





# **Programmieren lernen mit Lazarus**

**Object Pascal als erste Programmiersprache**

Simon Ameis

2. Januar 2014

\* Arbeitstitel! Bitte darüber noch abstimmen.

Dieses Werk ist unter einem Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Germany Lizenzvertrag lizenziert. Um die Lizenz anzusehen, gehen Sie bitte zu <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> oder schicken Sie einen Brief an Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

# Inhaltsverzeichnis

<b>I. Installation</b>	<b>9</b>
1. Installation von Lazarus und Free Pascal	11
2. Ubuntu 11.04	13
<b>II. Programmieren mit Pascal</b>	<b>15</b>
<b>3. Das erste Programm</b>	<b>17</b>
3.1. Lazarus starten . . . . .	17
3.2. Die verschiedenen Fenster . . . . .	17
3.2.1. Das Hauptfenster . . . . .	17
3.2.2. Fenster der automatisch angelegten Unit . . . . .	18
3.2.3. Objektinspektor . . . . .	18
3.2.4. Quelltexteditor . . . . .	18
3.2.5. Nachrichtenfenster . . . . .	21
3.3. Ein Programm übersetzen . . . . .	21
3.4. Das erste Steuerelement . . . . .	22
3.4.1. Eine Schaltfläche auf dem Formular platzieren . . . . .	22
3.5. Weiter Elemente . . . . .	27
3.6. Ein kurzer Dialog . . . . .	29
3.6.1. Eine Meldung ausgeben . . . . .	30
3.6.2. Eine Eingabe abfragen . . . . .	30
3.7. Speichern . . . . .	31
<b>4. Variablen und Konstanten</b>	<b>33</b>
4.1. Lotto . . . . .	34
4.1.1. Formularentwurf . . . . .	34
4.1.2. Berechnung programmieren . . . . .	35
4.1.3. Ausgabe der Lottozahlen . . . . .	39
4.1.4. Zufallszahlen in Pascal . . . . .	41
<b>5. Kontrollstrukturen</b>	<b>43</b>

<b>III. Komponenten</b>	<b>45</b>
<b>6. Einführung in die Datenbanken</b>	<b>47</b>
6.1. Datenbanktheorie . . . . .	48
6.1.1. Begriffe . . . . .	48
6.1.2. Was ist eine Datenbank . . . . .	48
6.1.3. Desktop und Client-Server Datenbankarten . . . . .	49
6.1.4. Relationale Datenbanken . . . . .	50
6.1.5. Grunddaten . . . . .	52
6.2. DDL Datendefinitionssprache . . . . .	54
6.3. DML Datenveränderungssprache . . . . .	54
6.3.1. SELECT . . . . .	54
6.3.2. Beispiele zu SELECT . . . . .	56
6.3.3. INSERT . . . . .	59
6.3.4. Beispiele zu INSERT . . . . .	59
6.3.5. UPDATE . . . . .	59
6.3.6. Beispiele zu UPDATE . . . . .	60
6.3.7. DELETE . . . . .	60
6.3.8. Beispiele zu DELETE . . . . .	61
6.4. DCL Datenkontrollsprache . . . . .	62
6.5. SQLdb - Serverdatenbank Komponenten . . . . .	63
6.5.1. Beschreibung . . . . .	63
6.5.2. TxxxConnection . . . . .	66
6.5.3. TMySQL50Connection . . . . .	66
6.5.4. TMySQL41Connection . . . . .	67
6.5.5. TMySQL40Connection . . . . .	67
6.5.6. TOracleConnection . . . . .	67
6.5.7. TPQConnection . . . . .	67
6.5.8. TODBCCConnection . . . . .	67
6.5.9. TSQLTransaction . . . . .	67
6.5.10. TSQLQuery . . . . .	68

# Anmerkungen, die noch zu klären sind

- Besondere Namen und andere Hinweise/weitergehende Erläuterungen sollten irgendwie vom Rest des Textes hervorgehoben werden.
- Wie werden Dateipfade/-namen ausgezeichnet? Ich habe `\file{datei}` als globalen Alias für `\texttt{}` definiert, damit man alles auf einmal einstellen kann.
- Wie werden Befehlsanweisungen fürs GUI ausgezeichnet (Folge dem Menü->Dann klicke auf->Fenster so und so)? Vorerst erfüllt `\menu{}` diese Aufgabe und verwendet direkt `\enquote{}`.
- Auszeichnung von Quelltexten, Bezeichnern, Typen, usw. im Fließtext
- Package `unicode-math` verwenden, sobald ich auf Debian Squeeze upgegraded habe.
- Ich habe das Paket `listings` eingebunden, daher:
  - Pascal ist Standard-Sprache; einige Anpassungen (Schlüsselwörter) müssen aber noch angegeben werden
  - Das Wort »Text« wird immer als Schlüsselwort erkannt.
  - `\lstinputlisting[caption=Anzeigename]{datei/pfad.pas}`
  - `\begin{lstlisting} //Pascal-Quelltext \end{lstlisting}`
  - Siehe auch Dokumentation unter <http://mirror.ctan.org/macros/latex/contrib/listings/listings.pdf>.
- Einbinden von Quelltexten
- Einbinden von Bildern
  - Wie erstellt man Screenshots mit einer Druckauflösung von 300 DPI? Das sind in etwa 1700 Pixel in der Breite (bei DIN A4: 14,5 cm \* 300 DPI).
  - Soll ein bestimmtes Widgetset für die Screenshots her halten? (ich meine nein; wenn doch, GTK 2)
  - Alle Screenshots sollten einer stabilen Lazarus-Version entstammen (oder zumindest so aussehen). 0.9.30 ist gerade aktuell
  - Vielleicht könnte man auch einen LCL-Hack entwickeln, der ein GTK-Fenster dank Cairo als PDF rendert.

## Inhaltsverzeichnis

- Wir sollten eine Wikiseite (o. Ä.) einrichten, wie bestimmte Dinge benannt bzw. übersetzt werden. Heißt das jetzt Form, Formular, Fenster; String oder Zeichenkette.
- Shortcuts/Tastaturbefehle? Verwirren die mehr oder findet der Benutzer die alleine? (ich denke letzteres)
- Unicode! Alle Dokumententeile ( $\text{\LaTeX}$ -Dokument, Quelltexte usw.) sollten in UTF-8 kodiert sein. Dank  $\text{\XeLaTeX}$  können dann sogar Umlaute im Paket *listings* genutzt werden.
- Verzeichnislayout, damit wir alle Dateien wiederfinden. Ich schlage ein Verzeichnis pro Kapitel vor. Jedes Verzeichnis enthält Unterverzeichnisse `src` und `img` für Quelltext bzw. Bilder.
- Wie kann man unter jedem Kapitelnamen die Autorennamen angeben?
- Wie kann man einzelne Worte in einer bestimmten Sprache markieren?

Weitere Hinweise zur Gestaltung und zur Erstellung:

- Satzsatz mit  $\text{\XeLaTeX}$
- Schriften:  
**Serifen** URW Bookman L  
**Sans** URW Gothic L  
**Mono** Inconsolata



# Vorwort

Wer schreibt das?



# **Teil I.**

## **Installation**



# 1. Installation von Lazarus und Free Pascal

Die Installation gestaltet sich dank fertiger Installationspakete unter den drei Betriebssystemen MacOS X, Linux und Windows gleichermaßen einfach. Andere Betriebssysteme, für die Sie Anwendungen mit Lazarus entwickeln können, – wie Windows CE oder Android – eignen sich kaum zur Anwendungsentwicklung.

Die häufigsten Probleme, die während der Installation auftreten, betreffen entweder die Konfiguration des Free Pascal Compilers oder die Installation wurde nicht vollständig oder in der vorgesehenen Reihenfolge durchgeführt.

Sollten Sie auf Problem stoßen, gibt es zwei Orte, an denen Sie auf jeden Fall Hilfe finden werden.

Als erstes seien hier die englischsprachigen Mailing-Listen genannt. Hier tauschen sich nicht nur Lazarus-Anwender aus der ganzen Welt aus, auch die Entwickler von Lazarus sind hier unterwegs. **TODO Link**

Für die deutschsprachige Community gibt es unter <http://www.lazarusforum.de/> ein kostenloses Internetforum, in dem Sie ebenfalls kompetente Hilfe von erfahrenen Lazarus-Benutzern finden können.

Während Sie für Software für Windows immer ein Installationspaket vom Hersteller der Software herunterladen und installieren, stellen fast alle Linux-Distributionen die Software direkt zur Verfügung. Hier haben Sie eine Paketverwaltung, die für Sie die einzelnen Software-Pakete aus dem Internet herunterlädt und installiert. Die beiden häufigsten Paketverwaltungen sind das Red Hat Package Management (RPM), welches unter anderem bei Red Hat und SuSe zum Einsatz kommt, und die Debian-Pakete, die von Debian stammen und damit auch bei der populären Distribution Ubuntu zum Einsatz kommen.



## 2. Ubuntu 11.04

Die gesamte Installation können Sie über die Synaptic-Paketverwaltung durchführen, die Sie über das Anwendungsmenü starten können.

Ubuntu selbst enthält nur ältere Versionen des Free Pascal Compilers und von Lazarus in den eigenen Paketquellen. Von der Lazarus-Community werden hingegen auch die aktuellen Lazarus-Versionen in fertigen Paketen zur Installation angeboten. Sie müssen also zuerst eine neue Paketquelle eintragen, damit Synaptic die neuen Pakete findet und installiert.

1. Starten Sie Synaptic und geben Sie das Passwort des Systemadministrators ein.
2. Wählen Sie den Menüpunkt »Einstellungen → Paketquellen«.
3. In dem Fenster »Software-Paketquellen« wechseln Sie zu dem Reiter »Andere Software«.
4. In der linken unteren Ecke finden Sie die Schaltfläche »hinzufügen ...«.
5. Als *APT-Zeile* tragen Sie »deb <http://www.hu.freepascal.org/lazarus/> lazarus-stable universe« ein und bestätigen mit einem Klick auf »Software-Paketquelle hinzufügen«.
6. Schließen Sie das Fenster und aktualisieren Sie die Paketquellen.
7. Nun können Sie alle benötigten Pakete installieren, wobei Sie darauf achten müssen, welche Version – die neue von der Lazarus-Community oder die alte aus den Ubuntu-Paketquellen – installieren.

Damit Lazarus ohne Einschränkungen funktioniert müssen Sie mindestens die folgenden Pakete auswählen, wobei Synaptic automatisch weitere Pakete, die ebenfalls zwingend benötigt werden, auswählt und installiert.

Überprüfen

- a) lazaurs-0.9.30
- b) lcl-0.9.30
- c) fpc





**Teil II.**

# **Programmieren mit Pascal**



## 3. Das erste Programm

Die Installation von Lazarus und Free Pascal ist abgeschlossen und alles ist fertig konfiguriert. Jetzt ist es an der Zeit mit dem ersten Programm zu starten.

### 3.1. Lazarus starten

**Windows** Wenn eine Verknüpfung auf dem Desktop angelegt wurde, können Sie Lazarus mit einem Doppelklick darauf starten.

Ansonsten gehen Sie über das Startmenü: »Start->Alle Programme->Lazarus->Lazarus«

**Linux** Hängt von der Installation ab

**Mac OS X** kenn' ich nicht

### 3.2. Die verschiedenen Fenster

In Lazarus arbeiten Sie mit im Gegensatz zu vielen anderen Programmierumgebungen mit mehreren Fenstern. Es gibt viele verschiedenen Fenster für alle Aufgaben, die beim Entwickeln von Anwendungen anfallen. Für den Anfang beschränken wir uns aber auf die Fenster, die Lazarus beim ersten Start präsentiert.

Ein Fenster ist in der Regel dauerhaft geöffnet und man kann damit arbeiten. In einem Dialog hingegen nimmt man Einstellungen vor oder bestätigt eine Aktion. Er ist nur kurzzeitig geöffnet und wird hinterher wieder geschlossen.



#### 3.2.1. Das Hauptfenster

Mit dem langen Hauptfenster kann man alle Aktionen steuern. Es bietet über das Menü Zugriff auf sämtliche Funktionen und Einstellungen. Dazu finden sich einige Schaltflächen für häufig benötigte Funktionen sowie die alle Komponente, die Sie auf einem Formular platzieren können. All diese Komponenten sind in verschiedene Gruppen je nach ihrer Funktion aufgeteilt.

#### Ein neues Projekt anlegen

Beim ersten Start erstellt Lazarus automatisch ein neues Projekt und bei jedem weiteren Start wird das letzte Projekt wieder geladen.

Ist das Standard?

### 3. Das erste Programm

Wenn Sie innerhalb dieses Buches an einem Punkt, an dem nichts mehr geht, ankommen, können Sie hier auch ein neues, leeres Projekt starten. Dazu gehen Sie wie folgt vor:

1. Im Menü »Datei->Neu ...« auswählen. Daraufhin öffnet sich ein neuer Dialog.
2. In der Liste wählen Sie unter »Projekt« den Projekttyp »Anwendung« aus und klicken dann den Button »OK« an.

Wie Sie sehen bietet Lazarus von Haus aus einige Vorlagen für verschieden Projekte und Dateien, die Sie in einem umfangreichen Projekt benötigen. Über zusätzliche Pakete (im Deutschen wird durchaus auch der englische Begriff *Package* verwendet), können noch weitere Vorlagen installiert werden.

Für Sie ist aber anfangs nur der Projekttyp »Anwendung« interessant. Er bietet die Unterstützung Programme mit einer grafischen Benutzeroberfläche – also genau das, was Sie von Ihrem Betriebssystem her kennen.

#### 3.2.2. Fenster der automatisch angelegten Unit

Eine grafische Anwendung besteht aus einem oder mehreren Fenstern. Daher hat Lazarus für das Projekt automatisch ein leeres Formular angelegt. Hier werden alle Steuerelemente abgelegt und eingestellt, wie sich das Programm präsentiert.

Eventuell hat sich dieses Fenster unter dem Quelltexteditor versteckt. Verschieben Sie den Fenster bis Sie es gefunden haben.

#### 3.2.3. Objektinspektor

Auf dem Hauptformular können wir Komponenten platzieren, verschieben, die Größe einstellen, aber das reicht natürlich nicht aus. Wenn Sie auf eine Komponente dem Hauptformular anklicken, können Sie hier alle Eigenschaften ändern. Andersherum können Sie genau so gut eine Komponente im Objektinspektor auswählen.

Dass eine Komponente (außer dem Formular) ausgewählt wurde, erkennen Sie an den kleinen Punkten, die in den Ecken der Komponente angezeigt werden.

#### 3.2.4. Quelltexteditor

Zu jedem Formular gehört auch immer ein Stück Quelltext, in dem steht, was passieren soll, wenn der Benutzer auf eine Schaltfläche oder ein Textelement klickt. Daher hat Lazaurus diesen Quelltext automatisch für Ihr Formular erstellt. Eine Datei, in der Pascal-Quelltext gespeichert wird, nennt man *Unit*.

Unit

```
1  unit Unit1;  
  
   {$mode objc}{$H+}  
  
5  interface
```

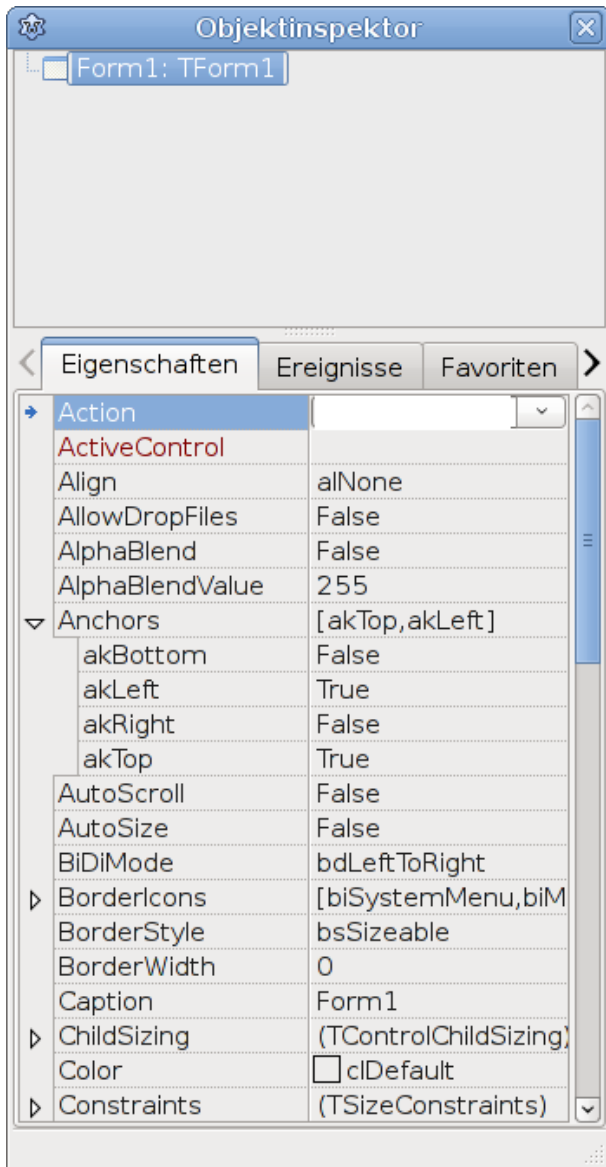


Abbildung 3.1.: Der Objektinspektor

### 3. Das erste Programm

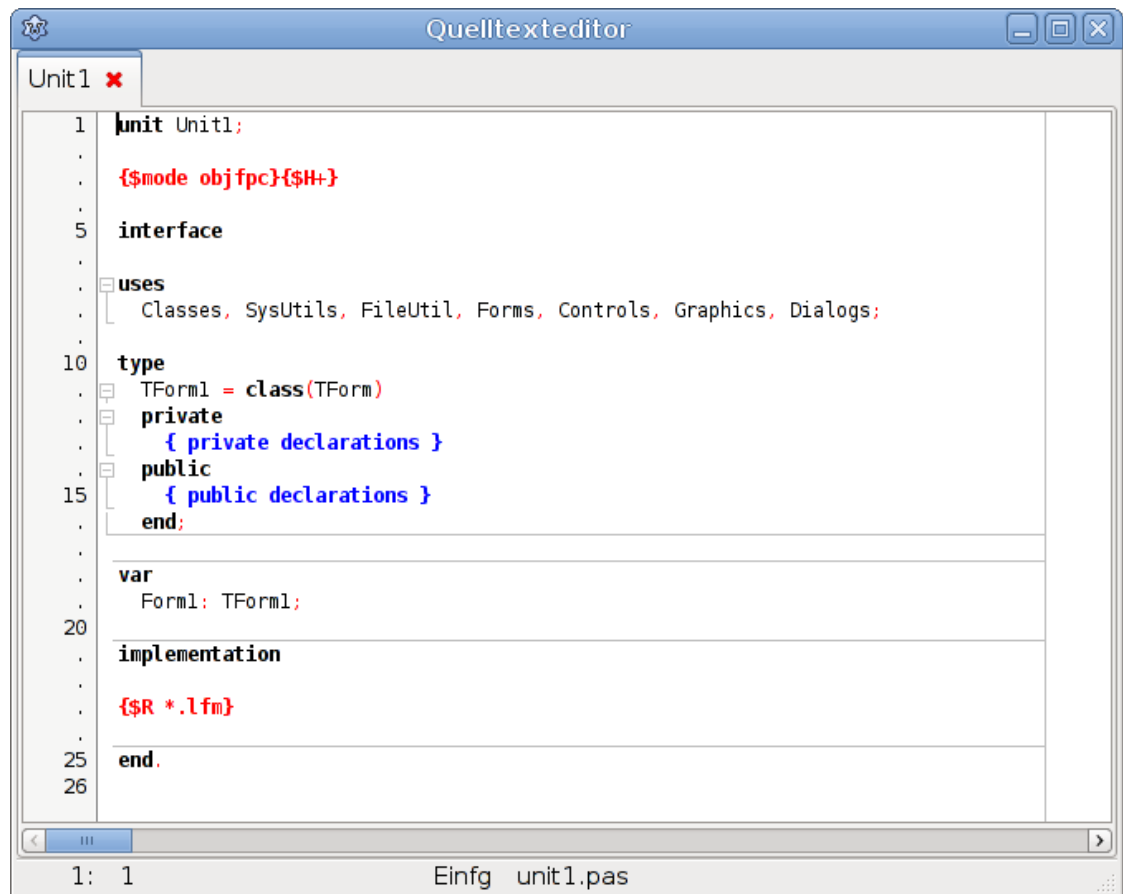


Abbildung 3.2.: Der Quelltexteditor in Lazarus

```

uses
  Classes , SysUtils , FileUtil , Forms , Controls , Graphics ,
    Dialogs ;
9
type
  TForm1 = class (TForm)
    private
13    { private declarations }
    public
    { public declarations }
    end ;
17
var
  Form1 : TForm1 ;

21 implementation

  {$R *.lfm}

25 end .

```

Listing 3.1: unit1.pas

In einer Unit sind aber nicht alle Daten zu einem Formular gespeichert. Sie enthält nur eine Auflistung aller Komponenten. Wie sie sich aber verhalten (d. h. ihre Eigenschaften) werden in einer zusätzlichen Datei gespeichert, die in Zeile 23 mit `{$R *.lfm}` eingebunden wird.

### 3.2.5. Nachrichtenfenster

Hier zeigt Lazarus alle wichtigen Meldungen, die beim Übersetzen des Quelltextes vom Free Pascal Compiler ausgegeben werden an. Stimmt irgendetwas nicht mit Ihrem Quelltext, finden Sie hier einen Hinweis darauf, was nicht stimmt.

Mit einem Klick auf eine Meldung springt der Quelltexteditor direkt an die Stelle, wo diese Meldung auftritt.

## 3.3. Ein Programm übersetzen

Zugegen, das Programm hat noch nicht besonders viel Inhalt. Trotzdem können wir es schon starten. Dazu wählen Sie im Menü des Hauptfensters »Start« den Eintrag mit dem Titel »Start« aus. Alternativ können Sie auch auf den kleinen grünen Pfeil auf der linken Seite des Hauptfensters klicken.

### 3. Das erste Programm

**Was passiert jetzt?** Der Free Pascal Compiler übersetzt den Quelltext des Programms und erstellt daraus ein lauffähiges Programm, dass gestartet werden kann. Dazu wird jede einzelne Unit des Projektes übersetzt und zusammen mit weiteren (bereits übersetzten) Units und anderen Programmteilen wie den Formulardaten in einer einzelnen Programmdatei zusammengesetzt.

Sobald das geschehen ist, startet Lazarus Ihr Programm und kurz darauf sehen Sie ein leeres Formular vor sich. Sehr viel können Sie damit leider noch nicht machen. Sie können die Größe verändern, es über den Desktop verschieben und es schließen. Tun Sie das.



Wenn ein Programm ausgeführt wird, wird dies als Laufzeit (engl. *runtime*) bezeichnet. Während Sie mit Lazarus ein Programm entwickeln befindet sich es sich hingegen in der Entwurfs- oder Designzeit. Während des Programmierens werden Sie des Öfteren von der Entwurfs- in die Laufzeit wechseln um gerade geschriebene Funktionen zu testen.

## 3.4. Das erste Steuerelement

Steuerelement

In Lazarus wird ganz grob zwischen Komponente und Steuerelementen unterschieden. Ein Steuerelement ist alles, was später im laufenden Programm zu sehen ist und vom Benutzer benutzt werden kann oder etwas anzeigt.

Komponente

Als Komponente hingegen bezeichnet man alles, was auf einem Formular platziert und dessen Eigenschaften über den Objektinspektor geändert werden können. Daraus folgt, dass alle Steuerelemente auch Komponenten sind, aber nicht alle Komponenten werden auch zur Laufzeit angezeigt.

### 3.4.1. Eine Schaltfläche auf dem Formular platzieren

Wählen Sie Auf dem Hauptformular den Reiter »Standard« aus und suche Sie ein Bild, das Ihren Vorstellungen einer Schaltfläche am nächsten kommt. Wenn Sie mit der Maus eine kurze Zeit über einem Bild verweilen, wird der Name dieser Komponente angezeigt. Wenn Ihnen »TButton« angezeigt wird, haben Sie das richtige gefunden.

Klicken Sie nun auf diese Komponente und sie wird ausgewählt. Jetzt klicken Sie an eine beliebige Position auf dem noch leeren Formular. Dort erscheint jetzt eine Schaltfläche (oder auch aus dem Englischen: Button). Über die Punkte in den Ecken der Komponente können Sie ganz bequem per Maus ihre Position und Größe verändern. Dazu einfach die Maus über einen solchen Punkt bewegen, die linke Maustaste drücken (und halten) und dann die Maus bewegen.

### Beschriftung ändern

Jedes Steuerelement, mit dem der Benutzer eine Aktion starten kann (am Häufigsten sind das Schaltflächen und Menüeinträge), sollte auf den ersten Blick verraten, was passiert, wenn es benutzt wird. Eine aussagekräftige Beschriftung ist daher Pflicht.



Um die Beschriftung der Schaltfläche zu ändern, benutzen wir den Objektinspektor. Dieser teilt sich in zwei Bereiche auf: Im oberen Bereich werden alle Komponenten, die auf dem Formular abgelegt sind, angezeigt. Die meisten Komponenten können untergeordnete Komponenten enthalten. Zum Beispiel enthält ein Menü alle Menüeinträge. Die Baumstruktur zeigt dabei diese Beziehungen an.

Auf Ihrem Formular befindet sich zur Zeit nur eine Schaltfläche, die direkt dem Formular untergeordnet ist. Wählen Sie jetzt die Schaltfläche aus.

Im unteren Bereich können die Eigenschaften des ausgewählten Objektes verändert werden. Wie Sie sehen, gibt es hier vier verschiedene Reiter:

**Eigenschaften** steuern das Verhalten einer Komponente.

**Ereignisse** werden ausgelöst, wenn irgendetwas passiert (z. B. Benutzeraktion).

**Favoriten** Häufig genutzten Eigenschaften und Ereignissen.

**Bedingte Eigenschaften** Eigenschaften, die nur unter einer bestimmten Plattform unterstützt werden.<sup>1</sup>

gibt es alle in unserer Version?

Sollte der Reiter »Eigenschaften« noch nicht ausgewählt sein, wählen Sie ihn jetzt aus. Unterhalb des Reiters sehen Sie jetzt eine Liste aller verfügbaren Eigenschaften der Schaltfläche mit ihren aktuellen Werten. Lazarus unterstützt Sie, indem es Ihnen eine Fehlermeldung anzeigt, wenn Sie einen ungültigen Wert für eine Eigenschaft eingeben.<sup>2</sup>

Suchen Sie die Eigenschaft *Caption* und ändern Sie den Wert in »Text anzeigen«.

Caption

Diese Eigenschaft gibt es bei jedem Steuerelement, das eine Beschriftung enthält.

#### Name

Eine Komponente sollte auch einen Namen haben, aus dem hervorgeht, wofür sie gebraucht wird. Festgelegt wird er in der Eigenschaft *Name*. Er darf nur alphanumerische Zeichen enthalten (a–z, A–Z, 0–9) sowie den Unterstrich (–) enthalten, wobei das erste Zeichen keine Ziffer sein darf. In jedem Formular darf jeder Name nur ein einziges Mal vorkommen.

Wie Sie Ihre Komponenten benennen, bleibt allein Ihnen überlassen. Damit Sie aber den Überblick behalten, sollten Sie sich aber ein Schema überlegen, dass Sie für ein Projekt einhalten. Häufig besteht der Name aus einer Abkürzung für die Art der Komponente (bspw. Btn für Schaltflächen von engl. buttons) und einer möglichst treffenden Bezeichnung seiner Funktion. Ob Sie die Funktion in Englisch oder in Deutsch (oder einer anderen Sprache) halten, hängt auch davon ab, ob Sie mit anderen Entwicklern zusammen an einem Projekt arbeiten und welche Sprache Sie bevorzugen.

In Ihrem Beispiel könnten Sie die Schaltfläche »BtnShowText« nennen, denn im nächsten Abschnitt wird sie ihre Funktion – einen Text ausgeben – erhalten.

<sup>1</sup>Mit Lazarus können grafische Programme für verschiedene Sammlungen von Steuerelementen erstellt werden. Diese sind teilweise vom Betriebssystem abhängig und unterstützen nicht immer alle Eigenschaften aller Komponenten.

<sup>2</sup>In einigen wenigen Fällen wird keine Fehlermeldung ausgegeben sondern stillschweigend auf den Wertebereich begrenzt.

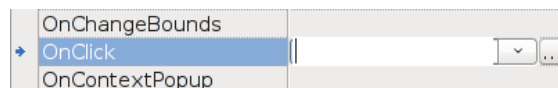
### 3. Das erste Programm

#### Ereignisse!

Wie bereits gesagt, definieren Ereignisse die Art und Weise, wie ein Steuerelement (oder allgemeiner: eine Komponente) auf ein eingetretenes Ereignis reagiert. Dazu müssen Sie für jedes Ereignis, auf das reagiert werden soll, eine Funktion im Quelltext verfassen. Diese wird beim übersetzen in ausführbaren Programmcode übersetzt und aufgerufen, wenn das Ereignis eintritt.

Funktionen die auf Ereignisse reagieren, nennt man *Ereignisbehandlungsroutinen* oder auf Englisch *Event-Handler*.

OnClick  
Bei einem Mausklick des Benutzers wird das *OnClick*-Ereignis ausgelöst. Auch wenn die Eigenschaft *OnClick* heißt, wird sie auch ausgelöst, wenn der Benutzer die **Enter**- oder die **Leertaste** betätigt.



Suche Sie im Objektinspektor nach dieser Eigenschaft. Wenn Sie sie auswählen, können Sie entweder einen Funktionsnamen eingeben und mit **Enter** bestätigen oder auf die kleine Schaltfläche mit den drei Punkten daneben klicken. Dann wählt Lazarus automatisch einen passenden Namen aus.

In beiden Fällen ergänzt Lazarus Ihren Quelltext im Quelltexteditor um ein Grundgerüst für diese Funktion. Zu diesem Grundgerüst gehören zwei Teile, die im Folgenden erläutert werden.

```
1  TForm1 = class(TForm)
    BtnShowText: TButton;
    procedure BtnShowTextClick(Sender: TObject);
4  private
    { private declarations }
    public
    { public declarations }
8  end;
```

Listing 3.2: unit1.pas: Definition des Formulars

In Zeile 14 steht `TForm1 = class(TForm)`. Wie Sie vielleicht schon gemerkt haben, steht hier der Name des Formulars, aber mit dem Buchstabe »T« davor. Dies hier ist nämlich die Definition des Formulars. Genau genommen ist es eine Definition einer Formular-Klasse. Man kann nämlich jedes Formular so oft man will erzeugen (auch wenn es in diesem Fall eher sinnlos wäre), und all diese Formulare verhalten sich, wie es diese Klasse beschreibt.

Formularklasse

In der Zeile darunter finden Sie den Namen der Schaltfläche. Lazarus hat ihn automatisch eingefügt, als Sie das Steuerelement auf dem Formular platziert haben. Darunter ist eine Funktion angegeben. Den Namen hat Lazarus automatisch gewählt.

Die Definition der Formulkasse endet erst mit dem Schlüsselwort **end**; in Zeile 21. Die Schaltfläche und die Funktion gehören also dazu. Um die beiden Schlüsselworte **private** und **public** müssen Sie sich keine Gedanken machen. Sobald sie wichtig sind, werden Sie erfahren, was sie bewirken.

**Kommentare** Vielleicht haben Sie schon die vielen Zeilen, in denen Text in geschweiften Klammern steht, bemerkt. Das sind Kommentare und werden vom Compiler beim Übersetzen des Quelltextes in ein ausführbares Programm ignoriert. Eine andere Möglichkeit ist (\* *Kommentar* \*). Beide Varianten können Sie auch über einen Zeilenumbruch hinaus verwenden.

Kommentare

Mit zwei Schrägstrichen hintereinander beginnen Sie einen Kommentar, der nur bis zum jeweiligen Zeilenende reicht (// am Ende der Zeile ist Schluss mit Kommentar).

Eine Sonderrolle nehmen geschweifte Klammern ein, deren erstes Zeichen im Kommentar ein Dollar-Zeichen ist (bspw. *{ \$mode objfpc }* in Zeile 3). Dies sind sogenannte Compiler-Schalter. Das heißt, sie haben eine besondere Bedeutung für den Compiler. Compiler-Schalter können sich entweder die *Verarbeitung* des Quelltextes oder die *Ausgabe* des Compilers auswirken.

```

1 procedure TForm1.BtnShowTextClick(Sender: TObject);
  begin
4 end;

```

Listing 3.3: unit1.pas: Funktion für das *OnClick*-Ereignis

**Funktionen** In der Definition der Formulkasse steht bisher nur der Funktionsname, nicht aber, *was* die Funktion überhaupt machen soll. Das wird an einer anderen Stelle gemacht, nämlich in Zeile 30–35. Die obere Definition der Formulkasse oder des Funktionsnamen nennt man auch *Deklaration* oder *Bekanntmachung*. Steht im Quelltext aber, was die Funktion macht, spricht man von der *Implementation* (engl. implementation = Realisierung).

Die Implementation besteht aus mehreren Teilen, die auch schon in der Deklaration auftauchen:

1. Das Schlüsselwort **procedure**.<sup>3</sup>
2. Dem Funktionsnamen TForm1.BtnShowTextClick. Er besteht hier aus dem eigentlichen Funktionsnamen, wie er in der Deklaration angegeben ist, und (davor) dem Namen der Formulkasse, zu dem die Funktion gehört. So können verschiedene Formulare Funktionen mit den gleichen Namen haben, aber die Eindeutigkeit der Namen ist weiterhin gewährleistet.

<sup>3</sup>Genau genommen gibt es noch eine weitere Funktionsart mit dem Schlüsselwort **function**, aber alles zu seiner Zeit.

### 3. Das erste Programm

3. Einer Parameterliste in runden Klammern (Sender: TObject). Zur Zeit benötigen Sie noch keine Parameter und können diese getrost ignorieren. Trotzdem ist es gut zu wissen, was hier steht.
4. Der Teil bis hier hin (Schlüsselwort, Name und Parameterliste) heißt Funktionskopf.
5. Dem Funktionsrumpf. Er wird durch die beiden Schlüsselworte **begin** und **end**;<sup>4</sup> begrenzt. Dazwischen kommt der Quelltext, der ausgeführt werden soll, wenn auf die Schaltfläche geklickt wird.

Ergänzen Sie nun Ihren Funktionsrumpf um die fehlende Zeile aus Listing 3.4 und testen Sie die Auswirkungen.

```
32 procedure TForm1.BtnShowTextClick(Sender: TObject);  
   begin  
       Caption = 'Hallo, ich bin ein Formular!';  
   end;
```

Listing 3.4: Eine Beschriftung ändern

Was hat sich geändert? Richtig – die Beschriftung des Formulars. Wenn Sie die Beschriftung der Schaltfläche ändern wollen, müssen Sie den Namen der Schaltfläche mit angeben:

```
34   BtnShowText.Caption := 'Hallo, ich bin eine Schaltfläche.';
```

Listing 3.5: Beschriftung einer Schaltfläche ändern

#### Zuweisung

Jetzt aber nochmal ganz langsam: Was hier passiert ist eine *Zuweisung*. Eine Zuweisung besteht immer aus zwei Teilen: auf der rechten Seite der Wert und auf der linken Seite das Ziel. Dazwischen steht der *Operator* :=.

Der Wert 'Hallo, ich bin eine Schaltfläche' muss in diesem Falle eine Folge von Zeichen (Zeichenkette), die von Apostrophen (') umschlossen ist, sein.

Bisher haben Sie nur zwei Elemente, die einen Wert entgegen nehmen können: das Formular selbst und die Schaltfläche darauf. Beide Steuerelemente können selbst aber keine Zeichenkette entgegennehmen, was in etwa die Bedeutung »Die Schaltfläche ist jetzt eine Zeichenkette« hätte.

Die Beschriftung hingegen ist eine Zeichenkette und daher kann ihr auch eine andere Zeichenkette zugewiesen werden. Die Beschriftung der Schaltfläche kann man über BtnShowText.Caption ansprechen. Im allgemeinen wird zuerst der Name der Komponenten und dann nach einem Punkt die Eigenschaft angegeben.

<sup>4</sup>Schon gemerkt? Hier taucht schon wieder das Schlüsselwort **end** auf. Es ist ganz so, wie es aussieht: **end** steht immer an einem Ende.

Bei dem Formular ist das ein wenig anders. Es besitzt zwar auch einen Namen und `Form1.Caption := 'Neue Ueberschrift'` führt oft zum gewünschten Ergebnis. `Form1` spricht aber immer ein bestimmtes Formular an. Da aber wie in Abschnitt 3.4.1 auf Seite 24 gesagt, eine ganze Klasse an Formularen definiert wird, sollte auch die Überschrift eines jeden Formulars dieser Klasse geändert werden. Dazu gibt es den Namen `Self`. Dieser greift immer auf das richtige Formular zu.

Wird keine Name einer Komponente angegeben, wird nachgeschaut, ob das Formular eine solche Eigenschaft besitzt. Im Erfolgsfall wird auf diese zugegriffen (`Self.Caption` ist also das Selbe wie nur `Caption`); ansonsten haben Sie fehlerhaften Quelltext und werden von Lazarus oder dem Free Pascal Compiler auch darauf hingewiesen.

Es ist vollkommen egal, wie Sie den Namen der Steuerelement oder deren Eigenschaften schreiben `BtnShowText.Caption` ist gleichbedeutend mit `BTNSHOWTEXT.cApTiOn` – Groß- und Kleinschreibung spielen keine Rolle.



### 3.5. Weiter Elemente

Ein Programm mit einer Schaltfläche, mit der man Aktionen auslösen kann, reicht aber in den seltensten Fällen aus. Nach der Schaltfläche sind Anzeigefelder für kurze Textnachrichten und Eingabefelder die wichtigsten Steuerelemente.

Suchen Sie in der Komponentenliste im Hauptfenster nach den Komponenten mit dem Namen »TLabel« und »TEdit« und platzieren Sie jeweils ein Element auf dem Formular. Die beiden neuen Steuerlemente werden von Lazaurus automatisch mit »Label1« und »Edit1« benannt.

#### TLabel

`TLabel` beschränkt sich ganz und gar auf die Anzeige eines Textes, den Sie in der Eigenschaft *Caption* angeben. Es folgt eine kurze Beschreibung der wichtigsten Eigenschaften, über die Sie steuern können, wie der Text angezeigt werden soll. Für den Rest dieses Kapitel sind sie jedoch eher nebensächlich.

**Autosize** stellt ein, ob das Label automatisch auf die Größe des Textes einstellt. Sie müssen diese Eigenschaft auf »False« stellen, damit die Auswirkungen anderer Eigenschaften sichtbar werden.

**Alignment** kontrolliert, wie der Text *horizontal* (Links, Rechts, Mitte) ausgerichtet wird.

**Layout** kontrolliert die *vertikale* Ausrichtung (Oben, Unten, Mitte) des Textes.

**WordWrap** kann entweder auf »True« oder »False« gestellt werden. Im ersten Fall wird der Text an einem Wortende umgebrochen, wenn er sonst nicht mehr in das Steuerelement hinein passen würde.

### 3. Das erste Programm

Bei allen anderen Eigenschaften möchten wir Sie dazu ermutigen, diese einfach auszuprobieren. Schiefgehen kann gar nichts. Im Notfall können Sie immer noch mit einem neuen Projekt beginnen, auch wenn es in der Regel ausreicht die Komponente einfach zu löschen und eine neue auf dem Formular zu platzieren.

#### TEdit

Wenn Sie sich die Eigenschaften dieses Eingabefeldes anschauen, werden Sie feststellen, dass es gar keine Eigenschaft *Caption* besitzt. Eine *Beschriftung* hat schließlich auch recht wenig Sinn. Den eingegebenen Text können Sie über die Eigenschaft *Text* einstellen und auslesen.

#### Die Komponenten miteinander verbinden

Nun ist es an der Zeit die Steuerelemente miteinander in Verbindung zu setzen. Mit den bisherigen Kenntnissen, können Sie noch nicht viel tun, aber immerhin ein wenig. Ändern Sie den Funktionsrumpf der Ereignisbehandlungsroutine der Schaltfläche entsprechend dem Listing:

```
1 Label1.Caption := 'Die Eingabe: ' + Edit1.Text;
```

Mit einem Klick auf die Schaltfläche wird nun dem Text aus dem Eingabefeld die Zeichenkette 'Die Eingabe: ' vorangestellt und auf dem Label angezeigt. Der Operator + verbindet zwei Zeichenketten miteinander. Die erste ist direkt fest in den Quelltext geschrieben; die zweite wird vom Eingabefeld ausgelesen.

**Zeilenumbruch einfügen** Ein Label kann auch Zeilenumbrüche anzeigen. Die oben angesprochenene Eigenschaft *WordWrap* kontrolliert den *automatischen* Zeilenumbruch. Manuelle Zeilenumbrüche können Sie über den Objektinspektor eingeben: Wenn Sie die Eigenschaft auswählen, erscheint eine Schaltfläche mit drei Punkten in dieser Zeile. Wenn Sie darauf klicken öffnet sich ein Dialog, in dem Sie auch länge, mehrzeilige Texte bequem eingeben können.

Im Quelltext funktioniert das ebenso. Vorher müssen Sie aber wissen, dass ein Zeilenumbruch nur ein bestimmtes Zeichen in einer Zeichenkette ist. Welches das ist, hängt vom verwendeten Betriebssystem zusammen. In Frage kommen die beiden Zeichen »Line Feed« (LF) mit dem Dezimalwert<sup>5</sup> 10 oder »Carriage Return« (CR) mit dem Wert 13.

**Windows** CR LF (beide Zeichen)

**Mac OS X** CR

---

<sup>5</sup>Für den Computer sind alles nur Zahlen. So wird auch jedem Buchstabe ein bestimmter Zahlwert zugeordnet.

**Linux LF**

In Free Pascal können Sie auch schreiben:

```
1  Label1.Caption := 'Die Eingabe: ' + LineEnding + Edit1.Text;
```

Wie Sie schon wissen, verbindet + zwei Zeichenketten miteinander. LineEnding ist also auch eine Zeichenkette und enthält immer das oder die richtigen Zeichen. Egal für welches Betriebssystem Sie Ihr Programm übersetzen.

### 3.6. Ein kurzer Dialog

Um mit einem Benutzer zu kommunizieren, reichen TLabel und TEdit aber nicht aus. Daher stellt Lazarus viele verschiedenen Steuerelemente bereit. Damit man nicht für jede Anwendung ein eigenes Fenster, das einfach nur eine kurze Statusmeldung ausgibt, erstellen muss, gibt es dafür bereits fertige Funktionen. Ein kleiner Vergleich:

```
1  procedure TForm1.BtnShowTextClick(Sender: TObject);
begin
    Label1.Caption := 'Guten Morgen!';
4   Label1.Caption := 'Ich bin dein Computer und spreche mit dir
    ...';
    Label1.Caption := 'Warum sagst du denn nichts?';
end;
```

Listing 3.6: Mehre Anweisungen in einem Block

Starten Sie das Programm und betrachten Sie das Ergebnis nachdem Sie auf die Schaltfläche geklickt haben. In Free Pascal werden alle Anweisungen in einem Funktionsrumpf nacheinander ohne Pause ausgeführt. *Zwischen* zwei Anweisungen steht immer ein Semikolon, das heißt, *nach* einer Anweisung muss kein Semikolon stehen, wenn danach keine weitere Anweisung folgt. Beispielsweise ist das Schlüsselwort **end**; keine Anweisung und daher können Sie das Semikolon unmittelbar davor auch weglassen<sup>6</sup>.

Sie können zwar explizit in den Quelltext hineinschreiben, dass eine Pause gemacht werden soll<sup>7</sup>, aber selbst wenn Sie dies täten, sähen Sie auf dem Label nur den Ursprünglichen Text »Label1«, da das Programm erst nach Ende der Funktion den neuen Text auf dem Bildschirm anzeigt.

<sup>6</sup>An ein paar Stellen, dürfen sogar keine Semikola stehen, aber darauf werden wir Sie extra hinweisew.

<sup>7</sup>Das geht durch Aufruf einer bestimmten Funktion. Welche das ist und wie sie funktioniert, möchten wir an dieser Stelle aber noch nicht verraten.

### 3.6.1. Eine Meldung ausgeben

Damit Sie mit Ihrem Programm sprechen können, brauchen Sie auf diese Weise jede Menge Labels und Schaltflächen. Anders, wenn Sie die oben erwähnten Funktionen verwenden.

```
1 procedure TForm1.BtnShowTextClick(Sender: TObject);  
  begin  
    ShowMessage('Guten Morgen!');  
4    ShowMessage('Ich bin dein Computer und spreche mit dir ...');  
    ShowMessage('Hey, deinen Namen hast du ja schon eingegeben, '  
      + Edit1.Text + '.');  
  end;
```

Listing 3.7: Statusmeldung anzeigen

Ändern Sie die Funktion TForm1.BtnShowText wie in Listing 3.7 und starten Sie Ihre Anwendung. Bevor Sie aber auf die Schaltfläche klicken, tippen Sie Ihren Namen im Textfeld ein.

Damit haben Sie schon eine andere Funktion aufgerufen und es war ganz einfach. Nicht nur das, Sie haben der Funktion, die den Namen ShowMessage trägt, eine Zeichenkette als Parameter übergeben. Ein Parameter ist also in etwa einfach ein Stück Daten, und mit diesen Daten kann die Funktion arbeiten und – wie in diesem Fall – in einem Fenster anzeigen.



Übergeben Sie einer Funktion einen Wert, dürfen Sie hinter den Wert nie ein Semikolon setzen. Nur ganz an's Ende (hinter die Funktion) können – oder müssen – Sie ein Semikolon setzen.

Jetzt könnten Sie einwenden: »Aber halt! Die Anwendung macht doch eine Pause, während die erste Meldung angezeigt wird. Schließlich werden die anderen Meldungen nicht auch sofort angezeigt.« Stimmt, die anderen Meldungen werden erst angezeigt, sobald Sie den Meldungsdialog (über die »OK«-Schaltfläche) geschlossen haben, aber die Anwendung macht keine Pause. Sie wartet darauf, dass Sie auf die Schaltfläche klicken.

### 3.6.2. Eine Eingabe abfragen

Auf ähnliche Weise können Sie dem Benutzer auch ein vorgefertigtes Dialogfenster anzeigen, in dem er aufgefordert wird, etwas einzugeben.

Wie Sie sehen können Sie das Ergebnis einer Funktion (TForm1.BtnShowText hat kein Ergebnis), auch direkt an eine andere Funktion übergeben. Weil die Funktion InputBox eine Zeichenkette zurückliefert, verbinden Sie das Ergebnis einfach mit dem übrigen Text, sodass dieser dann an die Funktion ShowMessage übergeben wird.



```

1 procedure TForm1.BtnShowTextClick(Sender: TObject);
  begin
    ShowMessage('Guten Morgen!');
4    ShowMessage('Ich bin dein Computer und spreche mit dir ...');
    ShowMessage('Hallo, ' + InputBox('Name eingeben', 'Sag mir
      doch bitte deinen Namen ...', 'Alfred') + '! Wie geht es
      dir?');
  end;

```

Listing 3.8: Eine Eingabe fordern

Die Funktion `InputBox` erwartet drei Parameter (`ShowMessage` erwartet nur einen), jeweils mit einem Komma getrennt. Als Erstes geben Sie eine Überschrift für das Fenster an, als Zweites eine Frage und als Drittes einen Vorgabewert für das Eingabefeld. Wollen Sie keinen Vorgabewert angeben, können Sie ihn nicht einfach weglassen. Stattdessen müssen Sie wie in Listing 3.9 eine leere Zeichenkette angeben.

```

1    ShowMessage('Hallo, ' + InputBox('Name eingeben', 'Sag mir
      doch bitte deinen Namen ...', '' ) + '! Wie geht es dir?');

```

Listing 3.9: Leere Zeichenkette

## 3.7. Speichern

Bevor Sie Lazarus beenden sollten Sie Ihr Projekt speichern. Ansonsten gehen alle Fortschritte verloren und Sie müssen von Vorne beginnen. Wenn Sie im Hauptmenü »Datei->Speichern unter ...« auswählen, wird Lazarus Sie nach zwei Dateinamen fragen. Der erste ist für die Projektdatei (Dateiendung `.lpi`). In ihr wird zwar kein Quelltext aber alle anderen Informationen, die nötig sind, um das Programm zu übersetzen, gespeichert. Danach müssen Sie noch einen Dateinamen für die Quelltextdatei zum Formular angeben (Dateiendung `.pas`).

Erstellen Sie für jedes Projekt ein eigenes Verzeichnis und speichern Sie dort die Projekt- und Quelltextdateien. Dadurch behalten Sie den Überblick, welche Dateien zu welchem Projekt gehören und nichts geht verloren. So können Sie auch nicht aus Versehen eine Datei eines anderen Projektes überschreiben oder löschen.

Bei einem Neustart lädt Lazarus automatisch das letzte Projekt. Wenn Sie ein anderes Projekt öffnen wollen, klicken Sie im Hauptmenü auf »Datei->Öffnen ...« und wählen

Leerzeichen in Dateinamen?

### 3. *Das erste Programm*

dann die Projektdatei aus. Daraufhin wird Lazars Sie fragen, ob Sie die Datei als XML-Datei<sup>8</sup> oder als Projekt öffnen wollen.

---

<sup>8</sup>*XML* ist ein standardisiertes Dateiformat um Daten zu speichern. Viele Anwendungen verwenden es um ihre Einstellungen zu speichern – so auch in Lazarus. Prinzipiell sind XML-Dateien nur gewöhnliche Textdateien, auch wenn sie manchmal sehr verwirrend aussehen.

## 4. Variablen und Konstanten

Im letzten Kapitel haben Sie gelernt, wie Sie vom Benutzer einen Wert entgegennehmen und wieder ausgeben. Mit diesem Wert können Sie aber nur sehr eingeschränkt arbeiten, da Sie dazu den Wert irgendwo zwischenspeichern müssen. Damit Ihr Programm darauf zugreifen und die Daten verwenden kann, gibt es nur einen geeigneten Ort dafür: den Hauptspeicher (RAM, engl. Random Access Memory) des Computers.<sup>1</sup>

Der Hauptspeicher wird von dem Betriebssystem verwaltet und an alle laufenden Programme (auf Anforderung) verteilt, wobei es zwei grundlegend verschiedene Wege gibt, auf denen ein Programm ein Stück Hauptspeicher anfordern kann.

**Stack** Der Stack wird jedem Programm automatisch bereitgestellt. Er wird vom Programm selbst verwaltet (Sie müssen sich nicht darum kümmern). Hier wird beim Aufruf einer Funktion (zur Erinnerung: Funktionen wurden auf Seite 25 besprochen) alle Parameter gespeichert und eine Funktionsadresse (auch Funktionen werden im Arbeitsspeicher abgelegt), wo das Programm nach am Ende der aufgerufenen Funktion weiterarbeiten soll.

**Heap** Auf dem Heap kann ein Programm beliebige Daten abgelegt werden. Wie viele Daten hier abgelegt werden müssen, steht in der Regel erst zur Laufzeit fest (zum Beispiel weil nicht fest steht, welche Datei der Benutzer bearbeiten möchte) und daher muss das Programm immer wieder beim Betriebssystem nach einem Stück Speicher fragen.

Sobald der Speicher nicht mehr benötigt wird (weil zum Beispiel das Programm beendet wird), muss das Programm dem Betriebssystem auch wieder mitteilen, dass der betreffende Speicher nicht mehr benötigt wird und an andere Programme vergeben werden darf.

**Adresse** Im vorhergehenden Abschnitt wurde wiederholt von *Adressen*. gesprochen. Adressen im Arbeitsspeicher funktionieren genau so wie Postadressen im realen Leben: sie sagen, *wo* etwas (ein Stück Speicher) gefunden werden kann, aber nicht *was*.

Im Gegensatz zu anderen Programmiersprachen, die näher an den Grundlagen der Computertechnik (in Bezug auf Hauptspeicherverwaltung) stehen, wie beispielsweise C, müssen Sie als Pascal-Programmierer nur sehr selten selbst Speicher vom Betriebssystem anfordern (und wenn Sie ihn nicht mehr benötigen wieder zurückgeben!). Sobald Sie

---

<sup>1</sup>Dateien, die auf einer Festplatte abgelegt wurden, müssen ebenfalls zuerst in den Arbeitsspeicher geladen werden, bevor das Programm mit den darin enthaltenden Daten arbeiten kann.

## 4. Variablen und Konstanten

dies aber machen müssen, werden Sie noch einmal darauf aufmerksam gemacht. Für den Anfang müssen Sie sich aber noch nicht darum kümmern.

### 4.1. Lotto

Als erste kleine Übung schreiben Sie ein kleines Programm, das die Gewinnzahlen der nächsten Lottoziehung *berechnet*, wobei die Ergebnisse in *Variablen* zwischengespeichert werden.

**Variablen** Variablen sind kleine Speicherbereiche, die auf dem *Stack* (siehe Seite 33) abgelegt werden. Dazu müssen Sie dem Free Pascal Compiler nur sagen, in welcher Funktion Sie eine Variable benötigen und er erledigt den Rest für Sie.

Aber zuerst sollten Sie sich Gedanken über das Programmfenster machen.

- Wie machen Sie Ihrem Benutzer deutliche, was Ihr Programm tut?
- Wie zeigen Sie das Ergebnis an?
- Wie kann der Benutzer das Berechnen der Lottozahlen starten?

In Kapitel 3 haben Sie bereits die Steuerlemente TLabel und TButton kennengelernt. Diese beiden reichen für für dieses kleine Programm völlig aus.

#### 4.1.1. Formularentwurf

Entwerfen Sie nun ein Formular mit den drei folgenden Elementen. Das **fettgedruckte** Wort vor jedem Absatz ist der Name des Steuerelementes, welches Sie in der Eigenschaft *Name* ändern können.

**LUeberschrift** Ein TLabel, welches in einem Satz anzeigt (Eigenschaft *Caption*), was berechnet wird, zum Beispiel: »Die Lottozahlen von Mittwoch:« Wenn sie dieses Textfeld eher oben im Formular platzieren, sollte jedem Benutzer klar sein, dass die nachfolgenden Schaltflächen irgendetwas mit den Lottozahlen von Mittwoch zu tun haben. Die Frage, ob das nun der letzte oder der nächste Mittwoch (oder doch ein ganz anderer) ist, lassen wir bewusst unbeantwortet.

**LZahlen** Ein weiteres TLabel, in dem die ermittelte Zahlenreihe ausgegeben wird. Da beim Start des Programms noch nichts berechnet wurde, geben Sie als Beschriftung (Eigenschaft *Caption*) den Text »<noch nicht berechnet ...>« vor.

**BtnZahlenBerechnen** Die Schaltfläche (TButton), über die die Berechnung der Lottozahlen gestartet oder auch wiederholt werden kann. Als Beschriftung reicht ein simples »Berechnen« aus.

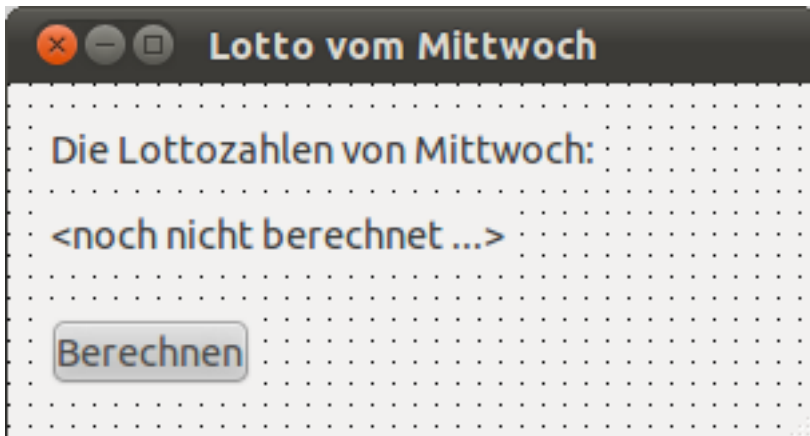


Abbildung 4.1.: So oder so ähnlich könnte Ihr Programmfenster im Entwurfsmodus aussehen.

#### 4.1.2. Berechnung programmieren

Als nächstes müssen in einer Funktion schreiben, was passieren soll, wenn der Benutzer auf die Schaltfläche mit der Beschriftung »Berechnen« klickt. Wie der Name es schon sagt, sollen die Lottozahlen berechnet werden, aber das ist nicht alles. Irgendwie muss der Benutzer auch noch das Ergebnis der Berechnung mitgeteilt bekommen, da die ganze Berechnung ansonsten recht sinnlos wäre.

Die Funktion muss also folgende Aufgaben erledigen:

1. Die Lottozahlen bestimmen.
2. Diese Zahlen irgendwie auf dem Textfeld LZahlen ausgeben.

Erzeugen Sie also zunächst mit Lazarus ein Grundgerüst für diese Funktion, indem Sie mit der Maus auf die Schaltfläche doppelklicken (Ergebnis in Listing 4.1).

```

34 procedure TForm1.BtnZahlenBerechnenClick(Sender: TObject);
36 begin
end;

```

Listing 4.1: Die leere Methode zum Berechnen der Lottozahlen

Wie oben bereits gesagt, müssen Sie nun dem Compiler mitteilen, dass Sie in dieser Funktion sechs Variablen benötigen um jeweils eine Zahl zu speichern. Pascal fordert im Gegensatz zu anderen Sprachen wie beispielsweise C oder PHP, dass zu Beginn einer Funktion alle darin genutzen Variablen bereits bekannt sind. In C können Sie auch mitten in einer Funktion eine neue Variable benutzen, die dann selbstverständlich erst ab dieser Zeile im Quelltext verfügbar ist.

#### 4. Variablen und Konstanten

Der Vorteil liegt auf der Hand: Wenn Sie den Quelltext betrachten, wissen Sie bereits welche Variablen genutzt werden, ohne dass sie die gesamte Funktion lesen müssen. Wenn diese dazu noch aussagekräftige Namen besitzen, können Sie schon ausmalen, was denn mit diesen Variablen passiert. Nicht zuletzt vereinfacht sich dadurch auch der Compiler, aber den benutzen Sie schließlich nur.

Gut aufgepasst? Wenn ja, wissen Sie jetzt:

- Variablen haben einen *Namen*, über den sie angesprochen werden.
- Sie müssen festlegen, *was* Sie in einer Variablen speichern möchten. Man spricht von einem *Datentyp* einer Variablen.

Der Name sollte in klar machen, wofür eine Variable genutzt wird beziehungsweise, welche Daten darin enthalten sind. Für den Name gelten die selben Regeln wie für die Namen von Komponenten (siehe Kaptiel 3.4.1 auf Seite 23), er kann also alle Zeichen aus a–z, A–Z, 0–9 und den Unterstrich verwenden. Wie immer gilt: Ziffern dürfen nicht am Anfang des Namens stehen und die Groß- und Kleinschreibung des Namens spielt keine Rolle.

In Listing 4.2 sehen Sie, wie Variablen deklariert (bekannt machen) haben: Nach der Parameterliste schreiben Sie das Schlüsselwort **var**. Zwischen diesem Schlüsselwort und dem Anfang der Funktion (Schlüsselwort **begin**) können Sie nun so viele Variablen deklarieren, wie sie möchten.

Um die Lottozahlen von Mittwoch (6 aus 49 ohne Zusatzzahl) zu speichern benötigen Sie genau sechs Variablen. Diese nennen Sie zahl1 bis zahl6. Nach dem Namen folgt ein Doppelpunkt und danach der Datentyp. Eine Übersicht über die wichtigsten Datentypen in Free Pascal finden Sie im Anhang.

Wenn Sie mehrere Variablen mit dem selben Datentyp deklarieren möchten, können Sie auch die einzelnen Namen mit Kommata voneinander trennen und am Ende den Datentyp für alle gemeinsam festlegen (in Listing 4.2 in den Zeilen 37 und 38).

Um einfache (ganze) Zahlen zu speichern eignet sich der Datentyp **Integer** hervorragend. Damit können Sie ganze Zahlen zwischen  $-2\,147\,483\,648$  und  $2\,147\,483\,647$ , was für die meisten Anwendungsbereiche völlig ausreicht.

```
34 procedure TForm1.BtnZahlenBerechnenClick(Sender: TObject);  
   var  
36     zahl1: Integer;  
     zahl2, zahl3: Integer;  
     zahl4, zahl5, zahl6: Integer;  
40 begin  
   end;
```

Listing 4.2: Die Variablen für Lottozahlen

Referenz!

## Lottozahlen ermitteln

Als nächstes müssen Sie die Lottozahlen ermitteln. Da die Lostrommel im Fernsehen auch nur eine Zufallsmaschine ist, wäre es eine gute Möglichkeit das Ermitteln der Zahlen in diesem Programm ebenfalls dem Zufall zu überlassen.

In Listing 3.4 auf Seite 26 haben Sie bereits einer Eigenschaft einen neuen Wert zugewiesen. Mit Variablen funktioniert das genau so, in Listing 4.3 sehen Sie wie der Variablen `zahl1` die Zahl 1 zugewiesen wird.

Bevor Sie einer Variablen einen Wert zugewiesen haben, können Sie nicht davon ausgehen, dass sie einen bestimmten Wert enthält. Sie enthält immer den Wert, der zuletzt an diese Stelle im Speicher geschrieben wurde – und das können Sie nur kontrollieren, wenn Sie es selbst tun.

Sollten Sie dennoch eine Variable auslesen, bevor Sie ihr einen Wert zugewiesen haben, wird der Free Pascal Compiler Sie darauf hinweisen. Im Nachrichtenfenster erscheint eine Meldung wie: »Warning: Variable ›zahl1‹ does not seem to be initialized« und ein Klick auf die Meldung bringt Sie an die entsprechende Stelle im Quelltext.

```
1  zahl1 := 1;
```

Listing 4.3: Einer Variablen einen Wert zuweisen

Der Unterschied liegt darin, dass Sie bei der Beschriftung von Steuerelementen den neuen Wert in Anführungszeichen gestzt haben, hier aber die Zahl ohne auskommt. Der Free Pascal Compiler unterscheidet anhand von Anführungszeichen, ob es sich um eine Zahlenkette oder um etwas anderes – in diesem Fall eine Zahl – handelt.

Warum das wichtig ist: Wie bereits gesagt, darf die Variable `zahl1` nur Zahlen enthalten, da Sie den Datentyp **Integer** gewählt haben. Eine Zeichenkette ist kein Zahl und passt also auch nicht in den bereitgestellten Speicherplatz.

Andersehrum können Sie natürlich auch nicht der Überschrift des Anwendungsformulars keine Zahl zuweisen. `Self.Caption := '42';` ist eine Zeichkette und funktioniert einwandfrei. `Self.Caption := 42;` wird der Compiler mit der Fehlermeldung »Error: Incompatible type for arg no. 1: Got ›ShortInt‹<sup>2</sup>, expected ›TTranslateString‹<sup>3</sup>« das Übersetzen abbrechen.

Daher müssen Sie erst die Daten zwischen den Datentypen konvertieren, wozu Free Pascal schon fertige Funktionen bereitstellt (sofern das Konvertieren sinnvoll ist). Wie das für Zahlen geht, erfahren Sie weiter unten in diesem Kapitel.

<sup>2</sup>ShortInt ist genau so ein Datentyp für Zahlen wie **Integer**, besitzt aber weniger Speicher und kann daher nur Zahlen zwischen –128 und 127 aufnehmen.

<sup>3</sup>TTranslateString ist ein spezieller Datentyp für Zeichenketten, den Sie erst benötigen, wenn Sie selbst Steuerelemente entwickeln. Vorher werden Sie mit dem Typ **String** wahrscheinlich alle Zeichenketten Ihrer Programme speichern können.

#### 4. Variablen und Konstanten

Dabei können Sie selbstverständlich auch Werte berechnen ( $1 + 1$ ) oder dies durch zusätzliche Funktionen erledigen. Zu diesem Zwecke können Funktionen einen Rückgabewert besitzen, der in weitere Berechnungen eingeht, zwischengespeichert wird oder auch verworfen wird (wenn die Funktion noch andere Dinge erledigt, die Sie wünschen, aber mit dem eigentlichen Ergebnis nichts anzufangen wissen).

Eine Zufallszahl können Sie mit der Funktion **Random** berechnen. Das Ergebnis einer Funktion können Sie genau so wie eine Zahl einer Variablen zuzweisen. Ein Funktionsaufruf ist ganz einfach: Sie müssen einfach nur den Funktionsnamen angeben und schon wird an dieser Stelle die Funktion mit diesem Namen aufgerufen. Wenn die Funktion weitere Daten als Parameter, mit denen sie rechnen oder arbeiten soll, benötigt, müssen diese in runden Klammern direkt (Leerzeichen sind erlaubt) hinter dem Funktionsnamen stehen. Werden keine Daten erwartet, können Sie die Klammern entweder ganz weglassen, oder Sie lassen sie einfach leer.

38

```
zahl1 := Random;
```

Wenn Sie dies aber wie in Listing 4.1.2 tun, erhalten Sie wieder eine Fehlermeldung wie schon auf der vorherigen Seite dargestellt. Es gibt nämlich drei verschiedene Varianten der Funktion **Random**, die alle den gleichen Namen besitzen. Sie unterscheiden sich nur anhand des Wertes, der in Klammern hinter dem Funktionsnamen als Parameter. Wenn Sie also eine Funktion mit mehreren Varianten (oder anders herum gesagt: mehrere Funktionen mit verschiedenen Parametern) besitzen, entscheidet der Free Pascal Compiler beim Übersetzen eines jeden Aufrufs, welche dieser Varianten am Besten zu den angegebenen Parametern passt.

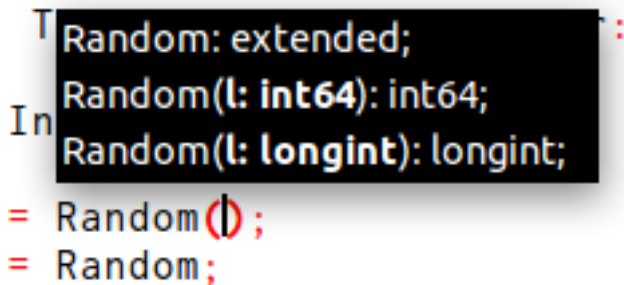
1. ein Parameter: Wie in Listing 4.1.2 liefert **Random** eine Zahl größer gleich 0 und kleiner 1. Da ein solcher Wert nur selten eine ganze Zahl ist, kann diese auch nicht im Datentyp **Integer** gespeichert werden.
2. Eine kleine Zahl zwischen  $-2\,147\,483\,648$  und  $2\,147\,483\,647$ : Beim Aufruf von beispielsweise **Random**(5) wird eine Zufallszahl zwischen 0 und 4 ( $5-1$ ) berechnet.
3. Eine große Zahl zwischen  $9\,223\,372\,036\,854\,775\,807$  und  $-9\,223\,372\,036\,854\,775\,808$ : Funktioniert genau so, wie die Variante mit kleinen Zahlen, aber hier können Sie wesentlich größere und auch negative Zufallszahlen, die dann zwischen 0 und dem angegebenen Wert liegen, berechnen.

Im Lotto 6aus49 gibt es alle ganzen Zahlen zwischen 1 und 49. Da **Random** aber immer eine Zahl zwischen 0 und der angegebenen Zahl minus Eins berechnet, müssen dies in Ihrer Lotto-Ziehung beachten: Wählen Sie also als Obergrenze die Zahl 49 und zählen Sie anschließend eins hinzu (Listing 4.4 auf der nächsten Seite).



```
38 zahl1 := Random(49) + 1;
```

Listing 4.4: Eine Zufallszahl zwischen 1 und 49 ermitteln



```
Random: extended;
Random(l: int64): int64;
Random(l: longint): longint;
= Random();
= Random;
```

Abbildung 4.2.: Lazarus zeigt auf Tastendruck alle Parameter einer Funktion an.

Sollten Sie einmal vergessen haben, welche Parameter eine Funktion erwartet, können Sie in Lazarus einfach den Funktionsnamen schreiben, diesen um die beiden Klammern für die Parameter ergänzen und den Tastatursor cursor zwischen den Klammern positionieren. Dann drücken Sie gleichzeitig die Tasten **Strg**, **Umschalt** und die **Leertaste** und Lazarus blendet alle Varianten dieser Funktion und deren Parameterlisten ein (Abbildung 4.1.2).

Nun müssen Sie dies für alle sechs Zahlen wiederholen (Listing ?? auf Seite ??).

```
34 procedure TForm1.BtnZahlenBerechnenClick(Sender: TObject);
   var
36   zahl1, zahl2, zahl3, zahl4, zahl5, zahl6: Integer;
   begin
       zahl1 := Random(49) + 1;
       zahl2 := Random(49) + 1;
40   zahl3 := Random(49) + 1;
       zahl4 := Random(49) + 1;
       zahl5 := Random(49) + 1;
       zahl6 := Random(49) + 1;
44 end;
```

### 4.1.3. Ausgabe der Lottozahlen

#### 4. Variablen und Konstanten

Nun ermittelt Ihr Programm sechs Gewinnzahlen sobald Sie auf die Schaltfläche klicken. Davon sehen Sie aber gar nichts. Das Berechnen von Daten, ohne dass sie in irgend einer Weise ausgegeben werden (beispielsweise auf dem Bildschirm, in einer Datei, in einer Datenbank, per Netzwerk versendet), ist nur in den seltensten Fällen wirklich sinnvoll, und daher werden Sie das jetzt ändern!

Wie in 4.1.2 auf Seite 37 schon angerissen, können Sie die Zahlen nicht einfach auf die Eigenschaft *Caption* des Labels zuweisen, vorher müssen Sie die Zahlen in eine Zeichenkette umwandeln.

Dazu verwenden Sie die Funktion `IntToStr()` und übergeben ihr als Parameter den Zahlenwert, den Sie als Zeichenkette haben wollen.

```
44 LZahlen.Caption := StrToInt(zahl1);
```

Listing 4.5: Eine Zahl in eine Zeichenkette umwandeln

Wie es bei Computern nunmal so ist, bestehen sowohl Zahlen als auch Zeichenketten nur aus einer Folge von Nullen und Einsen (Bits). Erst die Interpretation, also auf welche Art und Weise mit diesen Bitfolgen umgegangen wird, bestimmt welche Bedeutung sie haben.

Sie können also die Zahl 15 binär als Zahl speichern (Datentyp **Integer**), aber auch verschiedene Darstellungsformen als Zeichenkette wählen. Verschiedene Darstellungsformen heißt, dass diese Zeichenkette als ein bestimmtes Zahlensystem – häufig sind binär, oktal, dezimal, hexadizimal – dargestellt werden und dementsprechend nur die Ziffern aus diesem Zahlensystem enthalten.



Der Zahlwert 15 kann also als Zeichenkette so aussehen:

**binär** 1111

**oktal** 17

**dezimal** 15

**hexadezimal** F

In 4.1.1 auf Seite 34 haben Sie nur ein einziges TLabel-Element zur Ausgabe der ermittelten Gewinnzahlen platziert. Daher müssen Sie jetzt alle Gewinnzahlen nacheinander, mit Leerzeichen getrennt, auf diesem ausgeben.

Nun können Sie Ihr Programm übersetzen, ausführen und einige Lottoziehungen durchführen.

```

34 procedure TForm1.BtnZahlenBerechnenClick(Sender: TObject);
var
36   zahl1, zahl2, zahl3, zahl4, zahl5, zahl6: Integer;
begin
   zahl1 := Random(49) + 1;
   zahl2 := Random(49) + 1;
40  zahl3 := Random(49) + 1;
   zahl4 := Random(49) + 1;
   zahl5 := Random(49) + 1;
   zahl6 := Random(49) + 1;
44  LZahlen.Caption :=
      IntToStr(zahl1) + ' ' +
      IntToStr(zahl2) + ' ' +
      IntToStr(zahl3) + ' ' +
48  IntToStr(zahl4) + ' ' +
      IntToStr(zahl5) + ' ' +
      IntToStr(zahl6);
end;

```

Listing 4.6: Alle Lottozahlen ausgeben

#### 4.1.4. Zufallszahlen in Pascal

Starten Sie das Programm mehrmals, wird die erste Ziehung immer die Zahlenfolge 27, 30, 36, 42, 30 und 43 sein. Genau so wird die zweite oder dritte Ziehung nach jedem Programmstart immer die selben Zahlen beinhalten.

Das liegt daran, dass die Funktion **Random()** keine echten Zufallszahlen liefert. Sie berechnet nur sogenannte Pseudo-Zufallszahlen. Wirklich echt zufällige Zahlen können nicht mit einem Computer berechnet werden und müssen beobachtet werden. Als Quelle dient häufig das statische Rauschen von Hardwarekomponenten, die Zeit zwischen Tastaturanschlägen oder auch der Netzwerkverkehr.

Diese Beobachtung ist aber immer recht aufwändig und kann daher nur geringe Datenmengen liefern. Daher werden überall da, wo der echte Zufall nicht benötigt wird, Funktionen genutzt, die eine Zahlenreihe berechnen, die einem zufälligen Ergebnis möglichst nahe kommt (vor allem in der Verteilung der Zahlen). Damit das funktioniert, benötigt eine solche Funktion in der Regel einen Startwert (im Englischen *seed* genannt), auf dessen Basis die nächste Zahl berechnet wird; er ändert sich also nach jeder Berechnung.

Auch die Funktion **Random()** verwendet einen solchen Startwert, der bei jedem Programmstart der selbe ist. Mit der Funktion **Randomize()** wird er mit der aktuellen Systemzeit (Zeit und Datum, die das Betriebssystem bereitstellt) gefüllt. Rufen Sie diese Funktion bei Programmstart auf, werden Sie also immer eine andere Zahlenfolge erhalten.

```

1 procedure TForm1.BtnZahlenBerechnenClick(Sender: TObject);

```

#### 4. Variablen und Konstanten

```
var
  zahl1, zahl2, zahl3, zahl4, zahl5, zahl6: Integer;
4 begin
  Randomize;
  zahl1 := Random(49) + 1;
  zahl2 := Random(49) + 1;
8  zahl3 := Random(49) + 1;
  zahl4 := Random(49) + 1;
  zahl5 := Random(49) + 1;
  zahl6 := Random(49) + 1;
12 LZahlen.Caption :=
  IntToStr(zahl1) + ' ' +
  IntToStr(zahl2) + ' ' +
  IntToStr(zahl3) + ' ' +
16  IntToStr(zahl4) + ' ' +
  IntToStr(zahl5) + ' ' +
  IntToStr(zahl6);
end;
```

Listing 4.7: Die fertige Funktion zum Berechnen der Lotto-Zahlen

## 5. Kontrollstrukturen

Wie der Name schon sagt, kontrollieren diese Sprachelemente den Programmablauf.



# **Teil III.**

## **Komponenten**





## **6. Einführung in die Datenbanken**

## 6.1. Datenbanktheorie

Das Folgende Kapitel wendet sich speziell an den Personenkreis der in die Theorie von Datenbanken noch nicht so eingedrungen ist, beziehungsweise dient als Nachschlagwerk für die verschiedenen Begriffe. Man kann also bei entsprechenden Vorwissen die folgenden erklärenden Kapitel überspringen und bei Bedarf nachschlagen.

### 6.1.1. Begriffe

Um Missverständnisse auszuschließen und zu einer gemeinsamen Sprachregelung zu kommen, sollte man die verwendeten Begriffe möglichst genau definieren:

- Eine Datenmenge ist eine Menge von einzelnen Datensätzen. Jeder Datensatz besteht aus mindesten einem Feld. Die Herkunft dieser Datenmenge ist nicht festgelegt.
- Eine Tabelle ist als Datenbankbestandteil eine spezielle Ausführung einer Datenmenge. Die Tabelle speichert als physisches vorhandenes Element die Daten.
- Eine Abfrage ist eine virtuelle Datenmenge, die den Inhalt von tatsächlich vorhandenen Tabellen in einer frei wählbaren Anordnung abbildet, manchmal auch Projektion genannt.
- Eine Datenbank ist eine Zusammenstellung von logisch zusammengehörigen Tabellen.
- Ein Datenbankserver verwaltet verschiedene Datenbanken und stellt auch das Management, die Sicherung und andere Verwaltungsdienste zur Verfügung.

Wichtige Informationen bei der Entwicklung einer Datenbankanwendung sind das Verhalten der Datenbank bzw. des Datenbankservers selbst.

### 6.1.2. Was ist eine Datenbank

Eine Datenbank ist eine geordnete Sammlung von Daten, die auf irgendeine Weise miteinander in Beziehung stehen.

Die ersten Generationen von Datenbanken waren sogenannte File-Systeme. Zuerst auf Band, dann auch auf Festplatten. In diesen Datei-Systemen wurden die Daten nacheinander abgespeichert. Um auf einen bestimmten Datensatz zu zugreifen, muss man an den Anfang der Datei (oder Bandes) gehen und anschließend alle Datensätze durchlaufen, bis man den richtigen gefunden hat. Somit ist auch klar, dass ein einfaches sortieren oder einfügen von Daten in eine sortierte Datenmenge enormen Aufwand und Kapazität erfordert hat. Um diesen Beschränkungen zu entfliehen, (und durch neue, schnellere und größere Festplatten ermöglicht) haben sich aus diesen System die heutigen relationalen oder objektorientierten Datenbanken entwickelt. Die derzeit am meisten verwendeten Datenbanken sind die relationalen und im weiteren werde ich nur noch diese behandeln.

### 6.1.3. Desktop und Client-Server Datenbankarten

Gerade der Begriff Netzwerkdatenbank ist sehr verwirrend. Wird damit eine Access-Datenbank, die mehrere Benutzer über das Netzwerk verwenden so bezeichnet ? Oder eine Serverbasierende Datenbank? Die folgenden Erklärungen sollten die Begriffe klarer werden lassen.

**Stand-Alone Datenbank** Eine Stand-Alone Datenbank ist eine Desktop-Datenbank, es befinden sich daher die Daten auf dem Arbeitsplatzrechner. Auf die Daten kann immer nur ein Anwender, mit immer nur einem Programm zugreifen. Es ist zwar prinzipiell möglich über ein Netzwerk auf die Datenbankdatei zuzugreifen, aber es kann der eine nur in der Datenbank arbeiten, wenn der andere sein Programm geschlossen hat. Probleme die durch den gleichzeitigen Zugriff entstehen können daher gar nicht auftreten. Bei jeder etwas umfangreicheren Datenbank wird dieses Verhalten zu einem Engpass. Man stelle sich nur vor, der eine Benutzer öffnet die Datenbankdatei und vergisst auf das schließen des Programms. Kein anderer Benutzer kann die Daten in der Zwischenzeit benutzen!

**File-Share Datenbank** Moderne Netzwerke bieten die Möglichkeit, dass mehrer Anwender auf ein und dieselbe Datei zugreifen. Auf diese Weise ist es auch möglich das mehrer Programme auf ein und dieselbe Datenbankdatei zugreifen. Diese Version der Desktop-Datenbank nennt man File-Share Datenbank und damit ist bereits ein echter Mehrbenutzer Betrieb möglich. Das ganze hat jedoch (unter anderem) einen entscheidenden Nachteil: Die Datenverarbeitung erfolgt auf den Arbeitsplatzrechnern. Für Abfragen muss der jeweilige ganze Datenbestand zu den Arbeitsplatzrechnern gebracht werden, dementsprechend hoch ist die Belastung für das Netzwerk. Weiter können auch Störungen am Netzwerk leicht zu Datenverlust bzw. zu Inkonsistenz der Datenbeständen führen.

**Client-Server Datenbank** Bei Client-Server Datenbanken hat nur der Datenbankserver selbst direkten Zugriff auf die Dateien des Datenbestandes. Anfragen werden somit direkt an den Datenbankserver gestellt, von diesem bearbeitet und die Ergebnisse an den Arbeitsplatzrechner zurückgeliefert. Die Verwaltung beim gleichzeitigen Zugriff durch mehrer Arbeitsplatzrechner obliegt dem Datenbankserver. Falls eine Verbindung durch eine Störung abbricht, so wird dieses erkannt und die noch nicht kompletten Anfragen verworfen und somit die Konsistenz der Daten erhalten. Gerade zur File-Share Datenbank können Netzbelastungen drastisch gesenkt werden. Man stelle sich nur vor, das man den größten Wert aus einer unsortierten Tabelle mit 1 Millionen Datensätze habe will. Bei der File-Share Datenbank müssen alle Datensätze geholt und bearbeitet werden, bei der Client Server Datenbank nur das Ergebnis. Weitere Bereiche sind die Möglichkeit Backups zu erstellen während die Datenbank weiter in Verwendung ist.

#### 6.1.4. Relationale Datenbanken

Der Begriff relationale Datenbank geht auf einen Artikel von E. F. Codd zurück, der 1970 veröffentlicht wurde. Codd bezeichnet Datenbanken als "minimal relational", wenn sie folgende Bedingungen erfüllen.

- Die Informationen werden einheitlich in Form von Tabellen repräsentiert.
- Der Anwender sieht keine Verweisstrukturen zwischen den Tabellen.
- Es gibt mindestens die Operation der Selektion, der Projektion und des JOIN definiert.

1985 veröffentlichte Codd zwölf Regeln, die relationalen Datenbanken im strengeren Sinn definieren, Ende 1990 veröffentlichte er ein Buch über relationale Datenbanken, in dem er die einstigen zwölf Regeln des relationalen Modells auf 333 Regeln differenziert. Dadurch wird die Relationalität einer Datenbank bis ins letzte Detail festgeschrieben. Soweit die Theorie, normalerweise sprechen Hersteller von relationalen Datenbanken, wenn die Mehrheit der 12 Regeln eingehalten werden.

**Begriffe in relationalen Datenbanken** Man kann nicht über relationale Datenbanken sprechen, ohne zuvor einige Begriffe zu klären.

**Relationen** Eine Relation ist gleich einer Tabelle. Die Daten werden daher in Relationen gespeichert.

**Attribut und Tuples** Die Attribute sind die Spalten einer Relation (Tabelle), die Tuples sind die Datensätze.

**Degree und Kardinalität** Die Zahl der Attribute einer Relation nennt man Degree, das ist der Ausdehnungsgrad. Die Zahl der Tuples ist die Kardinalität. Eine Relation mit Degree Null macht keinen Sinn, eine Kardinalität von NULL Tuples hingegen ist eine leere Relation.

**Domain** Eine Domain ist ein Wertebereich. Man kann z.B. eine Domain Nachnamen vom Type „Zeichen mit Länge 20“ erstellen. Diese wird immer dann verwendet wenn man Datenspalten mit dem Type „Nachname“ erstellen muss. Warum dieser Umweg? Wenn man später Tabellen miteinander verknüpft (in Relation bringt), so müssen die Spalten (Attribute) der gleichen Domain unterliegen. Habe ich vorher eine Domain definiert, so gibt es keine Probleme. Weiter ist es kein Problem wenn man draufkommt das die „Nachnamen“ länger sind, so kann die Definition der Domain geändert werden und alle Spalten haben die richtige Länge. Würde ich beim händischen Editieren eine Spalte vergessen so würde die Datenbank nicht mehr korrekt arbeiten.

**NULL** Ein „Nichtwert“ der anfangs immer für Verwirrung sorgt ist NULL. NULL ist nicht gleich Null! Ein besserer Ausdruck wäre UNBEKANNT. NULL macht übrigens aus einer zweiwertigen Logik (Ja/Nein) eine dreiwertige (Ja/Nein/Unbekannt). Vorstellen kann man es sich am besten mit einem Beispiel. Jedem Konferenzraum kann ein Beamer mit seiner Nummer zugeteilt werden. Die Räume welche keinen Beamer besitzen (zuwenige Beamer vorhanden) bekommen als Wert NULL zugeteilt, da ja ein nicht vorhandener Beamer auch keine Nummer besitzt, folglich also unbekannt ist.

**Schlüssel** Im Zuge der Normalisierung (welche später erklärt wird) werden die Daten auf viele Tabellen verteilt. Um dieses Tabellen wieder richtig in Relation zu bringen werden verschiedene Schlüssel, auch Keys genannt, verwendet.

**Primärschlüssel (Primary Key)** Jede Relation (Tabelle) besitzt einen Primärschlüssel um einen Datensatz eindeutig zu identifizieren. Ausnahmen von dieser Regel gibt es nur bei „M:N Verknüpfungen“ für die Zwischentabellen verwendet werden (die meisten Datenbanksysteme können diese Verknüpfungen nicht direkt abbilden). Ein Primärschlüssel ist immer eindeutig und ohne Duplikate. Meistens wird dafür eine fortlaufende Nummer verwendet, ist aber nicht zwingend. Vorsicht bei bestimmten Fällen, denn selbst Sozialversicherungsnummern müssen nicht eindeutig sein !

**Sekundärschlüssel (Secondary Keys)** Werden dafür verwendet um bei bestimmten Datenbankoperationen die Effizienz zu steigern, da die Datensätze intern nicht ungeordnet sondern in sortierter Reihenfolge verwaltet werden. Beim verwenden sollte aber immer auf die zugrunde liegende Datenbank Rücksicht genommen werden, da die Vor- und Nachteile stark datenbankabhängig sind.

**Fremdschlüssel (Foreign Key)** Ist der Verweis in der Tabelle auf einen Primärschlüssel in einer anderen Tabelle. Gerade in relationalen Datenbanken gibt es sehr viele Verknüpfungen die auf Primär- und Fremdschlüsselpaaren aufbauen. Wichtig ist, das Primär- und Fremdschlüssel der gleichen Domain unterliegen.

**Referentielle Integrität** Die referentielle Integrität stellt sicher das die Daten zueinander (über Primär- und Fremdschlüssel) glaubhaft bleiben. Ein einfügen, ändern oder löschen ist zu verweigern wenn dadurch die Datenintegrität verletzt würde. Man kann zum Beispiel keine Datensätze aus der Personentabelle löschen, solange in der Tabelle der Bestellungen auf diese Personen verwiesen wird.

**Normalisierung** Unter der Normalisierung einer Datenbank, wird die Technik des logischen Datenbankdesigns bezeichnet. Es erfolgt meistens durch das Schrittweise optimieren der Datenbank zur Designzeit. Theoretiker haben insgesamt 5 Stufen der Normalisierung herausgearbeitet, wobei in der Praxis meist nur die ersten 3 Stufen verwendet werden.

## 6. Einführung in die Datenbanken

Ausgangslage	Alle Informationen in einer Tabelle
1. Normalform	Jede Spalte einer Tabelle enthält unteilbare Informationen. Die Datensätze verwenden keine sich wiederholenden Informationen, die nicht auch zu einer separaten Gruppe zusammengefasst werden könnten
2 Normalform	Es wird die 1. Normalform eingehalten und alle Informationen in nicht Schlüsselfeldern hängen nur vom kompletten Primärschlüssel ab
3 Normalform	Es wird die 2 Normalform eingehalten und alle Informationen in den nicht Schlüsselfeldern sind untereinander nicht abhängig.
4 Normalform	Es wird die 3. Normalform eingehalten und in gleichen Tabellen sind keine unabhängigen Objekte vorhanden, zwischen denen eine m:n Beziehung bestehen könnte
5 Normalform	Die normalisierte Datenbank kann nicht weiter in Tabellen mit weniger Attributen konvertiert werden. Es muss sich jederzeit der unnormalisierte Ursprungszustand ohne Informationsverlust herstellen lassen

Tabelle 6.1.: Datenbank Normalisierungsstufen Übersicht

Warum wird meistens nur bis zur 3. Normalform normalisiert? Bei der Anwendungsentwicklung besteht meistens nicht das Ziel, möglichst den exakten theoretischen Grundlagen zu entsprechen, sondern eine möglichst effiziente Komplettlösung für das Problem zu erhalten. Dazu gehört natürlich auch, die Rücksicht auf das verwendete Datenbanksystem und die damit zu erzielenden Performance. Es leuchtet jedem ein, dass die Aufteilung der Informationen auf viele Tabellen bei einer Auswertung zu schlechteren Ergebnissen führt. Somit kann es sein, dass in Teilbereichen sogar eine Denormalisierung aus Performancegründen nötig ist, oder der gewonnene Platz bzw. das Geschäftsmodell keine Normalisierung sinnvoll erscheinen lassen.

### 6.1.5. Grunddaten

Mit Grunddaten werden die Informationen bezeichnet, die Voraussetzung für die tägliche Arbeit sind und während des täglichen Betriebs anfallen. Sie stellen die Basis des Datenbanksystems dar. Die Grunddaten werden in zwei Kategorien, Stammdaten und Bewegungsdaten, eingeteilt.

**Stammdaten** Stammdaten sind diejenigen Grunddaten, die über einen längeren Zeitraum benötigt werden. Sie bilden den Grundbestand an Daten für das Datenbanksystem und werden auch als Bestandsdaten bezeichnet. Stammdaten weisen eine geringe Änderungshäufigkeit auf. Üblicherweise ist bei der Neuanlage von Stammdaten noch nicht bekannt, wann Änderungen zu erwarten sind und wie lange die Daten insgesamt gültig sind.

Da auf Stammdaten häufig zugegriffen wird, ist ihre aktuelle Pflege notwendig, so dass Änderungen unmittelbar im Datenbestand nachgezogen werden sollten. Somit ist auch die normale Zugriffsart festgelegt, auf Stammdaten wird meistens in Verbindung mit Abfragen lesend zugegriffen, nur die Wartung erfolgt schreibend.

**Bewegungsdaten** Im Gegensatz zu Stammdaten haben Bewegungsdaten eine begrenzte Lebensdauer, die durch einen vorgegebenen Lebenszyklus beschrieben ist. Bewegungsdaten haben einen konkreten Zeitbezug, der für die Bedeutung und Interpretation der Information wichtig ist. Des weiteren beziehen sich Bewegungsdaten auf Stammdaten, weshalb sie auch als abgeleitete Daten bezeichnet werden. Da Bewegungsdaten gegenüber Stammdaten in der Menge mächtiger sind, ist gerade hier auf ein gutes Design zu achten. Betrachten wir es am Beispiel eines Zählers einer Station: Die Zähler werden im 10 Minutenrhythmus in die Bewegungsdaten eingefügt, das sind 144 Datensätze pro Tag, währenddessen der Stammdatenteil gleich bleibt, nämlich 1 Datensatz.

Version: \$LastChangedRevision: \$<sup>1</sup>

---

<sup>1</sup> Autor: Andreas Frieß  
Lizenz: GFDL and CC BY-NC-SA 3.0

## 6.2. DDL Datendefinitionssprache

Die Datendefinitionssprache (Data Definition Language = DDL<sup>2</sup>) umfasst die Sprachteile von Datenbanksprache, mit deren Hilfe man Datenstrukturen wie Tabellen und andere ähnliche Elemente erzeugt. Es sind das die Befehle die mit *CREATE* beginnen, zum Beispiel *CREATE TABLE* und *CREATE INDEX*.

## 6.3. DML Datenveränderungssprache

Die Datenveränderungssprache (Data Manipulation Language = DML<sup>3</sup>) umfasst die Sprachteile von Datenbanksprache, die sich mit dem Lesen, Ändern und Löschen von Daten beschäftigen. Es sind das die Befehle *SELECT*, *INSERT*, *UPDATE* und *DELETE*.

Im folgend sehen wir uns die wichtigsten Befehle an, wobei ein Befehl besonders mächtig ist—*SELECT*—.

### 6.3.1. SELECT

*SELECT* ist einer der Vielseitigsten und am meisten eingesetzten Befehle überhaupt. Er dient zum Abfragen von Datenmengen aus der Datenbank. Er ermöglicht die Abfrage von Daten aus den Tabellen (die Projektion)

**Einfachste Darstellung** Bei der einfachsten Abfrage haben wir es mit wenigen Schlüsselwörter zu tun.

```
SELECT [DISTINCT] 'Auswahlliste'
FROM 'Quelle'
WHERE 'Where-Klausel'
[GROUP BY ('Group-by-Attribut')+
[HAVING 'Having-Klausel']]
[ORDER BY ('Sortierungsattribut' [ASC|DESC])+];
```

Mit *SELECT* wird die Abfrage eingeleitet, anschliessend kommt die Auswahlliste der gewünschten Tabellenspalten. Dann das *FROM*, das angibt welche Tabellen die Daten bereitstellen und das *WHERE*, das angibt nach welchen Kriterien die Daten vorselektiert werden.

**SELECT 'Auswahlliste'** Hier gibt man die einzelnen Tabellenspalten an, die sich später in der rückgelieferten Datenmenge finden. Sind die Name nicht eindeutig genug, besonders wenn mehrere Tabellen betroffen sind, so muß man den Tabellennamen und eventuell auch die Datenbank angeben, sprich die Eindeutigkeit qualifizieren. Somit kann ein *SELECT* so aussehen: *SELECT\_MyDB.STPerson.Vorname*, meistens wird die verkürzte Schreibweise verwendet wie *SELECT\_Vorname*.

<sup>2</sup>siehe auch [http://de.wikipedia.org/wiki/Data\\_Definition\\_Language](http://de.wikipedia.org/wiki/Data_Definition_Language)

<sup>3</sup>siehe auch [http://de.wikipedia.org/wiki/Data\\_Manipulation\\_Language](http://de.wikipedia.org/wiki/Data_Manipulation_Language)



In komplexeren Abfragen kann auch statt dem Tabellennamen alleine, eine Funktion stehen. Damit kann man in Datenmengen mathematische und statistische Funktionen verwenden. Auch Verzweigungen und Berechnungen sind möglich. Zu diesen Punkten kommen dann an späterer Stelle die entsprechenden Erklärungen.

**DISTINCT** erzwingt die Vermeidung von doppelten Datensätzen. Es überspringt in der Ausgabe die mehrfach vorkommenden Datensätze. Die Folge ist ein höherer Aufwand am Server, da das ursprüngliche Ergebnis (ohne **DISTINCT**) meist noch entsprechend nachbearbeitet werden muß. Bevor man **DISTINCT** zur Korrektur unerklärlicher doppelter Datensätze verwendet (Verschleiern von Problemen), sollte man sich zuerst seine **JOINS** und **WHERE** Klauseln ganz genau ansehen, denn dort liegt meistens das Problem.

Eine gerne verwendete Abkürzung stellt das beliebte **SELECT\_\*** dar. Hier darf dann das Programm zur Laufzeit erraten, welche Spalten es gibt und von welchen Typ sie sind. Es einfach zu verwenden, birgt aber in fertigen Programmen einiges an versteckten Fehlerquellen. Wird die dem **SELECT** zugrunde liegende Tabelle geändert oder auch nur Spalten getauscht, so kann es sein, daß Fehler nicht beim ausführen des Statements auffallen (Spalte xx nicht gefunden) und erst andere Programmteile dann unklare Fehlermeldung produzieren. Ich empfehle deshalb, die Spaltennamen wirklich anzugeben. Es gibt Programme die gerade diese Eigenschaft ausnutzen, dort wird es aber bewusst gemacht und entsprechend abgesichert.

**FROM 'Quelle'** **FROM** gibt die Quelle der Daten an. Es ist einfach der Tabellename, eine Verbindung von Tabellen (**JOIN**) oder auch eine untergelagerte **SELECT** Anweisung.

**WHERE 'Where-Klausel'** Die **WHERE** - Klausel schränkt die Ergebnisdatenmenge ein. Bei einfachen Beispielen wird sie gerne, oft zu Unrecht, weggelassen. Wir müssen aber immer Datenmengen betrachten, das Ergebnis kann eine Zeile oder eine million Zeilen sein. Wenn wir in einer Adressdatenbank eine Personen suchen, so wird es trotzdem sinnvoll sein, von Haus aus die Suche einzuschränken. Denn alles was nicht an Daten sinnlos übertragen werden muß, spart Bandbreite, Speicher und erhöht die Geschwindigkeit.

**GROUP BY ('Group-by-Attribut')** Die Daten werden nach dem Group-by-Attributen zusammengefasst (gruppiert). Dazu müssen auch in der Auswahlliste, für alle nicht durch **GROUP BY** erfassten Spalten, entsprechende Operationen verwendet werden (**SUM**, **AVG**, **MIN**, ...).

**HAVING 'Having-Klausel'** Ist als ein **WHERE** zu betrachten, das allerdings erst **nach** dem Gruppieren wirksam ist und deshalb nur auf die Gruppierungsfunktionen wirksam ist.

**ORDER BY ("Sortierungsattribut" [ASC|DESC])** Die ausgegeben Daten werden entsprechend sortiert. Das Sortierungsattribut sind die entsprechenden Spaltennamen in der Reihenfolge wie sortiert werden soll.

### 6.3.2. Beispiele zu SELECT

```
SELECT * FROM st_person;
```

Ohne Einschränkung oder Sortierung.

```
"STPerson", "cVName", "cFName", "cMName", "cRName"  
378, "Hope", "Giordano", "HoGi", "Hope0Giordano"  
379, "Rose", "Bruno", "RoBr", "Rose4Bruno"  
380, "Lauren", "Morgan", "LaMo", "Lauren4Morgan"  
381, "Megan", "Coleman", "MeCo", "Megan9Coleman"
```

```
SELECT * FROM st_person where STPerson = 875;
```

Die Person deren ID 875 ist.

```
"STPerson", "cVName", "cFName", "cMName", "cRName"  
875, "Hannah", "Collins", "HaCo", "Hannah3Collins"
```

```
SELECT * FROM st_person where cVName like 'H%';
```

Alle Personen deren Vorname mit „H“ beginnt. Das Prozentzeichen „%“ ist eine Wildcard. So wie bei manchen Betriebssystemen der Stern „\*“.

```
"STPerson", "cVName", "cFName", "cMName", "cRName"  
875, "Hannah", "Collins", "HaCo", "Hannah3Collins"  
406, "Hannah", "Cook", "HaCo", "Hannah9Cook"  
845, "Hannah", "Doyle", "HaDo", "Hannah9Doyle"  
1363, "Hannah", "Foster", "HaFo", "Hannah6Foster"
```

```
SELECT * FROM st_person order by cFName;
```

Alle Personen, aber nach Familienname aufsteigend sortiert.

```
"STPerson", "cVName", "cFName", "cMName", "cRName"  
1258, "Caitlin", "Adams", "CaAd", "Caitlin4Adams"  
687, "Taylor", "Alexander", "TaAl", "Taylor5Alexander"  
644, "Renee", "Alexander", "ReAl", "Renee8Alexander"  
885, "Taylor", "Alexander", "TaAl", "Taylor3Alexander"  
1131, "Sarah", "Alexander", "SaAl", "Sarah5Alexander"  
603, "Megan", "Allen", "MeAl", "Megan0Allen"  
1172, "Isabella", "Allen", "IsAl", "Isabella0Allen"  
472, "Isabelle", "Allen", "IsAl", "Isabelle6Allen"
```

```
SELECT * FROM st_person order by cFName desc;
```

Alle Personen, aber nach Familienname absteigend sortiert.

```
"STPerson", "cVName", "cFName", "cMName", "cRName"
842, "Isabella", "Young", "IsYo", "Isabella6Young"
1232, "Taylor", "Young", "TaYo", "Taylor3Young"
427, "Ashley", "Young", "AsYo", "Ashley3Young"
420, "Kyla", "Young", "KyYo", "Kyla2Young"
668, "Alexandra", "Wright", "AlWr", "Alexandra6Wright"
1070, "Madison", "Wright", "MaWr", "Madison5Wright"
```

*SELECT \* FROM st\_person where cVName like 'H%' order by cFName;*

Alle Personen deren Vorname mit „H“ beginnt und nach Familienname aufsteigend sortiert.

```
"STPerson", "cVName", "cFName", "cMName", "cRName"
932, "Hope", "Bell", "HoBe", "Hope3Bell"
1194, "Harmony", "Campbell", "HaCa", "Harmony7Campbell"
515, "Harmony", "Clark", "HaCl", "Harmony4Clark"
1279, "Holly", "Collins", "HoCo", "Holly7Collins"
875, "Hannah", "Collins", "HaCo", "Hannah3Collins"
406, "Hannah", "Cook", "HaCo", "Hannah9Cook"
1296, "Harmony", "Diaz", "HaDi", "Harmony1Diaz"
```

*SELECT cVName, cFName FROM st\_person where cVName like 'H%' order by cFName;*

Anzeige nur der Spalten „cVName“ und „cFName“ und alle Personen deren Vorname mit „H“ beginnt und nach Familienname aufsteigend sortiert.

```
"cVName", "cFName"
"Hope", "Bell"
"Harmony", "Campbell"
"Harmony", "Clark"
"Holly", "Collins"
"Hannah", "Collins"
"Hannah", "Cook"
"Harmony", "Diaz"
```

*SELECT cVName, cFName FROM st\_person where (cVName like 'H%') or (cVName like 'A%') order by cFName;*

Anzeige nur der Spalten „cVName“ und „cFName“ und alle Personen deren Vorname mit „H“ oder „A“ beginnt und nach Familienname aufsteigend sortiert.

```
"Alyssa", "Butler"
"Abby", "Butler"
"Ashley", "Butler"
"Ashley", "Byrne"
"Harmony", "Campbell"
"Harmony", "Clark"
```

## 6. Einführung in die Datenbanken

```
"Anna", "Clark"  
"Abby", "Coleman"  
"Hannah", "Collins"  
"Amelia", "Collins"  
"Anna", "Collins"
```

*SELECT count(cVName), cVName FROM st\_person group by cVName order by count(cVName) desc;* Anzahl der Vorkommen der Vornamen absteigend sortiert.

```
"count(cVName)", "cVName"  
19, "Amelia"  
19, "Angel"  
17, "Laura"  
16, "Elizabeth"  
16, "Paris"  
16, "Ruby"  
15, "Caitlin"  
14, "Sophie"  
14, "Abby"  
14, "Mikayla"
```

*SELECT count(cVName) as Anzahl, cVName as Vorname FROM st\_person group by cVName order by count(cVName) desc;* Anzahl der Vorkommen der Vornamen absteigend sortiert. Dazu noch die Spaltennamen geändert.

```
"Anzahl", "Vorname"  
19, "Amelia"  
19, "Angel"  
17, "Laura"  
16, "Elizabeth"  
16, "Paris"  
16, "Ruby"  
15, "Caitlin"  
14, "Sophie"  
14, "Abby"  
14, "Mikayla"
```

*SELECT count(cVName) as Anzahl, cVName as Vorname FROM st\_person group by cVName desc having Anzahl = 16;* Anzahl der Vorkommen der Vornamen, Dazu noch die Spaltennamen geändert und die Anzahl muß sechzehn sein

```
"Anzahl", "Vorname"  
16, "Elizabeth"  
16, "Paris"  
16, "Ruby"
```

### 6.3.3. INSERT

INSERT wird für das Einfügen von Datensätzen in eine Tabelle verwendet.

**Einfache Darstellung** Beim einfachen Einfügen haben wir es mit wenigen Schlüsselwörter zu tun.

```
INSERT 'Ziel' ('Spaltenliste')
VALUES ('Werteliste')
```

Bei dem Ziel handelt es sich um die Tabelle in die die Daten eingefügt werden sollen. Die Spaltenliste kann auch weggelassen werden, wenn die Werteliste in der exakt gleichen Reihenfolge ist, wie die Definition in der Tabelle.

Wenn die Spaltenliste vorhanden ist, so müssen die Werte in der Werteliste in der gleichen Reihenfolge stehen. Es muß dann aber nicht die Reihenfolge und Anzahl der Spalten mit der Tabellen Definition übereinstimmen. Das wird normalerweise verwendet, da an Spalten mit automatischen Zählern (meist Primärschlüssel) keine Werte übergeben werden dürfen!

### 6.3.4. Beispiele zu INSERT

*INSERT st\_person (cVName,cFName,cMName, cRName) VALUES ("Hope","Giordano","HoGi", "Hope0Giordano");* Bei *INSERT* und anderen Befehlen zur Erstellung von Daten, gibt es kein Ergebnismenge.

```
SELECT * FROM st_person;
```

```
"STPerson","cVName","cFName","cMName","cRName"
1,"Hope","Giordano","HoGi","Hope0Giordano"
2,"Harmony","Campbell","HaCa","Harmony7Campbell"
3,"Harmony","Clark","HaCl","Harmony4Clark"
4,"Holly","Collins","HoCo","Holly7Collins"
```

Die Spalte STPerson, ist zwar beim INSERT nicht angegeben, wird vom System automatisch vergeben.

Ein weiteres Beispiel befindet unter Projekt MySQLTestData

### 6.3.5. UPDATE

Mit *UPDATE* können die Daten in den Tabellen geändert werden.

**Einfache Darstellung** Beim einfachen Ändern haben wir es mit wenigen Schlüsselwörter zu tun.

```
UPDATE 'Ziel' SET 'Spalte1' = 'Wert1' , 'Spalte2' = 'Wert2'
WHERE 'Where-Klausel'
```

## 6. Einführung in die Datenbanken

Bei dem Ziel handelt es sich um die Tabelle in der die Daten geändert werden sollen. Nach dem Schlüsselwort *SET* erfolgt die Auflistung der Spalten mit den neuen Wert.

Wichtig ist hier die *WHERE*-Klausel! Fehlt sie so werden **alle Datensätze** der Tabelle geändert! Soll nur ein einziger Datensatz von der Änderung geändert werden, so muß die Einschränkung richtig definiert sein. Normalerweise wird hier der Primärschlüssel (auch zusammengesetzt) verwendet, denn so ist die Eindeutigkeit sichergestellt. Bezüglich der Möglichkeiten der *WHERE*-Klausel bitte beim Befehl *SELECT* nachzulesen.

### 6.3.6. Beispiele zu UPDATE

*UPDATE st\_person set cRName = "HoGi" WHERE STPerson = 1;* Bei *UPDATE* und anderen Befehlen zur Erstellung von Daten, gibt es kein Ergebnismenge.

```
SELECT * FROM st_person;
```

```
"STPerson","cVName","cFName","cMName","cRName"
1,"Hope","Giordano","HoGi","HoGi"
2,"Harmony","Campbell","HaCa","Harmony7Campbell"
3,"Harmony","Clark","HaCl","Harmony4Clark"
4,"Holly","Collins","HoCo","Holly7Collins"
```

*UPDATE st\_person set cVName = "Hopper", cRName = "HoGi1" WHERE STPerson = 1;* Hier werden zwei Spalten gleichzeitig geändert

```
SELECT * FROM st_person;
```

```
"STPerson","cVName","cFName","cMName","cRName"
1,"Hopper","Giordano","HoGi","HoGi1"
2,"Harmony","Campbell","HaCa","Harmony7Campbell"
3,"Harmony","Clark","HaCl","Harmony4Clark"
4,"Holly","Collins","HoCo","Holly7Collins"
```

*UPDATE st\_person set cVName = "HaHa" WHERE cVName like „Ha“;* Hier werden mehrere Datensätze gleichzeitig geändert

```
SELECT * FROM st_person;
```

```
"STPerson","cVName","cFName","cMName","cRName"
1,"Hopper","Giordano","HoGi","HoGi1"
2,"HaHa","Campbell","HaCa","Harmony7Campbell"
3,"HaHa","Clark","HaCl","Harmony4Clark"
4,"Holly","Collins","HoCo","Holly7Collins"
```

### 6.3.7. DELETE

Mittels dem Befehl *DELETE* werden Datensätze in der Datenbank gelöscht.

**Einfache Darstellung** Beim einfachsten DELETE haben wir es mit wenigen Schlüsselwörter zu tun.

```
DELETE 'Ziel'
WHERE 'Where-Klausel'
```

Aber Vorsicht, ohne entsprechende *WHERE*-Klausel werden alle betroffenen Datensätze gelöscht. Daher gilt, fehlt die *WHERE*-Klausel, so werden **alle Datensätze** unwiderruflich gelöscht!

### 6.3.8. Beispiele zu DELETE

*DELETE FROM st\_person WHERE STPerson = 1;* Bei *DELETE* und anderen Befehlen zur Erstellung von Daten, gibt es kein Ergebnismenge. Hier wird der Datensatz mit dem Wert 1 in der Spalte STPerson gelöscht.

```
SELECT * FROM st_person;
```

```
"STPerson","cvName","cfName","cmName","crName"
1,"Hope","Giordano","HoGi","HoGi"
2,"Harmony","Campbell","HaCa","Harmony7Campbell"
3,"Harmony","Clark","HaCl","Harmony4Clark"
4,"Holly","Collins","HoCo","Holly7Collins"
```

```
DELETE FROM st_person WHERE STPerson = 1;
SELECT * FROM st_person;
```

```
"STPerson","cvName","cfName","cmName","crName"
2,"Harmony","Campbell","HaCa","Harmony7Campbell"
3,"Harmony","Clark","HaCl","Harmony4Clark"
4,"Holly","Collins","HoCo","Holly7Collins"
```

*DELETE st\_person WHERE STPerson = 1;* Bei *DELETE* und anderen Befehlen zur Erstellung von Daten, gibt es kein Ergebnismenge. Hier werden **alle** Datensätze in der Tabelle st\_person gelöscht!

```
SELECT * FROM st_person;
```

```
"STPerson","cvName","cfName","cmName","crName"
1,"Hope","Giordano","HoGi","HoGi"
2,"Harmony","Campbell","HaCa","Harmony7Campbell"
3,"Harmony","Clark","HaCl","Harmony4Clark"
4,"Holly","Collins","HoCo","Holly7Collins"
```

```
DELETE FROM st_person;
SELECT * FROM st_person;
```

"STPerson", "cVName", "cFName", "cMName", "cRName"

## 6.4. DCL Datenkontrollsprache

Die Datenkontrollsprache (Data Control Language = DCL<sup>4</sup>) umfasst die Sprachteile von Datenbanksprache, mit deren Hilfe man die Rechte vergibt, Wartungen durchführt. Es sind das Befehle wie *GRANT* und *REVOKE*.

Version: \$LastChangedRevision: \$ <sup>5</sup>

---

<sup>4</sup>siehe auch [http://de.wikipedia.org/wiki/Data\\_Control\\_Language](http://de.wikipedia.org/wiki/Data_Control_Language)

<sup>5</sup> Autor: Andreas Frieß

Lizenz: GFDL and CC BY-NC-SA 3.0



## 6.5. SQLdb

### 6.5.1. Beschreibung

#### Einleitung

SQLdb wird speziell für Serverbasierende Datenbanken verwendet und besteht im Wesentlichen aus Komponenten für die Verbindung (TxxxConnection), für das Verwalten von Transaktionen (TSQLTransaction) und dem Verwalten von Datenmengen (TSQLQuery). Für Clientdatenbanken (Dbase, FoxPro,...) sind die Komponenten unter 'Client Access'<sup>6</sup> vorgesehen. Besonders die TSQLQuery ist eine mächtige Komponente, die einige automatismen eingebaut hat, die zwar das Leben erleichtern sollen, aber oft das Gegenteil bewirken können. In diesem Kapitel wollen wir uns die Komponenten einmal genauer ansehen. Das Bild aus der Wiki aus dem englischen Lazarusforum[?]<sup>7</sup> erklärt schön, wie die Komponenten zusammenhängen und in welchen Units sie sich befinden.

#### Debugging von SQLdb

SQLdb ist ein Teil der FCL und nicht direkt von Lazarus. Die FCL wird standardmässig nicht mit den für den Debugger notwendigen Informationen kompiliert, da der Kode so kompakter und kleiner ist. Wenn man für die Fehlersuche es anders benötigt, so muß man die fcl-db neu kompilieren. Dazu muß man die kompletten FCL Sourcen haben, dann kann man in das Verzeichnis 'fpc/packages/fcl-db' gehen und mit 'make clean all OPT='-gl' das Paket neu kompilieren, anschliessen die neuen PPU's über die alten kopieren. Die Fehlersuche wird aber nur Personen empfohlen, die entsprechendes Wissen über die SQLdb Komponenten haben.

#### Active, Open oder ExecSQL

... das ist hier die Frage ? Um diese Frage zu beantworten, muß man sich vor Augen halten, was man von der Komponente will. Mittels dem Befehl Open fordert man eine Datenmenge, die Kardinalität<sup>8</sup> einer Datenmenge<sup>9</sup> kann auch Null sein, an. Mit ExecSQL wird nur eine Aktion angefordert ohne das eine Datenmenge zurück erwartet wird. Somit ist klar, das bei allen Daten liefernden Statements das Open zu verwenden ist. Welche Statements in SQL liefern überhaupt Datenmengen zurück ? Eigentlich gilt das nur für das 'SELECT' Statement, alle anderen ('INSERT', 'UPDATE', 'DELETE', ...) führen etwas aus, liefern aber keine Daten zurück. Ob man jetzt 'Open' verwendet oder 'Active:=true;' macht ist letztlich egal, 'Open' führt genau dieses Statement aus.

Eine Besonderheit ist die Behandlung von 'Stored Procedure' und 'Functions' auf SQL-Servern. Diese können eine Kombination im Verhalten darstellen. Dort ist dann ein ExecSQL angebracht und es können Datenmengen zurückgeliefert werden.

Zusammenfassung:

---

<sup>6</sup>Siehe Kapitel ?? auf Seite ??

<sup>7</sup>[http://wiki.lazarus.freepascal.org/SQLdb\\_Programming\\_Reference](http://wiki.lazarus.freepascal.org/SQLdb_Programming_Reference)

<sup>8</sup>siehe Kapitel 6.1.4 auf Seite 50

<sup>9</sup>siehe Kapitel 6.1.1 auf Seite 48

## 6. Einführung in die Datenbanken

- Open, Active: Bei der Verwendung von 'SELECT'
- ExecSQL: Für alle anderen Statements

### Wie kommen die geänderten Daten in die Datenbank

Eine Änderung der Datenmenge alleine ist nicht ausreichend, um diese Änderung auch in der Datenbank sichtbar zu machen. Prinzipiell muß man jetzt zwei Wege unterscheiden. Einerseits kann man Änderungen im Zuge einer Transaktion in der Datenbank festschreiben durch das Abschliessen der Transaktion oder auch durch das dezierte schreiben durch ApplyUpdates. Ich bin der Meinung, das man sich für einen der Wege entscheiden sollte, wenn man eine Datenmenge öffnet beziehungsweise anfordert. Arbeitet man mit Transaktionen, dann sollte man ohne zwingenden Grund nicht mit ApplyUpdates das Transaktionsmanagement stören. Andererseits wenn man ohne explizite Transaktionen arbeitet, also Datenmengen öffnet ohne vorher Transaktionen geöffnet zu haben, dann darf man nicht vergessen die Änderungen entweder mittels ApplyUpdates zu übernehmen oder mittels CancelUpdates zu verwerfen. Vergisst man auf ApplyUpdates so ist dieses schlimmer als auf CancelUpdates zu vergessen. Denn ohne dem ApplyUpdates sind die Daten bei den meisten Datenbanken ganz einfach nicht in die Datenbank eingearbeitet und somit verloren.

### Filtern, aber wo ?

Die Standardantwort ist im Stile von Radio Eriwan: 'Dort wo es sinnvoll ist'. Dazu muß man sich vor Augen halten, wo man eine Datenmenge überhaupt filtern kann. Dazu muß man unterscheiden in Desktop Datenbanken und Server Datenbanken. Bei Desktop Datenbanken kann die Frage schon obsolet sein, weil die Datenmenge sowieso nur lokal gefiltert werden kann. Bei Datenbankenservern schaut die Sachlage ganz anders aus. Denn die können Datenmengen sehr wohl, effizient vorverarbeiten und nur die wenigen Ergebnisse zurück transportieren. Damit wird am lokalen Rechner Netzwerkleistung, Speicher und Ressourcen geschont. Somit kann man hier dem filtern am Server den Vorzug geben. Werden aber Daten erst am lokalen Rechner verknüpft, so kann man oft nur lokal filtern. Somit ist klar, das es stark auf das Design der Applikation an kommt, was sinnvoll ist.

### Anzahl der Datensätze abfragen

Hier kann man generell zwei verschiedene Fälle unterscheiden. Einmal den Fall, das man wissen will, ob überhaupt Datensätze vorhanden sind und dem Fall, das man die Anzahl wissen will.

Generell ist es nur dann sinnvoll die Anzahl der Datensätze zu bestimmen, wenn eine Abfrage aktiv ist.

Ob Datensätze überhaupt vorhanden sind, kann man über die Abfrage von EOF<sup>10</sup> und BOF<sup>11</sup> machen. Sind beide vorhanden ('true') so muß die Datenmenge leer sein. Genau diese macht die Methode 'IsEmpty'. Somit kann man diese genau für diesen Fall verwenden.

Die Anzahl selbst der Datensätze, kann man theoretisch mittels der Eigenschaft 'RecordCount' abfragen. Allerdings muß dazu auch der Datenbanktreiber<sup>12</sup> das unterstützen. Bis jetzt ist die Unterstützung auch nicht wirklich vorhanden. Weiters handelt es sich hier eher um eine Eigenschaft von Desktopdatenbanken, denn dort kann die Anzahl der Datensätze nicht anders festgestellt werden.

Die andere Variante die sich daher anbietet ist die SQL Abfrage selbst. So kann man mittels dem SQL-Statement `select count(row1) as Anzahl from table1 where ...` die Anzahl ermitteln.

### Navigieren durch eine Datenmenge

Für das Navigieren durch die Datenmenge stehen ein paar Befehle zur Verfügung. Mit 'first' kommt man zum ersten, mit 'last' zum letzten, mit 'next' springt man auf den nächsten und mit 'prior' zum vorhergehenden Datensatz. Größere Bewegungen kann man mit 'MoveBy' machen. Das funktioniert vorwärts mit positiven Zahlen, rückwärts mit negativen Zahlen. Allerdings muß man bedenken, das ein 'MoveBy' nicht zwingend die volle Distanz verfahren kann, wenn die Grenzen der Datenmenge erreicht werden. Wenn also ein EOF oder BOF nach dem 'MoveBy' ansteht, so wird nicht die volle Distanz erreicht worden sein, man weiß aber nicht um wie viel verfahren wurde.

### Was ist BOF und EOF

'BOF' bedeutet das man in der Datenmenge am Anfang, bei 'EOF' am Ende der Datenmenge steht. Wenn zum gleichen Zeitpunkt beide vorhanden sind, so ist das ein Zeichen, das die Datenmenge null ist.

### Zugriff auf Felder

Auf die Felder<sup>13</sup> kann über die Eigenschaft 'Fields' zugegriffen werden. Zusätzlich kann über die Methode 'FieldByName' mittels des Feldnamens oder 'FieldByNumber' einfach auf die einzelnen Felder zugegriffen werden. Die Werte werden über die 'Values' Eigenschaft als variant zugewiesen oder über die entsprechenden 'AsInteger', 'AsString' und 'As.....'.

---

<sup>10</sup>End of File - Anfang der Daten

<sup>11</sup>Beginn of File - Ende der Daten

<sup>12</sup>Ist genaugenommen die Verbindungskomponente

<sup>13</sup>auch Attribute genannt

### Zugriff auf Parameter

Auf die Felder der Parameter kann über die Eigenschaft 'Params' zugegriffen werden. Zusätzlich kann über die Methode 'ParamsByName' mittels des Feldnamens einfach auf die einzelnen Felder zugegriffen werden. Die Werte werden über die 'Values' Eigenschaft als variant zugewiesen oder über die entsprechenden 'AsInteger', 'AsString' und 'As....'.

Im SQL-Statement werden die Parameter durch einen Doppelpunkt am Anfang des Namens kenntlich gemacht. Zum Beispiel: `insert tablex (row1, row2) values (:param1, :param2)`. Hier ist ':param1' einer der Parameter. Die Zuweisung im Programm erfolgt über `TQ1.Params.ParamByName('param1').value := 'test';`.

### Schlüsselfelder

Als Schlüsselfelder werden die Felder<sup>14</sup> bezeichnet in dem der primäre Schlüssel der Tabelle gespeichert ist. Dieser sollte bei jeder änderbaren Datenmenge definiert sein, damit die Komponente richtig die Datensätze ändern oder löschen kann. Schlüssel erzwingen eine Eindeutigkeit.

#### 6.5.2. TxxxConnection

Genau genommen handelt es sich hier nicht nur um eine einfache Deklaration der Verbindung sondern um den lokalen Verwaltungsteil der Datenbank. Hier werden die gemeinsamen Methoden und Eigenschaften behandelt, im Folgenden die verschiedenen Connection mit den abweichenden Details behandelt.

#### UserName

`property UserName : string read FUserName write FUserName;`

Enthält den Benutzernamen für die Verbindung beziehungsweise für den Zugriff auf die Datenbank.

Eigenschaft von `TxxxConnection`>`TSQLConnection`

Zugriff: Lesend und schreibend

Jede Datenbank hat ihre spezielle Verbindung (Connection).

#### 6.5.3. TMySQL50Connection

Jede Datenbank hat ihre spezielle Verbindung (Connection). Dies hier ist die Verbindung zu MySQL 5.0

---

<sup>14</sup>Schlüssel können über mehrere Felder gehen um eindeutig zu sein, oder weil sie zusammengesetzt sind

#### 6.5.4. TMySQL41Connection

Jede Datenbank hat ihre spezielle Verbindung (Connection). Dies hier ist die Verbindung zu MySQL 4.1

#### 6.5.5. TMySQL40Connection

Jede Datenbank hat ihre spezielle Verbindung (Connection). Dies hier ist die Verbindung zu MySQL 4.0

#### 6.5.6. TOracleConnection

Jede Datenbank hat ihre spezielle Verbindung (Connection). Dies hier ist die Verbindung zu Oracle.

#### 6.5.7. TPQConnection

Jede Datenbank hat ihre spezielle Verbindung (Connection). Dies hier ist die Verbindung zu PostGreSQL Datenbanken.

#### 6.5.8. TODBCConnection

Jede Datenbank hat ihre spezielle Verbindung (Connection). Dies hier ist die Verbindung zu den ODBC Treibern.

#### 6.5.9. TSQLTransaction

Als Transaktion wird eine feste Folge von Operationen, die eine Einheit bilden, bezeichnet. Transaktionen müssen die ACID-Eigenschaft garantieren. A) Atomität, das heißt untrennbar. Es wird entweder alles oder nichts durchgeführt. C) Konsistenz, Nach der Transaktion müssen die Daten konsistent sein. Daher auch an allen geänderten Stellen den gleichen Inhalt haben. I) Isolation, mehrere gleichzeitig laufende Transaktionen dürfen sich nicht gegenseitig beeinflussen. D) Dauerhaft, die Auswirkungen der Transaktion müssen im Datenbestand dauerhaft sein. Auch bei widrigen Umständen dürfen die Transaktionen nicht verloren gehen oder vergessen werden, zB. bei Rücksicherungen nach Absturz.

Der Ablauf einer Transaktion ist relativ einfach. Die Transaktion wird eröffnet, dann die Handlungen an der Datenbank gesetzt und die Transaktion entweder mit 'Rollback' wenn sie zurückgenommen werden soll oder mit 'Commit' wenn die Änderungen dauerhaft übernommen werden sollten, abgeschlossen.

Man muß sich nur vor Augen halten, das das Datenbanksystem um die ACID Eigenschaften garantieren zu können, Aktionen wie Sperren, Duplizieren oder auch Warten durchführen muß. Deshalb soll eine Transaktion nur solange aufrecht erhalten werden wie es unbedingt nötig ist. Das heisst, auch, während einer Benutzereingabe oder sonstiger Wartezeit sollte keine Transaktion stattfinden. Besonders bei Mehrbenutzersystemen kann das bis zum Stillstand der Datenbank führen, wenn wegen vergessener Eingabe bei

Arbeitschluß eine Transaktion aktiv bleibt und deshalb eine Sperre auf einer Datenbank liegt.

### 6.5.10. TSQLQuery

#### Allgemeines

Der Name beschreibt nur unzureichend die Komponente. Es ist nicht nur ein Behälter für Abfragen (Query) sondern die Kapselung für die Datenmenge. Das heisst, über die Komponente läuft eigentlich alles bezüglich Daten und Datenmenge. Sie wird sowohl für DDL<sup>15</sup>, DML<sup>16</sup> und DCL<sup>17</sup> verwendet

#### SQL, UpdateSQL, InsertSQL, DeleteSQL

**property SQL : TStringlist;** In SQL befindet sich das SQL Statement was ausgeführt werden soll. Ist es ein einfacheres Select-Statement so kann TSQLQuery auch automatisch die Statements für Änderungen (UpdateSQL), Einfügen (InsertSQL) und Löschen (DeleteSQL) ausfüllen. Ist das SQL Statement komplexer so geht diese automatik manchmal ins Leere, kann die Statements nicht richtig auswerten und verursacht unerklärliche Probleme.

**property UpdateSQL : TStringlist;** Hier stehen die SQL Statements um Änderungen in der Datenmenge durchzuführen.

**property InsertSQL : TStringlist;** Die Statements um Datensätze einzufügen.

**property DeleteSQL : TStringlist;** Statements um Datensätze zu löschen.

Die SQL Statements interagieren mit den Eigenschaften ReadOnly, UsePrimaryKeyAsKey, ParseSQL, IndexDefs und Params (Vielleicht auch einigen mehr). Wenn die Komponente nicht so reagiert wie erwartet, dann sollte man sich das SQL und die vorher genannten Eigenschaften näher ansehen.

Version: \$LastChangedRevision: \$<sup>18</sup>

---

<sup>15</sup>Datendefinitionssprache siehe Kapitel 6.2 auf Seite 62

<sup>16</sup>Datenveränderungssprache siehe Kapitel 6.3 auf Seite 62

<sup>17</sup>Datenkontrollsprache siehe Kapitel 6.4 auf Seite 62

<sup>18</sup> Autor: Andreas Frieß

Lizenz: GFDL and CC BY-NC-SA 3.0

# Listings

3.1. unit1.pas . . . . .	18
3.2. unit1.pas: Definition des Formulars . . . . .	24
3.3. unit1.pas: Funktion für das <i>OnClick</i> -Ereignis . . . . .	25
3.4. Eine Beschriftung ändern . . . . .	26
3.5. Beschriftung einer Schaltfläche ändern . . . . .	26
3.6. Mehrere Anweisungen in einem Block . . . . .	29
3.7. Statusmeldung anzeigen . . . . .	30
3.8. Eine Eingabe fordern . . . . .	31
3.9. Leere Zeichenkette . . . . .	31
4.1. Die leere Methode zum Berechnen der Lottozahlen . . . . .	35
4.2. Die Variablen für Lottozahlen . . . . .	36
4.3. Einer Variablen einen Wert zuweisen . . . . .	37
4.4. Eine Zufallszahl zwischen 1 und 49 ermitteln . . . . .	39
4.5. Eine Zahl in eine Zeichenkette umwandeln . . . . .	40
4.6. Alle Lottozahlen ausgeben . . . . .	41
4.7. Die fertige Funktion zum Berechnen der Lotto-Zahlen . . . . .	41





# Abbildungsverzeichnis

3.1. Der Objektinspektor . . . . .	19
3.2. Der Quelltexteditor in Lazarus . . . . .	20
4.1. Berechnung der Lottozahlen . . . . .	35
4.2. Code-Insight . . . . .	39



# Tabellenverzeichnis

6.1. Datenbank Normalisierungstufen Übersicht . . . . .	52
---------------------------------------------------------	----