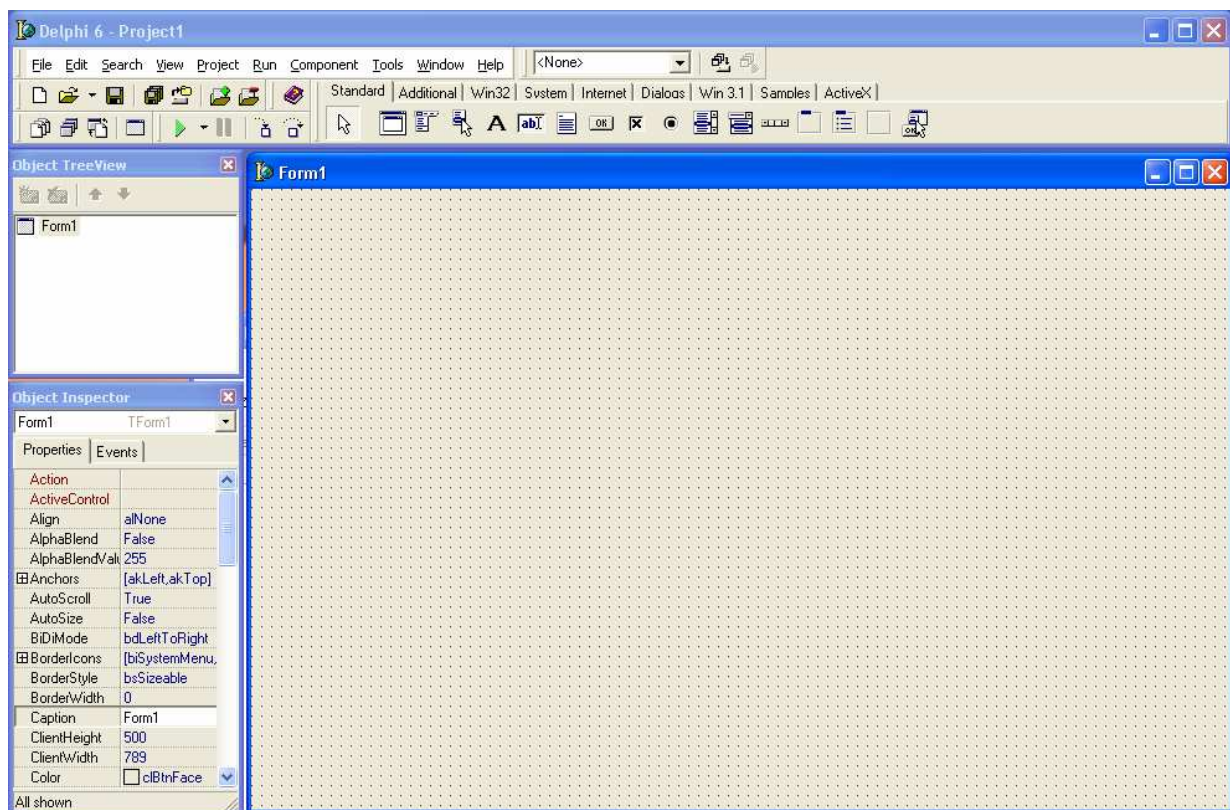


## EINFÜHRUNG IN DELPHI IN DER JAHRGANGSSTUFE 11.1

Delphi 6 ist ein mächtiges Werkzeug zur Erstellung leistungsfähiger Programme in ansprechendem Design auf der Windows-Oberfläche, eine sogenannte objektorientierte Programmiersprache (OOP), die zugleich alle herkömmlichen Mittel der Programmiersprache PASCAL zur Verfügung stellt. Im folgenden sollen dir die wesentlichen Befehle dieser Programmiersprache näher gebracht werden.

### 1. Erste Schritte mit Delphi

Mit dem Start von Delphi öffnen sich auf dem Bildschirm gleich vier Fenster: „**Form1**“, „**Object Inspektor**“, „**Object TreeView**“ und „**Unit1.pas**“ - letzteres liegt verdeckt unter dem Form1-Fenster und kann z. B. über den kleinen „Reiter“ am unteren Ende an die Oberfläche geholt werden. In der obersten Zeile siehst du „**Delphi - Projekt1**“ (weil du deinem neuen „Projekt“ noch keinen eigenen Namen gegeben hast), darunter die Windows-übliche **Menüleiste**, deren Möglichkeiten du dir der Reihe nach anschauen kannst, und darunter einen ganzen Haufen von **Buttons** auf der „**Werkzeugleiste**“. Wenn du den Mauszeiger auf diese Buttons setzt (aber noch nicht gleich „anklicken“), dann erhältst du schon ein paar Hinweise darauf, was sich dahinter verstecken könnte...



#### 1.1 Das erste lauffähige Programm ...

Klicke auf den „**OK**“ Button in der Werkzeugleiste, dann irgendwo auf das **Formular**. Es erscheint ein Button mit der Aufschrift „**Button1**“. Ein Doppelklick darauf, und es öffnet sich das **Unit1-Fenster**. Tippe einfach das Wort „**close**“ ein. Fertig ist dein erstes Programm! Bevor du es allerdings startest, solltest du es in deinem Homeverzeichnis speichern: *File/Save Project as...* Wähle das Laufwerk H und speichere sowohl die *Unit1.pas* als auch die *Project1.dpr*! **Nun kannst du dein Programm starten:** klicke dazu auf den Button mit dem grünen Dreieck (oben links, der so aussieht wie eine Play-Taste beim Videorecorder). Es öffnet sich ein Fenster mit deinem Programm, das genau so aussieht, wie du es auf dem Formular entworfen hast, lediglich die Rasterpunkte fehlen. Das einzige, was du aber im Moment tun kannst, ist ein Klick auf den „**Button1**“ – das war’s dann. Dein Programm ist beendet, und das System kehrt wieder zum Bearbeitungsmodus zurück. Schließlich ist dein Programm ja wohl auch noch ein bisschen ausbaufähig...

Klicke jetzt mal abwechselnd irgendwo auf das **Formular** und auf den **Button1** und beobachte, wie der **Objektinspektor** links daneben darauf reagiert, **welches** Objekt gerade „aktiviert“ ist! Im Objektinspektor kannst du nun irgendwelche „Eigenschaften“ für deine Objekte verändern und festlegen.

Manche Eigenschaften, wie z. B. die Größe und Lage deiner Objekte, kannst du auch direkt auf dem Formblatt mit den Windows-üblichen Mausaktionen verändern (ziehe z.B. mit gehaltener Maustaste den Button auf einen anderen Platz oder vergrößere bzw. verkleinere ihn). Möchtest du den Button sinnvoll beschriften – in unserem Fall am besten mit „Ende“ oder „beenden“ – dann musst du im Objektinspektor bei „aktiviertem“ Button in das Feld neben „Caption“ statt dem Wort „Button1“ das Wort „Ende“ bzw. „beenden“ eintragen. Soll's auch noch in einer anderen Schriftart sein, dann klicke auf die Eigenschaft „Font“ und anschließend auf die drei Punkte ... nebenan. Ändere danach z. B. – bei „aktiviertem“ Formular - die Hintergrundfarbe deines Formulars von grau auf blau. Verwende dazu die Eigenschaft „Color“. Gib schließlich deiner Form einen anderen Text in der Überschriftsleiste (wieder hinter „Caption“), speichere dein Projekt unter *File/Save all* und starte dein Programm erneut...

## 1.2 ... und nun ein wenig mehr ...

Erstelle ein neues Projekt mit dem Menüpunkt *File/New/Application*. Erinnerung dich an deine Untersuchung der **Werkzeugleiste**. Links neben dem **Button**-Objekt fandest du unter anderem einen Hinweis auf die Oberflächenobjekte „Edit“ und „Label“. **Edit**-Objekte werden wir für die Ein- und Ausgabe von Daten auf unserer Form verwenden. Sie besitzen keine Caption, sondern ihr „Inhalt“ wird in der Eigenschaft „Text“ im Objektinspektor festgehalten. **Label**-Objekte dienen dagegen im allgemeinen zur Beschriftung von Objekten, so z. B. auch zur Beschriftung von Edit-Objekten, um dem Benutzer mitzuteilen, welche Bedeutung der Text in den Input- bzw. Outputobjekten besitzt. Diese weiteren Oberflächenobjekte sollst du nun kennenlernen...

Leg dazu auf deiner Form zwei Buttons, ein Label und ein Editfeld an. Gib allen Objekten einen sinnvollen Namen. Ändere dazu im Objektinspektor die Eigenschaft „Name“ wie folgt ab: Der erste Button bekommt den Namen **B\_Ende**, Der zweite den Namen **B\_Loeschen**, das Label-Objekt bekommt den Namen **L\_Vorname** und das Editfeld den Namen **E\_Vorname**. Beschrifte über die Eigenschaft **Caption** die Buttons mit **Beenden** und **Löschen**, das Label mit **Vorname:** und lösche den Inhalt der Eigenschaft **Text** des Editfeldes. Gib deiner Form ebenfalls einen Namen, z. B. **F\_Beispiell**, und lasse dir in der zugehörigen **Caption** auch eine bezeichnende Überschrift einfallen. Beachte dabei, wo die Caption deiner Form angezeigt wird! **Doppelklicke** auf die Buttons und trage in das vorgegebene Prozedurraster zu B\_Ende das Wörtchen **close**, in das zu B\_Loeschen den Delphi-Befehl **E\_Vorname.Text := ''** ein. **Beachte dabei:**

- die Eigenschaft **Text** beinhaltet die Zeichenkette (den String), die in einem Edit-Objekt auf der Form angezeigt wird. **Text** stellt praktisch den „Wert“ eines Edit-Objekts dar! Über die Notation **E\_xxx.Text** greifst du daher auf den Inhalt des mit „E\_xxx“ bezeichneten Edit-Objektes zu.
- **:=** ist dabei das „**Zuweisungszeichen**“, das der „Variablen“ links davon den Wert des rechts davon stehenden Ausdrucks zuweist, in diesem Fall eine **leere Zeichenkette (String)**, realisiert durch zwei einzelne direkt aufeinanderfolgende Hochkommata (auf der Tastatur das Zeichen über dem Doppelkreuz #).

Das Programm tut zwar immer noch nicht viel (Nach dem Speichern kannst du es starten, mit der Maus in das Editfeld klicken, dort irgendetwas hineinschreiben, dann auf den Löschen-Button klicken, was deine Eintragung in dem Feld wieder löscht, usw. – solange, bis du den Beenden-Button drückst, der das Programm wieder abbricht), aber was willst du auch mehr erwarten, wenn du selber nur zwei Zeilen geschrieben hast?!

Erweitere dein Projekt gemäß der folgenden Aufgabe 1. Orientiere dich dabei an dem folgenden Bild:



**Aufgabe 1:** Erweitere das vorstehende Projekt um zwei weitere Editfelder (mit entsprechenden Labels dazu), und zwar eines für den Nachnamen und eines, in das per Klick auf einen weiteren Button die Verkettung von Vor- und Nachname (mit Leerzeichen dazwischen) eingetragen wird. Die **Verkettung von Strings** geschieht dabei einfach mit dem „+“ - Zeichen. Die „Variablenzuweisung“ müsste dazu wie folgt aussehen:

```
E_name.Text := E_vorname.Text + ' ' + E_nachname.Text
```

... wobei diesmal zwischen den beiden Hochkommata in der Mitte ein **Leerzeichen (Blank)** steht (andernfalls würden Vor- und Nachname einfach lückenlos hintereinander gesetzt!). Die Funktion des Lösch-Buttons sollte dazu noch auf alle drei Edit-Felder ausgedehnt werden. Damit kannst du jetzt im laufenden Programm einen Vornamen und einen Nachnamen eingeben und erhältst dann per Mausklick im dritten Feld den zusammenhängenden Namen, und du kannst auch alles wieder löschen – insgesamt allerdings auch noch nicht sonderlich berauschend, obwohl es schon ganz nett aussieht...

Übrigens, schau dir noch einmal die Liste der Eigenschaften im Objektinspektor an, insbesondere bei den Edit-Objekten die Eigenschaft „**Enabled**“ (mit den Eintragungsmöglichkeiten „**true**“ oder „**false**“, sogenannte Boolesche Werte oder auch **Booleans**) und probiere damit etwas herum. Letztlich solltest du diese Eigenschaft hier sinnvoll einsetzen können.

Was das (sinnvolle) **Speichern** deines Projektes betrifft, kannst du dir eigentlich schon einmal – wenn du willst – den **Punkt 1.3.2** im nachfolgenden Abschnitt zu Gemüte führen!!!

Diese Mühe lohnt aber vielleicht erst das folgende **Beispiel 2:**



d. h. bei jedem Klick auf den Button „Schalte Ampel“ soll die Ampel auf den Folgezustand übergehen, d. h. von **Rot** auf **Rot/Gelb**, auf **Grün**, auf **Gelb** und schließlich wieder auf **Rot** – usw..

Hier brauchst du bzgl. der Delphi-Oberflächenelemente nichts Neues dazulernen (falls du schon ein bisschen mit der Eigenschaft „**Color**“ der Edit-Felder herumexperimentiert hast) – dafür musst du jetzt über den logischen Ablauf ein wenig mehr nachdenken...

Schreibe dir einmal in einer (geschachtelten) „**Wenn – Dann – Sonst**“-Form auf, wie die Ampelfarben abhängig vom jeweils gerade vorliegenden Zustand auf den nächsten wechseln müssen ...

( ... weiterblättern gilt nicht – erst wenn du wirklich Bleistift und Papier hinreichend traktiert hast ... )

Wie sieht's aus? Wenn du wirklich intensiv genug nachgedacht hast, müsstest du die folgende Prozedur eigentlich gedanklich nachvollziehen können ( ... auch wenn deine Lösung eine andere Reihenfolge in der Abfrage verfolgte oder vielleicht logisch noch nicht so ganz richtig gewesen sein sollte ...) – und so ganz „nebenbei“ lernst du dann auch noch die **Syntax** der „**if – then – else – Anweisung**“ in Delphi kennen ...

```

procedure TF_ampel.B_SchalterClick(Sender: TObject);
begin
  if E_rot.Color = clRed
  then
    if E_gelb.Color = clWhite      { d.h. es liegt Rot/Weiß/egal vor ... }
    then E_gelb.Color := clYellow  { folgt: Rot/Gelb/egal ... }
    else begin                    { d.h. es liegt Rot/Gelb/egal vor ... }
      E_rot.Color := clWhite;
      E_gelb.Color := clWhite;    { folgt: Weiß/Weiß/Grün ... }
      E_gruen.Color := clGreen
    end
  else
    if E_gelb.Color = clWhite      { d.h. es liegt Weiß/Weiß/egal vor ... }
    then begin
      E_gelb.Color := clYellow;   { folgt: Weiß/Gelb/egal ... }
      E_gruen.Color := clWhite
    end
    else begin                    { d.h. es liegt Weiß/Gelb/egal vor ... }
      E_rot.Color := clRed;
      E_gelb.Color := clWhite;    { folgt: Rot/Weiß/Weiß ... }
      E_gruen.Color := clWhite
    end
  end;

```

**Also:** Das war der „Ampeltest“ (der erste „Härtetest“). Wenn du ihn bestanden hast, dann sage ich dir eine lange und erfolgreiche Zukunft in der Informatik voraus. Falls nein, bekommst du gleich die nächste Chance: Ich bin davon ausgegangen, dass der Ampelzustand zu Beginn (im Objektinspektor vorinitialisiert) Rot/Weiß/Weiß war. Wie reagiert die vorstehende Prozedur aber

- a) auf einen anderen zulässigen Anfangszustand,
  - b) auf einen unzulässigen Anfangszustand, also einen, der als Ampelzustand nicht infrage kommt (z.B. Rot/Weiß/Grün o.ä.) und
  - c) auf einen durchaus denkbaren Anfangszustand Weiß/Weiß/Weiß, der die anfangs ausgeschaltete Ampel repräsentieren könnte
- Überlege dir anschließend, ob man die Prozedur ggf. ändern müsste.

So, auch wenn die **Logik** in diesem Beispiel im Vordergrund stand, müssen wir uns jetzt aber auch um die **Syntax** kümmern. Die Grundstruktur der bedingten Anweisung kann man wie folgt notieren:

**if < Bedingung > then < Anweisung1 > [ else < Anweisung2 > ]**

Dabei bedeuten die **eckigen** [ ] Klammern – sie werden im Programm natürlich nicht geschrieben –, dass das darin ausgeführte „optional“ ist, d.h. es kann auch einfach entfallen [ wenn es nämlich nichts gibt, was denn „sonst“ zu tun wäre... ]. Und die **spitzen** < > Klammern werden ebenfalls nicht geschrieben, sondern sie bedeuten nur, dass hier im Programm etwas eingesetzt werden muss, was der darin aufgeführten Bedeutung entspricht, hier nach dem **if** also irgendeine Bedingung, von der klar entscheidbar sein muss, ob sie denn nun erfüllt ist (**true**) oder nicht (**false**), und nach dem **then** bzw. **else** eine Anweisung, was denn dann jeweils zu tun ist. Handelt es sich dabei um eine einzelne Anweisung, so kann man diese einfach so hinschreiben, handelt es sich dabei aber um einen ganzen „Block“ von mehreren Anweisungen, so müssen diese untereinander durch ein **Semikolon** getrennt und insgesamt durch **begin ... end** „geklammert“ werden.

Als Anweisung kennen wir bisher nur die Wertzuweisung mit dem **Zuweisungsoperator** := . Diese Anweisung kommt ja oben auch pausenlos vor, indem jeweils der Eigenschaftsvariablen „Color“ der Edit-Felder immer wieder ein anderer Farbwert zugewiesen wird. Die Bedingung besteht hier immer in einem Vergleich, indem eine Editfeld-Farbe ständig darauf abgefragt wird, welchen Farbwert sie denn

nun gerade hat. Beachte, dass hierbei der **Vergleichsoperator** = verwendet wird (also im Unterschied zum Zuweisungsoperator **ohne** den Doppelpunkt davor)! In anderen Fällen könnte hier auch ein > oder < Zeichen stehen, für größer oder gleich, bzw. kleiner oder gleich dann auch >= bzw. <= und für ungleich dann <> ! Außerdem sind auch zusammengesetzte Bedingungen möglich, indem mehrere solcher Einzelvergleiche noch durch „und“ bzw. „oder“ (**and / or**) verknüpft werden – doch dazu später mehr, wenn es gebraucht wird...

Tja, und dann noch etwas ganz Wichtiges: Oben steht ja nach dem **then** als Anweisung nicht immer nur eine einfache Zuweisung oder eine durch **begin ... end** geklammerte Folge von Zuweisungen, sondern mal folgt als Anweisung auch **wieder eine bedingte Anweisung** – das macht die ganze Sache ja auch erst so verzwickelt. Wie ist das denn mit dem folgenden Satz:

**„Wenn das Wetter heute schön ist, dann wenn du mitkommst, dann gehe ich in die Stadt, sonst mache ich Hausaufgaben.“?**

Stilistisch sicherlich unschön, aber ist er denn logisch wenigstens eindeutig? Zwei „wenn’s“, zwei „dann’s“, ein „sonst“ – okay, aber worauf bezieht sich das „sonst“? Delphi macht es hier genauso, wie wir es umgangssprachlich empfinden würden, nämlich dass die Hausaufgaben nur zum Zuge kommen, wenn Freund/Freundin keine Lust auf Stadt hat, aber bei Regen trotzdem keine Chance haben, falls Freund/Freundin mitkommen sollte. Das „sonst“ ist also im Zweifel immer nur die Alternative zum letzten „dann“. Meint man das anders (also Priorität für Hausaufgaben bei Regen), dann muss man es schon anders formulieren. Noch schlimmer wird’s, wenn du den Satz um ein weiteres „sonst“ ergänzt (eines verträgt er ja logisch noch), z. B. „..., sonst habe ich schlecht Laune.“ Weil’s regnet, oder weil du nicht mitkommst??? Delphi verwendet dazu die **begin ... end** Klammerung – schau es dir im Beispiel der obigen Prozedur noch einmal ganz genau an und mach dir klar, dass mit Hilfe der Klammerung keinerlei Missverständnisse mehr auftreten können! Von unschätzbarem Wert ist hierbei übrigens auch die Schreibweise durch geschicktes Einrücken und passendes Untereinanderschreiben zueinander gehöriger Bestandteile!!! Das ist keineswegs nur „(ästhetische?) Pingeligkeit“ sondern dient ganz erheblich der Lesbarkeit (deswegen möchte ich hier eigentlich gar nicht laut sagen, dass du theoretisch die obigen Zeilen auch beliebig hintereinanderweg oder mit beliebig anderen Zeilenumbrüchen und Einrückungen schreiben könntest – den Delphi-Compiler (Übersetzer in Maschinensprache) stört es nicht, mich dafür um so mehr, wenn ich so etwas von dir lesen, geschweige denn korrigieren muss – d.h. ich lese dann erst gar nicht dein Programm. Jetzt kannst du dir also noch überlegen, ob du Klausuren in der Informatik schreiben möchtest... Es hilft alles nichts)

### **1.3 Ein bisschen Ordnung muss sein ...**

... und zwar neben dem zuletzt genannten auch noch was die Wahl deiner Bezeichner für Objekte und Variablen innerhalb deines Projektes angeht, sowie was die Wahl der Namen für die Unterverzeichnisse und die Dateien betrifft, die Delphi beim Speichern auf Festplatte oder Diskette anlegt. Andernfalls versinkst du sehr schnell in einem hoffnungslosen Chaos!

#### **1.3.1 Vereinbarungen zu Objektnamen (Bezeichnen)**

du hast schon gesehen, dass jedes Objekt, das du auf deinem Formular anlegst, später im Programm unter einem bestimmten Namen angesprochen und manipuliert wird. Der Objektinspektor schlägt dir zwar immer schon einen Namen vor, aber es macht wenig Sinn, wenn du diese Namen akzeptierst. Du hast es dann irgendwann mit Button1 bis Button37, Edit1 bis Edit16, Label1 bis Label27 oder so ähnlich zu tun, und das liest sich dann im Programmtext deiner Unit wirklich nicht mehr gut ... (was war noch mal Edit7? – ach so, war das nicht das Feld, in dem der Straßename eingetragen werden sollte – oder nein, die Postleitzahlen? ... und Label21? ... usw.). Du kannst natürlich auch alle deine Objekte und Variablen mit x1 bis x269 durchnummeriert bezeichnen, egal ob es sich um einen Button, ein Editfeld, eine Zählschleifenvariable oder sonst was handelt - nur werde ich dann dein Programm ganz gewiss niemals lesen.


Also kurz und knapp: Da wir es vorerst hauptsächlich mit **Buttons**, **Labels** und **Editfeldern** zu tun haben werden, vereinbaren wir, dass die Namen dafür grundsätzlich mit **B\_**, **L\_** bzw. **E\_** anfangen,

gefolgt von einem Bezeichner, der an das erinnert, was gemeint ist (z. B. **B**\_Ende, **L**\_Vorname oder **E**\_Strasse, usw. – die Straße mit zwei s hat übrigens nichts mit der „neuen Rechtschreibung“ zu tun, sondern damit, dass in Delphi Bezeichnernamen kein ß und auch keine ä, ö, ü enthalten dürfen). Für die Namen anderer Objekte werden wir dann später genauso verfahren, für die **SpinEdit-Boxen** z.B. **SpE**\_zahl o.ä.!

Zu **unterscheiden** von den Objekt**Namen** sind grundsätzlich ihre „**Caption**“, d. h. ihre **Beschriftungen** bzw. ihre „**Text**“-Eintragung bei Editobjekten. Hier kannst du immer sinnvolle Wörter oder Texte eintragen, wobei du dann auch die Umlaute und das ß verwenden darfst.

### 1.3.2 Vereinbarungen zum Speichern von Projekten

Die **oberste Regel** vorweg: Bevor du überhaupt anfängst, ein neues Projekt in Delphi zu entwerfen, legst du dir mit Hilfe des **Windows-Explorers** ein eigenes **Unterverzeichnis** dafür in deinem Homeverzeichnis an, das auf den Inhalt deines neuen Projektes mit einem bezeichnenden Namen hinweist, z.B. für die Aufgabe 1 oben das Verzeichnis „Namenverkettung“. Der vollständige Verzeichnispfad für dein Projekt sieht dann z. B. so aus: „**h:\Namenverkettung**“.

Die **zweite Regel**: Wenn du dann Delphi gestartet hast, und noch bevor du überhaupt irgendwas auf deine Form gelegt hast, aktiviere als erstes den Menüpunkt *File/Save Project as...* Du bekommst dann das übliche Fenster mit einem Auszug des Explorers und aktivierst dort dein zuvor gerade neu eingerichtetes **Unterverzeichnis**. Dann akzeptierst du aber nicht den dir von Delphi vorgeschlagenen Dateinamen „**Unit1.pas**“, sondern gibst ihm einen „vernünftigen“ Namen, angeführt von den zwei Zeichen „**U**“ – für Aufgabe 1 oben z.B. „**U\_Nameneingabe.pas**“, da die Unit für die Ein-/Ausgabe zuständig ist. Nach Anklicken von „OK“ öffnet sich ein weiteres Fenster. Hier brauchst du jetzt das Verzeichnis nicht noch einmal neu einzustellen, wohl aber änderst du den vorgeschlagenen Namen „**Project1.dpr**“ sinnvoll um. Sinnvoll wäre hier z.B. „**Namenverkettung.dpr**“ – und auch dies quittierst du mit „OK“. Delphi hat dann auf deinem Unterverzeichnis ein – natürlich inhaltlich noch leeres – Projekt abgespeichert, und wenn du jetzt anfängst, an deinem Projekt zu arbeiten, kannst du jederzeit deinen erreichten Zwischenstand abspeichern, ohne dich noch um irgendwas kümmern zu müssen, und zwar entweder wieder unter dem Menüpunkt „*File/Save all*“ oder aber noch einfacher über den entsprechenden Button . Es öffnen sich dann nicht mehr die Fenster des Explorers, sondern alles geschieht automatisch im richtigen Verzeichnis und unter den vorab gewählten Namen.

#### Warum das alles so wichtig ist???

Delphi legt nämlich ( leider! ) zu deinem Projekt nicht nur eine einzige Datei auf der Platte an (die könnte man ja immer noch wieder leicht finden und identifizieren), sondern sage und schreibe gleich 6 (in Worten sechs) - und nachdem du das Projekt einmal gestartet hast, kommen sogar noch zwei weitere dazu. Schau dir im Explorer einmal an, was nach dem Speichern deines noch leeren Projektes auf deinem Unterverzeichnis so alles steht! Du findest dort – falls du so vorgegangen bist, wie ich es dir vorgeschlagen habe – folgende Einträge: „**Namenverkettung.dpr**“, „**Namenverkettung.cfg**“, „**Namenverkettung.dof**“, „**Namenverkettung.res**“, „**U\_Nameneingabe.pas**“ und „**U\_Nameneingabe.dfm**“! Wenn du dein fertiges Projekt zum erstenmal gestartet hast, dann schau noch einmal nach! Hinzugekommen sind dann noch: „**Namenverkettung.exe**“ und „**U\_Nameneingabe.dcu**“! Das muss dich aber alles jetzt nicht verwirren, und du brauchst das auch vorerst überhaupt nicht zu verstehen. Halte dich ganz einfach nur an die verabredeten Regeln.

Damit du nicht ganz im Regen stehst: Unter „**namen.dpr**“ ist das kurze „Hauptprogramm“ des **Delphi-Projektes** abgespeichert und unter „**u\_io.pas**“ der **Pascal-Quelltext** zu deiner Unit. Unter „**u\_io.dfm**“ stehen alle Informationen, die Delphi für den Aufbau des von dir gestalteten und in der zugehörigen Unit manipulierten Formulars braucht (**Delphi-Formular**). Besteht dein Projekt später mal aus mehreren verschiedenen Formularen, so kommen für jede Form wieder eine entsprechende „unit.pas“- und „unit.dfm“-Datei hinzu. Nach dem ersten Start des Projektes wird der „compilierte“ (d.h. in Maschinensprache übersetzte) Quelltext deiner Unit in „**U\_Nameneingabe.dcu**“ abgelegt (**Delphi-compiled-Unit**) und eine Datei „**Namenverkettung.exe**“ angelegt, die riesig groß ist – dafür aber dann ein auch unabhängig von deiner Delphi-Umgebung unter Windows direkt ausführbares (**executable**)



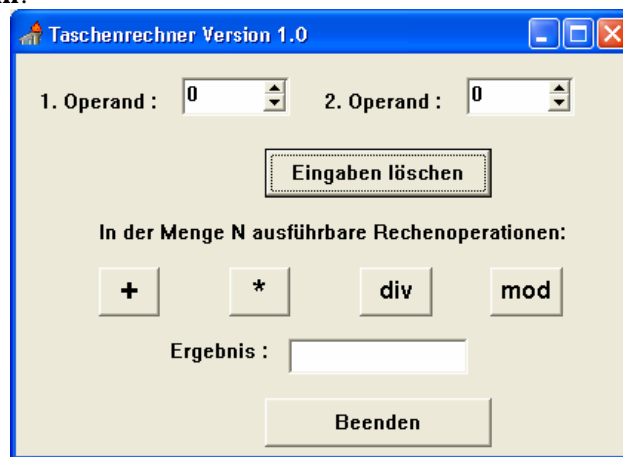
Programm ist. Solange du dein Programm immer nur unter Delphi laufen lässt (und es nicht z.B. mit einem „Icon“ direkt vom Programm-Manager starten können möchtest), kannst du diese große Datei aus Platzgründen auch immer wieder löschen. Ebenso kannst du nach eventuellen Umgestaltungen und Verbesserungen deines Projektes etwa angelegte „Backup“-Dateien alter Versionen (zu erkennen an einem „Schlängel“-Zeichen (~) in der Kennung nach dem Punkt) aus deinem Verzeichnis wieder löschen. Bleiben noch Namenverkettung.res und Namenverkettung.dof und Namenverkettung.cfg zu erklären – weiß ich auch nicht – könnte es jetzt nachschlagen, ist aber fürs erste unwichtig.

## 2. Learning by doing: Projekt „Taschenrechner“

Nach soviel „Theorie“ und „lästigen“ Vereinbarungen jetzt ein weiteres Beispielprojekt, bei dem wir unser bisheriges Wissen anwenden und natürlich auch Neues dazu lernen wollen. Wir werden ein zunächst einfaches Modell eines Taschenrechners entwerfen, das wir dann schrittweise immer komfortabler gestalten.

### 2.1 Version 1: Taschenrechner nur für Rechenoperationen in der Menge IN

Entwerfe dazu zunächst ein **Formular** mit dem folgenden Aussehen. **Beachte** dabei die in 1.3.1 und 1.3.2 vereinbarten **Regeln!**



Vielleicht hast du die „SpinEdit“-Boxen nicht gefunden? Sie befinden sich auf der Werkzeugleiste nicht unter dem Reiter „Standard“ sondern unter „Samples“ (der Button mit der 7 drauf). In diese Boxen kann man ganze Zahlen (**Integer**) eingeben und sie auch mit den beiden Dreieckchen am Rand rauf und runterzählen. Nenne sie hier „SpE\_1“ und „SpE\_2“. Den Zahlenwert, der in dieser SpinEdit-Box steht, kannst du dann im Programm syntaktisch unter den Variablennamen „SpE\_1.Value“ bzw. „SpE\_2.Value“ lesen bzw. auch neu beschreiben.

Die **normalen Edit-Felder** können dagegen nur **Zeichenketten** (Texte vom Typ **String**) aufnehmen und wiedergeben.

#### Wie kann denn dann aber in dem Ergebnisfeld eine Zahl stehen???

Dazu gibt es in Delphi Umwandlungsroutinen, die einen Integer-Wert in einen String umwandeln. Eine solche Funktion heißt z.B. „IntToStr( ... )“, wobei in den Klammern ein Integer-Zahlenwert stehen muss – oder natürlich auch eine Variable vom Typ Integer. Zurückgeliefert wird dann die Übersetzung dieser Zahl in eine Zeichenfolge aus Ziffern, die dann als Text betrachtet wird (du kannst also mit dieser Textzeichenkette nicht mehr „rechnen“, aber du kannst sie einem Edit-Feld als Text übergeben, und sie wird dort dann als Text angezeigt. Eine Anweisung, die z.B. die Summe der beiden Zahlenwerte in den SpinEdit-Feldern berechnet, sie in eine Zeichenkette umwandelt und diese dann als Ergebnis-Text dem Ergebnis-Edit-Feld (Name dazu z.B. „E\_Ergebnis“) zuweist und anzeigt, lautet dann:

```
E_Ergebnis.Text := IntToStr ( SpE_1.Value + SpE_2.Value )
```

**Und wo gehört diese Anweisung hin?** Richtig: In die **Klick-Ereignis-Prozedur**, die zu dem Button „+“ gehört! Also: **Doppel-Klick** auf diesen Button - das Fenster wechselt von deiner Form in die „Unit1.pas“ (die jetzt aber hoffentlich nicht mehr so, sondern z.B. „U\_Rechner.pas“ heißt, weil du Punkt 1.3.2 beachtet hast ...) und zeigt dir das zugehörige **Prozedur-Raster** an. Rücke einfach den Cursor um zwei Stellen ein, und schreibe dann die vorstehende Anweisung dorthinein ...

Jetzt kannst du auch die **anderen Buttons** zum Leben erwecken – die Rechenzeichen für die anderen Operationen sind in Delphi genauso, wie sie auf den Buttons stehen: **\***, **div**, **mod**, wobei **div** für die ganzzahlige Division steht und **mod** (lat. modulo) den Rest angibt, der beim ganzzahligen Dividieren übrigbleibt, also z.B.  $20 \text{ div } 12 = 1$  und  $20 \text{ mod } 12 = 8$ .

**Hinweis:** du könntest statt der SpinEdit-Felder für die beiden Eingabezahlen auch normale Edit-Felder verwenden. Nur stehen dann - auch wenn du z.B. 12 da reinschreibst - keine Zahlen darin, mit denen du rechnen könntest, sondern Zeichenketten (Texte, Strings). Nun wird's dich aber wohl kaum noch wundern, dass Delphi auch umgekehrt Texte in Zahlen umwandeln kann (vorausgesetzt natürlich, dass der eingegebene Text auch wirklich eine Ziffernfolge ist - und damit einer ganzen Zahl entspricht). Die Funktion, die dieses leistet, kannst du jetzt sicher schon erraten, sie heißt „**StrToInt(...)**“. Wenn du das vorhast, empfiehlt es sich, im Deklarationsteil der Klickprozedur zwei „lokale“ Variable vom Typ Integer zu deklarieren (die heißen „lokal“, weil sie nur im Rahmen dieser Prozedur definiert und bekannt sind) und ihnen die Zahlenwerte nach der Umwandlung zuzuweisen. Das sieht dann syntaktisch so aus (falls die Boxen „E\_zahl1“ und „E\_zahl2“ heißen):

```

procedure TF_Rechner1.B_plusClick(Sender: TObject);
  var a, b : Integer;
  begin
    a := StrToInt ( E_zahl1.Text ); b := StrToInt ( E_zahl2.Text );
    E_ergebnis.Text := IntToStr ( a + b )
  end;

```

(*Kursiv gedruckt* ist hier das, was du selber schreiben musstest, das restliche Prozedurraster war wie üblich von Delphi nach dem Doppelklick auf den Button „+“ bereits vorgegeben ...)

Wenn du dein Projekt jetzt auf diese Art umstellst, musst du allerdings aufpassen, was passiert, wenn der Benutzer dir in ein solches Editfeld z.B. „Otto“ reinschreibt statt einer „vernünftigen“ Ziffernfolge, oder aber auch weniger „böswillig“ z.B. 3,1416 - das gibt natürlich Ärger ... Wie man den vermeidet? Dazu später ...

Vorher lernen wir noch zwei andere neue Dinge. Es ist erstens unschön, dass der Benutzer in das Ergebnisfeld derzeit auch selber was reinschreiben kann (z.B. ein falsches Ergebnis - und das ist dann schon ein bisschen peinlich ...), und zum zweiten - jetzt wieder weniger „böswillig“ - steht auch was Falsches da, wenn der Benutzer oben die Eingabezahlen ändert, aber dann vergisst, einen neuen Berechnungsbutton anzuklicken ...

Das erste Problem ist leicht gelöst: Klicke in deiner Form auf das **Ergebnisfeld** und suche im Objektinspektor dazu die Eigenschaft „**Enabled**“. Du findest dort derzeit noch den voreingestellten Wert „**true**“. Setze ihn einfach auf „**false**“ und damit kann der Benutzer deines Programms zwar noch auf das Ergebnisfeld klicken, aber es nutzt ihm nichts – das Feld ist jetzt **gegen Eingaben von außen gesperrt**. Du könntest auch die Eigenschaft „**ReadOnly**“ auf true setzen.

Zur Lösung des zweiten Problems lernst du jetzt ein neues Ereignis. Bisher kennst du nur „**Click**“-**Ereignisse**, die irgendeine Aktion auslösen, wenn du auf einen Button klickst. Jetzt brauchen wir ein „**Change**“-**Ereignis**, das hier mit einem Edit- bzw. SpinEditfeld verknüpft ist, und das immer dann etwas auslöst, wenn der Inhalt des Feldes sich ändert bzw. von außen geändert wird. Wenn du in deiner Form auf das Feld für den ersten Operanden doppelt klickst, bietet dir Delphi sofort dazu in deiner Unit ein Prozedurraster für ein solches Change-Ereignis an, in das du nur noch eingeben musst, was im Falle einer Änderung des Feldinhaltes passieren soll - klar eigentlich (?!), nämlich: **E\_Ergebnis.Text := ''**,



d.h. also, der Inhalt deines Ergebnisfeldes soll gelöscht, d.h. durch einen LeerString ( $\cong$  zwei direkt aufeinanderfolgende Hochkommata) ersetzt werden.

```
procedure TF_Rechner1.SpE_1Change(Sender: TObject);
begin
    E_ergebnis.Text := '';
end;
```

Solltest du statt der SpinEdit-Box „SpE\_1“ ein normales Editfeld mit Namen „E\_zahl1“ verwendet haben, steht in deinem Prozedurkopf natürlich **TF\_Rechner1.E\_zahl1Change(...)**

Wenn du jetzt im laufenden Programm den **Zahlenwert** des ersten Feldes **änderst**, wird sofort das Ergebnis aus einer vorhergehenden Rechnung im **Ergebnisfeld gelöscht**. Führe dasselbe jetzt auch für das Feld des **zweiten Operanden** durch!

Ganz nützlich wäre auch noch ein **Button**, der die beiden Eingabefelder wieder auf 0 zurücksetzt. In das zugehörige Click-Ereignis gehören dann die beiden Befehle `SpE_1.Value := 0; SpE_2.Value := 0` (bzw. `E_zahl1.Text := '0' ; E_zahl2.Text := '0'`). Benenne den Button „**B\_Loeschen**“ und gib ihm die Beschriftung (Caption) „**Eingaben löschen**“! Probier's aus, und du siehst, dass bei einer Betätigung dieses Buttons nicht nur die beiden Eingabefelder auf 0 zurückgesetzt werden, sondern dass auch das Ergebnisfeld gelöscht wird. Woran liegt das wohl? Klar - weil mit der Änderung der Eingabefelder die zuvor eingeführten Change-Ereignisse ausgelöst wurden ... s.o.!

Experimentiere jetzt ein bisschen mit „großen“ Zahlen in deinem Programm herum - insbesondere was die Multiplikation angeht - die Ergebnisse können dann ziemlich groß werden ... Glaub' bitte nicht alles, was dir dein Programm als Ergebnis anbietet! Näheres über den Zahlbereich verschiedener „**Integertypen**“ findest du unter diesem Stichwort in der Delphi-Hilfe. Wir besprechen das erst später genauer, dann auch gleich im Zusammenhang mit den „Realtypen“, also den Kommazahlen. Falls dein Programm mit den SpinEdit-Boxen arbeitet, solltest du es aber jetzt so „absichern“, dass keine unsinnigen Ergebnisse bei der Multiplikation auftreten können. Unter den Eigenschaften dieser Felder findest du „**MaxValue**“. Ändere ihn jeweils ab auf 46340, dann bist du auf der sicheren Seite ... - warum ( $\sqrt{2147483647}$ )? Siehe dazu die Übersicht auf der nächsten Seite...

**Aufgabe 2 (ASCII-Umwandlung):** Die Verarbeitung von Zeichen (Typ **Char**) erfolgt gemäß dem **American Standard Code for Information Interchange (ASCII)**. Erstelle ein Projekt unter Verwendung von Klick- und Change-Ereignissen, das sowohl zu eingegebenem ASCII-Zeichen die zugeordnete Kodierungszahl wie auch umgekehrt zu eingegebener Kodierungszahl das zugeordnete ASCII-Zeichen ausgibt. Für diese Umwandlungen stellt dir Delphi die „Funktionen“ **Ord(<Zeichen>)** und **Chr(<Zahl>)** zur Verfügung. Da der Inhalt der Editfelder ein String (der Länge 1) ist, kannst du z.B. über **E\_Zeichen.Text[1]** auf das erste Zeichen des Strings zugreifen. Schau dir dazu außerdem z.B. in einem Druckerhandbuch einmal den ASCII-Zeichensatz an!

<b>Grundlegende Typen</b>		
Die grundlegenden Integertypen sind:		
Typ	Bereich	Format
Shortint	-128 .. 127	8-bit mit Vorzeichen
Smallint	-32768 .. 32767	16-bit mit Vorzeichen
Longint	-2147483648 .. 2147483647	32-bit mit Vorzeichen
Byte	0 .. 255	8-bit ohne Vorzeichen
Word	0 .. 65535	16-bit ohne Vorzeichen
Geltungsbereich und Format der grundlegenden Typen hängen nicht von der CPU oder dem Betriebssystem ab und bleiben in verschiedenen Implementationen von ObjectPascal konstant.		
<b>Generische Typen</b>		
Die generischen Typen sind Integer und Cardinal. Der Typ Integer stellt eine generische Ganzzahl mit Vorzeichen dar, der Typ Cardinal einen generische vorzeichenlose Ganzzahl. Die tatsächlichen Geltungsbereiche und Speicherformate der generischen Typen sind für verschiedene Implementationen von ObjectPascal verschieden; im allgemeinen sind es jedoch die Geltungsbereiche und Speicherformate, die die effizientesten Ganzzahloperationen mit der verwendeten CPU und dem verwendeten Betriebssystem ergeben.		
Typ	Bereich	Format
Integer	-32768 .. 32767	16-Bit mit Vorzeichen
Integer	-2147483648 .. 2147483647	32-Bit mit Vorzeichen
Cardinal	0 .. 65535	16-Bit ohne Vorzeichen
Cardinal	0 .. 2147483647	32-Bit ohne Vorzeichen

## 2.2 Ein zweiter Blick hinter die Kulissen ...

Deine Programmzeilen hast du immer nach Doppelklick auf ein Oberflächenobjekt in ein dir von Delphi vorgefertigtes Raster eingetragen. Vielleicht hattest du inzwischen schon einmal beim Compilieren deines Projekts Probleme mit der Compilermeldung: „RegisterClasses-Aufruf fehlt oder ist inkorrekt“. Hoffentlich konntest du diesen Fehler abklären! Die Ursache dieses Programmierfehlers liegt in einem notwendigen, aber fehlendem „**end.**“ in deinem Programmtext, das du aus Versehen gelöscht oder als „**end;**“ für den Abschluss einer Prozedur verwendet hattest. Daher wollen wir uns nun einmal genauer mit dem Aufbau eines Delphi-Projektes und einer Delphi-Unit befassen.

Starte ein neues Projekt und schaue dir das **Programmgerüst** an, das Delphi dir für ein noch leeres Projekt zur Verfügung stellt, nämlich: **Unit1** und **Project1** - letzteres erhältst du, wenn du aus dem Menu „*Project/View Source*“ anwählst (einen Ausdruck dazu siehst du in der linken Spalte der Tabelle auf der nächsten Seite). Keine Angst, du musst vorerst fast nichts davon verstehen. Du brauchst lediglich einen Blick auf die **Änderungen** zu werfen, die sich **nach** der Anlage eines eigenen Programms ergeben, was du in der rechten Spalte der Tabelle findest.

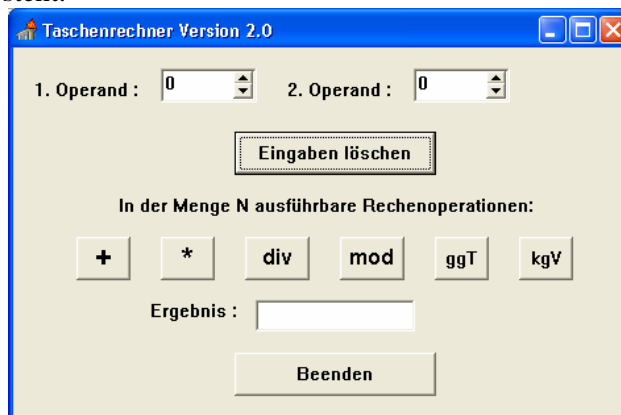
Aber zunächst noch einmal zum Listing eines leeren Projekts. Das Programm selbst (linkes oberes Feld) wird in der Regel immer von Delphi ganz automatisch für dich angelegt, sodass du in den Programmtext niemals irgendwelche Zeilen einzufügen hast. Anders bei der **Unit1**: diese enthält deine Oberflächenobjekte, wie auch die von dir erstellten Programmzeilen. Grundsätzlich erkennst du zwei wesentliche Abschnitte, einmal das „**interface**“ wie auch den „**implementation**“-Teil. Das reservierte Wort „**unit**“, gefolgt von ihrem Namen, leitet den Unittext ein, mit „**end.**“ wird der Unittext abgeschlossen (Beachte in der Notation den „**.**“ wie auch: zu diesem „**end.**“ fehlt offenbar ein „**begin**“!) - Erinnere dich hier an den oben aufgeführten Laufzeitfehler.

Das „**interface**“ besteht aus der **Deklaration** deiner Form, sowohl der Oberflächenobjekte wie auch aus den „**Köpfen**“ der Ereignisroutinen. Der „**implementation**“-Teil enthält die **Ausführungen** deiner im Interface aufgeführten Klick- und Change-Prozeduren. Dazu lade nun einmal ein eigenes Programm, z.B. dein Projekt „ASCII-Umwandlung“. Zum besseren Vergleich schalte das Unit-Fenster (das Code-Fenster) auf vollen Bildschirm (klicke dazu auf den rechten, oberen Button an der rechten, oberen Ecke deines aktiven Fensters!). Mit der „Bild-Oben-Taste“ gehe nun an den Anfang deiner Unit und nimm dir etwas Zeit für den Vergleich ...

<pre> <b>program</b> Project1; <b>uses</b>   Forms,   Unit1 in 'UNIT1.PAS' {Form1}; {\$R *.RES}  <b>begin</b>   Application.CreateForm(TForm1, Form1);   Application.Run; <b>end.</b> </pre>	<p>← An dem <b>Projekt Quelltext</b> hat sich nach deinen Aktionen auf dem Formblatt nichts bis auf den Unitnamen und den Formnamen geändert. (Denn du hast sicherlich die Ausführungen aus Abschnitt 1.3 beachtet!) Aber in der <b>Unit</b> hat sich einiges getan, wobei du selber lediglich nur etwa 20 Zeilen darin geschrieben hast ... Delphi hat im <b>Interface</b> in der „class“ zu deiner Form offenbar deine verwendeten <b>Objekte</b> und <b>Ereignisse</b> zur Kenntnis genommen und dort automatisch eingefügt. Im <b>Implementation</b>-Teil findest du dann die Ereignisse vollständig aufgeführt, einschließlich ihrer durch deine Anweisungen gefüllten begin-end-Blöcke...</p> <p>↓</p>
<pre> <b>unit</b> Unit1;  <b>interface</b> <b>uses</b>   SysUtils, WinTypes, WinProcs, Messages,   Classes, Graphics, Controls, Forms,   Dialogs;  <b>type</b>   TForm1 = <b>class</b>(TForm)    <b>private</b>     { Private-Deklarationen }   <b>public</b>     { Public-Deklarationen }   <b>end;</b>  <b>var</b>   Form1: TForm1;  <b>implementation</b> {\$R *.DFM}  <b>end.</b> </pre>	<pre> <b>unit</b> U_AsciiWandler;  <b>interface</b> <b>uses</b>   SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,   Controls, Forms, Dialogs, StdCtrls;  <b>type</b>   TF_ASCII = <b>class</b>(TForm)     L_Nr: TLabel;     L_Ch: TLabel;     E_nr: TEdit;     E_ch: TEdit;     B_clear: TButton;     B_end: TButton;     <b>procedure</b> B_clearClick(Sender: TObject);     <b>procedure</b> B_endClick(Sender: TObject);     <b>procedure</b> E_nrChange(Sender: TObject);     <b>procedure</b> E_chChange(Sender: TObject);   <b>private</b>     { Private-Deklarationen }   <b>public</b>     { Public-Deklarationen }   <b>end;</b>  <b>var</b>   F_ASCII: TF_ASCII;  <b>implementation</b> {\$R *.DFM}  <b>procedure</b> TF_ASCII.B_clearClick(Sender: TObject); <b>begin</b>   E_nr.Text:='';   E_ch.Text:='';   E_nr.Enabled:= true;   E_ch.Enabled:= true <b>end;</b>  <b>procedure</b> TF_ASCII.B_endClick(Sender: TObject); <b>begin</b>   close <b>end;</b>  <b>procedure</b> TF_ASCII.E_nrChange(Sender: TObject); <b>var</b> n: <b>integer</b>; <b>begin</b>   if E_nr.Text&lt;&gt;' '     <b>then</b> n:= StrToInt(E_nr.Text) {sonst StrToInt-Laufzeitfehler}     <b>else</b> n:= 0;   <b>if</b> (n &gt;= 32) <b>and</b> (n &lt;= 255)     <b>then begin</b> {der Bereich druckbarer Zeichen!}       E_ch.Text:= Chr(n); { ↑ so schreibt man Kommentare}       E_ch.Enabled:= false     <b>end</b> <b>end;</b>  <b>procedure</b> TF_ASCII.E_chChange(Sender: TObject); <b>begin</b>   <b>if</b> E_ch.Text&lt;&gt;' '     <b>then begin</b>       E_nr.Text:= IntToStr(Ord(E_ch.Text[1]));       E_nr.Enabled:= false     <b>end</b> <b>end;</b>  <b>end.</b> </pre>

### 2.3 Version 2: Taschenrechner in IN, erweitert um ggT- und kgV-Berechnung

Kehren wir nun wieder zu unserem „natürlichzahligen“ Taschenrechner zurück - die Erweiterung auf Dezimalzahlen sparen wir uns für die Version 3 auf. Bisher hast du dich im wesentlichen auf die Delphi-Oberfläche beschränken können und eine Reihe von Objekten mit ihren Eigenschaften und Ereignissen kennengelernt. Jetzt musst du aber mal „selber was programmieren“. Nach außen hin wird sich nicht viel ändern. Du sollst deine bisherigen Berechnungsbuttons um zwei neue erweitern, auf denen „ggT“ und „kgV“ steht.



Aber damit ist es nicht getan. Nach Aktivierung dieser Buttons soll auch tatsächlich der größte gemeinsame Teiler bzw. das kleinste gemeinsame Vielfache der beiden oben eingegebenen Zahlen im Ergebnisfenster erscheinen - aber dafür macht dir Delphi leider kein vorgefertigtes Angebot ... Der berühmte Doppelklick auf den neuen ggT-Button liefert dir zwar wieder ein Prozedurraster (falls du den Button B\_ggT genannt hast) ...

```
procedure TF_Rechner1.B_ggTClick(Sender: TObject);
begin

end;
```

... aber was soll nun da rein? Ein einfaches Rechenzeichen wie „+“ oder „mod“ zur Berechnung des ggT hat Delphi nicht anzubieten ...

Ich gehe im folgenden mal davon aus, dass die beiden Eingabezahlen nach wie vor in den SpinEditfeldern zur Verfügung stehen. Da es lästig ist, sie immer nur unter den langen Namen „SpE\_1.Value“ bzw. „SpE\_2.Value“ anzusprechen zu können, vereinbaren wir zunächst einmal (wie in der Mathematik üblich) kurze Variablennamen wie z.B. **a** und **b** für diese beiden Zahlen. Dies müssen wir dem Delphi-System natürlich mitteilen. Woher soll Delphi sonst auch wissen, was a und b sein sollen? Unter dem sog. „**Prozedurkopf**“ ist Platz für solche Vereinbarungen, wie du es schon in der Version 1 des Rechners z.B. in der B\_plusClick-Prozedur kennen gelernt hast. Die Zeilen zwischen „**procedure ...**“ und „**begin**“ stellen den sog. „**Deklarationsteil**“ der Prozedur dar. Hinter dem reservierten Wort „**var**“ können wir nun unsere eigenen Variablennamen mit Delphi vereinbaren und natürlich auch sagen, von welchem **Typ** diese sein sollen. Danach können wir diesen neuen Variablen dann mit dem schon bekannten **Zuweisungsoperator** „**:=**“ die Zahlenwerte der SpinEdit-Felder zuweisen. Merke dir aber schon mal gleich, dass die neuen Variablen a und b sogenannte „**lokale Variablen**“ sind, d.h. sie sind nur in dieser Prozedur „bekannt“, in anderen Programmteilen haben sie keine Gültigkeit.

Unsere Prozedur hat nach diesen Eingaben folgendes Aussehen:

```
procedure TF_Rechner1.B_ggTClick(Sender: TObject);
var a, b : longint;
begin
  a := SpE_1.Value; b := SpE_2.Value;
end;
```

Überlege nun einmal, wie du den ggT der beiden Zahlen a und b berechnen kannst. Wir werden in Zukunft auch sagen: überlege dir einen „**Algorithmus**“ zur Lösung dieses Problems ...

Vielleicht erinnerst du dich an deinen Mathematikunterricht früherer Jahrgangstufen. Dort wirst du verschiedene Lösungsmöglichkeiten zur ggT-Berechnung kennengelernt haben: eine basierend auf den beiden Primfaktorzerlegungen ..., und mindestens eine weitere basierend auf den Teilmengen beider Zahlen ...

Das letztere der genannten Verfahren greifen wir zunächst auf. Wie du feststellen kannst, ob eine Zahl eine andere ohne Rest teilt, kennst du schon aus dem bisherigen Taschenrechner in der Version 1: nämlich mittels der Operation „**mod**“. Wenn du also zunächst probierst, ob die größere der beiden Zahlen durch die kleinere teilbar ist (was i. a. recht selten sein wird...), dann die kleinere wiederholt um eins verringerst, solange bis diese „Laufzahl“ beide gegebenen Zahlen a und b ganzzahlig teilt, so hast du mittels dieser „Laufzahl“ den ggT von a und b bestimmt! Wir haben es hier also mit einer sogenannten „**Schleife**“, einer **Wiederholungsstruktur**, zu tun. Eine mögliche Formulierung dieses Algorithmus' findest du in der linken Spalte im folgenden Kasten ...

<b>Algorithmus 1 - berechne_ggT</b>	<b>Algorithmus 2 - berechne_ggT</b>
Input a, b : zwei ganze Zahlen	Input a, b : zwei ganze Zahlen
Output ggT : auch eine ganze Zahl	Output ggT : auch eine ganze Zahl
lokale Daten laufzahl : wieder eine ganze Zahl gefunden : eine boolesche Variable	lokale Daten keine
Falls $a < b$ dann $\text{laufzahl} \leftarrow a$ { $\text{laufzahl}$ erhält den sonst $\text{laufzahl} \leftarrow b$ Wert von a bzw. b } $\text{gefunden} \leftarrow \text{false}$ Solange noch kein gemeinsamer Teiler gefunden ist, mache folgendes: falls $(a \text{ MOD } \text{laufzahl} = 0)$ und $(b \text{ MOD } \text{laufzahl} = 0)$ dann $\text{gefunden} \leftarrow \text{true}$ sonst $\text{laufzahl} \leftarrow \text{laufzahl} - 1$ $\text{ggT} \leftarrow \text{laufzahl}$	Solange $a <> b$ mache folgendes: falls $a < b$ dann $b \leftarrow b - a$ sonst $a \leftarrow a - b$ $\text{ggT} \leftarrow a$

**Aufgabe 3:** Mache dir an verschiedenen Beispielen auch die Funktionsweise des zweiten der oben aufgeführten Algorithmen klar. Probiere es mit 150 und 54, oder mit 72 und 36, oder mit 11 und 12, ...

Vielleicht kennst du aber auch den **Algorithmus**, den schon vor ca. 2300 Jahren der große griechische Mathematiker **Euklid** herausgefunden hat, um den ggT von a und b zu bestimmen:

Bestimme den Rest, der bei der ganzzahligen Division von a durch b übrigbleibt, nimm dann das b als neues a und den Rest als neues b und mache so weiter - solange, bis irgendwann einmal der Rest 0 entsteht. Das letzte b, also die Zahl durch die du als letztes geteilt hast, ist dann der größte gemeinsame Teiler der ursprünglichen Zahlen a und b!

**Aufgabe 4:** Mache dir diesen rein umgangssprachlich formulierten Algorithmus an einigen Beispielen verständlich (z.B. Zahlenwerte s.o.), und versuche außerdem, diese rein sprachliche Fassung in ein Schema der oben gezeigten Art umzusetzen.

Zur Vervollständigung unserer ggT\_Click-Prozedur müssen wir jetzt nur noch die (in allen drei Algorithmen auftretende) **Wiederholungsstruktur** in korrekte **Delphi-Sprache** umsetzen. Die verwendete „**solange-Schleife**“ oder auch „**while-Schleife**“ hat nun die folgende Syntax:

```
while < ... Bedingung ... >
do begin
    < ... Anweisungsfolge ... >
end;
```

**Aufgabe 5:** Schreibe drei verschiedene Versionen der Click-Prozedur ... ( ... versuch's zuerst mal alleine ...)

Hast du es selbst herausbekommen? Wenn nein, dann solltest du die folgende auf dem Euklid-Algorithmus basierende Lösung jedoch nachvollziehen können:

```

procedure TF_Rechner1.B_ggTClick(Sender: TObject);
var a, b, rest : longint;
begin
  a := SpE_1.Value; b := SpE_2.Value;
  while a mod b <> 0
  do begin
    rest := a mod b;
    a := b;
    b := rest
  end;
  E_ergebnis.Text := IntToStr(b)
end;

```

Wenn die „while“-Schleife abbricht, weil der neue Rest bei  $a \bmod b$  gleich 0 ist, dann steht auf der Variablen b der Wert des ggT. Dieser Zahlenwert ist nur noch wieder in einen String zu konvertieren und der Textvariablen des Ergebnisfeldes zuzuweisen.

So weit, so gut. Was machen wir jetzt mit dem kgV? Ein Hilferuf an die Mathematik liefert uns eine einfache Möglichkeit: das kgV können wir über den ggT berechnen! Es gilt nämlich:

$$\text{ggT}(a,b) \cdot \text{kgV}(a,b) = a \cdot b,$$

eine Gleichung, die man ganz einfach nach  $\text{kgV}(a,b)$  auflösen kann:  $\text{kgV}(a,b) = a \cdot b \text{ div } \text{ggT}(a,b)$ . Müssen wir denn jetzt den gesamten ggT-Algorithmus in der  $\text{kgV\_Click}$ -Ereignisprozedur erneut schreiben? Schöner wäre es doch sicherlich, wenn wir die  $\text{kgV}$ -Prozedur jetzt einfach wie folgt schreiben könnten:

```

procedure TF_Rechner1.B_kgVClick(Sender: TObject);
var a, b, ggT, kgV : longint;
begin
  a := SpE_1.Value; b := SpE_2.Value;
  berechne_ggT(a, b, ggT);
  kgV := a * b div ggT;
  E_ergebnis.Text := IntToStr(kgV);
end;

```

Dazu müssen wir aber noch mit dem Delphi-System vereinbaren, was es beim Aufruf der Anweisung „**berechne\_ggT(a, b, ggT)**“ zu tun hat - denn diese Anweisung ist bislang kein Delphi bekannter Befehl. Wir haben also unsere erste eigenständige „**Prozedur**“ zu schreiben:

Was die Prozedur tun soll, sagt der **Name**, den wir ihr gegeben haben, mit welchen Daten sie was machen soll, das steht in der sogenannten „**Parameterliste**“ in runden Klammern hinter dem Prozedurnamen. Man nennt diesen Teil auch die „**Schnittstelle**“ der Prozedur zum Rest des Programms. Die Variablen a, b und ggT in dieser Schnittstelle müssen natürlich dem System bekannt sein, sind sie ja auch, wir haben sie ja als lokale Variable in der Click-Prozedur deklariert. Dabei sind a und b auch schon mit konkreten Zahlenwerten belegt, die Variable ggT dagegen ist so etwas wie ein noch leeres „**Einkaufskörbchen**“, das der Prozedur mitgegeben wird – in der Erwartung, es gefüllt mit dem größten gemeinsamen Teiler von a und b am Ende wieder **zurückzubekommen**. Unter einer solchen Prozedur können wir uns daher eine **Auftragserteilung an jemand dritten** vorstellen, wobei wir ihm alle nötigen Informationen, die er von uns als Beauftragter benötigt und die wir von ihm zurückerhalten wollen, mitgeben.

Es handelt sich hier also um zwei **unterschiedliche Arten der Variablenübergabe an eine Prozedur**. Im Falle von a und b übergibt (kopiert) man an die Prozedur lediglich zwei Zahlen, mit denen diese etwas anstellen soll, die man aber gar nicht irgendwie verändert zurückerwartet - es sind also einfach



nur „**Input**“-**Variable**, die nur in **einer** Richtung in die Prozedur hineingehen, aber nicht mehr herauskommen müssen. Im zweiten Fall handelt es sich um eine „**Input/Output**“-**Variable**, besagtes „Einkaufskörbchen“, das man in die Prozedur hineingibt (in unserem Beispiel mit einem noch völlig undefinierten Zahlenwert), und in dem man ein sinnvolles Ergebnis zurückerwartet, mit dem man dann weiterarbeiten kann. **Im Prozeduraufruf** „berechne\_ggT(a, b, ggT)“ ist diese unterschiedliche Bedeutung **nicht zu erkennen**, die Variablen werden in der Klammer einfach durch Kommata getrennt aufgelistet. **Wohl aber** muss man diesen Unterschied **im Kopf der Prozedur-Deklaration** deutlich machen, im „I/O“-Fall durch das schon bekannte Wörtchen „**var**“. Außerdem müssen in der Prozedur-Deklaration auch die **Datentypen** der Variablen angegeben werden. Es ist übrigens nicht erforderlich, dass die Variablen im Kopf der Prozedurdeklaration dieselben Namen haben wie die, mit denen man sie aufruft. Lediglich Anzahl, Reihenfolge und Typen der Variablen müssen übereinstimmen. „**Input**“-**Variable** kann ich übrigens **auch mit „Konstanten“ aufrufen**, „**I/O**“-**Variable dagegen nicht**, das verbietet schon der Mechanismus, den wir ihnen zugedacht haben. Diese beiden logisch zu unterscheidenden Übergabemodi werden im Fall eines reinen Inputs auch mit „**Call by Value**“ (**CbV**) und im Fall eines Outputs als „**Call by Reference**“ (**CbR**) bezeichnet.

Füllen wir in den Anweisungsteil der Prozedur noch den Euklidischen Algorithmus, so wie wir ihn bereits kennen, dann sieht das ganze folgendermaßen aus:

```

procedure TF_Rechner.berechne_ggT(a, b: longint; var ggT: longint);
var rest : longint;
begin
  while a mod b <> 0
  do begin
    rest := a mod b;
    a := b;
    b := rest
  end;
  ggT := b;
end;

```

Diese Prozedur musst du jetzt allerdings vollständig selber schreiben. Aber wohin? ... Irgendwo in den „**Implementation**“-Teil der Unit, am besten gleich oben vor den ganzen Ereignisprozeduren.

Da du die Prozedur nur in diesem Formular benötigst, musst du ihren Prozedurkopf im „**interface**“-Teil hinter dem Wort „**private**“ übernehmen. Dabei reicht es allerdings aus, den Prozedurkopf ohne das „TF\_Rechner.“ zu übernehmen. Später werden wir Prozeduren auch im „**public**“-Teil benötigen, doch dazu später mehr.

```

...
procedure B_ggTClick(Sender: TObject);
private
  { Private declarations }
=> procedure berechne_ggT(a, b: longint; var ggT: longint);
public
  { Public declarations }
end;

```

Ferner liegt es jetzt natürlich nahe, den Euklid-Teil auch aus der **ggTClick-Ereignis-Prozedur wieder zu entfernen** und dort ebenfalls durch den Aufruf von **berechne\_ggT(...)** zu ersetzen. Auch besteht jetzt eigentlich kein Grund mehr für die **lokalen Variablen a und b** in den Click-Prozeduren. Wir können die berechne\_ggT-Prozedur jetzt auch sofort mit den „**Value**“-**Werten der SpinEdit-Boxen aufrufen**.

Das sieht dann folgendermaßen aus:

```

procedure TF_Rechner1.B_ggTClick(Sender: TObject);
var ggT : longint;
begin
    berechne_ggT(SpE_1.Value, SpE_2.Value, ggT);
    E_ergebnis.Text := IntToStr(ggT)
end;

procedure TF_Rechner1.B_kgVClick(Sender: TObject);
var ggT, kgV : longint;
begin
    berechne_ggT(SpE_1.Value, SpE_2.Value, ggT);
    kgV := SpE_1.Value * SpE_2.Value div ggT;
    E_ergebnis.Text := IntToStr(kgV);
end;

```

Eleganter, als oben dargestellt, ist es, die ggT-Prozedur als sogenannte „**function**“ zu schreiben. Allein vom üblichen Sprachgebrauch her bietet sich dies an: die Sprechweisen und Notationen „ggT(a,b)“ oder auch „kgV(a,b)“ sind selbsterklärend und auch aus dem mathematischen Gebrauch bekannt. Somit ist der Auftrag „berechne\_ggT(a,b)“ in einen solchen umzuwandeln, der in sich schon einen Rückgabewert beinhaltet. Der Aufruf „**ggT(SpE\_1.Value, SpE\_2.Value)**“ bedeutet dann einerseits eine Auftragserteilung und andererseits stellt er selbst schon den Rückgabewert dar. Damit kann ein solcher Funktionsaufruf allerdings nicht mehr alleine stehen wie ein Prozeduraufruf! Daher kommt ein Funktionsaufruf meist innerhalb der rechten Seite einer Zuweisung vor!

Delphi bietet uns nun dazu auch das programmiertechnische Mittel. Allgemein hat eine **function** die folgende Struktur:

```

function <name> [(<Parameterliste>)] : <Datentyp>;
var ... ;           {eine Parameterliste kann, muss aber nicht auftreten}
begin
    <Anweisungen>;
    result := <Ergebnis> {abschließende Wertübergabe der Funktion}
end;

```

Wenden wir diese Überlegungen auf unser Rechnerprojekt an, so ändert sich die ggT-Routine zu ...

```

function TF_Rechner.ggT(a, b : longint) : longint;
var rest : longint;
begin
    while a mod b <> 0
    do begin
        rest := a mod b;
        a := b;
        b := rest
    end;
    ggT := b;
end;

```

... und die beiden Click-Ereignis-Prozeduren zu ...

( ... versuch's zuerst mal alleine ... )

```
procedure TF_Rechner1.B_ggTClick(Sender: TObject);
begin
  E_ergebnis.Text := IntToStr( ggT(SpE_1.Value, SpE_2.Value) )
end;

procedure TF_Rechner1.B_kgVClick(Sender: TObject);
var kgV : longint;
begin
  kgV := SpE_1.Value * SpE_2.Value div ggT(SpE_1.Value, SpE_2.Value)
  E_ergebnis.Text := IntToStr(kgV);
end;
```

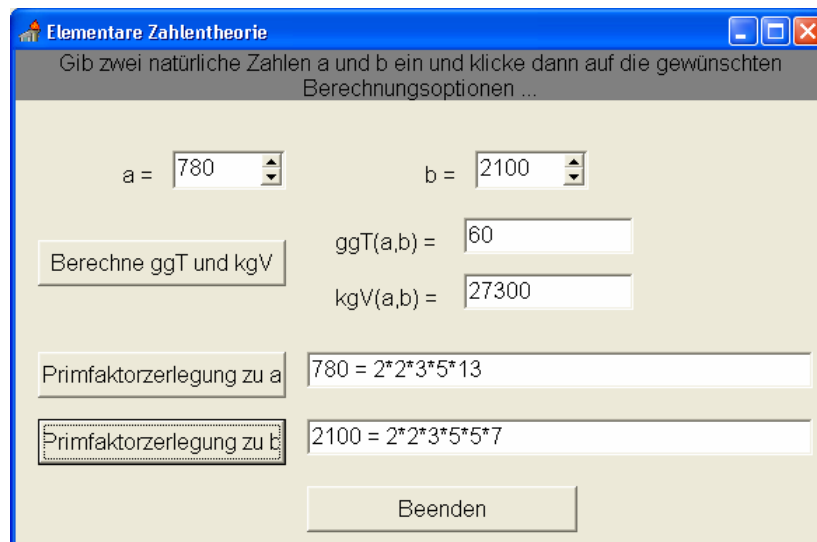
**Aufgabe 6:** Deklariere ähnlich dem ggT eine „function kgV(...)“ und variiere entsprechend dein Rechnerprojekt.

**Zeit, einmal kurz innezuhalten und sich die bisher gelernten Dinge noch mal vor Augen zu führen:**

**du kennst jetzt ...**

- ... den wesentlichen Aufbau eines Delphi-Projektes, das kurze Hauptprogramm mit der zur Form gehörigen Unit, weißt, welche Dateien Delphi dazu anlegt und wie du sie sinnvoll benennst und auf ein eigenes Unterverzeichnis abspeicherst ...
- ... einige wesentliche Objekte wie Form, Buttons, Labels, Edit- und SpinEdit-Felder, weißt, wie man sie sinnvoll benennt, wie man sie beschriftet, wie man weitere Eigenschaften im Objektinspektor einstellen kann, wie man diese Objekte auf der Form plaziert ...
- ... zwei Ereignis-Prozeduren, das Click- und das Change-Ereignis, weißt, dass du das zugehörige Raster durch einen Doppelklick auf das entsprechende Objekt in der Unit von Delphi vorgegeben bekommst ...
- ... die String- und Integer-Datentypen (auch Char und Boolean hast du schon kennen gelernt), weißt demnach auch zwischen z.B. edit.Text und SpinEdit.Value zu unterscheiden und kennst die Konvertierungsfunktionen IntToStr(...) und StrToInt(...), d.h. Du weißt, dass die Zahl 123, mit der man rechnen kann, datentypmäßig etwas ganz anderes ist als die Zeichenkette '123', mit der man nicht rechnen kann ...
- ... eigene Prozeduren und Funktionen, weißt, wo und wie man sie im „Implementation“-Teil plaziert, wie man sie aufruft und deklariert, weißt schon was über die Schnittstellen einer Prozedur, über die „I“- (CbV) und „I/O“-Parameter (CbR), über lokale Variable und du kennst auch schon eine der Wiederholungsstrukturen, die „While“-Schleife, sowie bedingte „if - then - else“-Verzweigungen ...

**Aufgabe 7:** Erweitere deinen „Rechner für N“ noch durch die Option, dass zu den beiden eingegebenen Zahlen auch noch auf Wunsch ihre **Primfaktorzerlegung** ausgedruckt wird (vielleicht erinnerst du dich, dass du früher damit auch den ggT und das kgV bestimmt hast). Das folgende Bild gehört zu einem anderen Programm - du kannst es aber ebensogut in dein Rechner-Projekt integrieren ...



Eine **algorithmische Idee** dazu wäre die folgende:

Die gegebene Zahl ist der Reihe nach durch alle Primzahlen zu teilen, und es ist immer zu prüfen, ob es „aufgeht“ oder nicht. Wenn die Division aufgeht, wird die Zahl durch diesen Quotienten ersetzt, und dann weitergeteilt. Da aber diese Primzahlen auch mehrfach in der Zahl enthalten sein können, musst du auf jeder Stufe diesen Prozess u. U. mehrfach wiederholen. Du hast es also hier mit **zwei (geschachtelten) Schleifen** zu tun:

- eine erste Schleife, die alle Primzahlen von 2 bis ? durchläuft, und abbricht, wenn der Quotient 1 erreicht ist,
- eine zweite Schleife, die für jede Primzahl die Division sooft ausführt, wie es geht.

Ein Problem ist allerdings, dass du die Folge der Primzahlen nicht so ohne weiteres verfügbar hast. Der Algorithmus wird aber nicht viel schlechter, wenn du zuerst die 2 probierst, dann die 3, die 5 und dann einfach alle ungeraden Zahlen weiter bis zu ? (bei 9 oder 15, usw. leistet der Rechner dann zwar unnütze Arbeit, aber das wird immer sofort erkannt, denn wenn's durch 3 oder 5 schon nicht mehr weiter zu teilen war, dann auch nicht durch 9 oder 15, usw.).

Im übrigen ist dies noch mal ein nettes Beispiel zur „String-Verkettung“, denn es bietet sich an, die Zerlegung als Zeichenkette anzusetzen und immer dann, wenn man einen weiteren Primteiler gefunden hat, diesen zusammen mit einem Malzeichen an den bereits existierenden String anzuhängen.

Man organisiert das Ganze am besten in einer eigenen Prozedur „Zerlege“, die als Input-Variable die Zahl bekommt und als I/O-Variable (denke an die „Einkaufskörbchen“ von S. 15!) den gewünschten ZerlegungsString. Versuch's zuerst selber, bevor du die Lösung auf der nächsten Seite anschaust ...!!! Formuliere zunächst den Algorithmus in sprachlicher Form, wie im Beispiel des ggT, dann erst als Delphi-Prozedur! Übrigens: auch eine Realisierung als Function wäre hier sinnvoll!!!

**Algorithmus zerlege\_in\_Primfaktoren**

Input	n:	eine ganze Zahl
Output	zerlegung:	die resultierende Zeichenkette, die mit '<Zahl>' + '=' vorinitialisiert ist

lokale Daten	teiler:	eine ganze Zahl
--------------	---------	-----------------

```

teiler ← 2
solange die Zahl n noch weiter zu zerlegen ist (d.h. n > 1), mache folgendes:
  - solange n MOD teiler = 0, mache folgendes:
    - zerlegung ← zerlegung + IntToStr ( teiler )
    - n ← n DIV teiler      {da n reine Inputvariable ist: ohne Auswirkung nach
außen}
  - falls noch ein weiterer Teiler vorliegt (d.h. n > 1),
    dann zerlegung ← zerlegung + '*'
  - falls teiler = 2, dann teiler ← 3
    sonst teiler ← teiler + 2

```

**Beachte** in dieser sprachlichen Fassung:

die erste Solange-Schleife enthält zwei Anweisungen, nämlich einerseits die zweite Solange-Schleife, andererseits die letzte bedingte Anweisung, die die Variable „teiler“ manipuliert. Die zweite Solange-Schleife beinhaltet drei Anweisungen: die zwei Zuweisungen bzgl. „zerlegung“ und „n“, als auch die bedingte Anweisung, die im Fall weiterer Teiler vorab schon das Multiplikationszeichen an die aktuelle Zeichenkette anhängt. Dieser logische Zusammenhang wird im Text am Besten durch Einrückungen und Gedankenstriche deutlichgemacht.

Übertragen in Delphi erhalten wir damit ...

```

procedure TF_Rechner.zerlege(n: longint; var zerlegung: String );
var teiler: longint;
begin
  teiler := 2;
  while n > 1
  do begin
    while n mod teiler = 0
    do begin
      zerlegung := zerlegung + IntToStr(teiler);
      n := n div teiler;
      if n > 1 then zerlegung := zerlegung + '*';
    end;
    if teiler = 2 then teiler := 3
      else teiler := teiler + 2
    end;
  end;
end;

```

Dabei bekommt diese Prozedur von der **aufzufindenden Stelle** zum einen die Zahl n übergeben und zum anderen den bereits vorinitialisierten String: **zerlegung := IntToStr(SpE\_1.Value) + '='** (wobei hier mit SpE\_1.Value der Wert der Zahl a oben gemeint ist), so dass nach Ablauf der Prozedur die Zeichenkette **zerlegung** direkt dem entsprechenden Edit-Ausgabefenster zugewiesen werden kann.

**Aufgabe 8:** Noch eine passende Ergänzung dazu - jetzt aber ohne jede Hilfestellung: Es sollen auch noch die **Mengen aller Teiler** der beiden Zahlen ausgegeben werden ...

**Aufgabe 9:** Schreibe einen **Taschenrechner für Bruchrechnung**, der zu zwei einzugebenden Bruchzahlen (Zähler und Nenner natürlich in getrennten übereinanderstehenden Edit-Feldern) die vier Grundrechenarten durchführt und das Ergebnis sowohl ungekürzt wie auch gekürzt ausgibt ...

### 3. Projekt: Spielautomat

An einem weiteren Beispiel wollen wir unsere bisher erworbenen Kenntnisse festigen und ein paar neue Dinge dazu lernen. Und zwar soll ein Spielautomat simuliert werden, so ein Ding, an dem man in den Kneipen (oder in Las Vegas) einen Haufen Geld verlieren kann ...

In einer einfachen Version soll der Automat aus drei „Spielrädern“ bestehen, auf denen jeweils die Ziffern 1 bis 4 oder ein Sternchen aufgemalt sind. Diese Symbole „laufen“ in einer schnellen Geschwindigkeit durch drei Fenster, und nach einer gewissen Zeit stoppen die Räder. Je nach Kombination der Fensterbilder gewinnt man dann etwa einen bestimmten Betrag - oder man verliert seinen Einsatz. Natürlich lassen wir hier nicht wirklich irgendwelche Räder laufen, sondern wir simulieren dies durch drei Edit-Fenster, in denen in zufälliger Reihenfolge und in schneller Abfolge die genannten Symbole erscheinen, bis dann nach einigen Sekunden ein Zufallsbild „eingefroren“ wird. Dazu brauchen wir ein Editfeld zur Anzeige des Kontostandes, einen Button zum Einwurf von Markstücken, einen Start-Button und einen Ende-Button, außerdem noch einen Gewinnplan.

Aus spielspsychologischer Sicht sollte man sich hier ein paar Gedanken über einen sinnvollen Gewinnplan machen. „Sinnvoll“ bedeutet: der „Erwartungswert“ der den Gewinn beschreibenden Zufallsfunktion sollte in etwa dem Spieleinsatz entsprechen bzw. „etwas“ geringer als der Spieleinsatz ausfallen. Dazu sind ein paar mathematische Überlegungen aus der Stochastik einzubringen. Bei drei Rädern und fünf Bildern pro Rad umfasst der Ereignisraum  $5^3 = 125$  gleichwertige Elementarereignisse. Die Wahrscheinlichkeit für ‘\*\*\*’ beträgt damit  $1/125$ , die Wahrscheinlichkeit für drei andere gleiche Radanzeigen  $4/125$  und für zwei Sternchen, egal auf welchen Rädern, dann  $3 \cdot 4/125 = 12/125$ , ...

Der Erwartungswert der Auszahlungsbeträge berechnet sich gemäß der (bekannt?! ) Formel:

$$E(X) = \sum_{i=1}^n x_i \cdot P(X = x_i)$$

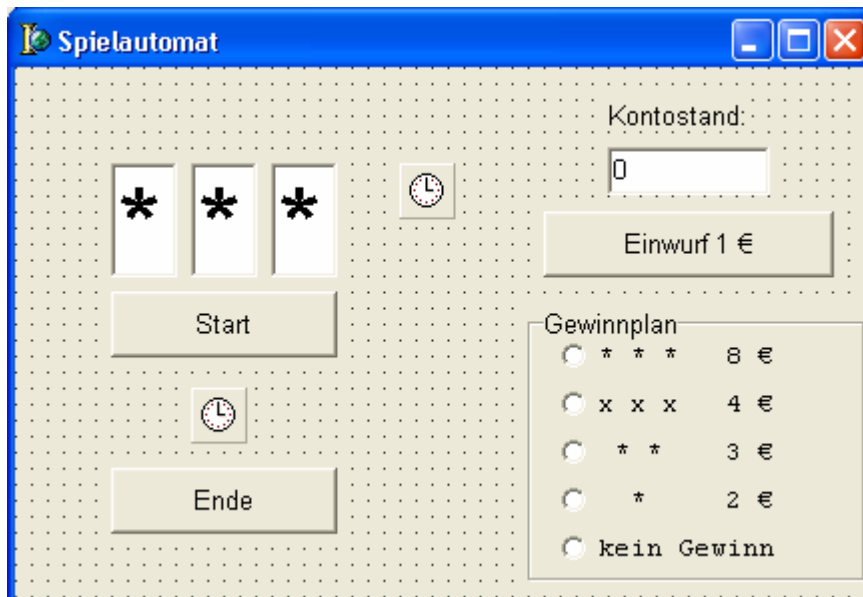
wobei hier gilt:

$n$  = Anzahl der unterschiedlichen Gewinnränge,

$x_i$  = die jeweilige Gewinnauszahlung.

... für den nachfolgend gezeigten Automaten gilt:  $E(X) = 92/125$ .

du kannst nach entsprechenden Überlegungen aber auch gerne einen anderen Gewinnplan für deinen Spielautomaten entwerfen...



Die meisten Oberflächenobjekte kennst du, wir nennen sie **E\_rad1**, **E\_rad2**, **E\_rad3** (farblich unterlegt und mit Sternchen-Symbolen im Objektinspektor vorinitialisiert), **E\_konto** ( mit '0' initialisiert ), **B\_einwurf**, **B\_ende** und **B\_start**, letzterer anfangs „disabled“ - ( warum wohl? - Klar, ohne Moos nix los ...). Die Editfelder werden ferner im Objektinspektor gegen Beschreibung seitens des Benutzers geschützt (disabled). Die Form selbst nennen wir z.B. **F\_spiel**.



Dann gibt es da noch was Neues: Die „**RadioGroup**“ mit der Beschriftung (Caption) „Gewinnplan“ und auf diesem Untergrund die „**Radiobuttons**“ die mit den einzelnen Gewinnsituationen beschriftet sind. Je nachdem, welche Gewinn- oder Verlustkombination am Ende eines Spieles aufgetreten ist, soll eine der Möglichkeiten durch einen Punkt im weißen Feld markiert werden, der dann zu Beginn eines neuen Spiels wieder entfernt werden muss. Du findest diese Radio-Objekte in der Menuleiste, und zwar in derselben Objektgruppe, wo auch die Buttons und die Edit-Felder stehen ...

Ferner brauchst du eine Information darüber, wie man im Delphi-System per „**Zufallszahlen-Generator**“ zufällige Zahlen (und - darüber gesteuert - dann auch beliebige zufällige Symbole oder Graphiken) erzeugen kann. Da wir hier nur **ganzzahlige** Zufallszahlen brauchen, erkläre ich auch zunächst nur, wie man diese bekommt:

Es gibt dazu eine Funktion namens **Random(n)**, die bei jedem Aufruf irgendeine ganze Zahl zwischen 0 und  $n - 1$  liefert,  $\text{Random}(5)$  also z.B. irgendeine ganze Zahl aus dem Bereich von 0 bis 4. Möchte man also z. B. irgendeine Lottozahl haben, dann kann man etwa einer Variablen „lottozahl“ vom Typ Integer einfach folgendes zuweisen:  $\text{lottozahl} := \text{Random}(49) + 1$ . Klar? Aber Lotto spielen wir erst später, für unseren Spielautomaten kommen wir mit  $\text{Random}(5)$  aus. Wenn wir ein Sternchen wollen, dann fragen wir einfach ab, ob die Zufallszahl gerade den Wert 0 hat und ersetzen dann den Wert auf dem gerade aktuellen Spielrad-Editfenster durch das Zeichen \* (falls dagegen irgendeine der Zahlen 1 bis 4 gekommen ist, müssen wir diese natürlich erst wieder in einen String umwandeln, bevor wir sie z.B. an `E_rad1.Text` zuweisen können ...).

Man erhält allerdings auf diese Weise bei jedem neuen Programmstart immer wieder dieselbe Folge von Zufallszahlen, was für Test- (oder „Pfusch“-) Zwecke ganz nützlich sein kann. Möchte man dieses vermeiden, so muss man anfangs einmal die Prozedur „**Randomize**“ aufrufen, die garantiert, dass bei jedem Neustart auch immer wieder eine andere Folge von Zufallszahlen gestartet wird. Der Zufallszahlengenerator wird damit irgendwie intern an die innere Uhr des Rechners angeknüpft. Im übrigen gibt es dicke Bücher darüber, wie man überhaupt auf einem Rechner gute Zufallszahlengeneratoren implementiert - das führt hier zunächst aber zu weit. Ein Hinweis nur noch: Die Funktion „**Random**“ (ohne die Klammern mit der Zahlenangabe dahinter) liefert eine zufällige Dezimalzahl zwischen 0 und 1.

Und noch was neues brauchen wir: Eine **Zeitsteuerung**, die es gestattet, den Inhalt der drei Räder-Fenster in Bruchteilen von Sekunden zu wechseln, und eine weitere, die ein Spiel nach wenigen Sekunden Durchlauf beendet. Diese „**Timer-Objekte**“ siehst du auf dem laufenden Programmfenster später nicht mehr, jedoch wohl **zur Entwicklungszeit** als **kleine Uhr-Symbole** auf deiner Form (s. o.!). Du erhältst ein solches Timer-Objekt aus der Menuzeile unter der Rubrik „System“. Ziehe dir also wie oben auf dem Bild zu sehen zwei solcher Timer-Symbole auf deine Form. Als erstes brauchst du natürlich wieder sinnvolle Namen für diese neuen Objekte. Wir wollen vereinbaren, diese Namen grundsätzlich mit **Ti\_** beginnen zu lassen. Den Timer für die Räder nennen wir z. B. „**Ti\_raeder**“, den für die gesamte Spielsteuerung „**Ti\_spiel**“. Im Objektinspektor siehst du nun nur erfreulich wenige manipulierbare Eigenschaften dieser Objekte. Die Eigenschaft **Name** hast du ja gerade schon geändert, die Eigenschaft **Enabled** setzt man sicherlich am besten erst mal auf **false**. In der Eigenschaft **Interval** kannst du jetzt eine Zeit in Millisekunden angeben, die später dafür verantwortlich ist, in welchen Zeitabständen immer wieder „irgendwas passieren“ soll. Gib hier für `Ti_raeder` einmal zunächst den Wert 50 ein, und bei `Ti_spiel` den Wert 5000.

Wo gehört nun das hin, was „passieren“ soll? Richtig, einfach ein Doppelklick auf das Timer-Symbol, und Delphi eröffnet dir ein Raster für die Timer-Ereignis-Prozedur, und da schreibst du jetzt einfach rein, was passieren soll ...

... bei **Ti\_raeder** sollen alle 50 ms drei neue zufällig ermittelte Zeichen in die entsprechenden Edit-Felder geschrieben werden. Wie das geht, haben wir gerade vorhin besprochen. Also:

```

procedure TF_Spiel.Ti_raederTimer(Sender: TObject);
var z : Byte;
begin
  z := Random(5);
  if z = 0 then E_rad1.Text := '*' else E_rad1.Text := IntToStr(z);
  z := Random(5);
  if z = 0 then E_rad2.Text := '*' else E_rad2.Text := IntToStr(z);
  z := Random(5);
  if z = 0 then E_rad3.Text := '*' else E_rad3.Text := IntToStr(z);
end;

```

Jetzt haben wir nur noch dafür zu sorgen, dass dieser Timer auch zum richtigen Zeitpunkt „Enabled“ wird, damit er alle 50 ms seinen Job macht – solange, bis er wieder „disabled“ wird. Dies muss natürlich in der **StartClick-Prozedur** geschehen, wo auch der Timer für einen gesamten Spieldurchlauf aktiviert wird. Dort müssen ferner der Kontostand um 1,- € Einsatz heruntersgesetzt werden, der Start-Button für die Zeit des laufenden Spiels deaktiviert werden und noch die Radio-Buttons alle auf nicht mehr „Checked“ gesetzt werden (um die im vorherigen Spiel angezeigte Gewinn-/Verlustsituation wieder „blank“ für das neu laufende Spiel zu schalten):

```

procedure TF_Spiel.B_startClick(Sender: TObject);
begin
  E_konto.Text      := IntToStr(StrToInt(E_konto.Text)-1);
  Ti_spiel.Enabled := true;
  Ti_raeder.Enabled := true;
  B_start.Enabled  := false;
  RB_1.Checked     := false;
  RB_2.Checked     := false;
  RB_3.Checked     := false;
  RB_4.Checked     := false;
  RB_5.Checked     := false;
end;

```

Was soll nun in der **SpielTimer-Prozedur** stehen? Sie soll ja dafür sorgen, dass ein Spiel 5 Sekunden (=5000 ms) dauert. Klar, diese Prozedur muss sich zum einen selbst „abschalten“, ebenso wie auch den Räder-Timer. Dann ruft sie eine von uns noch zu schreibende Auswertungsprozedur auf und aktiviert anschließend wieder den Start-Button, falls noch Geld für den Einsatz auf dem Konto ist:

```

procedure TF_Spiel.Ti_spielTimer(Sender: TObject);
begin
  Ti_raeder.Enabled := false;
  Ti_spiel.Enabled  := false;
  werte_aus;
  if E_konto.Text <> '0'
  then B_start.Enabled := true;
end;

```

In der **Auswertungsroutine** kannst du jetzt noch mal deine geschachtelten **if - then - else - Kenntnisse** testen. Probier's wieder zuerst selber, bevor du die Lösung auf der nächsten Seite nachliest ... Ein Hinweis sei dir aber gegeben:

Es gibt hier einen formal neuen Gesichtspunkt. In der Auswertungsprozedur arbeitest du zwar Günstigerweise mit einigen lokal deklarierten Variablen, aber ansonsten „nach außen“ hin nur mit den Objekten deiner Oberfläche. Insofern gehört diese Prozedur formal zu der „class“, die Delphi im Interface-Teil der Unit zu deinen Objekten angelegt hat. Wir führen den **Prozedurkopf** „**procedure werte\_aus;**“ daher dort unter dem Stichwort { **Private-Deklarationen** } auf. Wir müssen dann aber im Implementation-Teil der Unit die Zugehörigkeit dieser Prozedur zur „class“ dadurch deutlich machen, dass wir im Prozedurkopf dem von uns gegebenen Namen „**werte\_aus**“ noch **TF\_Spiel1.** vorsetzen, so wie das bei allen zur „class“ gehörigen Ereignisprozeduren von Delphi ansonsten automatisch gemacht wird:

```

procedure TF_Spiel.werte_aus;
var r1, r2, r3 : string;
    gewinn      : byte;
begin
  r1 := E_rad1.Text;
  r2 := E_rad2.Text;
  r3 := E_rad3.Text;
  if (r1 = '*') and (r2='*') and (r3 = '*')
  then begin gewinn := 8; RB_1.Checked := true end
  else if (r1 = r2) and (r2 = r3)
    then begin gewinn := 4; RB_2.Checked := true end
    else if ((r1 = '*') and (r2 = '*'))
      or ((r1 = '*') and (r3 = '*'))
      or ((r2 = '*') and (r3 = '*'))
    then begin gewinn := 3; RB_3.Checked := true end
    else if r2 = '*'
      then begin gewinn := 2; RB_4.Checked := true end
      else begin gewinn := 0; RB_5.Checked := true end;
  E_konto.Text := IntToStr(StrToInt(E_konto.Text) + gewinn)
end;

```

Tja, jetzt hab' ich dich möglicherweise doch ein bisschen „gelinkt“ - solltest du nämlich versucht haben, mit `if - then - else` und einfachen Bedingungen den Gewinnplan umzusetzen, dürftest du ganz fürchterlich ins Schleudern gekommen sein. Ich hatte auf S. 5 oben schon erwähnt, dass es da auch zusammengesetzte Bedingungen durch **Und-/Oder-Verknüpfungen** gibt. Wie das geht, und wie das syntaktisch gehandhabt wird, erklärt statt langer Worte einfacher das vorstehende Beispiel. Achte syntaktisch insbesondere auf die dann notwendigen runden Klammern um die einzelnen Bedingungen!

Was noch fehlt, ist die **Einwurf-Klick-Prozedur**, in der aber lediglich der Kontostand um 1,- € erhöht, sowie der Start-Button „Enabled“ werden muss, ferner natürlich noch das „close“ für die **Beendungsprozedur**. Und am Ende gar noch was Neues: durch Doppelklick irgendwo auf die Form tut sich auch ein Prozedurraster mit dem Namen „**FormCreate**“ auf. Schreibt man hier was rein, so wird dies **mit dem Start des Programms** jedesmal als erstes nach dem Aufbau des Bildschirms direkt mit ausgeführt, also noch vor jeder durch den Benutzer ausgelösten Aktion. Das bietet sich in diesem Beispiel dazu an, den oben erklärten Befehl „**randomize**“ hier reinzuschreiben, der bekanntlich bewirkt, dass später bei jedem Programmstart eine neue Folge von Zufallszahlen ausgelöst wird. Und nun viel Spaß bei deiner Realisierung. Übrigens, anstelle der Sternchen und Ziffern kannst du auch andere Symbole oder sogar Bilder in den Rädern anzeigen lassen ...

**Aufgabe 10:** Du kannst den Automaten noch ein bisschen wirklichkeitsgetreuer aussehen lassen, was natürlich einiges an Ergänzungen und Änderungen an deinem bisherigen Automaten nach sich zieht ...

- a) ... und ihm unter jedem Rad noch eine **eigene Stop-Taste** spendieren, sodass der Spieler das Gefühl hat, er könne sein Glück beeinflussen ... (dies erfordert natürlich einiges an Ergänzungen und kleinen Änderungen).
- b) ... auch ist es, wie schon erwähnt, überlegenswert, anstelle der Zeichen „\*,1..4“ andere Symbole (aus anderen Zeichensätzen!) zu verwenden. Noch schöner ist es, anstelle der Zeichen - und damit verbunden - anstelle der Edit-Objekte als Radanzeigen Oberflächenobjekte vom Typ `TImage` (in der Werkzeugleiste unter „zusätzlich“ zu finden) zu verwenden. Damit kannst du Icons oder auch `BitMaps` für die Räder einsetzen.
- c) ... ferner wäre es sehr interessant, eine **Statistik** darüber mitzuführen, wie oft die einzelnen Ereignisse während einer Serie von z. B. 100 oder 1000 Spielen aufgetreten sind. Die sich hieraus ergebenden zufälligen relativen Häufigkeiten kannst du dann den theoretischen Wahrscheinlichkeiten gegenüberstellen. Dies ist gleichzeitig ein **Test** für die Güte des **Zufallszahlengenerators**.

**Aufgabe 11:** Realisiere den nachstehend abgebildeten Fahrkartenautomaten. Nach Wahl einer Tarifzone (A = 1,80 €, B = 3,30 €, C = 4,70 €, D = 6,10€ und E = 7,40 €) soll der Fahrpreis im oberen Fenster angezeigt werden. Gleichzeitig werden die Münz-Buttons aktiviert, und man kann in beliebiger Münzstückelung Geld einwerfen. Dabei soll der Betrag im Fenster oben jeweils aktuell um den eingeworfenen Münzbetrag verringert werden. Solange der Fahrpreis noch nicht bezahlt ist, kann jederzeit auch die Geldrückgabebetaste betätigt werden, andernfalls wird jedoch sofort der Fahrschein ausgegeben und ggf. das Wechselgeld angezeigt. Soweit ist dieses Problem noch nicht von größerem Schwierigkeitsgrad, und du kannst dich sofort an die Realisierung machen. Etwas mehr Nachdenken erfordert es, wenn du auch noch in den Feldern unter dem Rückgeld anzeigt, aus wie vielen Münzen welcher Sorte das Wechselgeld besteht, ferner wenn du noch mitberücksichtigst, dass nicht immer unbeschränkt viele Münzen für das Wechselgeld im Automaten zur Verfügung stehen (z.B. wenn die Leute pausenlos immer nur 2 €-Stücke einwerfen. Dann müsste der Automat irgendwann auch mal mangels 10-Cent-Stücken o. ä. außer Betrieb gesetzt werden ...



Als Lösungsbeitrag hier nur eine Möglichkeit, zu einem gegebenen Rückzahlungsbetrag die Münzen auszurechnen (falls genügend vorhanden sind). Welche Strategie wird verfolgt?

```

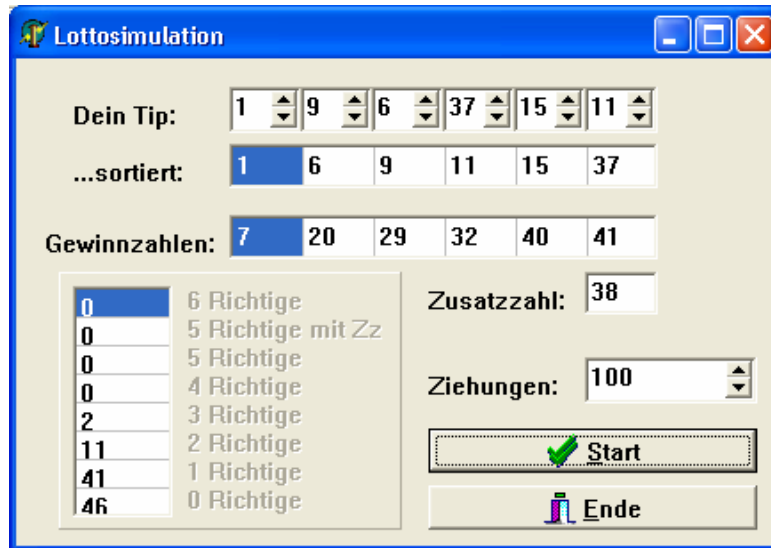
procedure berechne_Rueckgeld(betrag: extended; var z1, z50, z20, z10: integer);
begin
  while betrag >= 1
  do begin
    inc(z1);           {Inkrementieren bedeutet z1 := z1 + 1 }
    betrag := betrag - 1
  end;
  while betrag >= 0.5
  do begin
    inc(z50);
    betrag := betrag - 0.5
  end;
  while betrag >= 0.2
  do begin
    inc(z20);
    betrag := betrag - 0.2
  end;
  while betrag >= 0.09
  do begin
    inc(z10);
    betrag := betrag - 0.1
  end;
end;

```

Beachte die **I/O-Schnittstellen** für die Münzen-Anzahlen im Prozedurkopf, ferner die interne Prozedur **inc(z)**, die die in Klammern angegebene Variable z um 1 erhöht (inkrementiert) - Gegenstück dazu: **dec(z)** für das Dekrementieren einer Integer-Variablen (also Abziehen von 1), wie auch den neuen in der Schnittstelle der Prozedur auftretende Datentyp **Extended!**

## 4. Projekt Lottosimulation: 6 aus 49 - Einführung des Typs „ARRAY“

Anhand des folgenden Beispiels werden wir nun einen neuen und sehr wichtigen Datentyp kennenlernen. Die Oberfläche eines Programms zur Lotto-Simulation könnte etwa wie folgt aussehen:



Die logische „**Grob**“-**Struktur** ist klar: Der Benutzer gibt über die Spin-Edit-Felder seine Tippzahlen in beliebiger Reihenfolge ein, ferner die Anzahl der gewünschten Ziehungen - im Beispiel der Graphik hier also 100. Das Programm gibt dann nach Drücken des Buttons „Ziehung“ die Tippzahlen noch mal in sortierter Reihenfolge aus, zieht hundertmal neue Gewinnzahlen und wertet sie jedesmal im Vergleich mit den getippten Zahlen aus. Am Ende stehen die zuletzt gezogenen Gewinnzahlen (bei der wir diesmal keine „Richtige“ haben) noch in der Anzeige, und wir sehen ferner, dass wir in der vorstehenden Simulation z.B. nur zweimal drei Richtige hatten - ein leider etwas mageres Ergebnis, was wohl worauf schließen lässt? ...

Auf der gezeigten Form siehst du an drei Stellen ein neues Oberflächenobjekt: ein sogenanntes „**StringGrid**“. Das soll dich nicht erschüttern, denn solche Objekte kennst du aus Tabellenkalkulationsprogrammen. Es ist im Grunde nichts anderes als ein in „Spalten“ und „Zeilen“ unterteiltes Blatt, dessen Komponenten, die „Zellen“, über die Spaltennummern 0.. Spaltenzahl-1 (!) und 0..Zeilenzahl-1 „indiziert“ angesprochen werden können. Wichtige Eigenschaften - neben den dir bekannten - sind:

- **ColCount** und **RowCount** (Anzahl an Spalten und Zeilen),
- **FixedCols** und **FixedRows** (Kopf und Seitenzeilen z.B. zur Beschriftung der Spalten und Zeilen - vgl. Tabellenkalkulationsblatt), die wir hier nicht benötigen und daher auf 0 setzen,
- **ScrollBars** (mit den Einstellmöglichkeiten: both, vertical, horizontal und none; letzteres wählen wir hier für uns aus, da wir die Balken nicht benötigen)

Wie der Name schon sagt, können die Zellen nur Strings aufnehmen. Wollen wir z.B. - wie in unserem Fall - Zahlen eintragen, so benötigen wir wieder die uns bekannte Konvertierungsroutine `IntToStr()`! Die konkrete Belegung einer Zelle erfolgt dann gemäß

```
<StG_name>.Cells [<spaltennummer>,<zeilennummer>] := IntToStr ( ... )
```

Beachte die Spalten- und Zeilennummerierung beginnend ab der „0“! Mehr dazu am Ende dieses Abschnitts. Diese Informationen reichen aber aus, sodass du deine Form in Delphi erstellen kannst.

Kommen wir aber nun wieder auf das Kernproblem, nämlich die programmtechnische Realisierung zurück. Die SpinEdit-Felder `SpE_1 .. SpE_6` zur Eingabe der Tippzahlen sind zwar vielleicht nicht so schön, aber sie gewährleisten zumindest eine (fast) sichere Eingabe, da man den Zahlbereich von 1 bis 49 einstellen kann, und sinnlose Eingaben wie z.B. „-2,5“ oder gar „Otto“ nicht möglich sind. Aber was

passiert, wenn der Benutzer eine **Tippzahl mehrfach** eingibt? Selber schuld - oder können wir dem Benutzer diese Fehleingabe signalisieren und erst dann die Ziehungen durchführen lassen, wenn wirklich sechs verschiedene Zahlen eingegeben sind ( **Problem Nr. 1** )?

Wie **sortiert** das Programm die Tippzahlen und gibt sie dann sortiert wieder aus ( **Problem Nr. 2** )?

Wie wir eine Lottozahl erhalten, wissen wir schon. Wenn wir z.B. für die erste Gewinnzahl eine Variable `gzahl1` vom Typ Integer deklarieren, so leistet die Zuweisung `gzahl1 := Random(49)+1` das gewünschte. Wir könnten dann noch weitere Variable `gzahl2` bis `gzahl6` deklarieren und entsprechend mit zufälligen Lottozahlen beschreiben lassen, ebenso eine Variable `zzahl` für die Zusatzzahl - aber dann fangen die Schwierigkeiten erst an! Es könnte ja sein, dass bei den so gezogenen **Gewinnzahlen** auch wieder **gleiche** dabei sind. Wie kann ich das verhindern ( **Problem Nr. 3** )?

Und schließlich: Wie habe ich algorithmisch bei der **Auswertungsprozedur** vorzugehen, um die Anzahl der „Richtigen“ in meinem Tipp zu bestimmen, und das Ergebnis darüber hinaus jeweils auch für die Statistik nachzuhalten ( **Problem Nr. 4** )?

Also reichlich Probleme! Fangen wir mit der **Nr. 1** und der dazu ähnlichen **Nr. 3** an. Versuche einmal, durch „if-then-else“-Abfragen herauszufinden, ob unter den sechs vom Benutzer getippten Lottozahlen davon irgendwelche gleich sind ... - und wenn ja, was dann? Eine neue tippen lassen? Klar, aber stimmt die dann vielleicht mit einer anderen der schon getippten Zahlen überein? Also wieder diese ganze (ohnehin schon mühsame) Fragerei? Gleiches gilt für die über den Zufallszahlengenerator erzeugten und mit `gzahl1..gzahl6`, `zzahl` bezeichneten Gewinnzahlen!

Die Schwierigkeit liegt wesentlich darin, dass die verwendete einfache Datenstruktur, insbesondere bei den Gewinnzahlen mit lauter einzelnen Integer-Variablen, zu unflexibel ist. Wir bräuchten hier - ähnlich wie in der Mathematik z.B. bei Folgen oder Funktionen, aber allgemeiner verwendbar als das eben vorgestellte StringGrid - so etwas wie eine „indizierbare“ Variable, sodass wir etwa `gzahl[i]` mit `gzahl[j]` vergleichen könnten, wobei die Index-Variablen `i` und `j` die natürlichen Zahlen von 1 bis 6 durchlaufen. Wir brauchen also keine sechs verschiedenen Variablennamen, sondern nur einen einzigen für das ganze „**Feld**“ dieser Variablen und einen **indizierten Zugriff**. So etwas gibt es in Delphi, und wir müssen jetzt lernen, wie man damit umgeht. Im folgenden Kasten steht, wie's geht:

Im <b>Interface-</b> oder im <b>Implementationsteil</b> einer <b>Unit</b> oder im <b>Deklarationsteil</b> einer <b>Prozedur</b> schreibt man z.B. folgendes:	⇐ mit dem Ort unserer Deklaration werden wir uns noch konkret beschäftigen ...
<b>type</b> TLottofeld = <b>array</b> [1 .. 6 ] <b>of</b> Byte;	... wir haben damit einen <b>eigenen Datentyp</b> unter dem Namen <b>TLottofeld</b> deklariert, der aus einer „Kette“ von sechs Byte-Plätzen besteht ...
... <b>private</b> Gewinnzahlen, Tippzahlen : TLottofeld;	... auf deklarierte Variable von diesem Typ kann man nun im Programm „indiziert“ zugreifen, z.B.: Tippzahlen[3] := SpE_3.Value; i := 4; Gewinnzahlen[i] := Random(49)+1;

**Beachte:** der Zugriff auf die einzelnen **array**-Komponenten geschieht wie schon beim StringGrid unter Verwendung eckiger Klammern, in denen der jeweilige aktuelle Index steht!

Passend dazu gibt es eine sogenannte „**Zählschleife**“ (bislang kennen wir ja aus dem Skriptum als einzige Wiederholungsstruktur nur die „**While - do**“ - **Schleife**, die solange einen Anweisungsblock wiederholt, wie es die **Eintrittsbedingung** angibt bzw. diese abbricht, wenn die Eintrittsbedingung nicht mehr erfüllt ist). Die Zählschleife zur Wiederholung von Anweisungen ist immer dann angebracht, wenn von vornherein feststeht, **wie oft** die Wiederholung durchgeführt werden soll. Das folgende Beispiel erklärt sich in unserem Zusammenhang fast von selbst:



<pre>for i:=1 to 6 do begin   Gewinnzahlen[i]:= Random(49)+1 end;</pre>	... nach Ablauf dieser Schleife haben wir dann unsere 6 Gewinnzahlen, allerdings noch ohne die Zusatzzahl, auf unser „Hintergrund-Array“ Gewinnzahlen gespeichert ...
---	---

**Anmerkung:** Die **begin - end** - Klammerung wäre in diesem Beispiel verzichtbar, da es nur eine **einzige** Anweisung gibt, die sechsmal mit verschiedenen *i*'s wiederholt werden soll ...

Damit ist gegenüber der Speicherung auf einzelnen einfachen (nicht-indizierten) Variablen schon eine Menge gewonnen, allerdings noch nicht das Problem gleicher Zahlen gelöst. Dies versuchen wir zunächst einmal an der Prüfung der vom Benutzer eingegebenen Tippzahlen (**Problem Nr. 1**), weil das einfacher ist. Dazu speichern wir uns zuerst die Zahlen aus den SpinEdit-Feldern auf die oben deklarierte Array-Variable „Tippzahlen“ um. Da die SpinEdit-Felder ja leider nicht indiziert sind, bleibt uns dabei eine Kopie durch sechs einzelne Zuweisungen nicht erspart. Die Mühe lohnt sich aber, weil wir dann anschließend indiziert auf unsere Zahlen zugreifen können. Eine Prüfprozedur der Benutzereingaben, deren „Kopf“ wir, wie schon die Auswertungsroutine unseres Spielautomaten, in den Private-Abschnitt der class `F_Lotto` schreiben, und die außerdem das Hintergrund-ARRAY unserer Tippzahlen exportiert könnte dann etwa wie folgt aussehen - **wichtig:** der Datentyp **TLottofeld** muss vorher (d.h. noch vor der Class-Deklaration unserer Form!) typisiert werden! Warum wohl?

<pre>function TF_Lotto.Tipp_ok(var Tippzahlen:Tlottofeld): boolean; var i, j : Integer; begin   Tippzahlen[1] := SpE_1.Value; Tippzahlen[2] := SpE_2.Value;   Tippzahlen[3] := SpE_3.Value; Tippzahlen[4] := SpE_4.Value;   Tippzahlen[5] := SpE_5.Value; Tippzahlen[6] := SpE_6.Value;   result := true;   for i:=1 to 5 do     for j:=i+1 to 6 do       if Tippzahlen[i]=Tippzahlen[j]       then result:= false; end;</pre>
--

In den beiden geschachtelten For-Schleifen wird also jede getippte Zahl mit jeder anderen verglichen und im Fall der Gleichheit der boolesche Outputparameter `result` auf `false` gesetzt, sodass er dann außerhalb dieser Prüfprozedur abgefragt werden kann.

Bevor wir uns nun weiter mit der Lösung unserer gestellten Probleme befassen, machen wir uns nochmals in einer Übersicht klar, was ein `MouseClicked` auf unseren Ziehungsbutton zu bewirken hat:

<b>Algorithmus Spiel_starten</b> die I/O-Daten liegen im Oberflächenobjekt der Form vor
lokale Daten ok: ein boolescher „Switch“ zur Kontrolle der getippten Zahlen i: ein Zähler für die zu erfolgenden Ziehungen
- ok ← <b>Tipp_ok(Tippzahlen)</b> - Falls die Eingabe nicht ok ist, dann zeige dies dem Benutzer an sonst mache folgendes: - <i>sortiere (Tippzahlen)</i> - <i>zeige_an (Tippzahlen)</i> - für den Laufindex <i>i</i> mit Anfangswert 1 bis zur gewünschten Ziehungszahl mache folgendes: - <i>bestimme_Gewinnzahlen (Gewinnzahlen, Zusatzzahl)</i> - <i>zeige_Gew (Gewinnzahlen, Zusatzzahl)</i> - <i>zeige_Statistik ( Rang(Tippzahlen,Gewinnzahlen,Zusatzzahl) )</i>

Die Funktion „Tipp\_ok“ haben wir uns gerade schon klargemacht. Bei einer fehlerhaften Zahleneingabe kannst du dem Benutzer mittels der Delphi-Prozedur „`ShowMessage( '<Mld>' )`“ einen entsprechenden Hinweis geben (siehe dazu auch die Delphi-Hilfe). Wir müssen uns also noch um die im Kasten aufgeführten und kursiv geschriebenen Aktionen kümmern. Das Sortieren der

Tippzahlen übergehen wir zunächst und klären erst einmal, wie die Bestimmung der Gewinnzahlen, die schließlich mit unserer Eingabekontrolle eng verwandt ist, erfolgen kann:

<b>Algorithmus bestimme_Gewinnzahlen</b>	
Output	Gew: ein Feld mit den 6 üblichen Zahlen zZahl: eben die Zusatzzahl
lokale Daten	i, wert : Hilfszahlen
<ul style="list-style-type: none"> <li>- Für alle sechs Plätze im Gew-Feld (Laufvariable i!) mache folgendes:               <ul style="list-style-type: none"> <li>- wiederhole                   <ul style="list-style-type: none"> <li>- wert ← Random(49)+1</li> <li>bis <i>neu(Gew,i-1,wert)</i></li> </ul> </li> <li>- Gew[i] ← wert</li> </ul> </li> <li>- <i>sortiere (Gew)</i></li> <li>- wiederhole               <ul style="list-style-type: none"> <li>- wert ← Random(49)+1</li> <li>bis <i>neu(Gew,6,wert)</i></li> </ul> </li> <li>- zZahl ← wert</li> </ul>	

Wir können uns auf die Schulter klopfen, das war geschickt! Das Problem der Gleichheit haben wir auf eine weitere Routine, nämlich „neu (..)“ verlagert. Auch „sortiere (..)“ benötigen wir ein zweites Mal. Bevor du weiterliest, überlege doch einmal, wie dieses „neu (..)“ wohl arbeiten müsste ... Ebenso wie „Tipp\_ok(..)“ ordnen wir auch die Prozedur „bestimme\_Gewinnzahlen“ dem private-Bereich der class F\_Lotto zu.

```

procedure TF_lotto.bestimme_Gewinnzahlen (Var Gew: TLottofeld;
                                           Var zZahl: Byte      );
  function neu (t: TLottofeld; bisher, wert: Byte): Boolean;
  var bool: Boolean; j: Byte;
  begin
    bool:= true;
    for j:= bisher downto 1 do           {...eine „Abwärtsschleife“}
      if t[j]=wert then bool:= false;
    neu:= bool
  end;
  var i, wert: Byte;
  begin
    for i:= 1 to 6 do
      begin
        repeat                               {eine REPEAT-UNTIL-Schleife}
          wert:= Random(49)+1;
        until neu(Gew,i-1,wert);
        Gew[i]:= wert
      end;
      sortiere (Gew);
      repeat wert:= Random(49)+1 until neu(Gew,6,wert);
      zZahl:= wert
    end;
  end;

```

Wie du siehst, haben wir „neu ( )“ als Function umgesetzt unter Verwendung einer „**For-DownTo-Schleife**“. Außerdem hast noch etwas neues kennengelernt: die „**Repeat-Until-Schleife**“! So wie eine For-Schleife dann genutzt wird, wenn die Anzahl der Durchläufe von vornherein bekannt ist, eine While-Schleife eingesetzt wird, wenn ein durchlauf u. U. erst gar nicht erfolgt, so wird diese Wiederholungsstruktur verwendet, wenn einerseits die Anzahl der Durchläufe zu Anfang nicht bekannt ist und mindestens ein durchlauf zu geschehen hat. Die allgemeine Syntax ist ...

**Repeat**

<...Anweisungsfolge...>  
**Until** <...Abbruchbedingung...>

Wenden wir uns nun Problem **Nr. 4**, der Auswertung unseres Tipps zu. Der Tipp des Benutzers ist mit den gezogenen Zahlen zu vergleichen und einem **Gewinnrang** zuzuordnen.

```
function TF_Lotto.Rang (Tip,Gew: TLottofeld; zZahl: Byte): Byte;
  function Richtige(Tip,Gew: TLottofeld): Byte;
  ...
  function enthalten (z: Byte; Tip: TLottofeld): Boolean;
  ...
begin
  case Richtige(Tip,Gew) of
    0..2: Rang:= 0;
    3: if enthalten(zZahl,Tip)
        then Rang:= 2
        else Rang:= 1;
    4: Rang:= 3;
    5: if enthalten(zZahl,Tip)
        then Rang:= 5
        else Rang:= 4;
    6: Rang:= 6;
  end {of case}
end;
```

**Aufgabe 12:** Schreibe die beiden Funktionen „Richtige(..)“ und „enthalten(..)“ selbst!

Und wieder was neues ... die sogenannte „**Case-Anweisung**“. Die Zuordnung des Gewinnrangs aufgrund der Anzahl der Richtigen hätten wir auch über eine geschachtelte If-Then-Else-Struktur realisieren können! Aber, wie du dir sicherlich vorstellen kannst, wäre dies ein ziemlich unübersichtliches Geschäft geworden ... Da hilft uns dieser Anweisungstyp weiter, den wir später noch ausführlicher behandeln werden.

So bleibt uns schließlich noch Problem **Nr. 2**, nämlich die **Sortierung unserer Tippzahlen**. Dies kann auf sehr unterschiedliche Weisen geschehen. Bevor du dir den gewählten Algorithmus dazu anschaust, überlege einmal selbst, wie du eine ungeordnete Zahlenreihe oder auch einen ungeordneten Karteikasten oder deine Handkarten bei einem Kartenspiel sortieren könntest ...!

<b>Algorithmus Min-Sort:</b> Sortieren durch Auswahl des jeweils kleinsten Elementes	
I/O-Objekt	Daten - z.B. ein ungeordneter Karteikasten oder unsere Lottozahlen
lokale Objekte	i - der aktuelle linke Rand-Index des unsortierten Bereiches min_idx - eben derselbe im noch unsortierten Bereich
Für $i \leftarrow 1$ bis zum maximalen Index - 1 (!) mache folgendes:	
- min_idx $\leftarrow$ <i>Minpos</i> ( <i>Daten</i> , i )	
- falls $i < \text{min\_idx}$ dann <i>tausche</i> ( <i>Daten</i> [i], <i>Daten</i> [min_idx] )	

In Delphi könnte dies wie folgt aussehen:

```

procedure sortiere_durch_Auswahl (var f: TLottofeld);
    function Minpos (f: TLottofeld; vonIdx: Byte): Byte;
    ...
    var i, min_idx, hilf: Byte;
    begin {..... sortiere durch Auswahl .....}
        for i:=1 to 5 do begin
            min_idx:= Minpos(f,i);
            if i<>min_idx then begin {Tausche aus}
                hilf := f[i];
                f[i] := f[min_idx];
                f[min_idx]:= hilf
            end
        end {der For-Schleife}
    end;

```

**Aufgabe 13:** Schreibe die „function Minpos(..)“.

Abschließend noch einige Hinweise bezüglich der benötigten Ausgabeprozeduren. Dem Algorithmus auf Seite 27 kannst du entnehmen, dass wir drei verschiedene Ausgaben haben: die sortierte Ausgabe unseres Tipps, die ebenfalls sortierte Ausgabe der gezogenen Zahlen, sowie die Ausgabe der Statistik entsprechend der erreichten Gewinnränge. Alle Ausgaben sollen in den entsprechenden StringGrids angezeigt werden. Ich gebe dir die Prozedur „zeige\_Tip( )“ an. Beachte dabei, dass die Spalten- und Zeilennummerierung in StringGrids mit „0“ beginnt und in den Indexpaaren der Spaltenindex als erstes genannt werden muss! Die anderen beiden Prozeduren kannst du dann sicherlich selbständig implementieren ...

```

procedure TF_lotto.zeige_Tip (SortTip: TLottofeld);
    var i: Byte;
    begin
        for i:= 1 to 6 do
            StG_SortTip.Cells[i-1,0]:= IntToStr(SortTip[i])
        end;

```

Der vollständige private-Teil unserer Form sieht demnach wie folgt aus:

```

...
private
{ private-Deklarationen }
function Tipp_ok(Var TipZahlen: TLottofeld): boolean;
function Rang (Tip,Gew: TLottofeld; zZahl: Byte): Byte;
procedure bestimme_Gewinnzahlen (Var Gew: TLottofeld; Var zZahl: Byte);
procedure zeige_Tip (SortTip: TLottofeld);
procedure zeige_Gew (GewZahlen: TLottofeld; zZahl: Byte);
procedure zeige_Statistik (wert: Byte);

```

... nun viel Spaß beim Implementieren und Spielen ...

#### **Aufgabe 14:**

a) Erweitere dein Lotto-Projekt, sodass neben der Statistik der absoluten Daten auch die zugehörigen relativen Häufigkeiten, als auch die theoretischen Wahrscheinlichkeiten angegeben werden.

Anstelle des Typs „ARRAY[1..6] OF Byte“ hätten wir alternativ auch den folgenden Typ wählen können: „TFeld49 = ARRAY [1..49] OF Boolean“. Welche Konsequenzen hätte eine solche Wahl für die programmiertechnische Realisierung gehabt? Welche Auswirkungen hätte dies insbesondere auf die private-Routinen „pruefe\_Tip (..)“ und „bestimme\_Gewinnzahlen (..)“? Was wäre in diesem Fall die Konsequenz für unsere interne Prozedur „sortiere\_durch\_Auswahl (..)“?

- b) Die Eingabe der Tippzahlen des Benutzers könnte unter Verwendung eines booleschen Arrays viel eleganter als mit den SpinEdits realisiert werden: aus einem (zweidimensionalen) Raster mit 49 Feldern - ähnlich einem üblichen Tippzettel (ein StringGrid) - wählt der Benutzer per Mausclick seine Zahlen aus. Mehrfache Wahl ein und derselben Zahl ist auszuschließen, wie auch der Ziehungsbutton erst dann zu „enablen“ ist, wenn alle sechs Zahlen vom Benutzer ausgewählt wurden. Die Zahlen sind jeweils in einem booleschen Array festzuhalten; ebenso die sechs Gewinnzahlen ...

[Hinweis: schau dir einmal unter der Delphi-Hilfe das Ereignis *OnSelectCell* bzw. die Methode *SelectCell* bei StringGrids an, wie auch Beispiele dazu!]