LoadFromFile(FN);
stringliste.SaveToFile(FN);
```

Wollen wir der Stringliste neue Strings hinzufügen, so geschieht dies mit `Add`, `AddStrings` oder `Insert`. `Add` hängt einen String an, `Insert` fügt an der angegebenen Position ein und `AddStrings` hängt der Stringliste die Strings einer anderen Stringliste an. Strings sind hier zu verstehen als Zeilen einer Stringliste. Bei `Add` und `Insert` können die einzelnen Strings als Stringkonstante oder Stringvariable übergeben werden. Beim Einfügen eines Strings durch `Insert` sollte man beachten, dass Listen grundsätzlich 0-indiziert sind.

```
var
  s: AnsiString;
...
stringliste.Add('Übergabe als Konstante');
stringliste.Add(s);
stringliste.Insert(1, s); // 2. Position
stringliste1.AddStrings(stringliste2);
```

Das Löschen einer Zeile geschieht durch `Delete` unter Angabe des Index als Parameter. Soll der komplette Inhalt der Stringliste gelöscht werden, so bieten Listen dafür die Methode `Clear`.

```
stringliste.Delete(3); // Löscht die 4. Zeile
stringliste.Clear;
```

Der direkte Zugriff auf eine Zeile geschieht nicht durch einen Parameter, sondern über die Array-Eigenschaft `Strings`. Dadurch können wir den String direkt bearbeiten oder lesen. Die Anzahl der Zeilen einer Stringliste lesen wir aus der Eigenschaft `Count`. Zugreifen können wir auf Elemente von 0 bis `Count-1`.

```
var
  s: AnsiString;
  i: Integer;
...
stringliste.Strings[3] := 'Text'; // 4. Zeile ersetzen
stringliste[3] := 'Text'; // Alternative Schreibweise
stringliste[0] := stringliste[0] + 'Text'; // Anhängen an 1. Zeile
s := stringliste[stringliste.Count - 1]; // Letzte Zeile wird s zugewiesen
stringliste[1] := stringliste[3]; // 2. und 4. Zeile sind nun identisch
i := stringliste.Count; // i zeigt die Anzahl der Zeilen
```

Um die Strings innerhalb der Liste umzuordnen, verwenden wir die Methoden `Move` und `Exchange`, jeweils durch Übergabe zweier Indizes. `Move` verschiebt dabei den String mit Index1 nach Index2. Der String an Index2 und alle Strings zwischen diesen Indizes ändern dadurch ihre Position. `Exchange` vertauscht zwei Strings, die Indizes aller anderen Strings ändern sich dadurch nicht.

```
stringliste.Move(1, 3); // 2. Zeile ist nun 4. Zeile
stringliste.Exchange(0, stringliste.Count - 1); // vertauscht 1 und letzte Zeile
```

Wollen wir die Stringliste auf das Vorhandensein eines Strings testen, so müssen wir nicht die Liste prüfend durchlaufen, sondern verwenden dazu `IndexOf`. Diese Funktion gibt -1 zurück, wenn der Prüf-String kein String der Liste ist, andernfalls den Index des 1. Vorkommens. Die gesamte Zeile und der Prüf-String müssen dabei identisch sein! Berücksichtigen muss man hier den angegebenen Typ der Stringliste in der Variablendeklaration. Verwendet man, wie oben angegeben, `TStrings`, dann ist der Vergleich nicht unter Berücksichtigung der Groß-Kleinschreibung. Ist der angegebene Typ in der Variablendeklaration jedoch `TStringList`, dann kann man dieses Verhalten durch die Eigenschaft `CaseSensitive` steuern.

```
var
  stringliste: TStringList;
  i: Integer;
...
// Groß-Kleinschreibung soll berücksichtigt werden
stringliste.CaseSensitive := True;
// i zeigt -1 oder den Index des 1. Auffindens
i := stringliste.IndexOf('Hallo welt');
```

### 3.10.6.2 Stringlisten in Verbindung zur VCL

Stringlisten sind zwar nicht beschränkt auf Formularanwendungen, allerdings liegt hier ihr Haupteinsatzgebiet. Um mehrzeiligen Text zu verwalten, besitzen viele Komponenten der VCL einen Abkömmling von `TStrings`. Der Zugriff darauf geschieht über die Eigenschaft `Lines` bzw. `Items` und entspricht der bereits gezeigten Vorgehensweise.

Will man z.B. eine vorhandene Textdatei in einer Instanz von `TMemo` oder `TListBox` anzeigen, dann geht das relativ problemlos:

```
Memo1.Lines.LoadFromFile(FN);
ComboBox1.Items.LoadFromFile(FN);
```

Ähnlich unproblematisch gestaltet sich der Fall, diese Dateien aus den Komponenten heraus wieder zu speichern:

```
Memo1.Lines.SaveToFile(FN);
ComboBox1.Items.SaveToFile(FN);
```

Sowohl `Lines` als auch `Items` sind vom Typ `TStrings`, von dem `TStringList` abgeleitet ist. Die Inhalte können wir einander zuweisen:

```
Memo1.Lines.Assign(ComboBox1.Items);
```

### 3.10.7 Arbeiten mit Dateien

Mit Dateien kann man mehr machen, als sie zu lesen oder zu schreiben. Man kann z.B. das Datum der letzten Änderung auslesen oder setzen wollen oder prüfen, ob eine Datei überhaupt existiert.

Delphi bietet dafür die Unit `IOUtils`, welche den Typ `TFile` enthält. Wichtig ist, dass es sich hierbei um ein Record handelt, das eine ganze Reihe an Methoden enthält. Man sollte keine Variablen vom Typ `TFile` anlegen!

Beispielhafte Verwendung: Wir wollen wissen, wann die Datei erstellt worden ist:

```
uses System.IOUtils;  
...  
var creationDate: TDateTime;  
begin  
    creationDate := TFile.GetCreationTime(FN);
```

Eine weitere Methode liest den kompletten Inhalt einer Textdatei in ein String-Array (`ReadAllLines`) oder in einen String (`ReadAllText`) – hierbei ist sogar die Angabe des Encodings möglich.

Neben `TFile` gibt es auch die Records `TDirectory` und `TPath`, die ähnlichen Funktionalitäten für Verzeichnisse und Pfade bereitstellen.

### 3.11 Besondere Datentypen

In einem vorigen Kapitel haben wir die grundlegenden Datentypen in Delphi kennengelernt. Nun gibt es noch weitere Datentypen, die etwas komplexer sind und zum Teil in Form von Klassen zur Verfügung stehen. Aus diesem Grund kommt dieses Kapitel erst hier, nachdem wir die Objektorientierung kennengelernt haben. Konkret geht es z.B. um Listen, die einfacher zu handhaben sein können als simple Arrays, und um Datums- und Zeit-Werte.

#### 3.11.1 Datum und Zeit

Will man mit Datumswerten rechnen, z.B. Tage hinzuzählen oder abziehen, ist es ungünstig, das Datum in einer String-Variablen abgelegt zu haben. Auch jeweils eine Integer-Variable für Tag, Monat und Jahr anzulegen, ist nicht wirklich schön. Zumal dann immer die Bedingungen dazu kämen wie „Wenn Tag größer als 31, dann setze Tag auf 1 und zähle den Monat um eins hoch. Wenn es sich um April, Juni, September oder November handelt, dann darf der Tag höchstens 30 sein.“ Und was ist mit Schaltjahren im Februar? Natürlich wurde all das schon mal von jemandem programmiert. Und wenn wir das verwenden wollen, setzen wir in Delphi auf die Datentypen `TDate`, `TTime` und `TDatetime`. `TDate` beinhaltet lediglich ein Datum ohne Uhrzeit, `TTime` nur eine Uhrzeit und `TDatetime` beides. Alle drei Typen können ohne das Einbinden spezieller Units verwendet werden.

Die Units `SysUtils` und `DateUtils` bieten jede Menge Funktionen, um mit Datums- und Zeitwerten zu arbeiten.

Genau genommen ist `TDatetime` einfach ein Double-Wert, also eine Zahl. Der ganzzahlige Anteil stellt dabei die Tage seit dem 30.12.1899 dar, der Nachkommateil bestimmt die Uhrzeit als Bruchteil eines Tages.

##### 3.11.1.1 Heute und jetzt

Zunächst sehen wir uns an einem Beispiel an, wie man `TDate`-, `TTime`- und `TDatetime`-Variablen auf das aktuelle Datum bzw. die aktuelle Zeit setzt:

```
uses SysUtils;
var
  datum: TDate;
  zeit: TTime;
  datumZeit: TDateTime;
begin
  datum := Date; // datum enthält nun das aktuelle Systemdatum
  zeit := Time; // zeit enthält nun die aktuelle Systemuhrzeit
  datumZeit := Now; // datumZeit enthält sowohl Datum als auch Uhrzeit
  „Date“, „Time“ und „Now“ sind Funktionen der Unit SysUtils.
```

##### 3.11.1.2 Datum und Zeit umwandeln

Häufig bekommt man ein Datum aber über eine Benutzereingabe als String und muss diesen in einen Datumswert umwandeln:

```

uses SysUtils;
var
    datum: TDate;
begin
    datum := StrToDate(Edit1.Text);

```

Genauso gibt es auch `StrToTime` usw. sowie die umgekehrte Richtung (`DateToStr`, `TimeToStr`, ...) für den Fall, dass ein Datumswert in der GUI dargestellt werden soll. Ein Problem beim Umwandeln eines Strings in ein Datum kann natürlich sein, dass der Benutzer einen String eingegeben hat, der sich gar nicht in ein gültiges Datum konvertieren lässt. In diesem Fall fliegt eine Exception vom Typ `EConvertError`.

Möchte man auf das Abfangen dieses Fehlers verzichten, bietet sich auch die Funktion `TryStrToDate` an:

```

uses SysUtils;
var datum: TDate;
begin
    if (TryStrToDate(Edit1.Text, datum)) then ...

```

`TryStrToDate` erwartet zwei Parameter: als erstes den umzuwandelnden String und als zweites die `TDate`-Variable, in der das umgewandelte Datum abgelegt werden soll. Falls die Umwandlung nicht funktioniert, gibt die Funktion `false` zurück, so dass der Teil nach dem `then` in obigem Beispiel nicht ausgeführt werden würde.

Als dritte Variante zur Erzeugung eines Datums- oder Zeitwertes gibt es die Encode-Funktionen, denen man ein Datum oder eine Uhrzeit in ihren Einzelteilen mitgibt:

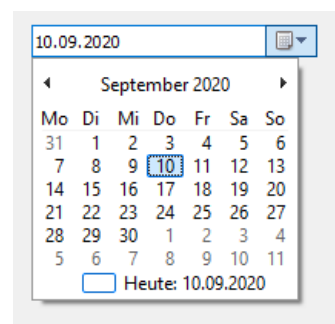
```

uses SysUtils;
var datum: TDate;
begin
    datum := EncodeDate(2020, 9, 10);

```

Und schließlich gibt es für VCL-Anwendungen auch die Komponente **TDateTimePicker**. Damit hat der Anwender die Möglichkeit, ein Datum in einem grafischen Kalender auszuwählen. In der Anwendung kann man sich dann sicher sein, dass das Datum korrekt ist. Falscheingaben sind nicht möglich.

Nun können wir mit den Datums- und Zeitwerten arbeiten. Hier ein paar Beispiele, die in den allermeisten Fällen das Einbinden der **Unit System.DateUtils** voraussetzen:





### 3.11.1.3 Ermittlung des Wochentags eines Datums

```
var
  day: Integer;
begin
  day := DayOfTheWeek(datum);
```

day enthält nun eine Zahl zwischen 1 (Sonntag) und 7 (Samstag). Will man einen Klartextnamen ausgeben anstatt einer Zahl, so hilft das vordefinierte Array in der Unit SysUtils namens LongDayNames. Die verwendete Sprache entspricht der Delphi-Installation.

```
uses System.SysUtils;

var
  day: Integer;
begin
  day := DayOfTheWeek(datum);
  displayDay := LongDayNames(day); // aktueller wochentag, z.B. 'Samstag'
```

### 3.11.1.4 Schaltjahr bestimmen

Eine wichtige Funktion beim Arbeiten mit Datumswerten ist die Prüfung, ob es sich bei einem Jahr um ein Schaltjahr handelt. Diese heißt IsLeapYear und befindet sich in der Unit SysUtils:

```
if IsLeapYear(2011) then ...
```

### 3.11.1.5 Zeitabstände prüfen

Die Unit DateUtils enthält jede Menge Funktionen, um den Abstand zwischen zwei Datums-/Zeitwerten zu prüfen. Beispiel: Liegt Datum X maximal Y Tage vor Datum Z?

```
var
  dateNow, datePast: TDateTime;
begin
  dateNow := EncodeDate(2011,8,13);
  datePast := EncodeDate(2011,6,13);
  if withinPastDays(dateNow, datePast, 30) then
    ShowMessage('ja')
  else
    ShowMessage('nein');
```

Im Beispiel wird geprüft, ob der 13.6.2011 maximal 30 Tage vor dem 13.8.2011 liegt. Die Ausgabe ist natürlich „nein“.

## 3.11.2 Listen mit TList

Die Delphi-RTL (Runtime Library) bietet Klassen an, mit denen sich Datenstrukturen einfacher handhaben lassen als mit den Standard-Datentypen. So ist z.B. eine Liste an Elementen mit `TList` einfacher zu verwenden als ein Array. Was ist der Unterschied?

`TList` ist eine Klasse, von der zunächst eine Instanz erzeugt werden muss. Dann lassen sich über die Methoden `Add` und `Insert` Elemente anhängen bzw. an einer bestimmten Position einfügen. Schon

hier hätten wir bei einem Array das Problem, den Eintrag, der bisher an der Position war, sowie alle folgenden um eine Position nach hinten zu verschieben. Dafür muss auch genügend Speicher reserviert werden (**SetLength**). Nicht so bei **TList**:

```
uses Generics.Collections;

procedure TForm1.MyMethod;
var
  MyList: TList<Integer>;
begin
  MyList := TList<Integer>.Create;
  try
    MyList.AddRange([1, 2, 3]);
    MyList.Insert(0, 9);
    ...
  finally
    MyList.Free;
  end;
```

Was passiert in diesem Beispiel? Wir definieren eine Variable vom Typ **TList** und geben dahinter in spitzen Klammern an, von welchem Typ die Elemente der Liste sein dürfen. In diesem Fall handelt es sich um eine Liste von Integer-Werten.

Diese Schreibweise nennt sich Generics und ist in Delphi erst seit Version 2009 möglich! Bis dahin enthielt eine Liste einfach nur Pointer, so dass beim Auslesen der Werte immer eine Typumwandlung durchgeführt werden musste.

Direkt nach **begin** wird eine Instanz der Liste erzeugt. Mit **AddRange** können gleich mehrere Werte in die Liste eingefügt werden. **Add** und **AddRange** fügen Werte immer am Ende an. **Insert** (und **InsertRange**) hingegen platzieren die neuen Werte an der angegebenen Position. Das **Insert** im Beispiel fügt den Wert 9 an der ersten Position (Index 0, weil die Zählung bei 0 beginnt) ein. Die bereits vorhandenen Werte 1, 2 und 3 rutschen dadurch eine Position weiter.

Die Werte der Liste lassen sich auch automatisch sortieren:

```
MyList.Sort();
```

Durch diesen Aufruf gelangt die in obigem Beispiel am Anfang eingefügte 9 ans Ende der Liste. Durch den Aufruf von

```
MyList.Reverse();
```

lässt sich die Reihenfolge der Elemente umdrehen, also rückwärts sortieren.

Auf die einzelnen Elemente einer Liste greift man über ihren Index zu. So gibt beispielsweise

```
ShowMessage(IntToStr(MyList.Items[2]));
```

den Wert an Indexposition 2 (also den dritten Wert) aus. Wer nach einem bestimmten Wert in einer sortierten Liste suchen möchte, sollte sich die Methode **BinarySearch** anschauen.

Als Erweiterung einer `TList` gibt es auch **TObjectList**. Diese ist hilfreich, wenn die Liste nicht einfache Zahlen (wie in obigem Beispiel), sondern Objekte beinhalten soll. Beim Entfernen eines Objekts aus der Liste kann es freigegeben werden. Wichtig hierfür ist die Eigenschaft **OwnsObject**. Über sie wird festgelegt, ob die Elemente der Liste gehören (`true`) oder nicht (`false`). Wenn Elemente der Liste gehören, werden sie beim Entfernen aus der Liste automatisch freigegeben. **OwnsObject** kann dem Konstruktor **Create** als Parameter mitgegeben werden. Standardwert ist `true`.

Für Strings gibt es die Klasse **TStringList**.

### 3.11.3 Dictionaries mit TDictionaary

Die RTL enthält einen weiteren hilfreichen Datentyp, **TDictionaary**. Dabei handelt es sich um eine Sammlung aus Schlüssel-Wert-Paaren, in vielen Programmiersprachen bekannt als **Map**. Die wichtigsten Funktionen sind bei einem Dictionary das Hinzufügen von Schlüssel-Wert-Paaren – der Schlüssel muss eindeutig sein und darf nicht `nil` sein – und das Auslesen. Das Hinzufügen geschieht mit **Add** oder **AddOrSetValue**. **Add** wirft eine **Exception**, falls bereits ein Eintrag mit demselben Schlüssel (**Key**) im Dictionary existiert. **AddOrSetValue** ersetzt in diesem Fall den bereits vorhandenen Eintrag.

```
uses Generics.Collections;

procedure TForm1.MyMethod;
var
  MyDict: TDictionaary<Integer, string>;
  Value: String;
begin
  MyDict := TDictionaary<Integer, string>.Create;
  try
    MyDict.Add(1, 'Eins');
    MyDict.Add(2, 'Zwei');
    Value := MyDict.Items[2]; // Value enthält nun den Wert ,Zwei‘
    MyDict.AddOrSetValue(2, 'Neue Zwei');
    Value := MyDict.Items[2]; // Value enthält nun den Wert ,Neue Zwei‘
    MyDict.Add(2, '2');      // Exception EListError wird geworfen,
                           // weil Key 2 schon existiert
  end;
end;
```

Zum Auslesen von Werten aus einem Dictionary gibt es verschiedene Methoden. Eine davon haben wir in obigem Beispiel schon gesehen: **Items** gefolgt vom **Key** in eckigen Klammern. In unserem Fall sind die **Keys** vom Typ **Integer**, weshalb in eckigen Klammern ein **Integer** steht. Rückgabewert ist der Wert, der zu dem **Key** gehört (in unserem Fall ein **String**). Existiert der gesuchte Schlüssel im Dictionary allerdings nicht, wird eine **EListError**-Exception geworfen. Das kann man verhindern, indem man vorher prüft, ob der Schlüssel existiert: **ContainsKey**.

```
if MyDict.ContainsKey(3) then  
    Value := MyDict.Items[3];
```

Will man sich das Überprüfen sparen, kann man die Methode `TryGetValue` verwenden, die als zweites Argument einen Out-Parameter bekommt. Die Methode an sich gibt `true` oder `false` zurück, je nach dem, ob der Schlüssel gefunden wurde oder nicht.

```
var value: string;  
begin  
    MyDict.TryGetValue(1, value);
```

Normalerweise wird auf den Inhalt von Dictionaries über die Schlüssel zugegriffen, die eindeutig sind. Es kann aber unter Umständen sinnvoll sein, auch nach dem Vorhandensein von Werten zu suchen. Auch das unterstützt `TDictionary`:

```
if MyDict.ContainsValue('Zwei') then ...
```

#### 3.11.4 Stacks und Queues

Die RTL enthält weitere Klassen außer `TList` und `TDictionary`, um Mengen von Werten bzw. Objekten zu verwalten. Zu erwähnen sind `TStack` und `TQueue`. Beide verwalten ihre Elemente in einer festen Reihenfolge und sind in der Unit `System.Generics.Collections` zu finden.

Bei einem Stack (Stapel) werden neue Elemente immer oben drauf gelegt (Push) und beim Auslesen immer das oberste Element wieder entfernt (Pop). Das entspricht dem sog. LIFO-Prinzip („Last in first out“). Das letzte Element das hinzugekommen ist, wird als erstes wieder entfernt.

Anders ist es bei der Warteschlange (Queue). Hier werden neue Elemente immer hinten angestellt (Enqueue), während beim Auslesen vorne beim ältesten Element begonnen wird (Dequeue). Das entspricht dem FIFO-Prinzip („First in first out“). Das Element, das als erstes hinzugefügt worden ist, wird auch als erstes wieder entfernt.

### 3.12 Generische Datentypen („Generics“)

Delphi unterstützt seit Version 2009 generische Datentypen. Diese werden gebraucht, um Datentypen sehr allgemein programmieren zu können, ohne dass sie dadurch in der Anwendung kompliziert werden. Ein gutes Beispiel dafür sind Listen. In Delphi gibt es dafür schon seit langer Zeit die Klasse `TList`. `TList` sollte in jeder Lebenslage einsetzbar sein, weshalb alle enthaltenen Elemente als Pointer aufgefasst werden. Ein Pointer zeigt auf eine Adresse im Arbeitsspeicher. Was dort zu finden ist, muss der Programmierer selbst wissen. D.h. beim Auslesen muss das Element in einen bestimmten Typ umgewandelt werden. Das ermöglicht es einem Programmierer nun tatsächlich, Objekte beliebigen Typs in eine Liste zu stecken. Allerdings gibt es nun zwei Probleme: Üblicherweise enthalten Listen nur Elemente des gleichen Typs, z.B. eine Liste von Strings. Das kann `TList` aber nicht sicherstellen. Dadurch ist es möglich, Objekte völlig unterschiedlicher Typen absichtlich oder unabsichtlich in dieselbe Liste zu stecken. Auf das andere Problem stößt man beim Auslesen der Liste: `TList` gibt alles als Pointer zurück. Der Programmierer muss das Objekt also selbst in das casten, was er eigentlich erwartet, was umständlich und fehleranfällig ist.

Wie wäre es also, wenn man der Liste schon beim Erzeugen sagen könnte, dass sie nur Strings aufnehmen und zurückgeben soll? Und in einer anderen Instanz nur Integers? Genau das ist der Einsatzzweck für Generics.

Wer auch die vorhergehenden Kapitel gelesen hat, kennt das Vorgehen schon. Hier trotzdem noch einmal das Beispiel:

```
var  
    MyList: TList<Integer>;
```

`MyList` ist nun eine Liste, die ausschließlich Integer-Werte aufnehmen kann. Beim Auslesen der Liste kommen direkt Integer-Werte zurück, die nicht gecastet werden müssen.

Selbstverständlich sind auch weitere Verwendungen möglich:

```
MyStringList: TList<string>;
```

`TList` ist also eine generische Klasse. Doch wie macht sie das?

Die Klasse `TList` weiß vor ihrer Instanziierung nicht, für welchen Datentyp sie zuständig sein soll. Deshalb können die Methoden in `TList` nicht mit einem bestimmten Typ arbeiten, auch nicht mit `TObject`. Stattdessen wird ein Platzhalter verwendet, üblicherweise kommt hier der Großbuchstabe `T` zum Einsatz. Allerdings können auch beliebige andere Pascal-Bezeichner verwendet werden. Die Klassendefinition von `TList` könnte also so aussehen (beispielhafte Implementierung, nicht an der Original-`TList`-Klasse orientiert):

```
type TList<T> = class
  public
    procedure add(element: T);
    function get: T;
end;
```

Man sieht an diesem Code: Alles läuft im Prinzip so, wie man es kennt. Nur dass eben kein konkreter Datentyp als Rückgabewert oder Parameter verwendet wird, sondern das ominöse T. Dieses wird erst bei der Instanziierung der Klasse durch einen konkreten Typ ersetzt wie in obigen Beispielen mit Integer- und String-Listen zu sehen.

***Nachdem nun Details von Object Pascal bekannt sind, wollen wir uns ansehen, wie man Fehler in eigenen Programmen findet und wie man vorgehen kann, um sie zu vermeiden.***

## 4 Fehlerbehandlung

Fehler können wir generell in drei Gruppen einteilen:

- **Syntaxfehler** werden bereits beim Kompilieren gefunden und angezeigt. Der Compiler prüft dabei den Quelltext auf seine syntaktischen und semantischen Eigenschaften. Die syntaktische Analyse bezieht sich auf die Grammatik von Object Pascal, z.B. auf ein falsch geschriebenes Schlüsselwort. Nicht deklarierte Bezeichner werden hingegen erst in der darauffolgenden semantischen Analyse festgestellt, da dort die Zusatzbedingungen im jeweiligen Zusammenhang überprüft werden. Tritt ein Syntaxfehler auf, dann wird das Programm nicht übersetzt und ist somit nicht lauffähig. Allerdings wissen wir, wo der Fehler auftritt, und können ihn somit schnell beheben.
- **Laufzeitfehler** entstehen erst beim Ausführen des Programms – die Syntaxprüfung war also erfolgreich. Typische Fehler wären z.B. eine Division durch Null oder der Zugriff auf eine noch nicht erzeugte Instanz einer Klasse. Im günstigsten Fall erhalten wir bei einem Laufzeitfehler sofort eine Meldung und kennen somit die Fundstelle. Möglich ist allerdings auch, dass irgendein Speicherbereich überschrieben wird und der Fehler erst sehr viel später auftritt.
- Bei **Logikfehlern**, welche nicht zu einem Laufzeitfehler führen, treten überhaupt keine Fehlermeldungen auf. Das Programm verhält sich lediglich anders als erwartet. Solche Fehler sind mitunter schwer zu finden, da wir unter Umständen von einer falschen Annahme ausgehen und uns deswegen selbst im Weg stehen.

Syntaxfehler können wir direkt im Quelltext beheben. Für Laufzeit- und Logikfehler benötigen wir jedoch ein Instrument, welches uns erlaubt, den Zustand des Programms auszuwerten. Erst durch den Einblick in aktuelle Variableninhalte, die Aufrufreihenfolge von Funktionen oder das Durchschreiten des Quelltextes in Einzelschritten, können wir uns der Fehlerstelle nähern. Das alles gehört zur Grundfunktionalität des Debuggers.

### 4.0 Die Vorbereitung

Wird ein Programm erfolgreich kompiliert, so werden die symbolischen Namen der Bezeichner entfernt. Erweiterte Compiler-Optimierungen können dazu führen, dass der Programmablauf für uns nicht mehr wirklich nachvollziehbar wird. Darum weisen wir den Compiler an, zusätzliche Debug-Informationen in die ausführbare Datei zu übernehmen und deaktivieren außerdem die Optimierungen. Die erzeugte Datei wird durch die Aufnahme der Debug-Informationen wesentlich

größer. Erlangt unser Programm dann irgendwann die Reife, so können wir diese, dann unnötigen Informationen, aus dem fertigen Programm entfernen. Delphi bietet uns hier einen relativ einfachen Mechanismus an, beim Kompilieren zwischen Endversion (Release) und Analyse (Debug), zu unterscheiden. Unter dem Menüpunkt Projekt/Optionen.../Erzeugen/Delphi-Compiler wählen wir als Projektoption die Build-Konfiguration Debug aus. Dies muss für jedes (neue) Projekt einmalig eingestellt werden.

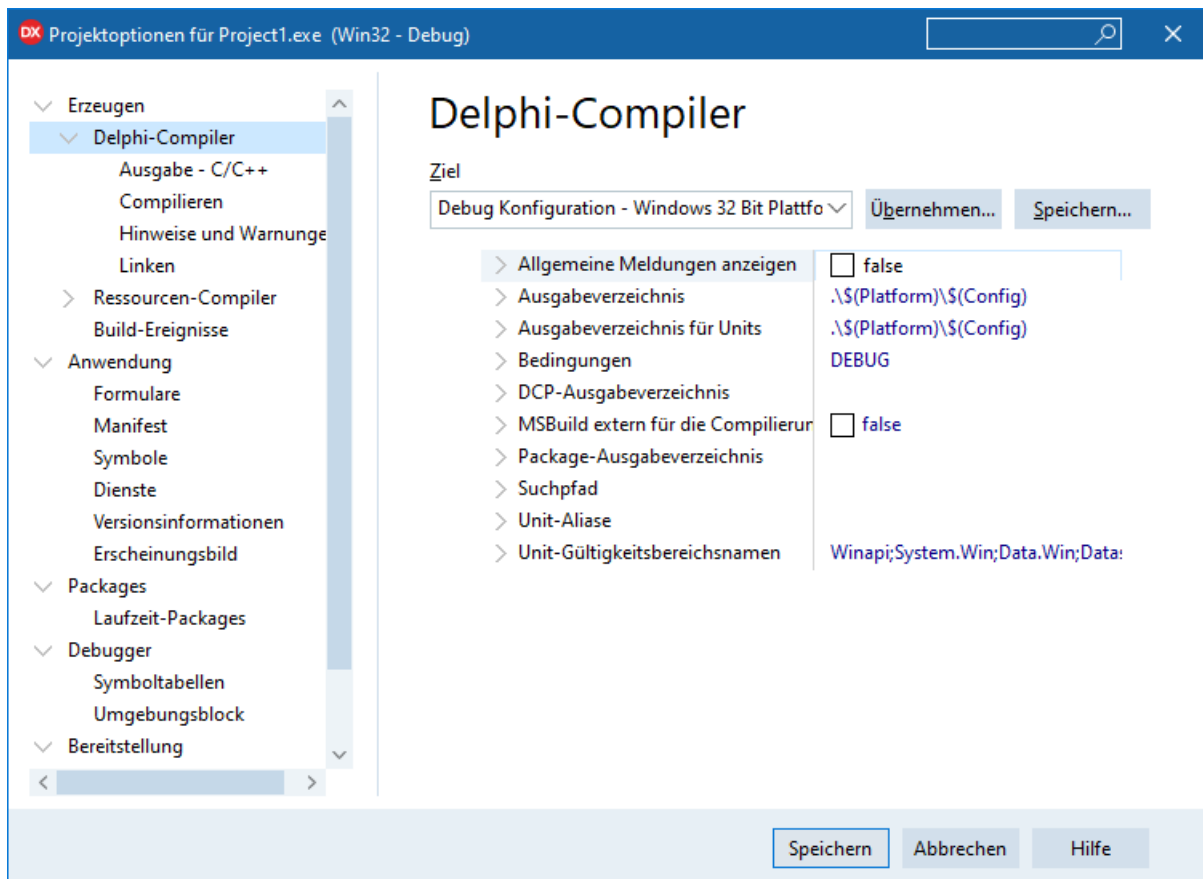


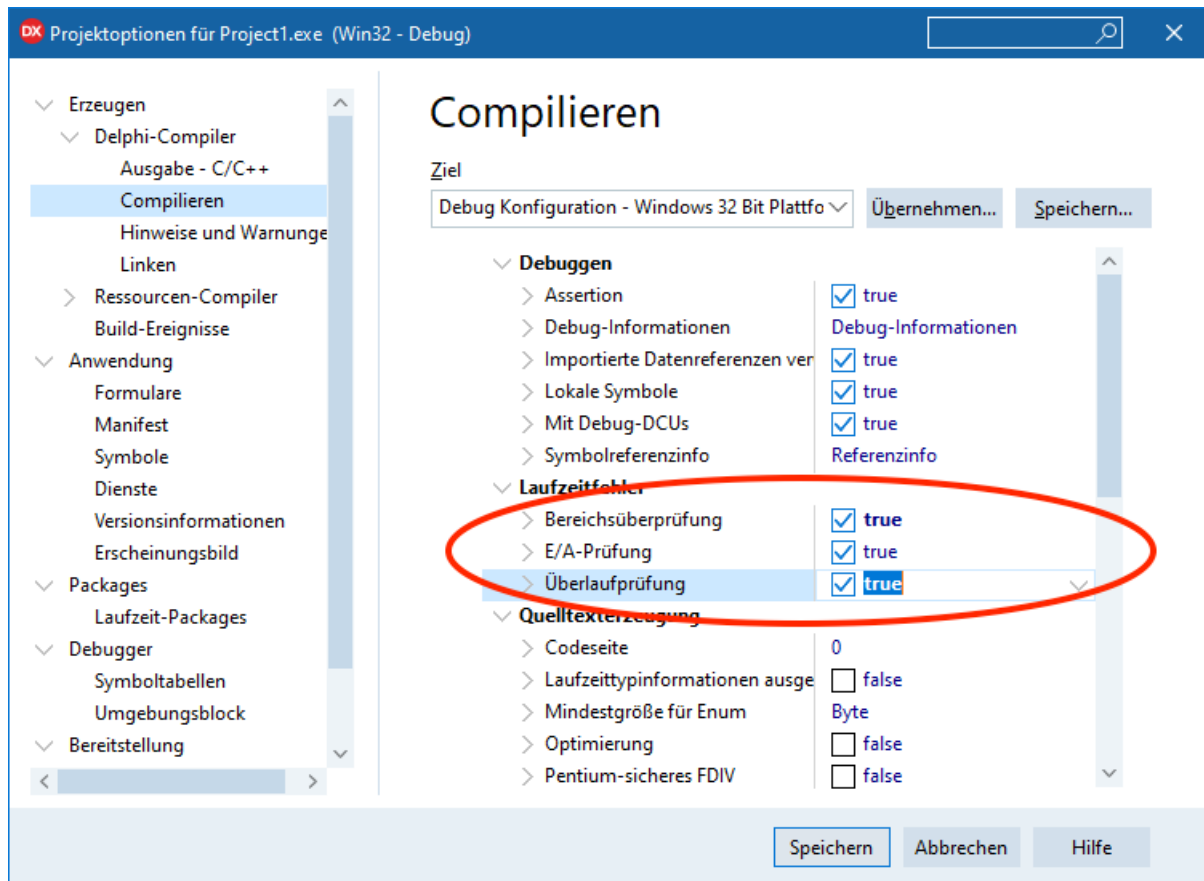
Abbildung 28: Projektoptionen

Daran erkennen wir auch, dass sich Debug-Informationen auf Projektebene bewegen. Ändern wir Einstellungen eines bestehenden Projekts, so reicht ein Kompilieren des Quelltextes durch Strg+F9 allein nicht, da dabei nur die Änderungen am Quelltext zur letzten Übersetzung berücksichtigt werden. Das Projekt muss also durch die Tastenkombination Umsch+F9 neu erzeugt werden.

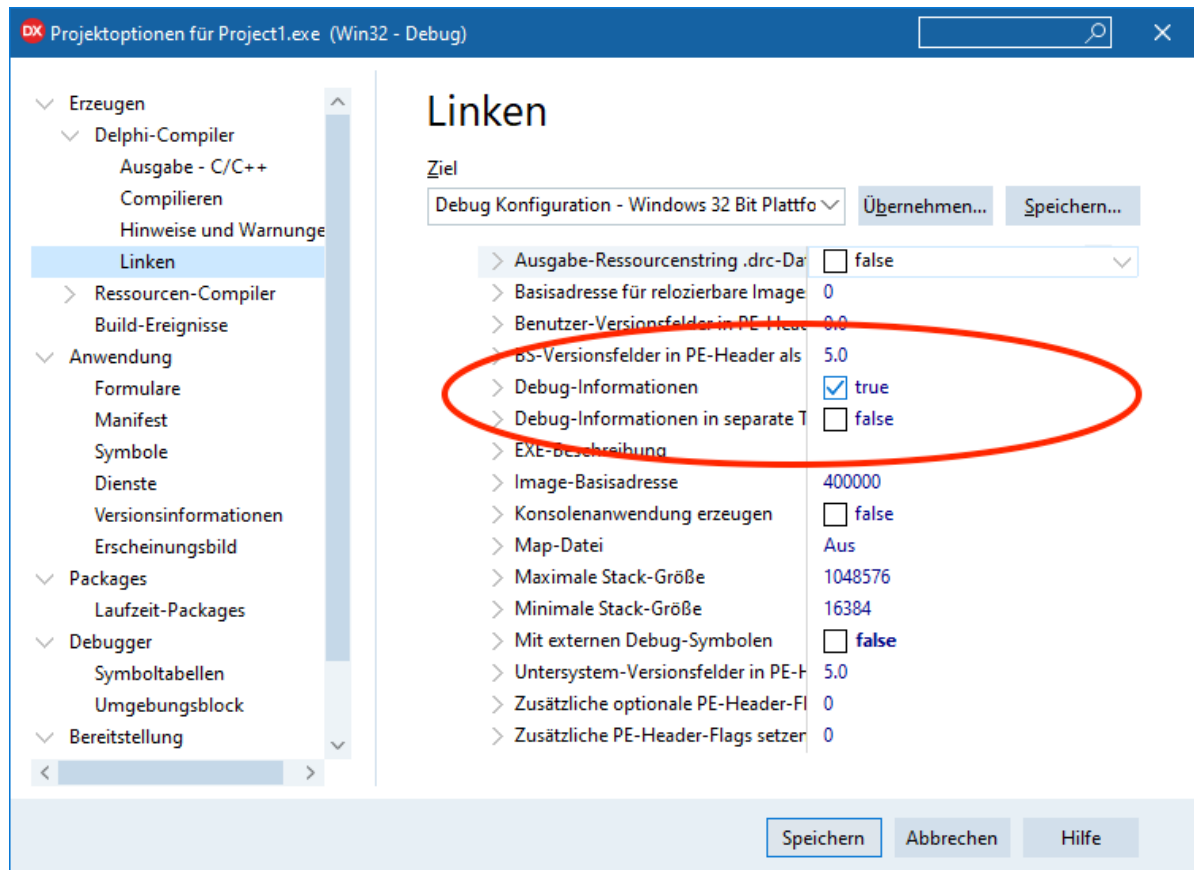
Wir nehmen noch einige ergänzende Einstellungen vor:

Im Unterpunkt Delphi-Compiler/Kompilieren aktivieren wir unter Laufzeitfehler die Bereichs- und Überlaufprüfung.





Und im Unterpunkt Delphi-Compiler/Linken aktivieren wir die Debug-Informationen, damit diese in die ausführbare Datei geschrieben werden.



Ist die Überlaufprüfung aktiviert, so wird bei arithmetischen Integer-Operationen auf ein Über- bzw. Unterschreiten der Grenzen des Datentyps geprüft. Falls das im folgenden Konsolenprogramm so ist, wird eine Fehlermeldung erzeugt und angezeigt. Ist die Prüfung deaktiviert, so zeigt sich das kommentierte Verhalten.

```

program Ueberlaufpruefung;
{$APPTYPE CONSOLE}
uses
    System.SysUtils;
var
    IntZahl: Integer;
begin
    try
        IntZahl := High(Integer); //Maximum im Integer-Bereich
        WriteLn(IntZahl);
        IntZahl := IntZahl + 1; //Maximum + 1 => Überlauf = Minimum im Integer-Bereich
        WriteLn(IntZahl);
    except
        on E: Exception do
            WriteLn(E.ClassName, ': ', E.Message);
        end;
    ReadLn;
end.

```

Bereiche von Datentypen können natürlich auch unterlaufen werden. Wollen wir ähnlich gelagerte Fälle mit Integer-Typen kleiner 4 Bytes ausprobieren, dann müssen wir allerdings die Bereichsprüfung und nicht die Überlaufprüfung deaktivieren, da alle arithmetischen Operationen intern mittels 32-Bit-Werten ausgeführt werden. Die Berechnung an sich läuft also fehlerfrei, nur die implizite Typumwandlung verletzt hier, beim Zuweisen des Ergebnisses an den kleineren Datentyp, die Bereichsgrenzen.

```

var
    ByteZahl: Byte;
begin
    try
        ByteZahl := Low(Byte); //Minimum im Byte-Bereich
        WriteLn(ByteZahl);
        ByteZahl := ByteZahl - 1; //Minimum - 1 => Unterlauf = Maximum im Byte-Bereich
        WriteLn(ByteZahl);
    except
        on E: Exception do
            WriteLn(E.ClassName, ': ', E.Message);
        end;
    ReadLn;
end.

```

Sofern solche Überläufe nicht ausdrücklich erwünscht sind, sollten diese Prüfungen im Debug-Modus immer aktiviert sein. Insbesondere die Bereichsprüfung ist hier sehr nützlich für uns, da sie anzeigt, wenn Zugriffe außerhalb der Grenzen z.B. eines Arrays oder Strings stattfinden.

```
program Bereichsprüfung;
{$APPTYPE CONSOLE}
uses
  System.SysUtils;

procedure ZeigeArrayBereichsverletzung;
var
  intArray: array[0..9] of Integer;
  i: Integer;
begin
  for i := 1 to 10 do //Nicht Index 0, dafür Index 10 beschreiben
    intArray[i] := i;
  for i := 0 to 10 do //Index 0..9 + 10 auslesen
    writeln(intArray[i]);
end;

begin
  try
    ZeigeArrayBereichsverletzung;
  except
    on E: Exception do
      writeln(E.ClassName, ': ', E.Message);
    end;
  ReadLn;
end.
```

Im gezeigten Beispiel wird, bei deaktivierter Bereichsüberprüfung, über die Grenzen des Arrays hinaus in den Speicher geschrieben und daraus gelesen. Was immer dort im Speicher stand, es wurde überschrieben und die Konsequenzen daraus sind nicht vorhersehbar. Zusätzlich sehen wir an Index 0 des Arrays einen uninitialisierten Wert.

#### 4.1 Die Elemente des Debuggers

Um Programme mit dem Debugger zu inspizieren, müssen wir natürlich seine Funktionalität kennen. Wir gehen dabei von folgendem, korrekten Konsolenprogramm aus:

```

program Primzahl;
{$APPTYPE CONSOLE}
uses
    SysUtils;

const
    MAX = 100; //Alle Primzahlen bis 100

function IstPrimzahl(APruefzahl: Integer): Boolean;
var
    teiler: Integer;
begin
    Result := True; //Zahl ist Primzahl (Annahme)
    teiler := 2;
    while Result and (Sqr(teiler) <= APruefzahl) do
    begin
        if APruefzahl mod teiler <> 0 then //Nicht ganzzahlig teilbar?
            Inc(teiler)
        else
            Result := False; //Keine Primzahl
        end;
    end;

var
    i: Integer;
begin
    try
        for i := 2 to MAX do
            if IstPrimzahl(i) then
                WriteLn(i);
    except
        on E: Exception do
            WriteLn(E.ClassName, ': ', E.Message);
    end;
    ReadLn;
end.

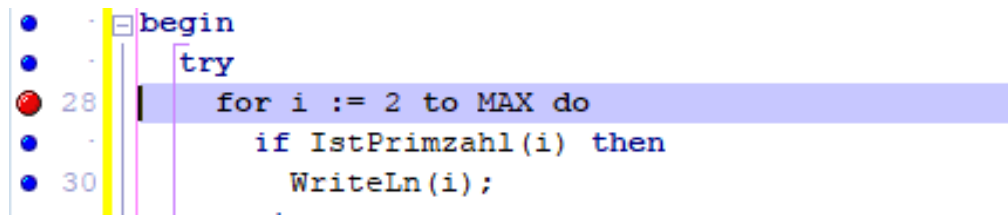
```

Dieses Konsolenprogramm speichern wir und erzeugen das Projekt mit Debug-Informationen. Starten wir nun das Programm mit Debug-Unterstützung durch F9, über den Menüpunkt Start/Start oder mit dem entsprechenden Symbol, dann ersehen wir die Funktion des Programms – es gibt alle Primzahlen <= 100 aus. Darunterliegend erkennen wir das Debug-Layout der Oberfläche, welches außerdem in der Desktop-Symbolleiste ausgewählt werden kann.

#### 4.1.1 Haltepunkte („Break points“)

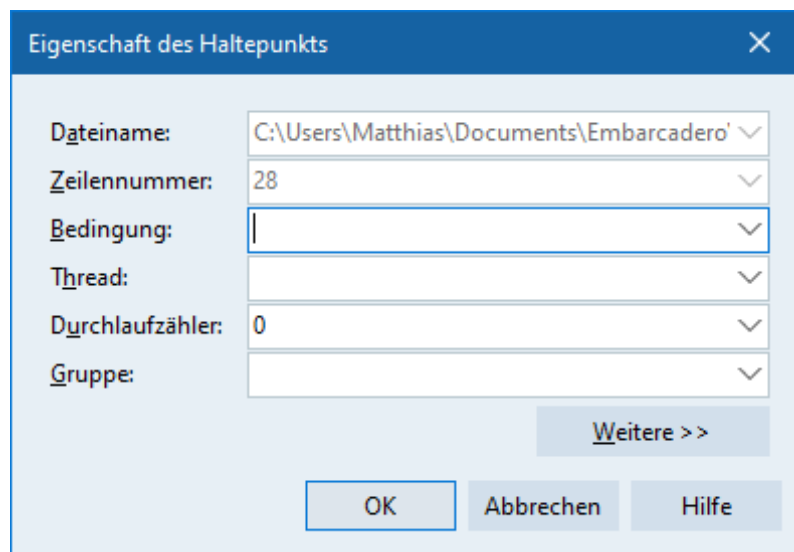
Um zu einem bestimmten Zeitpunkt nähere Informationen zum Programm zu erhalten, müssen wir den Programmlauf anhalten. Das erreichen wir durch Setzen eines Haltepunktes im Quelltext. Mit dem Kompilieren/Erzeugen des Programms erscheinen am linken Rand des Editors die möglichen

Kandidaten für solch einen Quelltexthaltepunkt, gekennzeichnet durch einen blauen Punkt. Klicken wir den Punkt neben der For-Schleife in Zeile 28 an, so sehen wir die farblich hinterlegten Merkmale eines aktivierten Haltepunktes.



Bei Starten des Programms mit F9 stoppt nun die Ausführung immer vor Abarbeitung dieser Anweisung in Zeile 28. Der blaue Pfeil und die Hinterlegung im Quelltext kennzeichnen die nächste auszuführende Zeile.

Wir können solche Haltepunkte aber auch mit Bedingungen versehen. Die Programmausführung stoppt dann also nicht immer, sondern nur, wenn die gesetzte Bedingung erfüllt ist. Dazu tätigen wir einen Rechtsklick auf den bereits gesetzten Haltepunkt und wählen „Eigenschaften des Haltepunkts...“ aus.



Als Bedingung können wir hier boolesche Ausdrücke wie z.B. „ $i \bmod 10 = 0$ “ oder „ $i \geq 30$ “ eintragen. Im ersten Fall ergibt die Bedingung True, wenn  $i$  die Werte 10, 20, ..., 100 annimmt und im zweiten Fall, bei Werten von 30, 31, ..., 100.

Eine weitere auswählbare Bedingung wäre in diesem Fenster der Durchlaufzähler. Tragen wir dort z.B. eine „10“ ein, dann stoppt die Anwendung, bevor die Haltepunktzeile zum 10. Mal abgearbeitet wird. Da die Schleife bei 2 beginnt, wäre zu diesem Zeitpunkt  $i=11$ . Danach wird der Durchlaufzähler wieder auf 0 gesetzt, und das Zählen der Durchläufe beginnt von vorn.

Die Verwaltung der Haltepunkte ist im unteren Bereich des Debug-Layouts zu finden. Dort, wie auch am Haltepunkt selbst, können die gesetzten Eigenschaften eingesehen und verändert werden.

Neben den Quelltexthaltepunkten haben wir zusätzlich die Möglichkeit, Adresshaltepunkte zu setzen. Hier hält die Ausführung an, wenn eine zuvor angegebene Speicheradresse im Arbeitsspeicher angesprochen wird. Das Setzen eines Adresshaltepunktes ist nur zur Laufzeit möglich und erreichbar über den Menüpunkt Start/Haltepunkt hinzufügen/Adresshaltepunkt. Interessant sind solche Adresshaltepunkte dann, wenn wir Fehlermeldungen erhalten, wo konstant auf eine bestimmte Speicheradresse zugegriffen werden soll.


### 4.1.2 Durchschreiten des Quelltextes

Die Haltepunkte allein nutzen relativ wenig. Wir müssen uns natürlich auch, nachdem wir die Ausführung angehalten haben, durch den Quelltext bewegen können. Steuern können wir das im Menüpunkt Start oder durch die Tasten F4, F7, F8 und F9.

F9 zeigt das bekannte Verhalten, die Anwendung läuft durch bzw. hält bei anfallenden Eingaben oder Haltepunkten.

Mit F4 läuft das Programm bis zur aktuellen Cursor-Position, wiederum unter Berücksichtigung der angesprochenen Einschränkungen.

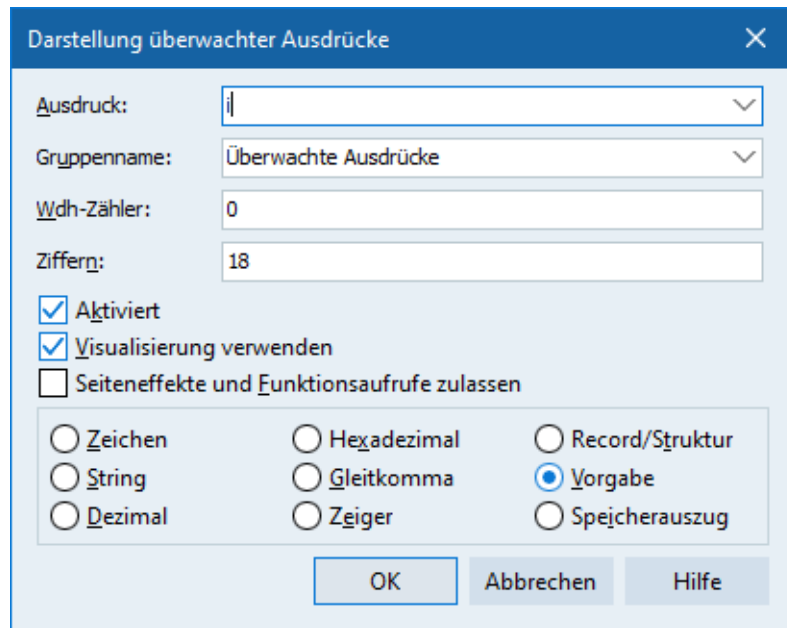
Für uns interessanter ist das Verhalten bei F7 und F8. Mit F8 wird nur die nächste auszuführende Zeile komplett abgearbeitet, bei F7 wird dabei in eine etwaige Routine verzweigt, sofern der Quelltext davon vorliegt.

Wir können das unterschiedliche Verhalten, mit dem Haltepunkt in Zeile 28 beleuchten. Mit dem Starten des Programms durch F9 gelangen wir zum Haltepunkt. Mit F9 können wir nun den gesamten Schleifendurchgang, das Prüfen und Schreiben, auf einmal erledigen und erreichen so wieder direkt den Haltepunkt. F8 steuert jede Zeile einzeln an und F7 verzweigt gar in die Funktion `IstPrimzahl` hinein. Da wir hier schrittweise arbeiten, müssen wir auch jeden Schritt, per Tastendruck, auslösen. Die Abarbeitung endet mit Ende des Programms oder dem Drücken von Strg+F2 bzw. dem Klick auf das Icon von „Programm abbrechen“  in der Symbolleiste zur Fehlersuche.

### 4.1.3 Überwachte Ausdrücke

Nachdem wir nun wissen, wie die Programmausführung gezielt angehalten und auch wieder fortgesetzt werden kann, widmen wir uns dem Inspizieren der Daten. Wir können durch den Menüpunkt Start/Ausdruck hinzufügen oder durch Strg+F5 der Liste überwachter Ausdrücke einen weiteren hinzufügen – je nach Cursor-Position und Markierungen im Quelltext auch ohne weitere Eingabe und Bestätigung. Die Verwaltung dieser Ausdrücke findet im linken Bereich des Debug-Layouts statt.

Mit unserem Haltepunkt in Zeile 28 starten wir nun das Programm durch F9. Mittels Strg+F5, Eingabe von `i` als Ausdruck und Bestätigung, fügen wir die Variable `i` den überwachten Ausdrücken hinzu.

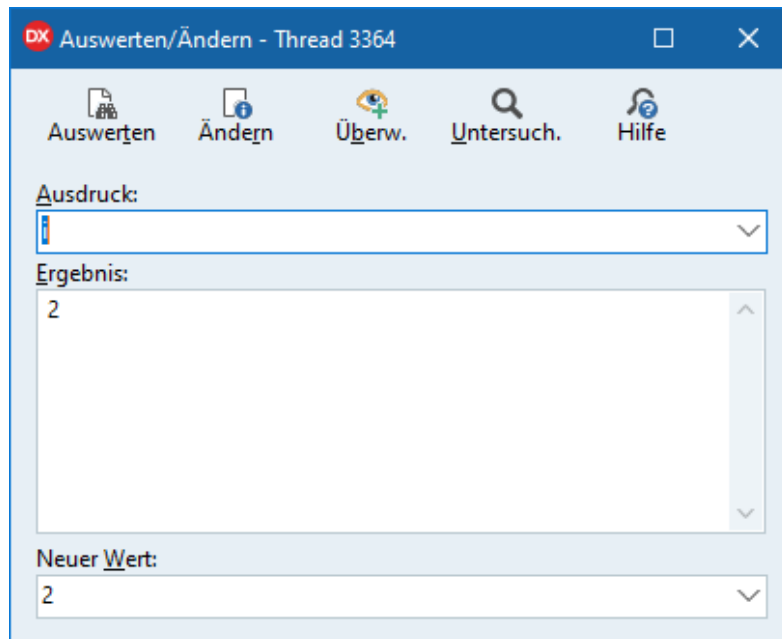


Zudem markieren wir den Ausdruck „APruefzahl mod teiler“ aus der Funktion `IstPrimzahl` in Zeile 17 und übernehmen ihn durch Strg+F5 direkt in die Liste. In der Liste der überwachten Ausdrücke sehen wir nun die beiden Ausdrücke und erkennen auch direkt eine Problemstellung – den Gültigkeitsbereich. Da wir uns momentan im Hauptprogramm befinden, kann die Restwertdivision, mit ihren lokalen Variablen, nicht ausgewertet werden. Erst wenn wir mit F7 in die Funktion hineinspringen, werden die lokalen Variablen gültig und dadurch der Ausdruck auswertbar.

Wir können außerdem Funktionsergebnisse auswerten und darstellen lassen. Dazu markieren wir den Aufruf „`IstPrimzahl(i)`“ in Zeile 29 und ergänzen die Liste überwachter Ausdrücke durch Strg-F5. Gehen wir jetzt schrittweise durch den Text, dann ist der Wert jedoch nicht verfügbar. Mit einem Rechtsklick auf den entsprechenden Listeneintrag und der Auswahl von Ausdruck bearbeiten bzw. durch Strg-E bei selektiertem Listeneintrag, erhalten wir seine Eigenschaften. Dort ändern wir die Standardeinstellungen und aktivieren „Seiteneffekte und Funktionsaufrufe zulassen“. Ab jetzt wird die Funktion, bei jedem Schritt im Quelltext ausgewertet und dargestellt, sofern dies möglich ist. Dabei sollte jedoch beachtet werden, dass das Ergebnis des Programmlaufs durch die ständige Auswertung der Funktion verfälscht werden kann. Nämlich dann, wenn durch den Aufruf der Funktion Daten bleibend verändert werden.

Über den Menüpunkt Start/Auswerten/Ändern bzw. durch Strg+F7 ist es uns zusätzlich möglich, den Wert von Ausdrücken zu ändern und somit deren Ergebnis zu manipulieren.





Die Aufnahme eines Ausdrucks geschieht identisch zu Strg+F5. Die Schaltfläche Auswerten erzeugt dabei im Anzeigefeld Ergebnis den aktuellen Rückgabewert, welchen wir im darunterliegenden Feld editieren können. Durch die Schaltfläche Ändern wird der Wert aktualisiert und hat damit Bestand bis zur nächsten Auswertung des Ausdrucks oder bis zum Verlust seines Gültigkeitsbereichs.

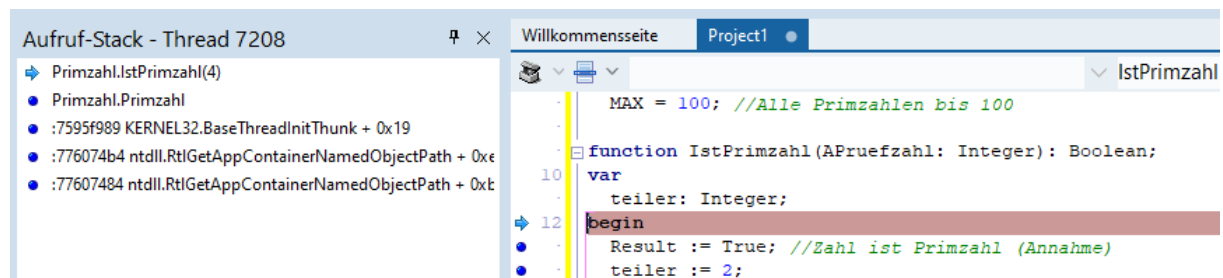
Die Anzeige selbst wird beim Bewegen durch den Quelltext nicht aktualisiert, sondern muss durch die Schaltfläche Auswerten jeweils angestoßen werden. Anders als die Liste überwachter Ausdrücke, eignet sich also dieses Fenster nur zum punktuellen Auswerten bzw. Ändern von Ausdrücken und nicht zur dauerhaften Überwachung.

Durch die Schaltfläche Überwachen können wir den aktuellen Ausdruck der Liste überwachter Ausdrücke hinzufügen.

#### 4.1.4 Aufruf-Stack

Im linken oberen Bereich des Debug-Layouts befindet sich die Darstellung aller aufgerufenen Routinen bis hin zur aktuellen Position. Der oberste Eintrag zeigt somit die Routine an, in der wir uns gerade befinden, der Eintrag darunter, durch welche Routine diese aufgerufen wurde. Das setzt sich dann nach unten fort. Die Aufrufreihenfolge, beginnend vom Start des Programms, wird also zeitlich von unten nach oben aufgebaut und ist nur dann einsehbar, wenn der Programmlauf durch einen Haltepunkt unterbrochen wurde.

Schauen wir uns den Aufruf-Stack unseres Programms, unterbrochen durch den Haltepunkt in Zeile 28 und fortgesetzt mit F7, beim Eintritt in die Funktion IstPrimzahl an:



Oben in der Liste sehen wir wie erwartet die Funktion IstPrimzahl. Wir erkennen ebenfalls den übergebenen Parameter, ausgewertet zum Betrachtungszeitpunkt. Würde dieser Parameter in der aufgerufenen Routine bearbeitet werden, so würde sich bei der schrittweisen Bewegung im Quelltext auch die Darstellung dieses Wertes im Aufruf-Stack ändern. In der zweiten Zeile wird die aufrufende Routine Primzahl gelistet, was unserem Hauptprogramm entspricht. Darunter sehen wir dann Initialisierungsarbeiten beim Programmstart, welche für uns, erkennbar durch den fehlenden blauen Punkt am Zeilenanfang, nicht im Quelltext erreichbar sind. Im Umkehrschluss können wir durch Doppelklick bzw. "Quelltext anzeigen" im Kontextmenü des 2. Eintrags im Aufruf-Stack, die aufrufende, dann hervorgehobene Stelle im Quelltext lokalisieren. Hier gibt es einen Unterschied zu einer VCL-Anwendung, denn dort wird die auf den Aufruf folgende Zeile markiert. Zudem wäre dort der Aufruf-Stack deutlich umfangreicher, da in einer Formularanwendung erheblich mehr Initialisierungsarbeit geleistet werden muss.

## 4.2 Verhalten bei Fehlern

Es ist unvermeidlich, dass Programmierer Fehler produzieren. Diese Fehler dann zu finden ist eine Sache der Erfahrung. Trotzdem gelingt es nicht immer, einen Fehler schnell zu lokalisieren und solch eine Fehlersuche kann sich auch manches Mal über Stunden hinziehen.

Kein Grund zu verzweifeln: Zumeist genügt es, eine Pause einzulegen, um mit klarem Kopf die Fehlersuche wieder aufzunehmen und die eigene Strategie in Frage zu stellen.

### 4.2.1 Hinweise und Warnungen

Bevor es mit den Fehlern losgeht, ein paar Worte zu einem, von Einsteigern oft unterschätzten Thema: Der Compiler gibt uns nicht nur Fehlermeldungen aus, er macht uns auch auf mögliche Fehlerquellen aufmerksam. Syntaktisch kann der Quelltext also in Ordnung sein, möglicherweise enthält er jedoch Fehler in der Programmlogik.

Unser Ziel muss es demnach sein, dass solche Meldungen gar nicht erst entstehen, damit potentielle Fehlerquellen von vornherein ausgeschlossen werden.

Betrachten wir die vom Compiler erzeugten Hinweise folgender Funktion:

```
function Maximum(AZahl1, AZahl2: Integer): Integer;  
var  
    i: Integer; //H2164  
begin  
    Result := 0; //H2077  
    if AZahl1 > AZahl2 then  
        Result := AZahl1  
    else  
        Result := AZahl2;  
end;
```

[dcc32 Hinweis] HinweiseUndWarnungen.dpr(5): H2077 Auf 'Maximum' zugewiesener Wert wird niemals benutzt  
[dcc32 Hinweis] HinweiseUndWarnungen.dpr(3): H2164 Variable 'i' wurde deklariert, aber in 'Maximum' nicht verwendet

Der untere Hinweis ist schnell abgehandelt, denn offensichtlich wurde hier nur vergessen, die Variablendeklaration von i zu entfernen. Ein Doppelklick auf diesen Hinweis im Meldungsfenster führt uns direkt zur Fundstelle im Quelltext, also zur Deklaration der überflüssigen Variable i in der angegebenen, auf die gesamte Datei bezogenen Zeile – hier und in der Folge allerdings angepasst auf den jeweiligen Textausschnitt.

Markieren durch Einfachklick und Anfordern der Hilfe durch Drücken von F1 zeigen uns weitere Informationen zu diesem Hinweis, ebenfalls zu erreichen durch Aufrufen der Hilfe und suchen nach der Meldungsnummer H2164.

Auch die Hinweismeldung der oberen Zeile spricht bereits für sich. Result wird in allen Fällen ein neuer Wert zugewiesen und damit ist die Initialisierung in Zeile 5 überflüssig und kann ebenfalls entfernt werden. Wie die Meldung außerdem zeigt, können innerhalb von Funktionen der Funktionsname und Result gleichrangig verwendet werden. Aus Gründen der Einheitlich- und Übersichtlichkeit verwenden wir jedoch ausschließlich Result.

Während Hinweise meist stilistischer Natur sind, deuten Warnungen auf mögliche Fehlerquellen hin.

```
function Potenz(ABasis: Integer; AExponent: Cardinal): Integer;  
var  
    i: Integer;  
begin  
    //Result := 1; //Fehlende Initialisierung  
    for i := 1 to AExponent do  
        Result := Result * ABasis; //Lesender Zugriff auf Result  
end;
```

```
[dcc32 warnung] HinweiseUndWarnungen.dpr(8): w1035 Rückgabewert der Funktion 'Potenz' könnte undefiniert sein
```

Abgesehen davon, dass man sich hier um die mögliche Größe des Ergebnisses wenig Gedanken macht, wird `Result` nicht mit dem richtigen Wert vorbelegt. Der lesende Zugriff darauf liefert also dort einen Zufallswert und stellt einen Sonderfall einer nicht initialisierten, lokalen Variable dar. Wäre die Funktion eine Prozedur und `Result` eine lokale Variable darin, dann würde sich folgende Warnmeldung ergeben:

```
[dcc32 warnung] HinweiseUndWarnungen.dpr(8): w1036 variable 'Result' ist möglicherweise nicht initialisiert worden
```

Glücklicherweise werden wir vom Compiler auf solche Schnitzer aufmerksam gemacht. Nicht immer ist jedoch direkt klar, was an unserem Quelltext eine Warnung hervorrufen sollte:

```
function Signum(AZahl: Integer): Integer;
begin
  if AZahl < 0 then
    Result := -1
  else
    if AZahl > 0 then
      Result := 1
    else
      if AZahl = 0 then //stiftet verwirrung
        Result := 0;
end;
```

```
[dcc32 warnung] HinweiseUndWarnungen.dpr(11): w1035 Rückgabewert der Funktion 'Signum' könnte undefiniert sein
```

Offensichtlich ist hier jeder mögliche Fall für `AZahl` abgehandelt worden und `Result` wird auch das richtige Ergebnis zugewiesen. Der Compiler erkennt jedoch in der letzten `If`-Anweisung eine Bedingung, für die es dann natürlich auch eine Alternative innerhalb eines `Else`-Zweiges geben könnte. Abhilfe schafft hier das einfache Entfernen der überflüssigen Bedingung, denn das abschließende `Else` behandelt alle anderen Fälle.

Gerade fehlende oder falsche `Else`-Zweige innerhalb von `If`- und `Case`-Anweisungen führen immer wieder zu schwer lokalisierbaren Fehlern. Handelt es sich nämlich nicht gerade um einen Initialwert, dann werden wir auch nicht auf einen fehlenden oder semantisch falschen Wert in der Fallunterscheidung hingewiesen. Ist man sich hier unsicher, so könnte man einfach im abschließenden `Else` eine Exception erzeugen, die auf den unbehandelten Wert in der Fallunterscheidung hinweist.

Häufig sieht man auch folgende Warnmeldung:

```
var
  sTemp: AnsiString; //sTemp: string;
begin
  sTemp := '123';
  writeln(StrToInt(sTemp)); //Implizite Typumwandlung
```

```
[dcc32 warnung] HinweiseUndWarnungen.dpr(5): w1057 Implizite String-Umwandlung von
'AnsiString' zu 'string'
```

Seit Delphi 2009 verweist der Alias String nicht mehr auf einen AnsiString, sondern auf einen UnicodeString. Da die internen String-Funktionen natürlich weiterhin mit dem Alias arbeiten, wandelt Delphi den AnsiString in einen UnicodeString um, was, in diese Richtung umgewandelt, auch nicht weiter fehleranfällig ist. Trotzdem bereinigen wir den Quelltext, in dem wir mittels `string(sTemp)` den Parameter für `StrToInt` explizit umwandeln oder direkt `sTemp` als `string` deklarieren. Das nackte Ergebnis der beiden Möglichkeiten ist erstmal gleich gut – die Warnung verschwindet.

Typumwandlungen haben aber immer eine latente Fehleranfälligkeit, denn man muss die beiden Typen bewerten können, um zu wissen, ob hier unter Umständen Datenverlust möglich ist. Außerdem sollte man, solange kein gegenteiliger Grund vorliegt, immer mit den generischen bzw. dynamischen Typen arbeiten, denn der Compiler ist darauf optimiert. Eine Ausnahme wäre z.B. die programmexterne Kommunikation mit einer Datei, bei der fundamentale bzw. statische Typen mit ihrer konstanten Darstellungsbreite, über Delphi-Versionen hinweg, angebracht sind.

Wie auch in den nächsten Unterkapiteln, können wir hier nur einige wenige Meldungen exemplarisch zeigen. Aber allein das Lesen und Verstehen der Hinweise und Warnungen sollte uns, in Kombination mit dem entsprechenden Hilfetext, zur Lösung des Problems führen.

#### 4.2.2 Fehler zum Zeitpunkt der Kompilierung

Ein Programmierer kann ohne Einsicht in die Hilfe nicht programmieren, denn niemand hat alle Deklarationen und deren Aufbau und Zusammenhänge im Kopf. Syntaxfehler, welche jetzt behandelt werden, widersprechen diesem Aufbau und damit einer erfolgreichen Überprüfung durch den Compiler. Da aber alle Sprachmerkmale von Delphi in der Hilfe beschrieben sind, ist das unser Ansatzpunkt beim Beheben von Fehlern zum Kompilierungszeitpunkt.

Der wohl häufigste Fehler, ist ein unbekannter und damit undeklartierter Bezeichner:

```
program Syntaxfehler;  
{$APPTYPE CONSOLE}  
uses  
    System.SysUtils{, System.Math};  
  
var  
    Eingabe: Integer;  
    Ausgabe: Integer; //Ausgabe: TValueSign;  
begin  
    try  
        write('Ganzzahl: ');  
        ReadLn(Eingabe); //Tippfehler  
        Ausgabe := Sign(Eingabe); //Unit Math nicht eingebunden  
        WriteLn('Signum= ', Ausgabe);  
    except  
        on E: Exception do  
            WriteLn(E.ClassName, ': ', E.Message);  
        end;  
        ReadLn;  
    end.  
end.
```

```
[dcc32 Fehler] Syntaxfehler.dpr(12): E2003 Undeklariertes Bezeichner: 'ReadLn'  
[dcc32 Fehler] Syntaxfehler.dpr(13): E2003 Undeklariertes Bezeichner: 'Sign'
```

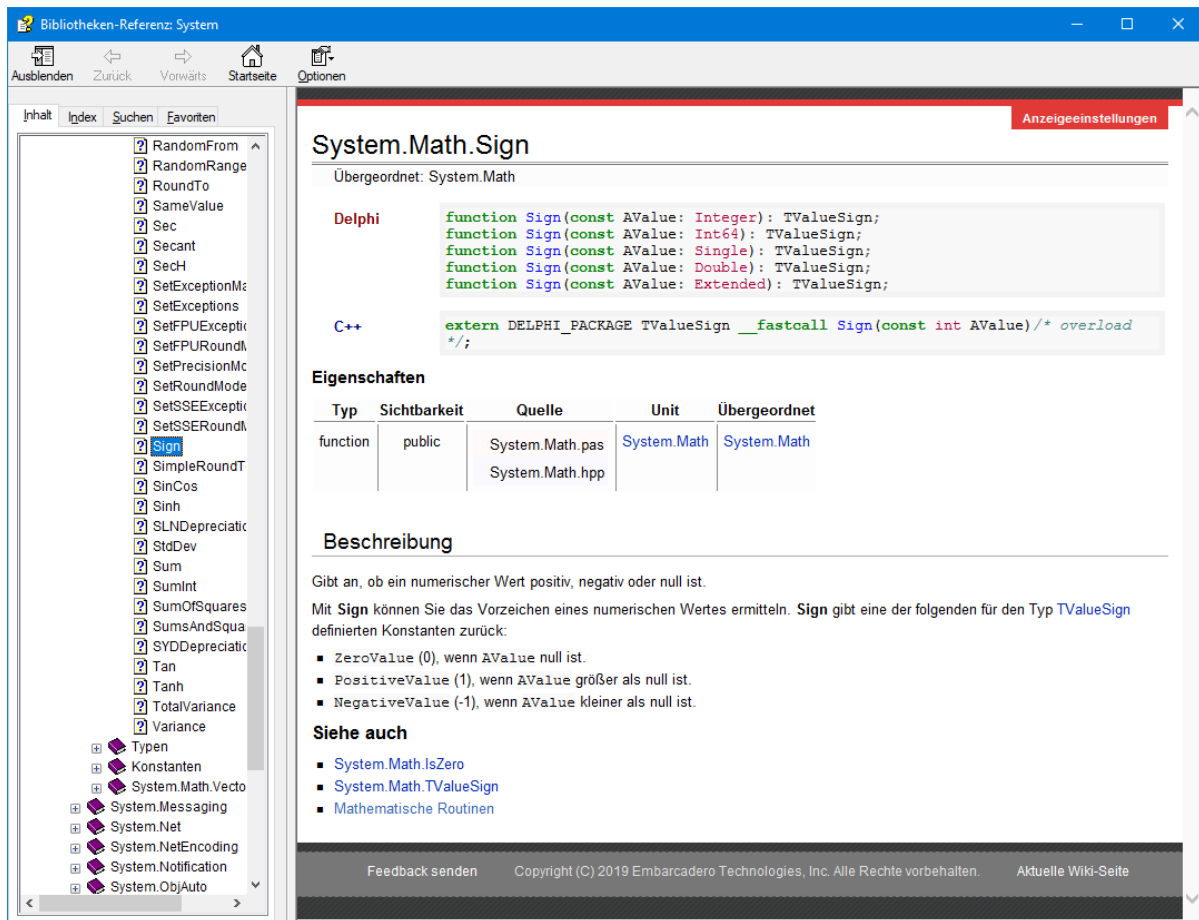
Alle Deklarationen – also Konstanten, Funktionen, Klassen etc. – werden in Units abgelegt. Mit Ausnahme der Unit System, welche automatisch eingebunden wird, müssen wir dem Compiler durch Einbinden der entsprechenden Unit diese Deklarationen zugänglich machen. Findet der Compiler dann einen Bezeichner nicht, so kann das nur drei Gründe haben:

- Der Bezeichner existiert grundsätzlich nicht, wie z.B. bei einem Tippfehler,
- die Sichtbarkeit in der Unit ist an dieser Stelle nicht vorhanden, so z.B. beim Vertauschen der Reihenfolge von definierenden Deklarationen bzw. einer fehlenden Forward-Deklaration oder
- die benötigte Unit ist nicht eingebunden.

Dass ReadLn nicht durch die Hilfe zu erfassen ist, liegt an einem simplen Tippfehler im Quelltext.

Ein weiterer Grund für das Nichtauffinden eines Hilfeeintrags könnte sein, dass es sich um Bestandteile der Windows-Anwendungs-Programmierschnittstelle handelt. Um die Dokumentation der WinAPI einzusehen, muss im Menüpunkt Hilfe/Plattform SDK-Hilfe/Windows Plattform SDK-Bibliothek nach dem Begriff suchen.

Da wir einen Hilfeeintrag zu Sign vorfinden, interessiert uns die dort angegebene und von uns bisher nicht berücksichtigte Unit Math. Zusätzlich sehen wir eine Beschreibung der Funktionalität und der Eigenschaften von Sign, sowie die überladenen Deklarationen der Funktion.



Hier erkennen wir im Rückgabetypp `TValueSign` zunächst einen Widerspruch zu unserer Variablendeklaration von Ausgabe, welche vom Typ `Integer` ist. Der im Hilfeintrag zu `TValueSign` ersichtliche `Integer`-Teilbereichstyp wird hier aber niemals Probleme machen, weil ein `Integer` einen Teilbereich seiner selbst natürlich vollständig aufnehmen kann. Wenn man allerdings bedenkt, dass alle `Integer`-Typen untereinander zuweisungskompatibel sind, so könnte man hier auch einen vorzeichenlosen `Cardinal`, als Typ der Rückgabe, fehlerfrei kompilieren lassen.

Das Überführen von Daten in einen anderen Datentyp sollte somit immer sorgfältig geprüft werden – ein erfolgreiches Kompilieren allein reicht nicht aus!

Vielen Typumwandlungen schiebt aber bereits der Compiler einen Riegel vor:

```
var
  iwert: Integer;
  fwert: Real;
begin
  iwert := 1;
  fwert := iwert; //zuweisungskompatibel
  iwert := Integer(fwert); //Trunc(fwert);
```

[dcc32 Fehler] Syntaxfehler.dpr(7): E2089 Ungültige Typumwandlung

Während der Integer-Wert noch zuweisungskompatibel zum Real-Wert ist, muss man für den umgekehrten Weg schon die Hilfe der Funktion Trunc in Anspruch nehmen, denn Integer- und Real-Werte können generell nicht per Typumwandlung ineinander überführt werden.

Andere häufig auftretende Fehlermeldungen, auf die man gerade als Einsteiger trifft:

```
program Syntaxfehler;
{$APPTYPE CONSOLE}
uses
  System.SysUtils;

var
  izahl1, izahl2: Integer;
begin
  try
    Randomize //Fehlendes Semikolon
    izahl1 := Random(10);
    izahl2 := Random(1,0); //2 Parameter
    if izahl1 >> izahl2 then //Nur >
      writeln(izahl1, '>', izahl2); //Abschluss der Anweisung durch ;
    else
      writeln(izahl2, '>=', izahl1);
  except
    on E: Exception do
      writeln(E.ClassName, ': ', E.Message);
  end;
  ReadLn;
end.
```

[dcc32 Fehler] Syntaxfehler.dpr(11): E2066 Operator oder Semikolon fehlt  
[dcc32 Fehler] Syntaxfehler.dpr(12): E2034 Zu viele Parameter  
[dcc32 Fehler] Syntaxfehler.dpr(13): E2029 Ausdruck erwartet, aber '>' gefunden  
[dcc32 Fehler] Syntaxfehler.dpr(15): E2153 ';' nicht erlaubt vor einem 'ELSE'

Aufgelistet sind hier ausnahmslos Tipp- und Flüchtigkeitsfehler, und die Fehlermeldungen sprechen alle für sich. Interessant dabei ist, dass die Meldungen nicht immer auf die entsprechende Zeile verweisen.

Delphi erlaubt es, dass Anweisungen über mehrere Zeilen hinweg geschrieben werden dürfen und erst ein Semikolon beendet eine solche Anweisung. Fehlt ein Semikolon, so geht der Compiler zunächst davon aus, in der folgenden Zeile einen Operator zum Verknüpfen der Anweisungen vorzufinden. Ist das nicht der Fall, so ergibt sich der Widerspruch, dann jedoch in dieser nachfolgenden Zeile.



Gerade Fehlermeldungen mit fehlenden Semikolons gibt es in einigen Ausprägungen und Fehler dieser Art können auch Folgefehler hervorrufen. Deswegen ist es angebracht, solche Fehlerlisten von oben herab abzuarbeiten und zwischendurch einfach mal neu zu kompilieren.

### 4.2.3 Interpretieren von Laufzeitfehlern


Laufzeitfehler können sich nicht nur für Einsteiger zu einem echten Problem entwickeln. Die Art und Weise wie sie entstehen, sind vielfältig, und für deren Lösung gibt es kein Patentrezept, denn neben Programmierfehlern können hier z.B. auch hardware-spezifische Faktoren eine Rolle spielen. Bei Zugriffsverletzungen kann es sich um Speicherbereich handeln, der schon sehr viel früher im Quelltext falsch angesprochen wurde, die Auswirkungen darauf aber erst später zum Tragen kommen und das möglicherweise bei verschiedenen Programmläufen an unterschiedlichen Stellen. Daher ist es wichtig, die Fehlermeldung richtig einzuordnen, insbesondere, um in einem eventuellen Debugging-Prozess die notwendigen Rückschlüsse ziehen zu können.

Betrachten wir folgenden Klassiker der Laufzeitfehler, eine Division durch 0:

```
program Laufzeitfehler1;
{$APPTYPE CONSOLE}

var
  i: Integer;
begin
  i := 0;
  writeln('1 geteilt durch 0 = ', 1 div i); //Division durch 0
  ReadLn;
end.
```

Speichern und erzeugen wir das Projekt und starten das Programm **ohne** Unterstützung des

Debuggers  (Umsch + Strg + F9), so sehen wir nur ein kurzes Aufflackern der Konsole – das Programm terminiert direkt, trotz der abschließenden Aufforderung zum ReadLn. Was wirklich passiert ist, können wir hier nicht erkennen. Dazu starten wir eine separate Konsole und führen die erzeugte Datei Laufzeitfehler1.exe dort direkt und somit außerhalb von Delphi aus.

Durch Ausführen von cmd.exe im Windows-Startmenü, startet die Eingabeaufforderung im Verzeichnis des Benutzerprofils. Die ausführbaren Dateien werden standardmäßig im Unterordner Win32\Debug\ des in der Umgebungsvariablen BDSPROJECTSDIR angegebenen Verzeichnisses gespeichert. Einzusehen ist dieser Wert im Hauptmenü unter Tools/Optionen/Umgebungsoptionen/Umgebungsvariablen. Wurde das Projekt nicht gezielt an einem anderen Ort gespeichert oder diese Variable geändert, dann wechseln wir nun, exemplarisch auf einem deutschen Windows 7, mit

```
cd "Documents\Embarcadero\Studio\Projekte\Win32\Debug"
```

ins entsprechende Verzeichnis und starten das Programm durch Eingabe von

```
Laufzeitfehler1.exe
```

Wir erhalten folgende Fehlermeldung: In der Konsolenausgabe einen Laufzeitfehler 200 an einer bestimmten Adresse.

```
1 geteilt durch 0 = Runtime error 200 at 0040B11D
```

Diese Fehlerausgabe wurde ebenfalls durch das Betriebssystem erzeugt und nicht etwa durch Delphi. Das ist auch nicht weiter verwunderlich, da wir im Quelltext überhaupt keine Fehlerbehandlung implementiert haben.

Fehler werden solange in der Hierarchie nach oben weitergereicht, bis sich jemand dafür verantwortlich fühlt, d.h. eine entsprechende Fehlerbehandlung vorgesehen hat. In letzter Instanz also Windows selbst, das die Anwendung in einem geschützten Speicherbereich ausführt. Was genau die Ursache des Programmabsturzes war, bleibt uns bei einer solchen Fehlermeldung natürlich verborgen. Es muss jedoch unser Anliegen sein, solche Programmabstürze generell abzufangen.

Die Funktionalität, die dafür notwendig ist und die Laufzeitfehler in Exceptions umwandelt, liegt in der Unit Sysutils. Was bei einer Windows-VCL-Anwendung jedoch noch weitgehend in der Methode Application.Run vom Benutzer ferngehalten wird, ist in einer Konsolenanwendung direkt im Quelltext sichtbar: Das Umschließen des relevanten Quelltextes mit einer Try-Except-Anweisung. Im Gegensatz zur Formularanwendung ist der Programmablauf hier aber nicht ereignisgesteuert, sondern folgt linear dem Hauptprogramm. Daraus ergibt sich die Notwendigkeit, eben dieses Hauptprogramm vor Laufzeitfehlern, welche zu Exceptions bzw. Ausnahmen führen, zu schützen.

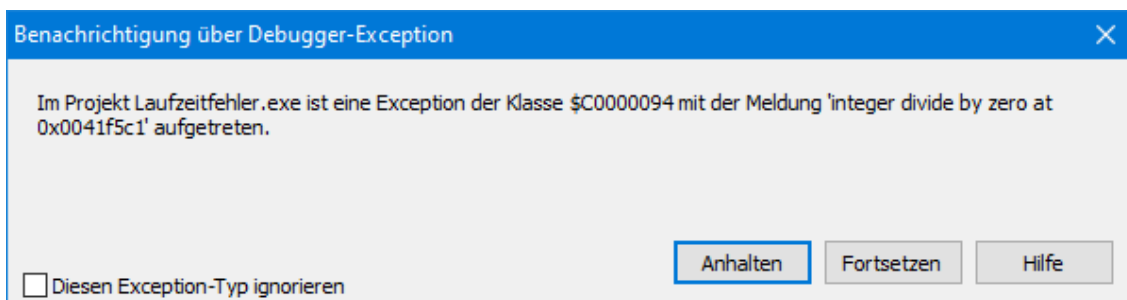
Wir ergänzen also den Text von oben, mit dem von Delphi vorgesehenen Gerüst der Fehlerbehandlung und starten das Programm erneut mit Unterstützung des Debuggers.

```

program Laufzeitfehler;
{$APPTYPE CONSOLE}
uses
    System.SysUtils;

var
    i: Integer;
begin
    try
        i := 0;
        writeln('1 geteilt durch 0 = ', 1 div i); //Division durch 0
    except
        on E: Exception do
            writeln(E.ClassName, ': ', E.Message);
        end;
    ReadLn;
end.

```



Diese Meldung stammt nun von der Delphi-Laufzeitumgebung, in der unser Programm ausgeführt wird. Der Unterschied zur Windows-Ausgabe ist deutlich, denn es wird uns hier ein klarer Grund für die Ausnahme genannt. Wir wissen also jetzt, was passiert ist, kennen aber noch nicht die Stelle des Quelltextes, an der die Ausnahme aufgetreten ist. Der Dialog bietet uns aber zwei erwähnenswerte Möglichkeiten: Anhalten und Fortsetzen.

Fortsetzen lässt das Programm einfach weiterlaufen. Es verzweigt darauf in die Ausnahmebehandlung und wartet dann auf das abschließende Return. Das Programm stürzt also nicht ab, der Fehler wurde abgefangen. Da Konsolenanwendungen jedoch selten interaktiv sind und üblicherweise über Programmparameter gesteuert, direkt aus der Konsole gestartet werden, ist ein abschließendes ReadLn eher unerwünscht und hier nur zum Offenhalten der Konsole eingebaut.

```
1 geteilt durch 0 = EDivByZero: Division durch Null
```

Diese Ausgabe schlussendlich stammt von unserem Programm und wäre auch das Einzige, das wir sehen würden, würden wir Laufzeitfehler.exe in einer separaten Konsole starten. Dadurch erkennen wir, dass die komplette Fehlerbehandlung, durch Einbinden der Unit Sysutils, im Programm vorhanden ist.

Die Meldung selbst listet uns die Fehlerklasse und die entsprechende Fehlermeldung. Was weiterhin fehlt ist der Ort des Fehlers.

Dazu wählen wir im Dialog nicht Fortsetzen, sondern Anhalten. Im Quelltext wird dadurch die Zeile markiert, in der die Ausnahme aufgetreten ist. Hier können wir jetzt die Variablen auswerten oder uns z.B. schrittweise durch den Quelltext weiterbewegen.

In der Praxis ist es aber meist sinnvoller, den Zustand aller relevanten Variablen direkt vor Auftreten des Fehlers zu kennen, sofern dieser konstant in einer Zeile auftritt. Man würde also einen Haltepunkt in einer Zeile darüber setzen und einen neuen Programmablauf damit dort stoppen, die Variablen auswerten und sich dann schrittweise der Problemstelle nähern.

In einer realen Konsolenanwendung würde jede Ausnahme, die ausschließlich auf die gezeigte Art und Weise behandelt wird, schlussendlich zu einem kontrollierten Beenden des Programms führen, da das Programm in den abschließenden Except-Block verzweigt. Das ist ein Unterschied zu einer Windows-VCL-Anwendung, die nach einer von uns unbehandelten Ausnahme ein Meldungsfenster anzeigt und danach weiterhin auf Benutzerinteraktion wartet.

Genauso wie in einer Formularanwendung würden wir aber auch hier einzelne Funktionen und Bereiche, die Ausnahmen erzeugen können, durch eine eigene Fehlerbehandlung schützen und somit einem vorzeitigen Programmende entgehen. Das soll aber nicht heißen, dass dann jede kritische Funktion in einen Try-Except-Block eingeschlossen wird, denn eine Exception-Behandlung kostet Zeit und Ressourcen. Außerdem erzeugen z.B. WinAPI-Funktionen überhaupt keine Exceptions, sondern liefern zumeist einen Rückgabewert, der ausgewertet werden muss.

Im gezeigten Beispiel der Division durch Null wurde der Fehler natürlich hinkonstruiert. Liegt die Division in einer etwas komplexeren Form vor, so würden wir den Divisor in einer eigenen Variablen berechnen und diese gegen 0 prüfen. Solch eine Überprüfung der Eingangsdaten ist ein probates Mittel zur Fehlerminimierung. Genauso sollten beim Austesten der Funktionalität gerade die Randstellen des Wertebereichs ausgiebig untersucht werden. Handelt es sich um Benutzereingaben, dann ist die Validierung der Eingangsdaten sogar unerlässlich, denn neben gezielten Falscheingaben müssen auch immer Tippfehler in die Überlegung mit einbezogen werden.

Schauen wir uns eine solche Datenvalidierung am Beispiel einer Formularanwendung genauer an:

```

unit ULaufzeitfehler2;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms,
  Vcl.Dialogs, Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
  dividend, divisor, division: Integer;
begin
  dividend := StrToInt(Edit1.Text);
  divisor := StrToInt(Edit2.Text);
  if divisor <> 0 then
    begin
      division := dividend div divisor;
      ShowMessage(IntToStr(division));
    end
  else
    ShowMessage('Division durch 0');
  end;

end.

```

Auf dem Formular befinden sich zwei TEdit und ein TButton, deren Namen nicht verändert wurden. Warum eine sinnvolle Namensgebung aber durchaus wichtig ist, wird in einem späteren Abschnitt noch gezeigt. Die beiden TEdit nehmen Dividend und Divisor auf, und der TButton zeigt uns, ausgelöst

durch die entsprechende Ereignisbehandlungsroutine, das Ergebnis bzw. den abgefangenen Fall einer Division durch 0. Die Benutzereingaben wurden dabei allerdings keiner gesonderten Überprüfung unterzogen und werden durch `StrToInt` genauso verarbeitet, wie sie in den `TEdits` stehen. `StrToInt` erwartet dabei einen String, den es in einen Integer umwandeln kann.

Was aber passiert, wenn dieser String eben nicht in eine Ganzzahl umgewandelt werden kann? Wenn z.B. der String leer ist, ein Komma, einen Punkt oder generell irgendwelche Sonderzeichen enthält? Wir testen das durch die Eingabe von „a“. Die Laufzeitumgebung zeigt uns daraufhin folgende Fehlermeldung:

Im Projekt Laufzeitfehler2.exe ist eine Exception der Klasse `EConvertError` mit der Meldung '"a" ist kein gültiger Integer-Wert' aufgetreten.

Wie schon im Beispiel der Division durch 0 gibt uns die Fehlerklasse auch hier an, in welche Kategorie wir den Fehler einordnen können. Fehlerklassen gibt es viele und jede Fehlerklasse ist eine grobe Einteilung in die Art der Fehler. So können unterschiedliche, aber in ihrer Art gleiche Fehler ein und dieselbe Fehlerklasse hervorrufen.

`EConvertError`, wie der Name schon sagt, behandelt dabei Konvertierungsfehler. Dazu zählen z.B. Umwandlungen von String oder zu String, aber auch fehlerhafte Zuweisungen typfremder Komponenten. Hier bietet uns die Hilfe zur entsprechenden Fehlerklasse immer einen grundsätzlichen Überblick an.

Die darauf folgende Fehlermeldung zeigt uns dann einen genaueren Aufschluss zu dem konkreten Fehler. Im Allgemeinen können wir mit diesen Informationen und dem Wissen um die Stelle des Fehlers das Problem genügend eingrenzen und bestenfalls direkt lösen.

"a" ist kein gültiger Integer-Wert

Diese vom Programm erzeugte Fehlermeldung ist aus Benutzersicht die einzige Information, die er vom fertigen Programm erhält, und obwohl der Fehler nicht speziell behandelt wurde ist das Programm weiterhin bedienbar. Trotzdem gilt es natürlich auch hier diesen Fehler abzufangen und gar nicht erst entstehen zu lassen.

Delphi bietet dazu eine ganze Reihe von Funktionen an, die Konvertierungen zwischen Strings auf der einen Seite und Ganzzahlen, reellen Zahlen, Datumswerten und ähnlichem mehr auf der anderen Seite ermöglichen.

Diese Funktionen mit dem Aufbau `TryStrToXXX` und `TryXXXTToStr` geben den Erfolg der Konvertierung als Boolean-Wert zurück. Eine weitere Möglichkeit Strings in einen bestimmten Datentyp zu konvertieren sind Funktionen des Aufbaus `StrToXXXDef`, die bei Misserfolg einen übergebenen Default-Wert zurückgeben.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dividend, divisor, division: Integer;
begin
  if TryStrToInt(Edit1.Text, dividend) and TryStrToInt(Edit2.Text, divisor) then
  begin
    if divisor <> 0 then
    begin
      division := dividend div divisor;
      ShowMessage(IntToStr(division));
    end
    else
      ShowMessage('Division durch 0');
    end
  else
    ShowMessage('Ungültige Ganzzahl-Eingabe');
  end;
end;
```

Kann einer der beiden Strings nicht erfolgreich umgewandelt werden, so wird die Berechnung nicht gestartet, und für den Benutzer erscheint eine aussagekräftige Mitteilung. Im Zuge einer guten Benutzerführung könnte man auch die beiden Edits einzeln auswerten und bei Misserfolg zusätzlich den Fokus dem entsprechenden Edit zuweisen.

Bisher hatten wir es immer mit sehr aussagekräftigen Fehlermeldungen zu tun, die uns schnell der Lösung des Problems näherbrachten. Bei **AccessViolations** bzw. **Zugriffsverletzungen** ist das nicht der Fall. Zugriffsverletzungen entstehen dann, wenn auf Ressourcen zugegriffen wird, die nicht erreichbar oder geschützt sind. In der Praxis bedeutet das, dass wir lesend oder schreibend auf Speicher zugreifen, der dafür nicht vorgesehen ist. In den meisten Fällen werden dabei Zeiger dereferenziert, die nil (not in list) oder ungültig sind:

```
procedure ErzeugeZugriffsverletzung;
var
  sListe: TStringList;
begin
  //sListe := nil;
  sListe.Create;
```

Das Create wird hier nicht an der Klasse aufgerufen sondern fälschlicherweise direkt an der Instanz. Eine Instanz hat aber nur dann eine gültige Adresse, wenn sie erzeugt wurde. Damit misslingt die Dereferenzierung und führt zu folgender Zugriffsverletzung:

Im Projekt Zugriffsverletzung1.exe ist eine Exception der Klasse EAccessViolation mit der Meldung 'Zugriffsverletzung bei Adresse 0043B5FC in Modul 'Zugriffsverletzung1.exe'. Schreiben von Adresse 00421396' aufgetreten.

Die angegebenen hexadezimalen Adressen, welche je nach Compiler-Version, Betriebssystem, Programmaufruf u.ä. variieren können, sind hilfreicher als man das im ersten Augenblick vermuten könnte.

Die erste Adresse (\$0043B5FC) beschreibt den Ort des Fehlers, dort ist die Zugriffsverletzung aufgetreten. Praktischen Nutzen hat diese Adresse als Adresshaltepunkt beim Debugging.

Rückschlüsse zur zweiten Adresse können wir nur ziehen, wenn die Adresse nahe bei 0 liegt. Um das zu verdeutlichen entkommentieren wir im Quelltext die nil-Zuweisung und schauen uns die Fehlermeldung erneut an.

Im Projekt Zugriffsverletzung1.exe ist eine Exception der Klasse EAccessViolation mit der Meldung 'Zugriffsverletzung bei Adresse 0043B5FC in Modul 'Zugriffsverletzung1.exe'. Schreiben von Adresse 0000000C' aufgetreten.

Der Ort des Fehlers ist derselbe. An der zweiten Adresse ist jedoch eine Veränderung zu sehen und diese wird als 0+Offset interpretiert. 0 bedeutet, dass der Zeiger nil ist. Das Offset von 12 Bytes (hexadezimal = C) für Create muss uns nicht weiter interessieren, es ist unterschiedlich bei verschiedenen Klassen bzw. entfällt ganz. Relevant ist der nil-Zeiger, denn auf nil können und sollten wir prüfen, bevor wir auf eine Instanz zugreifen. So auch bei dem seltenen Fall, dass Instanzen häufig erstellt und wieder gelöscht werden sollen.

```
procedure TForm1.Button1Click(Sender: TObject);  
//Stringliste erstellen  
begin  
    if not Assigned(sListe) then  
        sListe := TStringList.Create;  
end;  
  
procedure TForm1.Button2Click(Sender: TObject);  
//Stringliste löschen und nil setzen  
begin  
    FreeAndNil(sListe);  
end;
```

Wir erstellen die Liste nur dann, wenn sie nicht existiert und löschen sie nur dann, wenn sie existiert. Die Prüfung erfolgt durch Assigned, das intern auf nil prüft. Somit muss dieser Zustand nil explizit gesetzt werden, was FreeAndNil für uns erledigt.



Ein einfaches Free würde die Instanz zwar auch freigeben, allerdings wäre der Zeiger darauf noch immer belegt und eine Prüfung würde uns eine vorhandene Instanz vortäuschen.

Probleme in der Analyse dürften eigentlich nur bei Logik- und Laufzeitfehlern entstehen. Ein allgemeingültiges Konzept zur Lösung kann allerdings nicht gezeigt werden, da die Herangehensweise von Fall zu Fall variiert. Es ist es aber grundsätzlich sinnvoll die Problemstelle einzukreisen und den Quelltext auf das Nötigste zu reduzieren.

### 4.3 Vermeiden von Fehlern

Der Titel ist hoch gegriffen, denn Fehler kann man nicht vermeiden. Wohl aber kann man die Anzahl der eigenen Fehler minimieren, wenn man sich an bestimmte Verhaltensmuster und Programmiertechniken beim Erstellen des Quelltextes hält. Die folgenden Abschnitte sollen dem Einsteiger ein Gespür dafür geben.

#### 4.3.1 Lesbarkeit des Quelltextes

Das Verständnis für einen Quelltext steht und fällt mit seiner Lesbarkeit. Die Stilmittel, die einem Programmierer dafür zur Verfügung stehen sind allerdings recht begrenzt. Umso wichtiger ist es deshalb, diese Mittel voll auszuschöpfen. Die Beachtung von nur wenigen grundsätzlichen Regeln wird uns die Lesbarkeit des Textes und die Fehlersuche darin erheblich erleichtern.

**Tipp:** Man kann mittels Strg+D den Quelltext automatisch formatieren lassen!

Wir betrachten jetzt einen voll funktionsfähigen Quelltext, so wie er im schlimmstmöglichen Fall, in einem Delphi-Forum, gepostet werden könnte:

```

unit UEinrueckungUndBenennung;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    Edit1: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
  arr: array of integer;
  i, a: integer;
begin
  if trystrtoint(edit1.Text, a) and (a > 2) then
  begin
    setlength(arr, a);
    arr[0] := 0; arr[1] := 1;
    for i := 2 to high(arr) do arr[i] := arr[i-1] + arr[i-2];
    memo1.Clear;
    for i := 0 to high(arr) do memo1.Lines.Add(inttostr(arr[i]));
  end
  else showmessage('Ungültige Ganzzahl-Eingabe');
  end;

end.

```

Die Funktionalität spielt sich einzig in dieser obigen Methode ab, in der absolut keine Struktur zu erkennen ist. Das ist jedoch eine Grundvoraussetzung, um dem Programmablauf gedanklich folgen zu können. Eine einfache oder blockweise Einrückung von zwei Leerzeichen hat sich hier etabliert.

Zusätzlich wird die Anzahl an Anweisungen pro Zeile auf maximal eins reduziert, denn das zeilenweise Abarbeiten des Debuggers würde die Betrachtung von Variablen unmöglich machen. Die beiden For-Schleifen z.B. würden vom Debugger am Stück ausgewertet werden und das gäbe uns keine Möglichkeit zur Einsicht oder Manipulation der Daten.

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    arr:array of integer;  
    i,a:integer;  
begin  
    if trystrtoint(edit1.Text,a) and (a>2) then  
        begin  
            setlength(arr,a);  
            arr[0]:=0;  
            arr[1]:=1;  
            for i:=2 to high(arr) do  
                arr[i]:=arr[i-1]+arr[i-2];  
            memo1.Clear;  
            for i:=0 to a-1 do  
                memo1.Lines.Add(inttostr(arr[i]));  
        end  
    else  
        showmessage('Ungültige Ganzzahl-Eingabe');  
end;
```

Die Veränderung ist groß und so manchen könnte auch schon klar sein, was dieser Text bewirkt. Um die Lesbarkeit weiter zu steigern, vergeben wir allen verwendeten Bezeichnern sprechende Namen. Es ist überaus wichtig, aus der Benennung heraus bereits Rückschlüsse auf Bedeutung, Typen oder Funktionsweisen ziehen zu können. So werden Prozeduren nach ihrer Verwendung benannt, zumeist beginnend mit einem Verb in Befehlsform und Funktionsnamen sollten auf den entsprechenden Rückgabewert schließen lassen. Bei Komponenten wird der Typ als Präfix, mit 3 bzw. 4 Buchstaben abgekürzt, vorangestellt bzw. als Postfix angehängt. Unterstützt wird die Benennung durch InfixCaps bzw. CamelCase-Schreibweise, die jedes Wort mit einem Großbuchstaben beginnen lässt. Typen wird ein T, Argumenten ein A und Feldern ein F in ihrer Benennung vorangestellt. Mit Ausnahme von lokalen Schleifenvariablen sollten einbuchstabige Variablennamen vermieden werden.

Soviel erstmal zu einer groben Einteilung, für genauere Informationen schaue man in den Delphi-Styleguide. Unabhängig jedoch von der verwendeten Ausrichtung und Benennung ist wichtig, dass sie konsistent über den ganzen Quelltext hinweg gegeben ist.

Ein großes Thema ist auch immer wieder die verwendete Sprache: der Programmierstandard ist Englisch; Delphi benutzt englische Bezeichner und Quelltexte ausschließlich in englischer Sprache wirken natürlicher. Es macht allerdings keinen Sinn darauf zu beharren, wenn man diese Sprache nicht, oder nur eingeschränkt beherrscht. Schlecht gewählte Bezeichner oder gar Gemischtschreibung verwirren den Lesenden.

```

type
  TFrmFibonacci = class(TForm)
    MemFibonacciAusgabe: TMemo;
    BtnBerechneUndZeigeFibonacci: TButton;
    EdtFibonacciAnzahl: TEdit;
    procedure BtnBerechneUndZeigeFibonacciClick(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  FrmFibonacci: TFrmFibonacci;

implementation

{$R *.dfm}

procedure TFrmFibonacci.BtnBerechneUndZeigeFibonacciClick(Sender: TObject);
var
  fibonacci: array of Integer;
  i, anzahl: Integer;
begin
  if TryStrToInt(EdtFibonacciAnzahl.Text, anzahl) and (anzahl > 2) then
    begin
      SetLength(fibonacci, anzahl);
      fibonacci[0] := 0;
      fibonacci[1] := 1;
      for i := 2 to High(fibonacci) do
        fibonacci[i] := fibonacci[i-1] + fibonacci[i-2];
      MemFibonacciAusgabe.Clear;
      for i := 0 to High(fibonacci) do
        MemFibonacciAusgabe.Lines.Add(IntToStr(fibonacci[i]));
      end
    end
  else
    ShowMessage('Ungültige Ganzzahl-Eingabe');
  end;
end;

```

Nach Anpassen aller Bezeichner kann sich das Ergebnis bereits sehen lassen. Dass es hier um Fibonacci-Zahlen geht ist inzwischen auch klar geworden. Trotzdem besteht weiterhin Steigerungspotenzial in der Lesbarkeit, denn die Methode wirkt immer noch unaufgeräumt. Das erkennt man auch an ihrer Benennung, denn es sind zwei Vorgänge in ihr vereint – berechne und zeige an. Das modulare Prinzip sieht jedoch eine Aufteilung in kleine Teilaufgaben vor, was enorme Vorteile bei der Fehlersuche und der Wartung des Quelltextes bietet.

Wir lagern also diese beiden Vorgänge in eigene Prozeduren aus und übergeben die zu verarbeitenden Daten als Parameter.

```

type
  TFibonacciArray = array of Integer;

  TFrmFibonacci = class(TForm)
    MemFibonacciAusgabe: TMemo;
    BtnBerechneUndZeigeFibonacci: TButton;
    EdtFibonacciAnzahl: TEdit;
    procedure BtnBerechneUndZeigeFibonacciClick(Sender: TObject);
  private
    { Private-Deklarationen }
    procedure BerechneFibonacci(AFibonacci: TFibonacciArray);
    procedure ZeigeFibonacci(AAusgabe: TStrings; AFibonacci: TFibonacciArray);
  public
    { Public-Deklarationen }
  end;

var
  FrmFibonacci: TFrmFibonacci;

implementation

{$R *.dfm}

procedure TFrmFibonacci.BerechneFibonacci(AFibonacci: TFibonacciArray);
var
  i: Integer;
begin
  AFibonacci[0] := 0;
  AFibonacci[1] := 1;
  for i := 2 to High(AFibonacci) do
    AFibonacci[i] := AFibonacci[i-1] + AFibonacci[i-2];
  end;

procedure TFrmFibonacci.ZeigeFibonacci(AAusgabe: TStrings; AFibonacci:
TFibonacciArray);
var
  i: Integer;
begin
  AAusgabe.Clear;
  for i := 0 to High(AFibonacci) do
    AAusgabe.Add(IntToStr(AFibonacci[i]));
  end;

procedure TFrmFibonacci.BtnBerechneUndZeigeFibonacciClick(Sender: TObject);
var
  fibonacci: TFibonacciArray;
  anzahl: Integer;
begin
  if TryStrToInt(EdtFibonacciAnzahl.Text, anzahl) and (anzahl > 2) then
    begin

```

```
    SetLength(fibonacci, anzahl);  
    BerechneFibonacci(fibonacci);  
    ZeigeFibonacci(MemFibonacciAusgabe.Lines, fibonacci);  
end  
else  
    ShowMessage('Ungültige Ganzzahl-Eingabe');  
end;
```

Der Text wurde zwar insgesamt länger, doch die gewonnenen Vorteile überwiegen klar. Wegen der Kürze der Routine ist auf den ersten Blick zu erkennen, was im ButtonClick passiert. Die beiden ausgelagerten Prozeduren sind jeweils genau auf eine Funktion beschränkt. Um sie nun aber universell gebrauchen zu können, müssten sie vom Formular entkoppelt werden. BerechneFibonacci z.B. ist immer noch davon abhängig, dass das übergebene Array existiert und mindestens zwei Elemente enthält. Das Hauptanliegen dieses Abschnitts war jedoch die Steigerung der Lesbarkeit und die Mittel, die dafür zur Verfügung stehen.

#### 4.3.2 Speicherlecks

Speicherlecks sind Speicherbereiche, die angefordert, aber nicht wieder freigegeben werden. In seltenen Fällen werden Speicherlecks von Einsteigern überhaupt wahrgenommen, da beim Programmende der vom Betriebssystem für unser Programm reservierte Speicher wieder aufgeräumt wird. Mit Ausnahme von geteilten Speicherbereichen, wie z.B. Speicher bei gemeinsam genutzten DLLs, haben Speicherlecks nur dann merkliche Relevanz, wenn der von Windows zur Verfügung gestellte Speicher zur Neige geht. Das ist dann der Fall, wenn unser Programm ständig neuen Speicher anfordert, aber diesen nicht wieder freigibt. Der Rechner reagiert dann irgendwann träge, da Windows Speicher auf die Festplatte auslagert.

Aber soweit muss es natürlich nicht kommen. Es gibt einige Grundregeln im Bezug auf Erzeugen und Freigeben von Speicher, die befolgt werden sollten, damit keine Speicherlecks auftreten und vor Veröffentlichung eines Programms sollte auch eine Prüfung dahingehend stattfinden.

Aber zunächst einmal müssen wir ein Speicherleck erzeugen, um die Sachverhalte zu klären:

```

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    OpenDialog1: TOpenDialog;
    procedure FormCreate(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

function ErzeugeRueckwaertsListe(AListe: TStrings): TStrings;
var
  i: Integer;
begin
  Result := TStringList.Create;
  for i := AListe.Count-1 downto 0 do
    Result.Add(AListe[i]);
    //Result.Free;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  liste: TStringList;
begin
  if OpenDialog1.Execute then
  begin
    liste := TStringList.Create;
    try
      liste.LoadFromFile(OpenDialog1.FileName);
      Memo1.Lines.Assign(ErzeugeRueckwaertsListe(liste));
    finally
      liste.Free;
    end;
  end;
end;
end;

```

Auf dem Formular befinden sich ein TMemo und ein TOpenDialog. Der OpenDialog wird im OnCreate des Formulars geöffnet und erwartet die Übergabe einer Textdatei. Diese wird eingelesen, von der Funktion umgedreht und im Memo dann ausgegeben. Die Funktion steht hier stellvertretend für einen

komplizierten, ausgelagerten Ablauf. Das alles funktioniert reibungslos. Wie also erfährt man von einem Speicherleck?

Die globale Variable `ReportMemoryLeaksOnShutdown` veranlasst den Speichermanager beim Beenden der Anwendung, den Speicher nach Speicherlecks zu durchsuchen. Diese boolesche Variable müssen wir zum frühest möglichen Zeitpunkt im Programm aktivieren. Im Falle einer Konsolenanwendung in der ersten Zeile des Hauptprogramms. Für unsere Formularanwendung lassen wir uns in der Projektverwaltung die Projektdatei als Quelltext anzeigen und ergänzen entsprechend:

```
program Speicherleck;

uses
  Forms,
  USpeicherleck in 'USpeicherleck.pas' {Form1};

{$R *.res}

begin
  ReportMemoryLeaksOnShutdown := True; //Aktivierung
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Wählen wir beim Programmstart nun eine Textdatei aus und beenden danach das Programm, so sehen wir den Report des Speichermanagers. Je nach Größe der ausgewählten Datei kann diese Ausgabe sehr umfangreich sein. Sie beinhaltet Einträge mit einer bestimmten Anzahl an UnicodeStrings, Unknown und TStringList. Einzig relevant für uns ist das, was wir selbst erzeugt haben, nämlich die TStringList. Der Rest ist eine Folgeerscheinung und verschwindet automatisch, wenn die Instanz von TStringList wieder freigegeben wird.

Ähnlich verhält es sich, wenn z.B. ein TImage erzeugt und nicht freigegeben wird. Hier erscheinen neben dem TImage auch ein TPicture, TFont, TBrush, TPen etc. in der Ausgabe. Alle wurden durch das TImage erzeugt und sind somit auch abhängig von der Freigabe des TImage.

Es stellt sich natürlich nachfolgend die Frage: wie kommt es zu diesem Speicherleck?

Dazu müssen wir nur die Erzeugung der beiden Stringlisten in den beiden Routinen miteinander vergleichen. Im `FormCreate` wird die Liste im Ressourcenschutzblock mit `Free` entsorgt. In `ErzeugeRueckwaertsListe` entfällt das Freigeben mit `Free` und so entsteht genau hier das Speicherleck.

Testweise können wir dort das `Free` entkommentieren, denn dann ist das Speicherleck behoben. Allerdings haben wir dann auch keine Ausgabe mehr, denn die Liste wurde ja freigegeben bevor sie als Rückgabe dienen konnte.



Das Beispiel wollte also nicht nur die Anwendung von `ReportMemoryLeaksOnShutdown` zeigen, sondern auch gleichzeitig auf eine häufig gemachte Fehlerquelle aufmerksam machen. Folgende Anpassung unseres Quelltextes behebt dieses Problem auf einfache Art:

```
procedure BefuelleRueckwaertsListe(AListe, ARueckwaertsliste: TStrings); //übergabe  
als Parameter  
var  
    i: Integer;  
begin  
    for i := AListe.Count-1 downto 0 do  
        ARueckwaertsliste.Add(AListe[i]);  
end;  
  
procedure TForm1.FormCreate(Sender: TObject);  
var  
    liste, rueckwaertsliste: TStringList;  
begin  
    if OpenDialog1.Execute then  
        begin  
            liste := nil;  
            rueckwaertsliste := nil;  
            try  
                liste := TStringList.Create;  
                rueckwaertsliste := TStringList.Create; //Erzeugen beim Aufrufer  
                liste.LoadFromFile(OpenDialog1.FileName);  
                BefuelleRueckwaertsListe(liste, rueckwaertsliste);  
                Memo1.Lines.Assign(rueckwaertsliste);  
            finally  
                rueckwaertsliste.Free;  
                liste.Free;  
            end;  
        end;  
end;
```

`BefuelleRueckwaertsListe` ist jetzt eine Prozedur und nicht mehr eine Funktion, die ihre eigene Rückgabe erzeugen muss. Beide Stringlisten werden in `FormCreate` erzeugt und dann der Prozedur als Parameter übergeben. Dort wird die leere `rueckwaertsliste` befüllt, nach Rückkehr aus der Prozedur dem Memo zugewiesen und dann werden beide Listen im Ressourcenschutzblock freigegeben.

Der Quelltext zeigt eine Möglichkeit, zwei oder mehr verschachtelte Try-Finally-Blöcke beim Erzeugen von zwei oder mehr Instanzen zu umgehen. Wird der Try-Block betreten, so wird der Finally-Block abschließend immer durchlaufen. Da die Konstruktoren hier aber im Try-Block aufgerufen werden und eine Erzeugung grundsätzlich auch scheitern kann, muss sichergestellt werden, dass die Freigabe nur an einer gültigen Instanz erfolgt. Die initiale Zuweisung auf `nil` lässt sich dadurch begründen, dass `Free` intern auf `nil` prüft und nur beim fehlerfreien durchlaufen des Konstruktors, die Instanz eine Wertzuweisung erhält und somit auch nur dann freigegeben wird.

Der entscheidende Unterschied ist aber, und darauf liegt das Hauptaugenmerk, dass rueckwaertsliste dort erzeugt wird, wo auch die Freigabe stattfindet.

Diese Erkenntnis lässt sich dahingehend verallgemeinern, dass erzeugte Elemente auf der Ebene freigegeben werden sollten, auf der sie auch erzeugt werden, d.h. lokal, wie im Beispiel gezeigt innerhalb eines Try-Finally-Blocks. Elemente auf Klassen/Formularebene werden in Create/FormCreate erzeugt und im dazugehörenden Gegenpart Destroy/FormDestroy zerstört, wobei FormCreate und FormDestroy hier die Ereignisbehandlungsroutinen OnCreate und OnDestroy des Formulars beschreiben. Auf Unitebene heißt das Pärchen dann initialization- und finalization-Abschnitt.

Erzeuger und Zerstörer treten auch immer paarweise auf. Create und Free haben wir schon gesehen und auf jedes Create folgt irgendwo ein Free. Es sei denn, es wird z.B. beim Erzeugen einer Komponente ein Eigentümer bestimmt, der sich dann eigenverantwortlich um die Freigabe kümmert. Andere Paare sind z.B. GetMem bzw. AllocMem und FreeMem oder auch New und Dispose.

***Dieses Kapitel widmet sich der Gestaltung von grafischen Benutzeroberflächen.***

## 5 Grafische Benutzeroberflächen

### 5.0 Frameworks

Bereits im „Hallo Welt“-Kapitel zu Beginn dieses E-Books haben wir uns kurz mit grafischen Benutzeroberflächen beschäftigt. Auf allen folgenden Seiten ging es dann aber – mit Ausnahme des Abschnitts „Benutzereingaben“ – hauptsächlich um die Programmiersprache Object Pascal an sich und um Programmiertechniken. Deshalb ist es jetzt mal wieder an der Zeit, uns der Oberfläche zuzuwenden.

Wie schon an den „Hallo Welt“-Beispielen zu sehen war, gibt es in Delphi zwei Frameworks, um GUI-Anwendungen (GUI = Graphical User Interface) zu erstellen: die VCL und FireMonkey. Die **VCL** (Visual Component Library) ist immer dann zu wählen, wenn man eine Anwendung erstellen möchte, die unter Windows läuft. Zieht man es aber in Betracht, die Anwendung auch für andere Betriebssysteme zu kompilieren, sollte man das mit Delphi XE2 eingeführte **FireMonkey** verwenden. Aktuell unterstützt FireMonkey neben Windows auch das Erzeugen nativer macOS, iOS und Android Anwendungen.

Während die VCL bei der Darstellung der bekannten Steuerungselemente (Buttons, Menüs, Listboxen usw.) direkt auf die Windows-Programmierschnittstelle zugreift und dadurch dafür sorgt, dass alles so aussieht wie in jeder anderen Windows-Anwendung, zeichnet FireMonkey alle Steuerungselemente komplett selbst. Das macht es unabhängig vom darunter liegenden Betriebssystem.

### 5.1 Oberflächen-Stile

Seit Delphi XE2 ist es möglich, dass der Entwickler seiner Anwendung einen Stil verpasst. Das ist auch für VCL-Anwendungen möglich. Bei einem Stil handelt es sich um eine Ansammlung grafischer Definitionen für Steuerelemente, also Farben, Schriften usw. Das lässt sich ganz leicht ausprobieren.

#### 5.1.1 VCL-Stile

Wir erstellen eine neue Windows-VCL-Anwendung mit einem Fenster und setzen einen Button drauf. Ohne eine Zeile Code starten wir die Anwendung durch Druck auf die Taste F9. Wie erwartet sieht sie so aus:

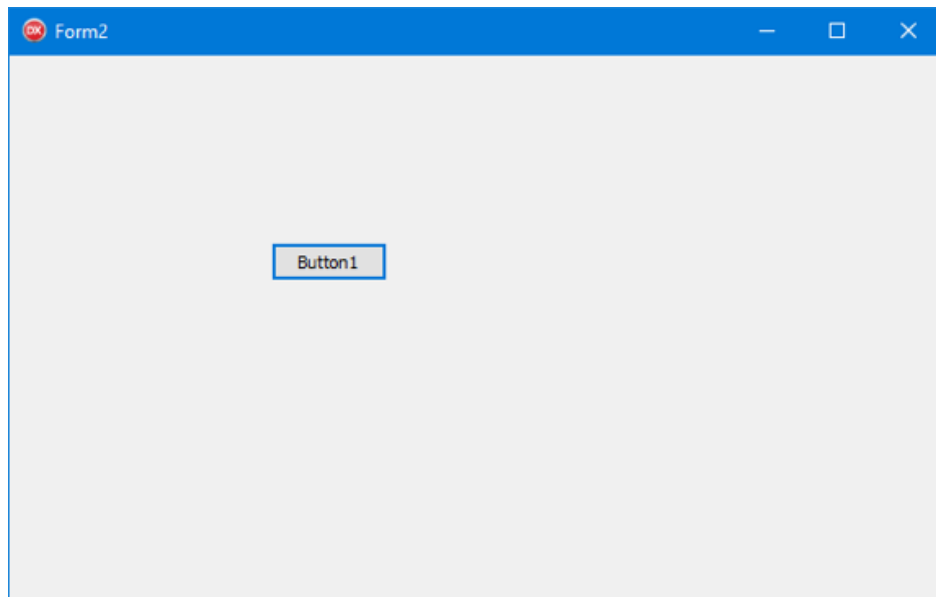


Abbildung 29: Anwendung mit Standard-Stil "Windows"

Natürlich bewirkt ein Klick auf den Button nichts, aber darauf kam es ja auch nicht an. Schließen wir die Anwendung und öffnen die Projektoptionen (Menü „Projekt“ – „Optionen...“). In der Baumansicht auf der linken Seite gibt es den Punkt „Anwendung“ mit dem Unterpunkt „Erscheinungsbild“. Hier ist eine Liste aller vorhandener VCL-Stile zu finden:

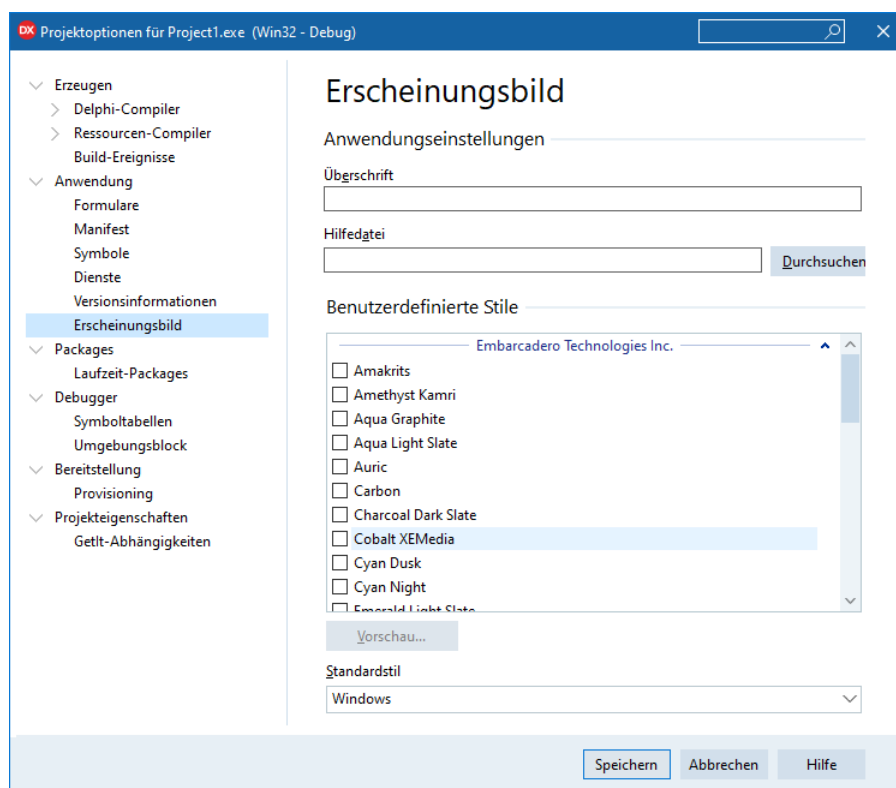


Abbildung 30: Projektoptionen mit Stilauswahl

Wählen wir nun beispielsweise den Stil „Cyan Night“ aus, klicken „Speichern“ und starten die Anwendung wieder mit F9. Schon hat sich die Darstellung drastisch geändert:

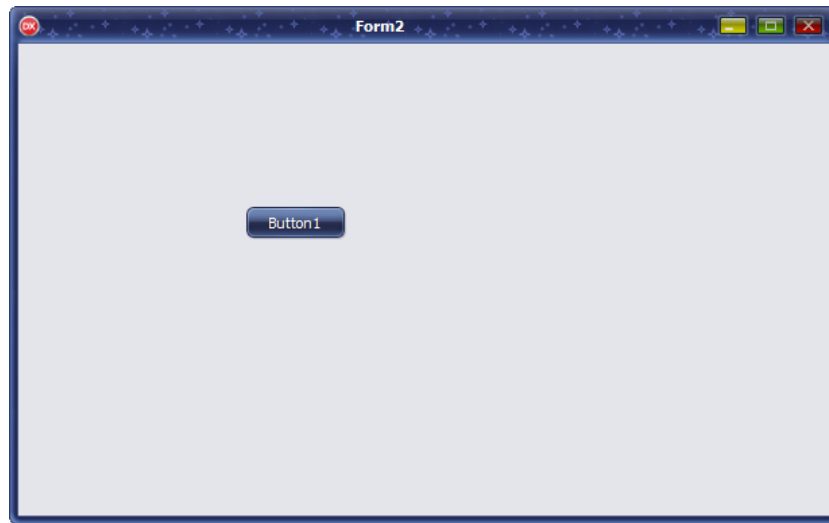


Abbildung 31: Anwendung mit Stil "Cyan Night"

Wem die mitgelieferten Stile nicht ausreichen, der findet im Menü „Tools“ den „Bitmap-Stil-Designer“, über den er bestehende Stile verändern oder neue Stile schaffen kann.

### 5.1.2 FireMonkey-Stile

In FireMonkey funktionieren Stile etwas anders. Grundsätzlich ist der Standard-Stil einer FireMonkey-Applikation abhängig von der gewählten Zielplattform. Das führt dazu, dass eine Windows-Anwendung anders aussieht als eine macOS-Anwendung.

Für ein kleines Beispiel legen wir eine neue FireMonkey-Anwendung an (Datei / Neu / Geräteübergreifende-Anwendung / Leere Anwendung). Auf das Fenster setzen wir – wie im VCL-Beispiel – einen Button. Nach Druck auf F9 sieht die Anwendung so aus:

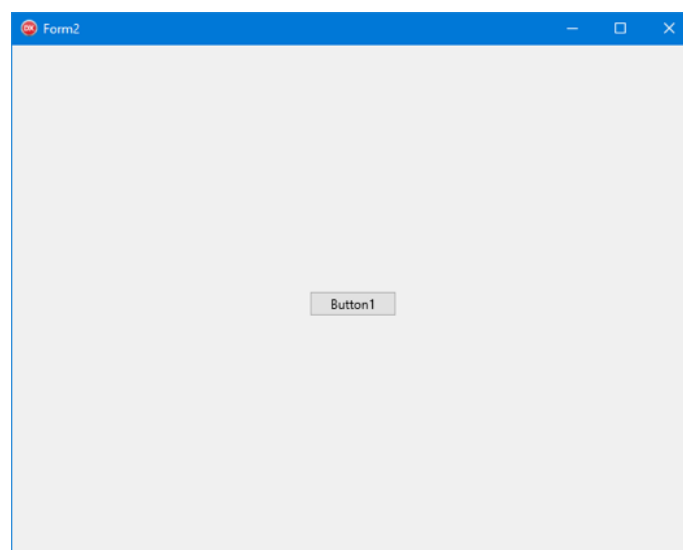
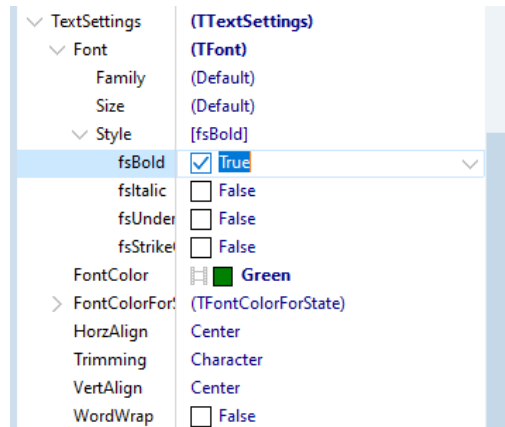


Abbildung 32: FireMonkey-Anwendung mit Standard-Stil

Auch unter FireMonkey kann man die grundlegenden, textuellen Eigenschaften der visuellen Komponenten direkt im Objektinspektor bearbeiten. Unter den TextSettings kann man diese (wie Farbe, Größe, Ausrichtung) direkt ändern.

Es gibt aber auch die Möglichkeit die grundlegenden Stile für alle Komponenten zentral zu verwalten. Diese Vorgaben, die man in den Stilen definiert, können dann individuell einer oder mehreren visuellen Komponenten vorgegeben werden. Diese werden in einem Stylebook zusammengefasst.



Beachten sollte man, daß die Übernahme der Stile aus einem Stylebook auch für jede visuelle Komponente explizit ein- bzw ausgeschaltet werden können, so daß man sich von der Vorgabe „lösen“ kann. Bei Veränderung der TextSettings, an der Komponente selbst, werden diese automatisch losgelöst. Beispiel: Der Vorgabestil hat eine rote Schriftfarbe. Über StyledSettings -> FontColor kann man die Übernahme dann individuell ein- bzw ausschalten. Dazu ein Beispiel.

Möchte man nun eigene Stile für Buttons verwenden, so klickt man mit der rechten Maustaste auf die TButton-Komponente und wählt „Benutzerdefinierten Stil bearbeiten...“. Es öffnet sich der Stil-Editor.

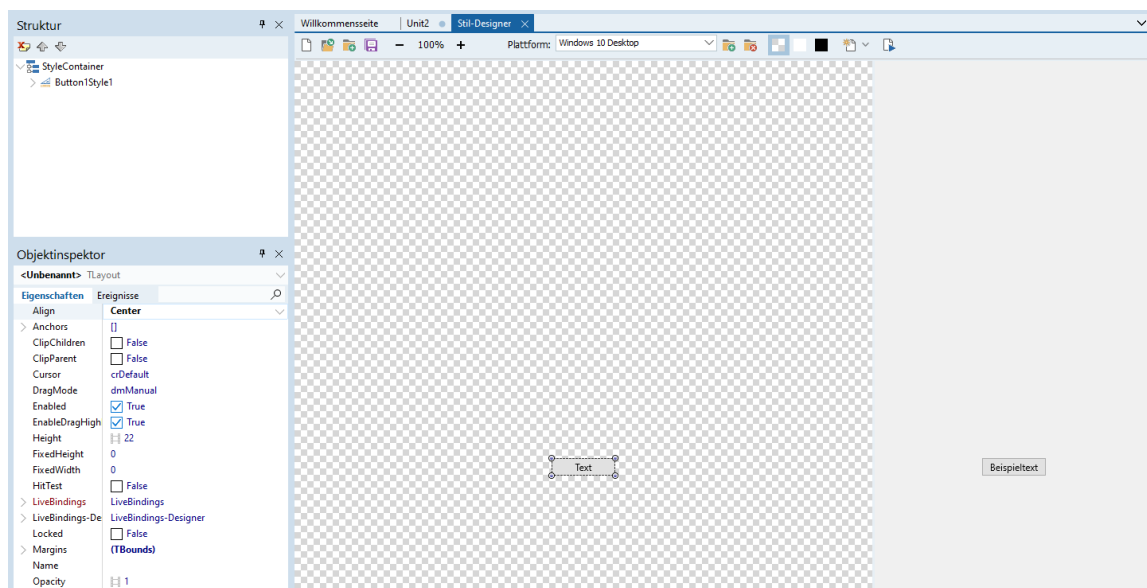
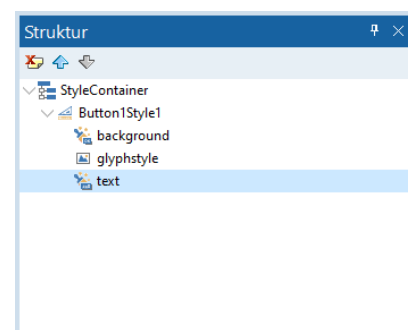
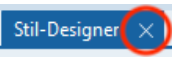


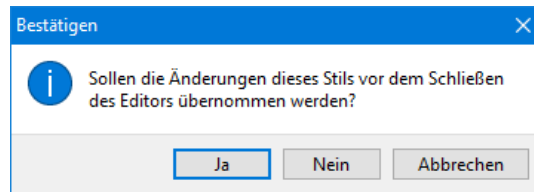
Abbildung 33: Button-Stil bearbeiten

Angenommen, wir wollen die Button-Beschriftung für diesen Button und alle Buttons, denen wir diesen Stil in Zukunft zuweisen wollen, vergrößern und in der Farbe ändern, so wählen wir in der Strukturansicht Stylecontainer -> Button1Style1 -> text aus. Im Objektinspektor erscheinen daraufhin die Eigenschaften eines TButtonStyleTextObject. Hier können wir alle Eigenschaften einer



Button-Beschriftung ändern, also z.B. NormalColor und TextSettings -> Font. „NormalColor“ überschreibt die Eigenschaft „TextSettings/FontColor“

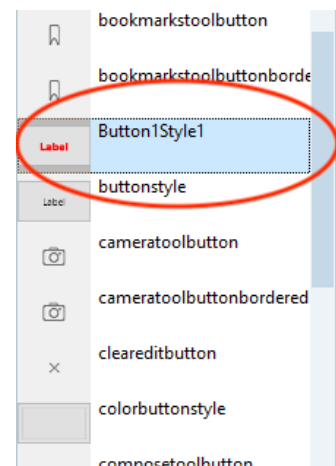
Den Stil-Designer kann man einfach schliessen . Speichern nicht vergessen




Auf unserem Formular können wir nun zwei Änderungen feststellen:

1. Unser Button hat sich der geänderten Darstellung angepasst.
2. Auf dem Formular existiert nun eine neue Komponente mit dem Namen StyleBook1 vom Typ TStyleBook. Diese Komponente ist für die Verwaltung der Stile zuständig.

Setzen wir einen zweiten Button auf das Formular, sieht dieser wieder ganz gewöhnlich aus mit schwarzer Beschriftung. Soll dieser den gleichen Stil verwenden wie unser erster Button, so müssen wir hier nicht erneut alle Eigenschaften von Hand ändern. Stattdessen werfen wir einen Blick auf die Eigenschaft StyleLookup im Objektinspektor. Hier existiert nämlich eine Auswahlmöglichkeit. In dieser Liste befindet sich unser neuer Button-Style.



Natürlich ist es auch in FireMonkey möglich, vorgefertigte Stile zu verwenden. Auch hierfür kommt das StyleBook zum Einsatz. Ein Doppelklick auf das Stylebook öffnet den Stil-Designer, wo man vorgefertigte Stile laden kann. Ganz links befindet sich der Button

„Laden...“ . Ein Klick darauf öffnet einen normalen Datei-Öffnen-Dialog. Wir navigieren in das Verzeichnis der öffentlichen Dokumente und dann in das Unterverzeichnis der Stile (z.B. C:\Users\Public\Documents\Embarcadero\Studio\20.0\Styles). Hierin sind einige FireMonkey-Style-Definitionen zu finden. Wir wählen z.B. den GoldenGraphite.Style aus und schließen das StyleBook mit „Übernehmen und schließen“. Nach einem Druck auf F9 sieht unsere Anwendung so aus:

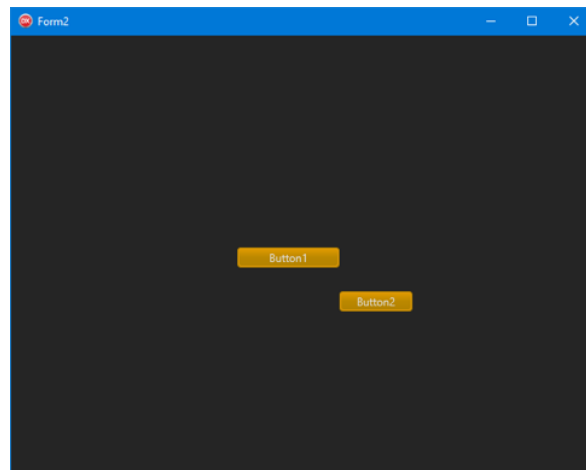


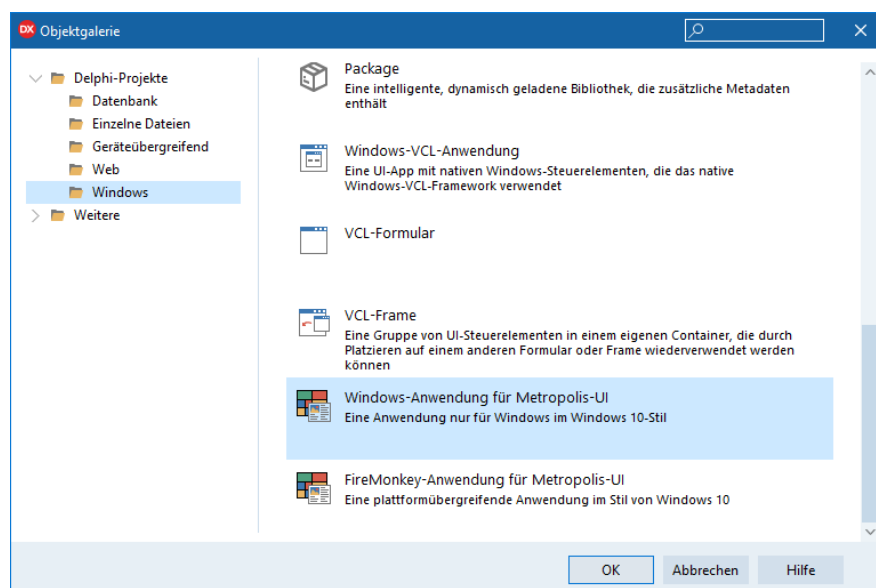
Abbildung 34: FireMonkey-Anwendung mit Stil "GoldenGraphite"

### 5.1.3 Metropolis – der Windows 8-Stil

Mit Windows 8 hat Microsoft das Look & Feel von Anwendungen stark verändert. Aus abgerundeten wurden wieder eckige Ecken, 3D-Effekte und Farbverläufe sind verschwunden. Microsoft nannte diesen neuen Stil „Metro“, bis es zu Namensstreitigkeiten mit einem gleichnamigen deutschen Handelsunternehmen kam. Seitdem wird der Stil „Windows 8 UI Style“ genannt.

Mit Delphi ab Version XE3 ist es möglich, sowohl VCL als auch FireMonkey-Anwendungen zu erstellen, die diesen Stil nachahmen. Entsprechend wird er „Metropolis“ genannt. Zu beachten ist, dass die Verwendung dieses Stils lediglich Auswirkung auf die Darstellung der Anwendungen hat. Sie werden dadurch nicht Windows RT-kompatibel o.ä. Deshalb sind entsprechende Anwendungen auch unter älteren Windows-Versionen lauffähig und sehen trotzdem so aus wie Windows 8-Applikationen. Zu finden sind die Vorlagen in der Objektgalerie unter Datei -> Neu -> Weitere... Delphi Projekte -> Windows

Metropolis Anwendungen können auf VCL oder FireMonkey basieren.





### 5.1.3.1 VCL-Anwendung für Metropolis-UI

Beim Erstellen einer VCL-Anwendung für Metropolis-UI steht ein Assistent zur Verfügung, der entweder eine leere Anwendung erzeugt oder eine mit vorgefertigtem Layout (Raster oder Teilbereich):

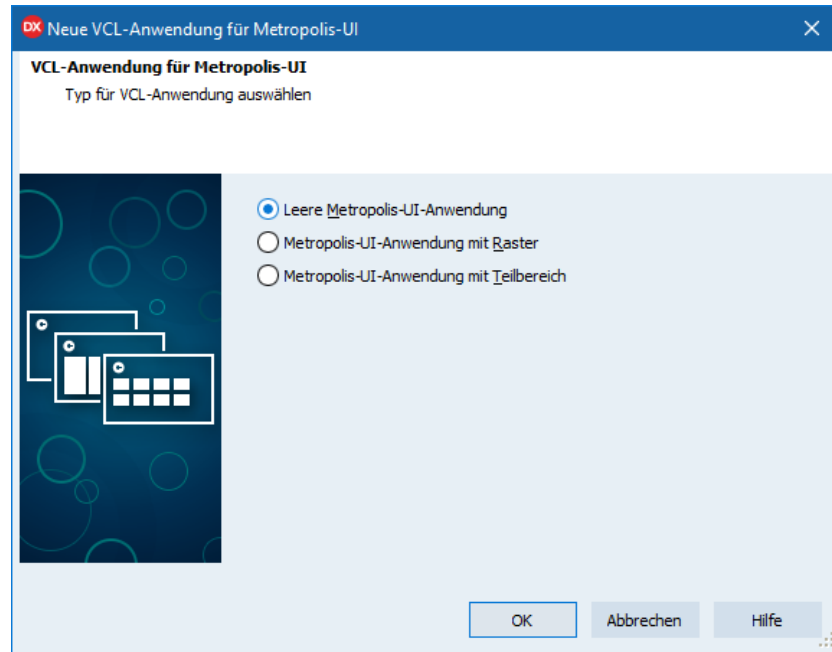


Abbildung 35: Assistent zur Erstellung von Metropolis-UI-Anwendungen

Gemeinsam haben alle drei Anwendungen, dass folgende Komponenten bereits enthalten sind:

- Die **AppBar** (ein einfaches TPanel) am unteren Rand des Fensters. Sie wird eingeblendet, wenn der Benutzer ESC drückt bzw. bei Touch-Oberflächen an den unteren Fensterrand kommt. Auf der AppBar ist ein Schließen-Button enthalten, der die Anwendung beendet. Alternativ ist das nach wie vor über die Windows-übliche Tastenkombination Alt+F4 möglich.
- Außerdem ist ein **Gesten-Manager** enthalten (TGestureManager), der eine Standardgeste für das Anzeigen der AppBar enthält..
- Und schließlich gibt es auch eine **ActionList** mit einer vorgegebenen Action1, die für das Ein- und Ausblenden der AppBar zuständig ist.

Die Anwendung nutzt standardmäßig den VCL-Stil „Metropolis UI Dark“. Metropolis-Stile stehen auch in Black, Blue und Green zur Verfügung.

Beispiel einer Metropolis-UI-Anwendung mit der Vorlage „Teilbereich“:

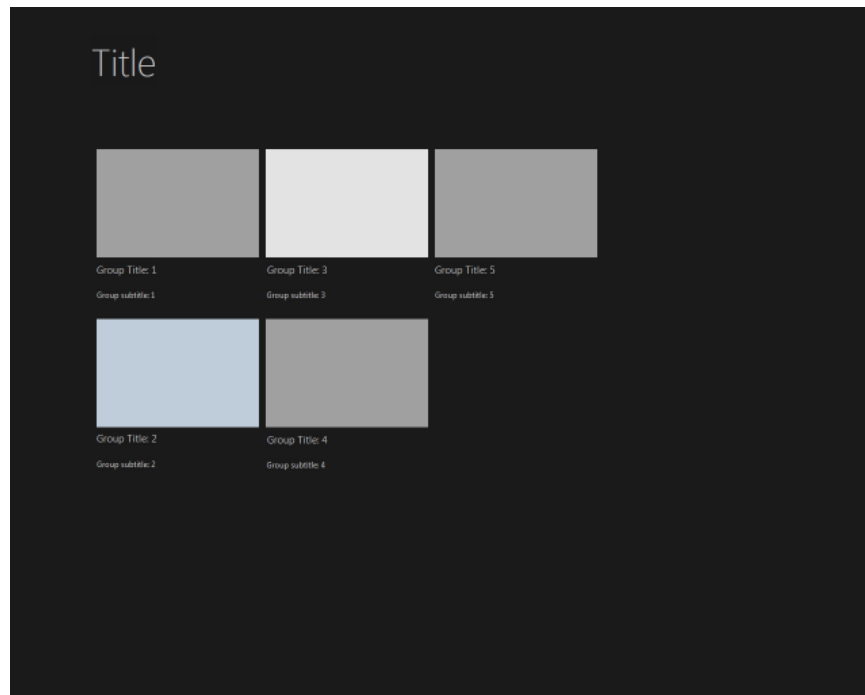


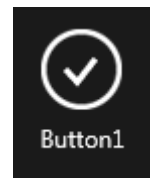
Abbildung 36: VCL-Metropolis-UI-Anwendung mit Teilbereichs-Vorlage

Die einzelnen Flächen sind anklickbar und führen zu einer Detailseite.

#### 5.1.3.2 FireMonkey-Anwendung für Metropolis-UI

Das Erstellen einer FireMonkey-Anwendung für Metropolis-UI funktioniert sehr ähnlich dem oben beschriebenen Vorgehen bei VCL-Anwendungen. Auch hier steht zu Beginn ein Assistent zur Verfügung, der die Vorlagen „Raster“ und „Teilbereich“ anbietet.

Allerdings bietet FireMonkey auch von seinen Komponenten her eine größere Unterstützung für den Windows 8-UI-Style. Z.B. bietet jede **TButton**-Komponente über die Eigenschaft `StyleLookup` im Objektinspektor eine recht große Auswahl an Button-Typen mit dazu passenden Icons (s. Screenshot mit Apply-Button).



Außerdem gibt es die Komponente **TAniIndicator**. Setzt man diese auf ein Formular, wählt unter `StyleLookup` „aniindicatorstyle“ aus und setzt `Enabled` auf `true`, so zeigt sich zur Laufzeit der neue „unbestimmte Statusring“ mit den fliegenden Punkten (s. Screenshot), wie er auch vom Windows 8-Start bekannt ist.



***Im Folgenden stellen wir ein paar einfache Projekte vor. Diese verwenden Komponenten, die von Drittherstellern stammen und kostenlos erhältlich sind.***

## 6 Beispielprojekte

### 6.0 Datenübertragung mit Indy-Komponenten

Delphi enthält in der Tool-Palette die Indy-Komponenten. Dabei handelt es sich um eine große Sammlung von Komponenten zum Thema Netzwerk. Mit ihnen kann man Anwendungen schreiben (sowohl Clients als auch Server), die alle möglichen Protokolle sprechen können, angefangen bei so grundlegenden wie UDP und TCP, aber auch komplexere wie http (Web), FTP (Dateiübertragung) oder SMTP und POP3 (E-Mail).

Die Indy-Komponentensammlung (Indy steht für „Internet Direct“) stammt nicht von Embarcadero, sondern wird in Form eines Open Source-Projekts entwickelt. Die Projektwebseite ist unter [www.indyproject.org](http://www.indyproject.org) zu finden. Dort kann man auch eine Anleitung sowie Demo-Anwendungen zu den Komponenten herunterladen.

In diesem Kapitel wollen wir ein einfaches Beispiel umsetzen, nämlich den Austausch von Textnachrichten zwischen zwei Delphi-Anwendungen über TCP. Die eine Anwendung wird der Client sein, von dem aus man Nachrichten abschicken kann, die andere der Server, der die Nachrichten empfängt, darstellt und eine Bestätigung an den Client zurückschickt.

Wenn du nicht weißt, was es mit TCP und anderen Netzwerkbegriffen auf sich hat, solltest du dich im Internet informieren – als Einstieg z.B. bei Wikipedia. Für das Umsetzen der folgenden Beispiele ist dieses Wissen allerdings nicht erforderlich. Dafür reichen folgende Kurzerklärungen:

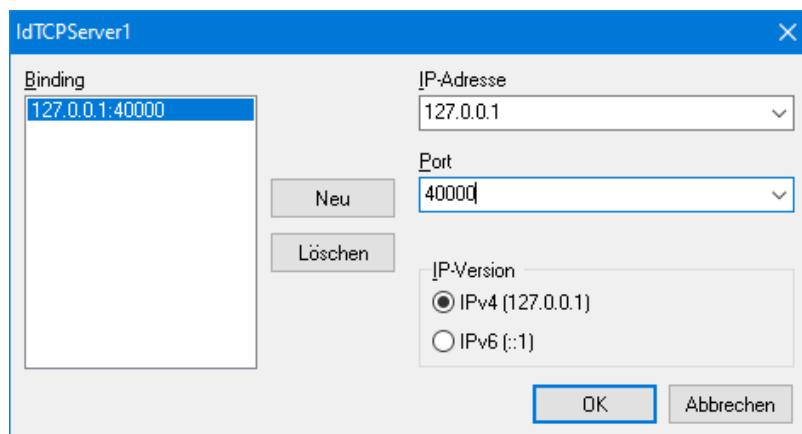
- **TCP** („Transmission Control Protocol“) ist ein Netzwerkprotokoll, also eine Möglichkeit, Daten im Netzwerk zwischen zwei Anwendungen zu übertragen.
- **Server** ist eine Anwendung auf einem Rechner in einem Netzwerk, die von sich aus keinen Kontakt zu anderen aufbaut, sondern nur auf Anfrage etwas tut und das Ergebnis zurückliefert. Ein HTTP-Server (auch Web-Server genannt) liefert z.B. auf Anfrage Webseiten aus.
- **Client** ist eine Anwendung (z.B. ein Webbrowser) auf einem Rechner in einem Netzwerk, die Anfragen an Server schickt und evtl. auf eine Antwort wartet. Eine Anwendung ist nicht darauf beschränkt, nur Client oder nur Server zu sein. Auch beides zusammen ist möglich.

- Rechner in einem Netzwerk werden über **IP-Adressen** identifiziert, darauf laufende Anwendungen werden über **Ports** identifiziert, wobei eine Anwendung auch mehrere Ports öffnen kann, über die sie Daten empfängt oder sendet.

### 6.0.1 Server

Beginnen wir mit der Server-Anwendung. Dazu erstellen wir eine neue Windows-VCL-Anwendung und platzieren eine TIdTCPServer-Komponente und ein TLabel darauf. Das Label soll die Nachrichten, die der Server empfangen hat, darstellen. Die TIdTCPServer-Komponente muss im Objektinspektor wie folgt konfiguriert werden:

- Active: True
- Bindings: Neues Binding mit IP-Adresse 127.0.0.1 und Port 40000 hinzufügen:



Bei 127.0.0.1 handelt es sich um „localhost“, also den Rechner selbst. Das ist völlig unabhängig von einem Netzwerk, weshalb es auch auf deinem PC funktionieren wird, selbst wenn er mit keinem Netzwerk verbunden ist. 40000 als Port ist willkürlich gewählt. Es gibt eine Reihe von Ports, die standardmäßig genutzt werden (z.B. Port 80 für HTTP-Verbindungen). Es gibt 65535 Ports. Alle Ports ab 49152 sind frei verwendbar, da sie keiner festen Anwendung zugeordnet sind.

Nun brauchen wir noch Code, der ausgeführt wird, wenn ein Client den Server aufruft. Dafür bietet die IdTcpServer-Komponente ein OnExecute-Ereignis. Das soll bei uns so aussehen:

```
procedure TForm1.IdTCPServer1Execute(AContext: TIdContext);
var cmd: String;
begin
    try
        cmd := Trim(AContext.Connection.IOHandler.ReadLn);
        Label1.Caption := cmd;
        AContext.Connection.IOHandler.WriteLine('Bye!');
    finally
        AContext.Connection.Disconnect;
    end;
end;
```

Hier wird der Text entgegengenommen, den der Client schickt, in der Variablen „cmd“ gespeichert und in Label1 (muss natürlich auf dem Formular platziert werden) angezeigt. Anschließend wird als Antwort der Text „Bye!“ zurückgeschickt und die Verbindung geschlossen.

Das war's auch schon. Wenn wir die Anwendung starten, sehen wir das Hauptfenster – mehr nicht. Die Anwendung wartet nun darauf, dass jemand Kontakt zu ihr aufnimmt.

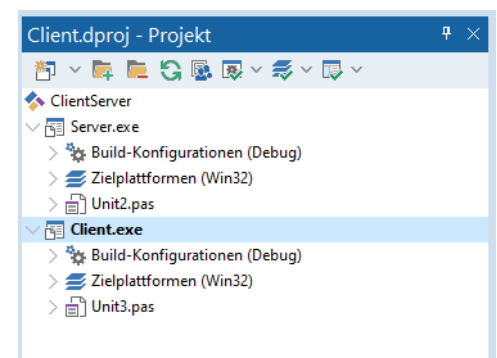
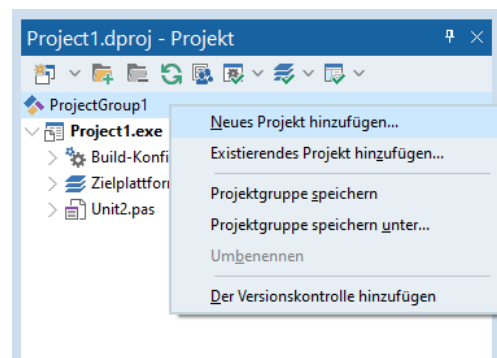
## 6.0.2 Client

Nun kommt die Client-Anwendung dran. Wir erstellen also eine neue Windows-VCL-Anwendung. Dies geht am besten als weiteres Projekt innerhalb der vorhandenen Projektgruppe. Dazu drückt man die rechte Maustaste auf der vorhandenen „ProjectGroup1“ (diese fasst ein oder mehrere Projekte zusammen) und wählt „Neues Projekt hinzufügen...“. In der erscheinenden Objektgalerie wählen wir „Windows-VCL-Anwendung“. Dadurch hat man zwei Projekt gleichzeitig, zwischen denen man bequem per Doppelklick auf den Projektnamen wechseln kann. Speichert man die Projekte unter sinnigen Namen, behält man den Überblick (Datei -> Alles speichern).

Dort platzieren wir diesmal eine TIdTCPClient-Komponente darauf. Außerdem noch ein TEdit zur Eingabe einer Nachricht, die an den Server geschickt werden soll, einen TButton zum Abschieken der Nachricht und ein TLabel zur Anzeige der Antwort, die vom Server kommt.

Der TIdTCPClient-Komponente muss noch mitgeteilt werden, wo sie den Server findet. Dazu sind folgende Einstellungen im Objektinspektor notwendig:

- Host: 127.0.0.1 (eben der Host, der beim Server eingetragen ist)
- Port: 40000 (ebenfalls der Port, der vom Server verwendet wird)



Und nun fehlt nur noch der Code, der ausgeführt wird, wenn man auf den Button klickt:


```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    IdTCPCliant1.Connect;  
    try  
        IdTCPCliant1.IOHandler.WriteLine(Edit1.Text);  
        Label1.caption := IdTCPCliant1.IOHandler.ReadLn();  
    finally  
        IdTCPCliant1.Disconnect;  
    end;  
end;
```

Was passiert hier? Zuerst lassen wir den Client eine Verbindung zum Server aufbauen (Connect). Falls der Server nicht erreichbar ist, z.B. weil wir die Server-Anwendung noch nicht gestartet haben, fliegt hier eine `EIdSocketError`-Exception.

Konnte die Verbindung aufgebaut werden, wird der Text aus dem Eingabefeld Edit1 per `WriteLn` an den Server geschickt und anschließend direkt die Antwort mittels `ReadLn` gelesen und in Label1 angezeigt. Zum Schluss wird die Verbindung wieder getrennt.

### 6.0.3 Der Test

Nun wollen wir natürlich auch ausprobieren, ob das alles so funktioniert. Dazu müssen beide oben erstellte Anwendungen kompiliert werden. Als erstes starten wir die Server-Anwendung, wobei die Start-Reihenfolge egal ist. Der Server muss nur laufen, wenn jemand im Client auf den Button klickt. Ansonsten führt das zu o.g. Exception. Die Server-Anwendung läuft nun vor sich hin und ist nur damit beschäftigt, auf Client-Anfragen zu warten.

Ds Starten vom Server geht am besten indem man in der Projektgruppe die „Server“ Anwendung doppelklickt und diese ohne Debuggerkontrolle mittels Umsch+Strg+F9 startet (oder einfach den einfachen Startknopf drücken ).

Als nächstes starten wir die Client-Anwendung und geben in das Eingabefeld „Delphi rocks!“ ein. Nach einem Klick auf den Button sollte genau dieser Text in der Server-Anwendung erscheinen:

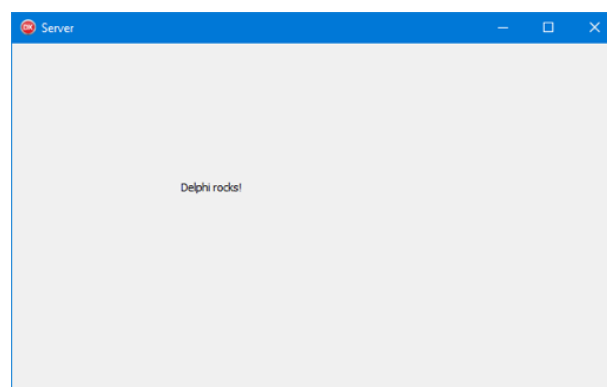


Abbildung 37: Der Server

Auch beim Client hat sich nun etwas verändert. Der Server hat den Text „Bye!“ als Antwort zurückgeschickt, die der Client im Label1 anzeigt:

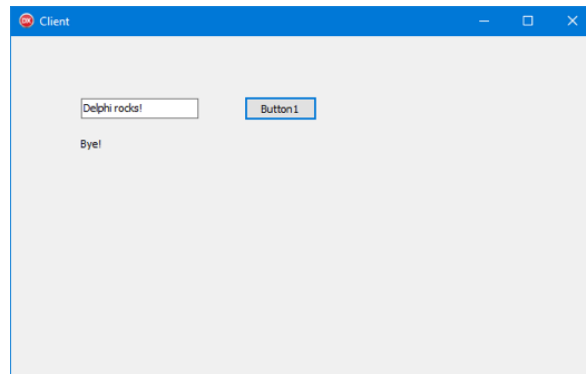


Abbildung 38: Der Client mit Antwort vom Server

Alles hat also wunderbar funktioniert.

## 6.1 Datenbankprogrammierung – SQLite mit Delphi

Im Vergleich zu den höherpreisigen Delphi-Editionen (Enterprise, Architect) sind die Datenbankfähigkeiten der Community-Edition etwas eingeschränkt: Es kann nur auf lokale Datenbanken (mittels FireDAC) zugegriffen werden. Für dieses E-Book werden wir SQLite verwenden, weil hierfür kein eigener Datenbank-Server erforderlich ist, sondern die Datenbank quasi in die Anwendung integriert wird. Zudem ist SQLite kostenlos verfügbar. Bevor wir uns aber um die Datenbank kümmern können, sind ein paar Grundlagen erforderlich.

### 6.1.1 Was ist eine Datenbank?

Zunächst einmal das Wichtigste: Was ist überhaupt eine Datenbank?

In diesem Zusammenhang unterscheidet man verschiedene Begriffe: **Datenbankmanagementsystem** (DBMS), **Datenbanksystem** (DBS) und **Datenbank** (DB). Ein DBMS ist eine Software, die Daten speichert und auf Anfrage wieder zur Verfügung stellt, z.B. MySQL.

Die Datenbank (DB) sind die Daten, die vom DBMS gespeichert werden. Und ein Datenbanksystem ist beides zusammen.

Was ein DBMS tut, hört sich zunächst einfach an. Jedoch sind DBMS optimiert für das, was sie tun sollen, so dass sie performant und zuverlässig sind. Es gibt unterschiedliche Arten von Datenbankenmanagementsystemen.

Die bislang häufigste ist das **relationale Datenbanksystem**. In ihr werden Daten in Tabellen (auch „Relationen“ genannt) abgelegt. Jede Zeile derselben Tabelle (eine Zeile wird „Datensatz“ genannt) ist gleich aufgebaut. Die Daten verschiedener Tabellen können untereinander in Beziehung gebracht werden. Dazu in einem späteren Abschnitt mehr. Bekannte Vertreter hiervon sind Oracle, DB2, MySQL, InterBase, Microsoft SQL Server usw.

Eine weitere Art sind **objektorientierte Datenbanksysteme**. Hierin werden direkt Objekte gespeichert zusammen mit ihren verbundenen Objekten, ohne dass der Entwickler die Daten eines Objekts erst in eine Tabellenform bringen muss. Objektorientierte Datenbanken haben jedoch nur eine geringe Verbreitung.

Eine relativ neue Gattung sind die sog. **NoSQL-Datenbanksysteme**. Auch sie haben ihren Einsatzzweck, z.B. wenn es um Performance geht oder wenn man kein starres Datenbankschema haben will, sondern jeden Datensatz einer Tabelle unterschiedlich aufbauen will. NoSQL-Datenbanken zerfallen in weitere Unterarten wie dokumentenorientierten Datenbanken, Key-Value-Caches usw. Bekannt sind z.B. CouchDB, Cassandra und Berkeley DB.

Wir werden uns hier um die herkömmlichen relationalen Datenbanksysteme kümmern.

### 6.1.2 Was ist SQL?

SQL steht für „Structured Query Language“ und ist – wie der Name schon sagt – keine richtige Programmiersprache, sondern eine Abfragesprache, die für die Kommunikation mit relationalen



Datenbanken verwendet wird. SQL erlaubt es, bestimmte Daten aus der Datenbank abzufragen, aber auch neue Daten hinzuzufügen, bestehende zu verändern und auch ganze Tabellen anzulegen oder zu entfernen.

SQL wird von allen großen Datenbankservern (Oracle, DB2, MS SQL Server, InterBase usw.) unterstützt. Diese Datenbankserver sind allerdings nicht Inhalt dieses Kapitels. Vielmehr soll ein Einblick in die Arbeit mit SQLite gegeben werden.

### 6.1.3 Was ist SQLite?

SQLite ist ein relationales Datenbankmanagementsystem. Es unterstützt den Großteil der im SQL-92-Standard festgelegten SQL-Sprachbefehle. SQLite kommt ohne Datenbankserver aus. Die Datenbank wird als Datei abgelegt, was einen Vorteil gegenüber MySQL darstellt. Es sind keine umständlichen Server-Installationen notwendig, es muss nur die SQLite-DLL mitgeliefert werden, was von den Machern des Datenbanksystems erlaubt ist. Mehr und aktuelle Informationen dazu findet man auf der Website des Projektes (<http://sqlite.org/>).

SQLite ist für den Einsatz innerhalb einer Anwendung, also ohne Server, entworfen und findet Verwendung in bekannten Produkten von Apple, Mozilla, Adobe, Microsoft und Google. Auch die teilweise in Delphi geschriebene Videotelefonie-Anwendung Skype setzt auf SQLite.

Es existieren mehrere Delphi-Wrappers um die SQLite-DLL. Bei einem Wrapper handelt es sich um eine Zwischenschicht, die sich einfach in einer Programmiersprache (bei uns Delphi) verwenden lässt und die Aufrufe intern übersetzt in die Form, wie die DLL sie erwartet. In diesem Kapitel wird die Verwendung des integrierten FireDAC Wrappers domonstriert. Man muss also nichts nachträglich installieren.

### 6.1.4 Relationale Datenbanksysteme

Ein relationales Datenbanksystem besteht, wie oben bereits erwähnt, aus Tabellen (Relationen), die über sogenannte Fremdschlüsselbeziehungen miteinander verknüpft sind.

Ein simples Beispiel für eine Fremdschlüsselbeziehung ist folgendes: Für eine kleine Bücherei soll ein Ausleihsystem implementiert werden. Dazu benötigen wir in der Datenbank eine Tabelle mit Büchern, eine mit den Daten der Leser und eine, in der steht, welcher Leser welche Bücher ausgeliehen hat. Also in etwa so (auf die Details kommt es nicht an, es geht mehr um das grundsätzliche Vorgehen):

#### **Tabelle Buch**

Titel

Autor

#### **Tabelle Leser**

Name

Adresse

Jetzt stellt sich die Frage, wie die Tabelle „Ausleihe“ auszusehen hat. Diese verbindet ja die Leser mit den Büchern, die sie ausgeliehen haben. Dazu müssen die Bücher und die Leser zunächst einmal eindeutig identifizierbar gemacht werden. Der Buchtitel reicht dafür nicht aus, weil dieser nicht

eindeutig ist. Die ISBN wäre geeignet, allerdings bieten Bibliotheken häufig ja auch andere Medien wie CDs oder Zeitschriften an. Bei den Lesern besteht das gleiche Problem. Namen sind nicht eindeutig. Deshalb bekommen beide Tabellen eine zusätzliche Spalte „ID“ verpasst. Die ID wird eine fortlaufende Zahl sein. Datenbankmanagementsysteme bieten in der Regel einen entsprechenden Automatismus an, so dass man sich als Entwickler nicht selbst um das Hochzählen der IDs kümmern muss.

Diese IDs, die jeden Datensatz einer Tabelle eindeutig identifizierbar machen, nennt man **Primärschlüssel**.

Nun wird auch klarer, wie die Tabelle „Ausleihe“ aussehen könnte. Wenn Max Mustermann (ID 37) das Buch „Delphi Starter“ (ID 1892) ausleiht, muss einfach ein Datensatz mit diesen beiden Informationen angelegt werden können. Natürlich kopieren wir nicht die Daten aus den Tabellen „Buch“ und „Leser“ (sonst hätten wir einiges zu ändern, wenn Max Mustermann umzieht), sondern speichern nur die IDs des Buchs und des Lesers. Das nennt man „**Fremdschlüssel**“, da sich der Wert auf den Primärschlüssel einer fremden Tabelle bezieht.

Die Tabelle „Ausleihe“ könnte also so aussehen:

#### **Tabelle Ausleihe**

Leser-ID (Fremdschlüssel)

Buch-ID (Fremdschlüssel)

Ausleihdatum

Rückgabedatum

### 6.1.5 Wichtige SQL-Befehle

Wie schon erwähnt, geschieht das Abfragen von Daten aus einer relationalen Datenbank mit Hilfe der Sprache SQL. Hier die wichtigsten Funktionen/Schlüsselwörter:

#### 6.1.5.1 *SELECT*

Jede SQL-Abfrage wird durch SELECT eingeleitet. Die einfachste Form lautet

```
SELECT * FROM <Tabellenname>
```

Das Sternchen bedeutet, dass alle Felder ausgelesen werden sollen. Um nur gewisse Felder auszulesen, kann man hier eine Liste der gewünschten Felder anführen. Das könnte beispielsweise so aussehen:

```
SELECT nr, bezeichnung FROM artikel
```

#### 6.1.5.2 *WHERE*

Um die Abfrage einschränken zu können, kann mittels WHERE eine Bedingung gestellt werden. Der Aufbau lautet

```
SELECT <Felder> FROM <Tabelle> WHERE <Bedingung>
```

Wenn eine Tabelle „Artikel“ Preise enthält, kann man beispielsweise alle Artikel heraussuchen, die weniger als 10 Euro kosten:

```
SELECT * FROM artikel WHERE preis<10
```

WHERE-Bedingungen lassen sich wie auch die logische Bedingung in Pascal durch AND und OR verknüpfen:

```
SELECT * FROM artikel WHERE preis>10 AND preis<100
```

#### 6.1.5.3 JOIN

Joins lesen Daten aus verschiedenen Tabellen und vereinigen diese nach bestimmten Kriterien. Dieser Abschnitt kann nicht in die Details gehen, zumal die Syntax sich von DBMS zu DBMS unterscheidet. Deshalb nur eine einfache Variante am Beispiel unserer Bücherei: Möchte man wissen, wann welcher Titel (und nicht nur die ID) ausgeliehen wurde, muss man alle Tabellen „Buch“ und „Ausleihe“ miteinander verbinden:

```
SELECT a.ausleihdatum, b.titel FROM buch b, ausleihe a WHERE a.buch_id = b.id
```

Die Tabelle „Buch“ bekommt hier den Alias-Namen „b“, „Ausleihe“ den Namen „a“, damit alles weitere übersichtlich bleibt. Dann wird der Fremdschlüssel in der Ausleihe-Tabelle dem zugehörigen Primärschlüssel in der Buch-Tabelle gleichgesetzt. Dadurch stehen im Select-Ergebnis die passenden Spalten der Buchtabelle in jeder Zeile der Ausleihe-Tabelle zur Verfügung.

#### 6.1.5.4 INSERT

Mit dem Befehl INSERT können neue Datensätze in die Datenbank eingefügt werden. Die Struktur sieht folgendermaßen aus:

```
INSERT INTO <Tabelle> (<Felder>) VALUES (<werte>)
```

Will man in die bereits erwähnte Artikel-Tabelle einen neuen Artikel aufnehmen sieht der SQL-Befehl so aus:

```
INSERT INTO artikel (bezeichnung, preis) VALUE ("Delphi Community Edition", 149)
```

#### 6.1.5.5 UPDATE

Soll ein bereits bestehender Datensatz geändert werden, wird der UPDATE-Befehl benötigt.

```
UPDATE artikel SET preis=20 WHERE bezeichnung="USB-Stick 32GB";
```

ändert den Preis des Artikels USB-Stick 32GB auf 20 Euro.


Weitere Befehle und wie diese von SQLite verstanden werden findet man auf <http://www.sqlite.org/lang.html>.

### 6.1.6 SQLite-Zugriff über FireDAC mit Delphi

Der Zugriff auf eine SQLite Datenbank ist in Delphi Community Edition gleich integriert. Hat man die „Samples“ installiert (in der IDE unter Tools -> Plattformen verwalten -> Weitere Optionen -> [X] Samples), dann ist auch gleich eine Beispieldatenbank mitinstalliert worden, die wir hier nutzen werden.

C:\Users\Public\Documents\Embarcadero\Studio\20.0\Samples\data\FDDemo.sdb

Auch gibt es dazu auch ein Beispielprojekt

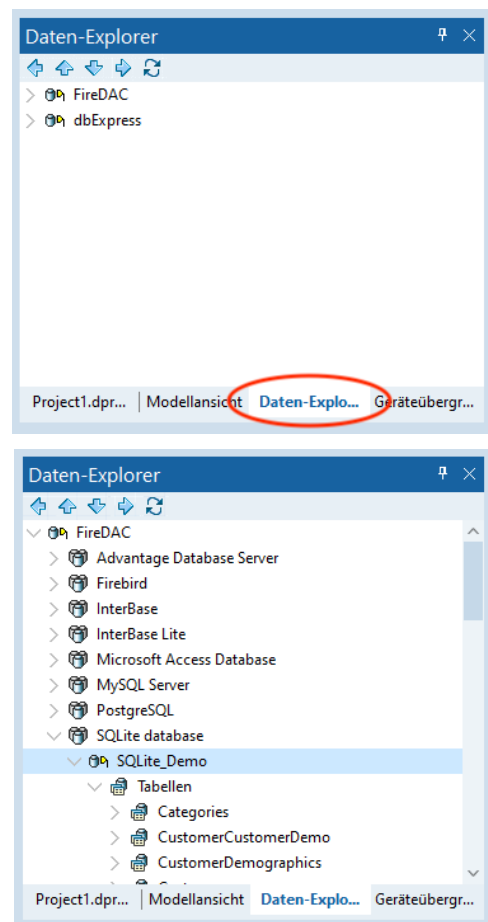
C:\Users\Public\Documents\Embarcadero\Studio\20.0\Samples\   
Object Pascal\Database\FireDAC\Samples\Getting Started\SQLite\GettingStarted.dproj

#### 6.1.6.1 Datenbankverbindung herstellen und trennen

Die Datenbankverbindung zur SQLite Demodatenbank ist in der Delphi Community Edition gleich vorkonfiguriert. Man kann diese sofort einsetzen. In einer ersten, kleinen Demoanwendung wollen wir diese nutzen. Dazu Datei -> Windows-VCL-Anwendung. Es erscheint eine neue Windows Anwendung und im rechten, oberen Bereich sehen wir, neben der Projektverwaltung, den Daten-Explorer. In diesem gibt es zwei Einträge: FireDAC und dbExpress. Wir beschränken uns hier auf das neuere FireDAC und greifen darüber das das vorkonfigurierte SQLite\_Demo zu.

Einige Hintergrundinformationen:

- SQLite speichert seine Dateien in einer einzelnen Datei. Diese kann irgendwo in einem beschreibbaren Bereich des Dateisystems liegen. Das „Program Files/Program Files (x86)“ Verzeichnis ist iA nicht für den Benutzer beschreibbar. Man sollte seine Datenbank im Bereich der „Eigenen Dateien“ ablegen.
- Der Name der Datei, insbesondere die Dateiendung, kann beliebig sein. Weit verbreitete Endung für SQLite Datenbankdateien sind .sqlite3, .sqlite, .db, .s3db oder .sdb
- Delphi selbst braucht für den Zugriff nur die SQLITE3.DLL, die bei Delphi gleich mitgeliefert wird  
C:\Program Files (x86)\Embarcadero\Studio\20.0\bin\sqlite3.dll



- Die Verbindungseinstellungen zur Datei/Datenbank können über die rechte Maustaste im Datenbank-Explorer geändert werden: Auf dem Eintrag der Verbindungsdefinition -> Ändern. Auch kann man natürlich neue Verbindungseinstellungen hier im Daten-Explorer mittels „Neu“ anlegen: Direkt auf dem Eintrag „SQLite database“ mittels rechter Maustaste.
- Weitere Informationen findet man auch auf der DocWiki Seite unter [http://docwiki.embarcadero.com/RADStudio/Sydney/de/Tutorial:\\_Herstellen\\_einer\\_Verbindung\\_zu\\_einer\\_SQLite-Datenbank\\_mit\\_FireDAC](http://docwiki.embarcadero.com/RADStudio/Sydney/de/Tutorial:_Herstellen_einer_Verbindung_zu_einer_SQLite-Datenbank_mit_FireDAC)

In der neuen Geräteübergreifenden Anwendung (Datei > Neu > Geräteübergreifende Anwendung) platzieren wir nun die folgenden Komponenten

- zwei `TButton`-Steuerelemente; setze im Objektinspektor die Eigenschaften `Name` der beiden Schaltflächen auf `connectButton` bzw. `executeButton` und die Eigenschaften `Text` auf `Connect` bzw. `Execute`.  
Setzen Sie die Eigenschaft `Enabled` von `executeButton` auf `False`.
- Eine `TFDConnection`-Komponente
- Ein `TMemo`-Steuerelement; setze im Objektinspektor die Eigenschaft `Name` auf `outputMemo`.  
Setze die Schriftart des Memos mittels `TextSettings.Font.Family` auf `Consolas`
- Eine `TFDPhysSQLiteDriverLink`-Komponente für die Verbindung des SQLite-Treibers mit der Anwendung

Jetzt sollte das Formular folgendermaßen aussehen:

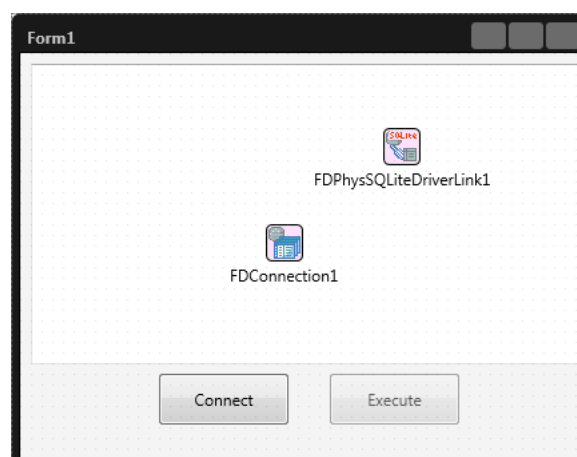


Abbildung 36: Datenbankzugriff

Füge der `OnClick`-Ereignisbehandlungsroutine von `connectButton` den folgenden Code hinzu:

```

procedure TForm3.connectButtonClick(Sender: TObject);
begin
    outputMemo.Text := '';
    // Den Pfad auf die Datenbankdatei setzen.
    // 'C:\Users\Public\Documents\...\Employees.s3db ' durch den absoluten Pfad
    // zu der SQLite-Datenbankdatei ersetzen.
    FDConnection1.DriverName := 'SQLITE';
    FDConnection1.Params.Values['Database'] :=
    C:\Users\Public\Documents\Embarcadero\Studio\20.0\Samples\data\FDDemo.sdb';
    try
        // Die Verbindung herstellen.
        FDConnection1.Open;
        executeButton.Enabled := True;
        outputMemo.Lines.Add('Connection established!');
    except
        on E: EDatabaseError do
            outputMemo.Lines.Add('Exception raised with message' + E.Message);
    end;
end;

```

#### 6.1.6.2 Datensatz abfragen

Abfragen richten sich an eine bestimmte Tabelle. Möchte man z.B. aus der oben verwendeten Datenbank die Artikel aus „Products“ abfragen, verwendet man folgenden Code:

```
'SELECT * FROM Products WHERE ProductID = 57'
```

Diese Abfrage kann man nun auf verschiedene Weisen in seiner Anwendung einbauen (und auch unterschiedliche Darstellungen erhalten). Als reiner Text: Eine TFDQuery Komponente kann einen beliebigen SQL Text gegen eine Datenbank ausführen. Diese kann natürlich auch dynamisch zur Laufzeit erzeugt werden („Create“). Füge also den folgenden Quelltext dem executeButton als OnClick Ereignis hinzu

```

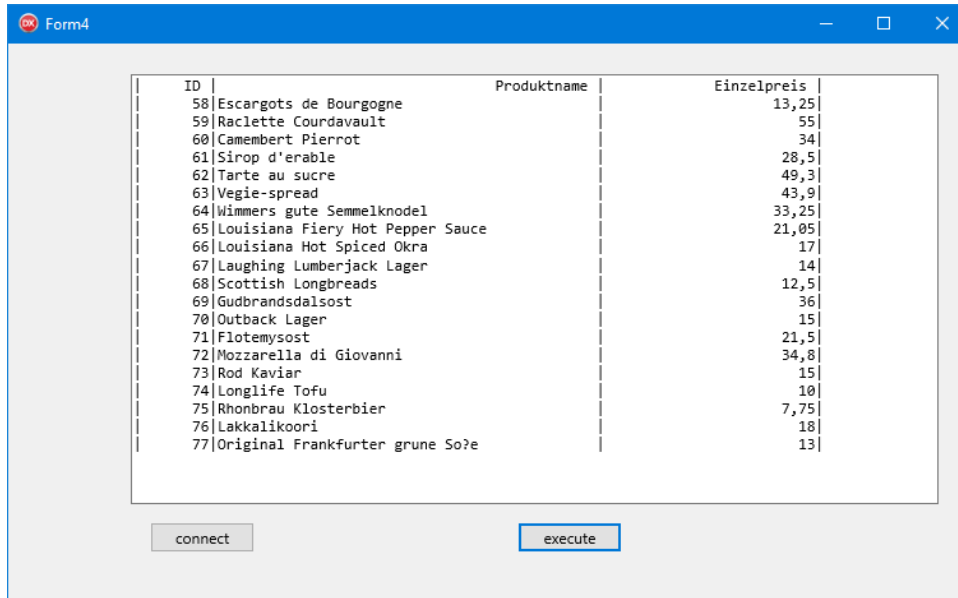
procedure TForm1.executeButtonClick(Sender: TObject);
var
    query: TFDQuery;
begin
    query := TFDQuery.Create(nil);
    try
        // Die SQL-Abfrage festlegen
        query.Connection := FDConnection1;
        query.SQL.Text := 'SELECT * FROM Products WHERE ProductID > 57';
        query.Open();
        outputMemo.Text := '';
        // Die Feldnamen aus der Tabelle hinzufügen.
        outputMemo.Lines.Add(String.Format('%8s%-45s%-25s|',
            [' ID ',
              ' Produktname ',
              ' Einzelpreis '])));
        // Dem Memo eine Zeile pro Datensatz in der Tabelle hinzufügen.
        while not query.Eof do
            begin
                outputMemo.Lines.Add(String.Format('%8d%-45s%-25s|',
                    [query.FieldByName('ProductID').AsInteger,
                     query.FieldByName('ProductName').AsString,
                     query.FieldByName('UnitPrice').AsString]));
                query.Next;
            end;
        finally
            query.Close;
            query.DisposeOf;
        end;
    end;

```

Auf die einzelnen Felder kann man dann über die entsprechenden Methode der Instanz einer TQuery zugreifen:

```
bezeichnung := query.FieldByName('ProductName').AsString; // bezeichnung : string
```

Zu beachten ist hier: `String.Format` formatiert einen String mit einem oder mehreren Platzhaltern („%“) und der Angabe der Längen („8s“: 8 Zeichen (s=String) / „-45s“: Link ausgerichtet (Minus-Zeichen)). Mit einer angepassten Schriftart, kann das dann so aussehen:



ID	Produktname	Einzelpreis
58	Escargots de Bourgogne	13,25
59	Raclette Courdavault	55
60	Camembert Pierrot	34
61	Sirop d'érable	28,5
62	Tarte au sucre	49,3
63	Veggie-spread	43,9
64	Wimmers gute Semmelknodel	33,25
65	Louisiana Fiery Hot Pepper Sauce	21,05
66	Louisiana Hot Spiced Okra	17
67	Laughing Lumberjack Lager	14
68	Scottish Longbreads	12,5
69	Gudbrandsdalsost	36
70	Outback Lager	15
71	Flotemysost	21,5
72	Mozzarella di Giovanni	34,8
73	Rod Kaviar	15
74	Longlife Tofu	10
75	Rhonbrau Klosterbier	7,75
76	Lakkalikoori	18
77	Original Frankfurter grüne Soße	13

Abbildung 37: Datenbankzugriff Ergebnis

#### 6.1.6.3 Darstellung eines DataSets

Das war jetzt der schwierige, komplizierte Weg. Es gibt einfachere Wege und vorgefertigte Komponenten, um mit Datensätzen zu arbeiten. So lassen sich TFDQuery und/oder TFDTable Komponenten direkt auf das Formular (oder ein Datenmodul) platzieren, um mit diesen direkt zu arbeiten oder um diese mit „datensensitiven“ Komponenten zur Darstellung zu verbinden.

Dazu wählen wir eine neue Windows-VCL-Anwendung (Datei -> Neu) und gehen in den Daten-Explorer unter „FireDAC“. Hier klappen wir die vorhandene SQLite Datenbank auf und bewegen uns auf die Tabelle „Products“:

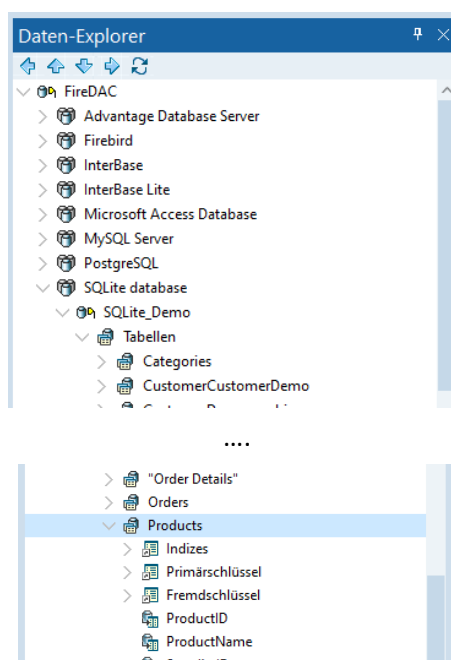


Abbildung 38: Tabelle „Products“ unter FireDAC im Daten-Explorer



Diese („Products“) ziehen wir jetzt einfach per Drag’n’Drop in das Formular der Windows-VCL-Anwendung:

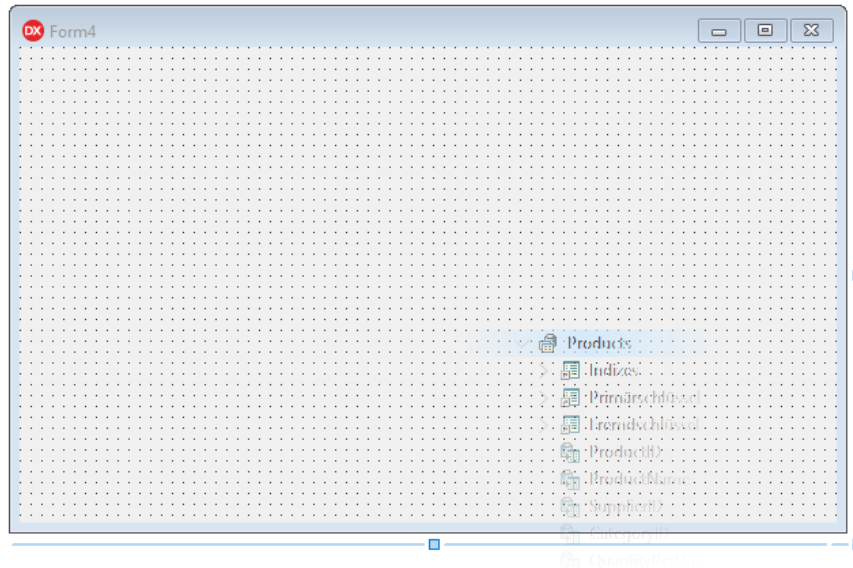


Abbildung 39: Drag’n’Drop vom Daten-Explorer

Es wird eine **TFDConnection** (Name: „Sqlite\_demoConnection“) und eine **TFDQuery** (Name: „ProductsTable“) auf dem Formular platziert. Diese beinhalten alle notwendigen Informationen zur Konnektivität zu dieser Datenbank/Tabelle. Die **TFDConnection** stellt die Verbindung zur Datenbank her und die **TFDQuery** definiert die passende Abfrage für die Tabelle „Products“. Was hier automatisch konfiguriert wurde:

- **TFDConnection / Sqlite\_demoConnection**  
DriverName    SQLite  
Params.Strings ConnectionDef=SQLite\_Demo  
LoginPrompt   False
- **TFDQuery / ProductsTable**  
Connection    Sqlite\_demoConnection  
SQL.Strings   'SELECT \* FROM Products'

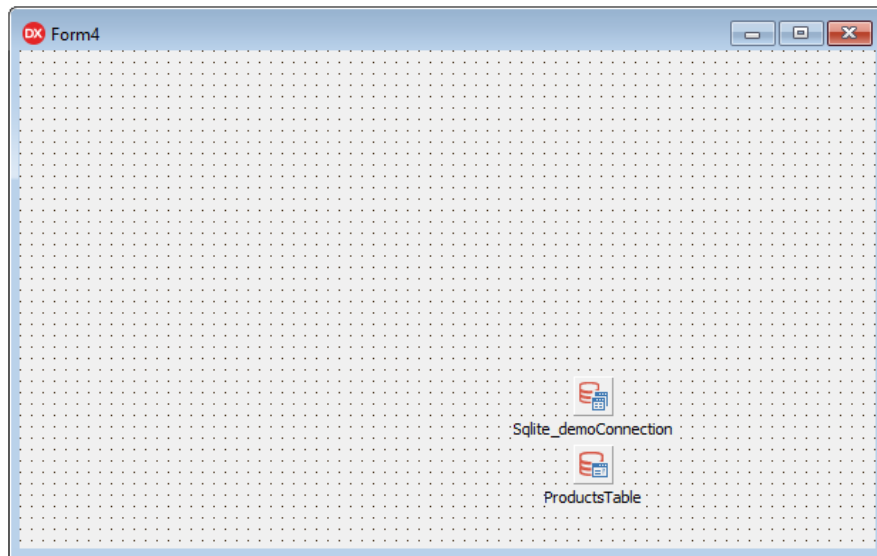


Abbildung 40: Neue Komponenten für den Datenzugriff auf dem Formular

Setzt man die Eigenschaft „Active“ der TFDTable/ProductsTable auf „True“ (was auch automatisch die Eigenschaft „Connected“ der TFConnection auf True setzt), so hat man schon im Designer die Möglichkeit der aktiven Datenbankanbindung: Es wird versucht die Verbindung herzustellen und die Datenmenge (die Datensätze) werden auch schon geladen. Ohne, daß man die Anwendung ausführen muss.

Um die Daten nun auch darzustellen fügen wir zwei weitere Komponenten hinzu: Eine TDataSource und ein TDBGrid. Räumt man das ein wenig auf, so kann das so aussehen:

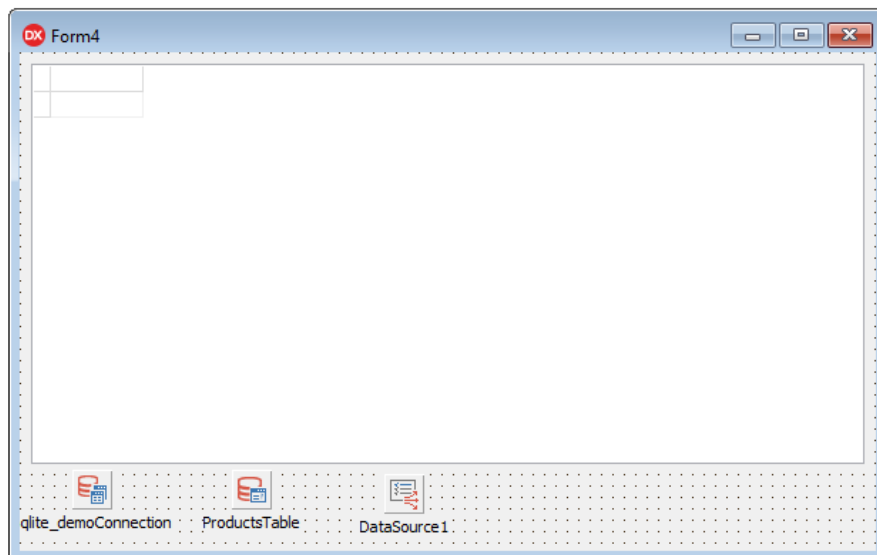
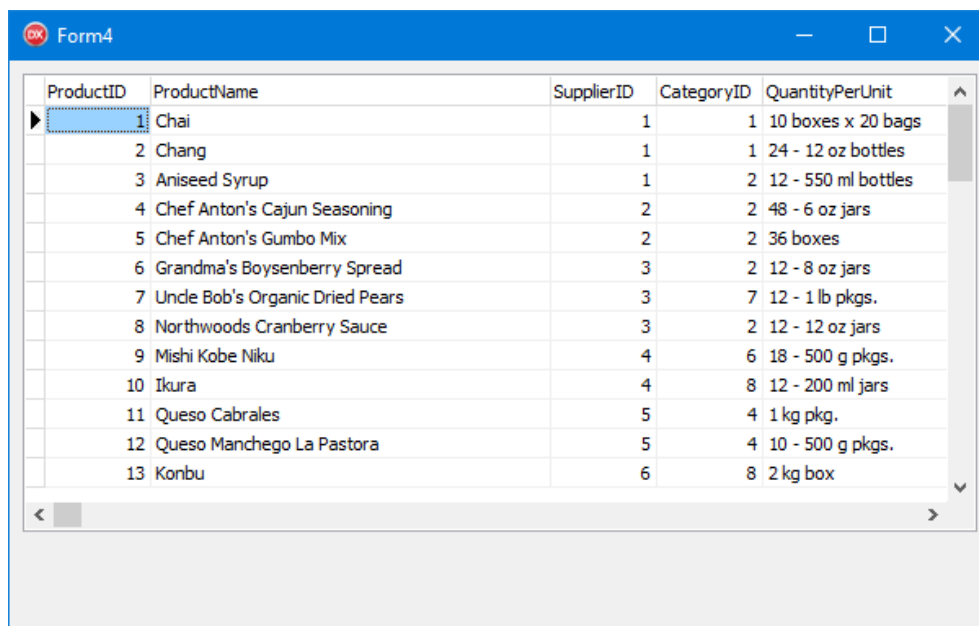


Abbildung 41: TDataSource hinzugefügt und ein wenig aufgeräumt

Wir verbinden nun noch die DataSource, die ein Bindeglied zwischen einer Datenmenge (Abfrage/Tabelle) und den visuellen, datensensitiven Komponenten darstellt (DBGrid, Tabellendarstellung der Datenmenge):

- TDataSource / DataSource1  
DataSet                      ProductsTable
- TDBGrid / DBGrid1  
DataSource      DataSource1

TFDConnection, TFDQuery und TDataSource sind dabei nicht-visuelle Komponenten, die zur Laufzeit nicht sichtbar sind. Startet man die Anwendung, dann sollte das ungefähr so aussehen:



The screenshot shows a Delphi application window titled 'Form4'. Inside the window is a table with the following data:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit
1	Chai	1	1	10 boxes x 20 bags
2	Chang	1	1	24 - 12 oz bottles
3	Aniseed Syrup	1	2	12 - 550 ml bottles
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars
5	Chef Anton's Gumbo Mix	2	2	36 boxes
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.
10	Ikura	4	8	12 - 200 ml jars
11	Queso Cabrales	5	4	1 kg pkg.
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.
13	Konbu	6	8	2 kg box

Abbildung 42: Die „fertige“ Anwendung

Mit dieser Anwendung kann man schon:

- In den Datensätzen navigieren
- Datensätze hinzufügen (ganz am Ende der Tabelle)
- Datensätze verändern (mittels F2 in den Änderungsmodus für ein Feld; sonst wird überschrieben)
- Datensätze löschen (Strg+Entf)

Änderungen werden generell beim Verlassen des Datensatzes automatisch in die Datenbank/Tabelle eingetragen. All das macht Delphi automatisch.

#### 6.1.6.4 Demoanwendung, Dokumentation und Beispiele

Wie bereits erwähnt, werden mit FireDAC auch viele Beispiele mitgeliefert. Man findet auch viel Dokumentation und auch Tutotrials. Die Beispiele finden sich hier:

C:\Users\Public\Documents\Embarcadero\Studio\20.0\Samples\Object  
Pascal\Database\FireDAC

Tutorials (auf dem Stand der aktuellen Delphi Community Edition) hier:

[http://docwiki.embarcadero.com/RADStudio/Rio/de/Tutorials\\_zu\\_Datenbanken\\_und\\_LiveBindings](http://docwiki.embarcadero.com/RADStudio/Rio/de/Tutorials_zu_Datenbanken_und_LiveBindings)

## 7 Anhang

### 7.0 Lösungen zu den Übungsaufgaben

#### 7.0.1 Kapitel 3 (Schnellstart)

Angegeben wird jeweils der relevante Code-Teil.

##### Aufgabe 1:

```
var i: Integer;  
begin  
  ListBox1.Clear;  
  for i := 12 to 144 do  
    ListBox1.Items.Add(IntToStr(i));  
  end;
```

##### Aufgabe 2:

```
procedure TForm1.Button1Click(Sender: TObject);  
var number: Integer;  
begin  
  number := TryStrToInt(Edit1.Text);  
  Label1.Caption := IntToStr(number div 2);  
end;
```

##### Aufgabe 3:

```
procedure TForm1.Button1Click(Sender: TObject);  
var number: Integer;  
begin  
  number := TryStrToInt(Edit1.Text);  
  Label1.Caption := IntToStr(number mod 2);  
end;
```

##### Aufgabe 4:

```
procedure TForm1.Button1Click(Sender: TObject);  
var number: Integer;  
begin  
  number := TryStrToInt(Edit1.Text);  
  if (number mod 2 = 0) then  
    Label1.Caption := 'Durch 2 teilbar'  
  else  
    Label1.Caption := 'Nicht durch 2 teilbar';  
end;
```

##### Aufgabe 5:

```
var i: Integer;  
begin  
  for i := 12 to 144 do begin  
    if (i mod 2 = 0) then  
      writeln(IntToStr(i));  
    end;  
  end;  
end;
```

**Aufgabe 6:**

```
var i: Integer;  
begin  
  for i := 12 to 144 do begin  
    if (i mod 3 = 0) and (i mod 5 = 0) then  
      writeln('FizzBuzz')  
    else if (i mod 3 = 0) then  
      writeln('Fizz')  
    else if (i mod 5 = 0) then  
      writeln('Buzz')  
    else  
      writeln(IntToStr(i));  
    end;  
  end;  
end;
```

**7.0.2 Kapitel 4.1 (Variablen und Konstanten)**

**Aufgabe 1:** Bezeichner dürfen in Delphi nicht mit einem Dollarzeichen und nicht mit einer Ziffer beginnen. Deshalb sind \$i und 2teZahl nicht zulässig. Ein Unterstrich ist hingegen erlaubt und seit Delphi 2006 auch Umlaute.

**Aufgabe 2:** An Position 1 würde es sich um eine globale Variable handeln, an Position 2 eine innerhalb der Unit globale Variable und an Position 3 eine lokale Variable innerhalb der Prozedur. Nur an Position 4 darf keine Variablendeklaration stehen. Hinweis: Seit Delphi 10.3 ist auch das aber auch erlaubt 😊

**Aufgabe 3:** Die Variable ist vom Typ Integer (ganze Zahl), weshalb ihr keine Zahl mit Nachkommastellen zugewiesen werden darf. Der Compiler reagiert in solchen Fällen mit einem Kompilierfehler („Inkompatible Typen „Integer“ und „Extended““. Die erste Antwortmöglichkeit ist also die richtige.

**DOWNLOAD DER KOSTENFREIEN COMMUNITY EDITION UNTER:** [www.embarcadero.com/de/downloads](http://www.embarcadero.com/de/downloads)

Embarcadero Inc | 10801 North Mopac Expressway | Building 1, Suite 100 | Austin, TX | 78759 U.S.A.

Embarcadero Germany GmbH | D-63225 Langen | Südliche Ringstrasse 175

Copyright © 2020 | Alle Rechte vorbehalten.