3.2 TApplication, TScreen, TForm

Die drei Komponenten *TApplication*, *TScreen* und *TForm* sind nicht auf der Komponentenpalette zu finden. Von *TApplication* und *TScreen* wird jeweils automatisch eine Instanz erzeugt (namens *Application* beziehungsweise *Screen*).

Um einer Anwendung ein neues Formular hinzuzufügen, verwenden Sie den Menüpunkt DATEI | NEUES FORMULAR.

3.2.1 TApplication

Die Komponente *TApplication* kapselt allgemeine Eigenschaften, Methoden und Ereignisse, die nicht an ein bestimmtes Formular geknüpft sind. *TApplication* ist von *TComponent* abgeleitet.

Delphi erstellt automatisch eine Instanz namens *Application* von *TApplication*. Sie sollten *TApplication* nicht selbst instantisieren.

Anwendung

- Title (Eigenschaft, öffentlich), ExeName (Eigenschaft, öffentlich, nur Lesen)

  ```
  property Title: string;
  property ExeName: string;
  ```

  Den Namen der Anwendung setzen Sie mit Title oder zur Entwurfszeit unter PROJEKT | OPTIONEN | ANWENDUNG. Den Namen der Datei (inklusive dem Pfad) können Sie mit *ExeName* ermitteln.

- Icon (Eigenschaft, öffentlich)

  ```
  property Icon: TIcon;
  ```

  Das Icon für die Anweisung weisen Sie der Eigenschaft *Icon* zu. Sie können zur Entwurfszeit auch mit PROJEKT | OPTIONEN | ANWENDUNG ein Icon zuweisen.

- OnActivate, OnDeactivate (Ereignisse)

  ```
  property OnActivate(Sender: TObject);
  property OnDeactivate(Sender: TObject);
  ```

  Das Ereignis *OnActivate* tritt auf, wenn eine Anwendung den Fokus erhält, *OnDeactivate*, wenn sie ihn verliert.

Ereigniswarteschlange

- ProcessMessages (Methode)

  ```
  procedure ProcessMessages;
  ```

  Mittels der Methode *ProcessMessages* werden alle anstehenden Ereignisbehandlungsroutinen abgearbeitet.

  Rufen Sie bei langdauernden Opperationen hin und wieder diese Methode auf, damit die Anwendung auf Benutzereingaben reagieren und gegebenenfalls die Operation auch abbrechen kann.

```
procedure TForm1.Button1Click(Sender: TObject);
{Operation starten}
var
  i: integer;
begin
  Screen.Cursor := crHourGlass;
  FAbbruch := false;
  for i := 1 to 100000 do
  begin
      ...
    if i mod 1000 = 0 then
    begin
      Application.ProcessMessages;
      if FAbbruch = true
        then exit;
    end; {if i mod 1000 = 0 then}
  end; {for i := 1 to 100000 do}
  Screen.Cursor := crDefault;
end; {procedure TForm1.Button1Click}

procedure TForm1.Button2Click(Sender: TObject);
{Operation abbrechen}
begin
  FAbbruch := true;
end;
```

- OnMessage, OnShortCut (Ereignisse)

```
property OnMessage(var Msg: TMsg; var Handled: Boolean);
property OnShortCut(var Msg: TWMKey; var Handled: Boolean);
```

Das Ereignis *OnMessage* tritt auf, wenn die Anwendung eine Windows-Botschaft empfängt. Die Botschaft selbst wird als Parameter *Msg* übergeben. Wenn Sie in der *OnMessage*-Ereignisbehandlungsroutine abschließend auf die Botschaft reagieren, dann können Sie verhindern, dass die Botschaft an die einzelnen Formulare weitergereicht wird, indem Sie *Handled* auf *true* setzen.

Eine Windows-Anwendung kann – ohne dass etwas besonderes passiert – mehrere tausend Botschaften in der Sekunde erhalten. Die *OnMessage*-Ereignisbehandlungsroutine sollte deshalb möglichst schnell sein – zumindest bei allen Botschaften, auf die nicht reagiert werden soll.

Das Ereignis *OnShortCut* tritt bei allen Tastaturereignissen auf und ähnelt ansonsten *OnMessage*.

Die folgende Prozedur implementiert die Steuerung eines Lichtsteuersystems über die Funktionstasten. Zunächst wird sichergestellt, dass bei allen anderen Botschaften außer *WM_KeyDown* und *WM_KeyUp* die Prozedur sofort wieder verlassen wird.

Im zweiten Schritt wird geprüft, ob es sich bei der Taste um eine Funktionstaste bis F8 handelt und ob gleichzeitig die SHIFT-Taste betätigt wurde. Ist dies der Fall, dann werden die Methoden *ClickButton* und *FreeButton* der Komponente *TMasterFade* aufgerufen. (*TMasterFade* ist nicht in der VCL enthalten.)

```
procedure TForm1.ApplicationMessage(var Msg: TMsg;
  var Handled: Boolean);
var
  Master: TMasterFade;
begin
  if not ((Msg.Message = WM_KeyDown)
    or (Msg.Message = WM_KeyUp)) then exit;
  if (GetKeyState(VK_SHIFT) < 0)
    and (Msg.wParam > 111) and (Msg.wParam < 121) then
  begin
    Master := TMasterFade(Form12.FMasters[Msg.wParam - 112]);
    if Msg.Message = WM_KeyDown
      then Master.ClickButton(Master, mbLeft, [], 8, 8);
    if Msg.Message = WM_KeyUp
      then Master.FreeButton(Master, mbLeft, [], 8, 8);
  end; {if GetKeyState(VK_SHIFT) < 0 then}
end; {procedure TForm1.ApplicationMessage}
```

- OnIdle (Ereignis)

```
property OnIdle(Sender: TObject; var Done: Boolean);
```

Das Ereignis *OnIdle* tritt auf, wenn die Anwendung nichts zu tun hat. Normalerweise wird nach dem Abarbeiten der *OnIdle*-Ereignisbehandlungsroutine diese wieder aufgerufen, wenn die Anwendung immer noch nichts zu tun hat. Soll das verhindert werden, ist der Variablenparameter *Done* auf *true* zu setzen.

Ein etwas ausgefallenes Beispiel für die Verwendung von *OnIdle* ist das Öffnen von Tabellen und Abfragen beim Programmstart. Ist die Eigenschaft *Active* der *TTable*- und *TQuery*-Instanzen gleich *true*, dann dauert es etwas länger, bis die Anwendung gestartet ist, weil erst alle diese Datenmengenkomponenten geöffnet werden müssen.

Bei der folgenden Konstruktion werden der Reihe nach alle Datenmengenkomponenten geöffnet, wenn die Anwendung nach dem Start gerade nichts zu tun hat. Sobald alles geöffnet ist, wird die Ereignisbehandlungsroutine vom Ereignis getrennt.

```
procedure TForm1.FormShow(Sender: TObject);
begin
  FStep := 0;
  Application.OnIdle := Application.Idle;
end;

procedure TForm1.ApplicationIdle(Sender: TObject;
  var Done: boolean);
begin
  case FStep of
    0: Table1.Open;
    1: Table2.Open;
    2: Form2.Query1.Open;
       ...
    27: Application.OnIdle := nil;
  end; {case FStep of}
  inc(FStep);
  Done := false;
end; {procedure TForm1.ApplicationIdle}
```

- OnSettingsChange (Ereignis)

```
property OnSettingChange: (Sender: TObject; Flag: Integer; const
Section: string; var Result: Longint);
```

Das Ereignis *OnSettingsChange* tritt dann auf, wenn Windows-Systemeinstellungen geändert werden, beispielsweise das Hintergrundbild des Desktops ausgetauscht wird. Nähere Informationen dazu finden Sie in der Online-Hilfe.

Online-Hilfe

- HelpFile (Eigenschaft, öffentlich)

```
property HelpFile: string;
```

Der Eigenschaft *HelpFile* wird der Dateiname der Online-Hilfe zugewiesen. Die Hilfe-Datei der Anwendung wird nur dann verwendet, wenn dem aktuellen Formular keine eigene Hilfe-Datei zugewiesen ist.

Sie können zur Entwurfszeit auch mit PROJEKT | OPTIONEN | ANWENDUNG eine Hilfedatei zuweisen. Dabei wird der Dateiname der Hilfedatei inklusive dem vollständigen Pfad gespeichert.

Wird das Programm auf einem anderen Rechner installiert, so stimmt vermutlich der Pfad des zugewiesenen Dateinamens nicht mehr. Während die aktuelleren Delphi-Versionen die Hilfe-Datei auch dann finden, vorausgesetzt, sie befindet sich im selben Pfad wie die Exe-Datei, machen hier die frühen Delphi-Versionen Probleme. Diese können Sie umgehen, wenn Sie die Eigenschaft *HelpFile* erst zur Laufzeit setzen:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.HelpFile := ExtractFilePath(Application.ExeName)
    + '\DEMO300.HLP';
end;
```

- HelpCommand, HelpContext, HelpJump (Methoden)

```
function HelpCommand(Command: Word; Data: Longint): Boolean;
function HelpContext(Context: LongInt): Boolean;
function HelpJump(const JumpID: string): Boolean;
```

In den meisten Fällen wird die Online-Hilfe durch die Funktionstaste F1 aktiviert. Es wird dann diejenige Hilfe-Seite aufgerufen, die der Eigenschaft *HelpContext* der gerade fokusierten Komponente entspricht.

Mit Hilfe der Methoden *HelpCommand*, *HelpContext* und *HelpJump* kann man die Online-Hilfe auch programmgesteuert öffnen. Mit *HelpContext* rufen Sie eine Seite der Online-Hilfe auf, wobei die Kontext-Nummer der betreffenden Seite als Parameter übergeben wird:

```
Application.HelpContext(1);
```

Um eine Hilfe-Seite nach dem Kontext-String zu öffnen, wird die Anweisung *HelpJump* verwendet.

```
Application.HelpJump('How_to_use_Macros');
```

Spezielle Anweisungen an die Online-Hilfe rufen Sie mit HelpCommand auf. Um die Suchen-Funktion der Online-Hilfe aufzurufen, verwenden Sie das Kommando *Help_PartialKey*. Im folgenden Beispiel wird die Suchen-Funktion mit dem Begriff *Macros* initialisiert. Soll die Suchen-Funktion ohne einen speziellen Begriff aufgerufen werden, so ist ein Zeiger auf einen leeren String zu übergeben.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  s: string;
begin
  s := 'Macros';
  Application.HelpCommand(HELP_PartialKey, integer(@s[1]));
end;
```

Es ist auch möglich, die Online-Hilfe der Online-Hilfe aufzurufen:

```
procedure TMain.Hilfebenutzen2Click(Sender: TObject);
begin
  Application.HelpCommand(HELP_HelpOnHelp, 0);
end;
```

- OnHelp (Ereignis)

```
property OnHelp(Command: Word; Data: Longint;
  var CallHelp: Boolean): Boolean;
```

Das Ereignis *OnHelp* wird dann ausgelöst, wenn vom Programm die Online-Hilfe aufgerufen wird – beispielsweise durch F1. Um das Aufrufen der Online-Hilfe zu verhindern, muss der Variablen-Parameter *CallHelp* auf *false* gesetzt werden.

Die möglichen Werte für die Parameter *Command* und *Data* finden Sie in der *Win32*-Hilfe unter dem Stichwort *Help Functions*. Beachten Sie bitte, dass das Ereignis *OnHelp* eine Funktion als Ereignisbehandlungsroutine erwartet.

Hints

Hints sind Hinweistexte, die dann angezeigt werden (können), wenn der Mauszeiger länger auf einer Komponente verweilt.

- Hint (Eigenschaft, veröffentlicht), OnHint (Ereignis)

```
property Hint: string;
```

Mit der Eigenschaft *Hint* kann der (lange) gerade aktuelle Hinweistext ermittelt werden.

Das Ereignis *OnHint* tritt auf, wenn ein Mauszeiger länger auf einer Komponente verweilt. *OnHint* wird auch dann ausgelöst, wenn die Eigenschaft *ShowHint* der betreffenden Komponente den Wert *false* hat, nicht jedoch, wenn deren Eigenschaft *Hint* ein leerer String zugewiesen ist.

Es ist möglich, der *TControl*-Eigenschaft *Hint* zwei Hinweistexte zuzuweisen, die durch einen senkrechten Strich zu trennen sind:

```
Edit1.Hint := 'Eingabe|Geben Sie hier den Text ein';
```

Der erste Hinweistext wird immer an der Komponente angezeigt, während der zweite Text beispielsweise in einer Statusleiste angezeigt werden kann:

```
type
  TForm1 = class(TForm)
    ...
  private
    procedure ApplicationHint(Sender: TObject);
    ...
  end;

procedure TForm1.ApplicationHint(Sender: TObject);
begin
  StatusBar1.SimpleText := Application.Hint;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := ApplicationHint;
end;
```

- HintPause, HintHidePause (Eigenschaften, öffentlich)

```
property HintPause: Integer default 500;
property HintHidePause: Integer default 2500;
```

Mit *HintPause* stellen Sie ein, wie viele Millisekunden der Mauszeiger über der jeweiligen Komponente verweilen muss, bis der Hinweistext eingeblendet wird. Mit *HintHidePause* wird spezifiziert, wann der Hinweistext wieder ausgeblendet wird.

- HintColor (Eigenschaft, öffentlich)

```
property Color: TColor default clInfoBk;
```

Mit *HintColor* kann man die Hintergrundfarbe derjenigen Hinweistexte ändern, die direkt an der Komponente angezeigt werden.

- OnShowHint (Ereignis)

```
property OnShowHint(var HintStr: string; var CanShow: Boolean;
  var HintInfo: THintInfo);
```

Das Ereignis *OnShowHint* wird ausgelöst, bevor ein Hinweistext angezeigt wird. Um das Anzeigen zu verhindern, wird der Variablen-Parameter *CanShow* auf *false* gesetzt.

Der anzuzeigende Hinweistext kann mittels *HintStr* abgeändert werden. Wenn Sie Details, beispielsweise die Hintergrundfarbe, individuell einstellen möchten, dann können Sie das mit *HintInfo* tun. Der Typ *THintInfo* ist in der Online-Hilfe detailliert beschrieben.

Exceptions

- OnException (Ereignis)

```
property OnException(Sender: TObject; E: Exception);
```

Das Ereignis *OnException* wird ausgelöst, wenn eine Exception auftritt, welche nicht durch eine *try..except..end* oder eine *try..finally..end*-Konstruktion abgefangen wurde.

- ShowException (Methode)

```
procedure ShowException(E: Exception);
```

Ist *OnException* keine Ereignisbehandlungsroutine zugewiesen, dann wird *ShowException* automatisch bei jeder Exception aufgerufen und zeigt in einem Meldungsfenster den Exceptiontext an.

Soll das Meldungsfenster erscheinen, obwohl *OnException* eine Ereignisbehandlungsroutine zugewiesen wurde, muss ShowException dort explizit aufgerufen werden.

```
procedure TForm1.ApplicationException(Sender: TObject;
  E: Exception);
begin
    ...
  Application.ShowException(E);
end;
```

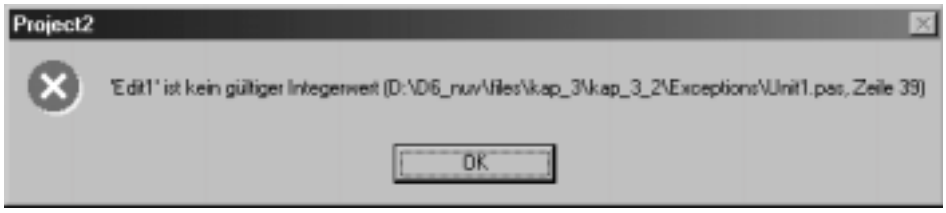
Effektives Fehlerhandling

Vielen Programmen mangelt es an einem effektiven Fehler-Handling. Manchmal werden *except..end*-Blöcke einfach leer gelassen (»ich schlage mir meine Fehlermeldungen tot«), oder man findet Standard-Meldungen wie *Es ist ein Fehler aufgetreten*. (Damit kann weder der Anwender noch der Entwickler wirklich etwas anfangen.)

Vermutlich ist ein wirklich anwenderfreundliches Fehlermanagement gar nicht zu realisieren. Dann sollte aber wenigstens der Entwickler mit den Fehlermeldungen etwas anfangen können, vor allen sollte er möglichst genau die Stelle lokalisieren können, wo das Problem aufgetreten ist. Zu diesem Zweck lassen sich Assertions etwas zweckentfremden:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: integer;
begin
  try
    i := StrToInt(Edit1.Text);
    Label1.Caption := IntToStr(i * 2);
  except
    on E: Exception
      do Assert(false, E.Message);
  end;
end;
```

Die Prozedur *Button1Click* ist ein ganz simples Beispiel für die Möglichkeit eines Fehlers. Im *except..end*-Block wird mittels einer Assert-Anweisung, deren Parameter *Condition* auf *false* gesetzt wird, eine erneute Exception ausgelöst, der die Fehlermeldung der ursprünglichen Fehlermeldung weitergereicht wird.



Nun wird auf das *TApplication*-Ereignis *OnException* dadurch reagiert, dass diese Exceptions neben Programmversion sowie Datum und Uhrzeit in eine Datei geschrieben werden. Anschließend wird auch noch eine Fehlermeldung angezeigt.

```
procedure TForm1.ApplicationEvents1Exception(Sender: TObject;
  E: Exception);
var
  sl: TStringList;
  s: string;
begin
  s := ExtractFilePath(Application.ExeName) + 'fehler.txt';
  sl := TStringList.Create;
  try
    if FileExists(s)
      then sl.LoadFromFile(s);
    sl.Add(PROG_VERS + FormatDateTime('dd.mm.yyyy hh.mm.ss', now));
    sl.Add(E.Message);
    sl.SaveToFile(s);
  finally
    sl.Free;
  end;
  Application.ShowException(E);
end; {procedure TForm1.ApplicationEvents1Exception}
```

Im Support-Fall kann man nun den Anwender bitten, diese Datei per eMail-Anhang einem zuzusenden und sieht dann gleich, an welcher Stelle (sprich: in welcher Unit und an welcher Zeilennummer) der Fehler aufgetreten ist. Zu diesem Zweck ist es erforderlich, alle herausgegebenen Programmversionen zu archivieren. Mittels der auch angezeigten Programmversion sieht man dann, welche Version gerade verwendet wurde.

Prinzipiell wäre es auch denkbar, beim Auftreten einer Exception gleich eine eMail an den Entwickler zu veranlassen. Man könnte auch nach einiger Zeit diese Dateien einsammeln und daraus Anregungen für die weitere Entwicklung und den Support gewinnen.

3.2.2 TScreen

Die Komponente *TScreen* kapselt die Anzeige des Programms auf dem Bildschirm. *TScreen* ist von *TComponent* abgeleitet.

Delphi erzeugt automatisch eine Instanz von *TScreen* namens *Screen*. Sie sollten *TScreen* nicht selbst instantisieren.

- Width, Height, PixelPerInch (Eigenschaften, öffentlich, nur Lesen)

  ```
  property Height: Integer;
  property Width: Integer;
  property PixelsPerInch: Integer;
  ```

  Die Höhe und die Breite des Bildschirms in Pixeln wird mittels der Eigenschaften *Height* und *Width* ermittelt. Mit *PixelPerInch* lässt sich ermitteln, wie viele Pixel pro Inch (nach Ansicht von Windows) gezeichnet werden.

  Mit der folgenden Anweisung kann man dann die Zollgröße des Monitors ermitteln – oder auch nicht. Mangels Einstellmöglichkeit »degradiert« NT 4.0 meinen *Vision Master Pro 21* zum schnöden 15-Zöller.

  ```
  Label1.Caption := FloatToStr(SQRT(SQR(Screen.Height)
    + SQR(Screen.Width)) / Screen.PixelsPerInch);
  ```

- MonitorCount, Monitors (Eigenschaften, öffentlich, nur Lesen)

  ```
  property MonitorCount: Integer;
  property Monitors[Index: Integer]: TMonitor;
  ```

  Seit Windows 98 werden auch Multi-Monitor-Systeme unterstützt. Die Anzahl der Monitore lässt sich mit *MonitorCount* herausfinden, Informationen über die einzelnen Monitore über *Monitors*. Näheres in der Online-Hilfe.

- Fonts (Eigenschaft, öffentlich, nur Lesen)

  ```
  property Fonts: TStrings;
  ```

  Mit *Fonts* kann ermittelt werden, welche Schriftarten vom Bildschirm unterstützt werden. Die folgende Anweisung weist die Namen dieser Schriften einer Listbox zu:

  ```
  ListBox1.Items := Screen.Fonts;
  ```

  Die für den Drucker verfügbaren Schriften ermitteln Sie mit der *TPrinter*-Eigenschaft *Fonts*.

Cursor

- Cursor (Eigenschaft, öffentlich)

```
property Cursor: (-32768..32767;);
```

Der Cursor wird mit der Eigenschaft *Cursor* eingestellt. Das folgende Beispiel zeigt, wie man vor einer längeren Operation den Sanduhr-Cursor einstellt und hinterher wieder den Normalzustand herstellt.

```
try
  Screen.Cursor := crHourGlass;
    ...
finally
  Screen.Cursor := crDefault;
end;
```

- Cursors (Eigenschaft, öffentlich)

```
property Cursors[Index: Integer]: HCursor;
```

Die verfügbaren Cursor finden sind in der Array-Eigenschaft *Cursors* aufgelistet. Die folgende Abbildung zeigt die vordefinierten Cursor:

Um selbstdefinierte Cursor aus einer Ressourcen-Datei zu laden, gehen Sie wie folgt vor:

```
implementation

{$R *.DFM}
{$R Lupe.RES}

const
  crLupe = 1;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ...
  Screen.Cursors[1]
    := LoadCursor(HINSTANCE, 'Lupe');
end;
```

Zunächst muss die Ressourcen-Datei eingebunden werden. Desweiteren sollte eine Konstante für die Cursor definiert werden. Die in Windows vorbelegten Konstanten bewegen sich alle im negativen Bereich, so dass für selbstdefinierte Konstanten der positive Bereich bleibt. Der Cursor wird dann mit der API-Funktion *LoadCursor* aus der Ressource geladen.

Formulare und Controls

- ActiveControl, ActiveForm (Eigenschaften, öffentlich, nur Lesen)

  ```
  property ActiveControl: TWinControl;
  property ActiveForm: TForm;
  ```

  Mit *ActiveControl* kann das Steuerelement referenziert werden, das gerade den Fokus hat. *ActiveForm* liefert einen Zeiger auf das gerade aktive Formular.

- OnActiveControlChange, OnActiveFormChange (Ereignisse)

  ```
  property OnActiveControlChange(Sender: TObject);
  property OnActiveFormChange(Sender: TObject);
  ```

  Wenn ein anderes Steuerelement den Fokus erhält, tritt das Ereignis *OnActiveControlChange* auf. Wenn ein anderes Formular aktiviert wird, wird *OnActiveFormChange* ausgelöst.

- CustomFormCount, CustomForms (Eigenschaft, öffentlich, nur Lesen)

  ```
  property CustomFormCount: Integer;
  property CustomForms[Index: Integer]: TCustomForm;
  ```

  Mit *CustomFormCount* kann die Anzahl der vorhandenen Formulare ermittelt werden, die Array-Eigenschaft *CustomForms* liefert Zeiger auf die einzelnen Formulare.

- DataModuleCount, DataModules (Eigenschaft, öffentlich, nur Lesen)

  ```
  property DataModuleCount: Integer;
  property DataModules[Index: Integer]: TDataModule;
  ```

  Mit *DataModuleCount* erfährt man die Zahl der vorhandenen Datenmodule, *DataModules* liefert Zeiger auf die einzelnen Datenmodule.

3.2.3 TForm

Die Komponente *TForm* kapselt ein Windows-Hauptfenster. *TForm* ist von *TWinControl* abgeleitet.

Formular anzeigen

- Show, Close, ShowModal (Methoden), ModalResult (Eigenschaft, öffentlich)

```
procedure Show;
function ShowModal: Integer;
property ModalResult: integer;
procedure Close;
```

Mit der Prozedur *Show* zeigen Sie ein Formular an. Mit der Funktion *ShowModal* zeigen Sie ein Formular »modal« an – das heißt, mit der Programmausführung wird erst dann fortgefahren, wenn das Formular geschlossen wird. Ist ein Formular modal geöffnet, dann können Sie derweil nicht die anderen Formulare aktivieren (mit der Ausnahme derjenigen Formulare, die vom modalen Formular aus geöffnet wurden).

Die Funktion *ShowModal* gibt einen Integer-Wert zurück, der dem der Eigenschaft *ModalResult* des modal angezeigten Formulars entspricht. Indem Sie ModalResult einen Wert ungleich null zuweisen, schließen Sie gleichzeitig das Formular. Für die Eigenschaft ModalResult sind einige Konstanten definiert (*mrNone*, *mrOk*, *mrCancel*, *mrAbort*, *mrRetry*, *mrIgnore*, *mrYes*, *mrNo*, *mrAll*).

Das folgende Listing zeigt, wie Sie ein Formular zur Laufzeit erstellen und dann modal anzeigen.

```
Form2 := TForm2.Create(nil);
try
  if Form2.ShowModal = mrOk then
  begin
    ...
  end;
finally
  Form2.Free;
end;
```

Mit der Methode *Close* wird das Formular geschlossen.

- OnShow, OnClose, OnCloseQuery (Ereignisse)

```
property OnShow(Sender: TObject);
property OnCloseQuery(Sender: TObject; var CanClose: Boolean);
property OnClose(Sender: TObject; var Action: TCloseAction);
```

Nach dem Öffnen eines Formulars tritt das Ereignis *OnShow* auf.

Wird ein Formular geschlossen, tritt zunächst das Ereignis *OnCloseQuery* auf. Sie können das Schließen des Formulars verhindern, indem Sie den Variablen-Parameter *CanClose* auf *false* setzen.

Die folgende *OnCloseQuery*-Ereignisbehandlungsroutine prüft zunächst, ob die Daten geändert und noch nicht gespeichert wurden. Ist dies der Fall, wird eine Sicherheitsabfrage durchgeführt. Äußert der Anwender während dieser Sicherheitsabfrage den Wunsch, die Datei zu speichern oder den Vorgang abzubrechen, wird *CanClose* auf *false* gesetzt.

```
procedure TChild.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
var
  Ergebnis: TModalResult;
begin
  if Fgeaendert = true then
  begin
    Ergebnis := MessageDlg(' Datei ' + caption +
      ' wurde geändert. Speichern ?',
      mtwarning, [mbYes, mbNo, mbAbort], 0);
    if Ergebnis = mrNo then exit;
    CanClose := false;
    if Ergebnis = mrYes then
    begin
      if Speichern1.enabled = true
        then Speichern1Click(Sender)
        else Speichernunter1Click(sender);
    end; {if Ergebnis = mbYes then}
  end; {if geaendert = true}
end; {procedure TChild.FormCloseQuery}
```

Nach *OnCloseQuery* tritt das Ereignis *OnClose* auf. Der Variablenparameter *Action* kann auf folgende Werte gesetzt werden:

- *caNone*: Das Formular wird nicht geschlossen.
- *caHide*: Das Formular wird verborgen. Ein Zugriff auf das Formular ist weiterhin möglich. Dies ist die Voreinstellung des Variablenparameters.
- *caMinimize*: Das Formular wird zum Symbol verkleinert.

- *caFree*: Die Instanz des Formulars wird freigegeben. Ein Zugriff auf das Formular ist dann nicht mehr möglich.

- Visible (Eigenschaft, veröffentlicht), Hide (Methode), OnHide (Ereignis)

```
property Visible: Boolean default true;
procedure Hide;
property OnHide(Sender: TObject);
```

Um ein Formular zu verbergen, wird *Visible* auf *false* gesetzt oder die Prozedur *Hide* aufgerufen. Das Ereignis *OnHide* tritt auf, wenn das Formular verborgen wird, also auch dann, wenn *Close* mit *caHide* aufgerufen wird.

Aussehen des Formulars

- BorderStyle (Eigenschaft, veröffentlicht)

```
property BorderStyle: (bsNone, bsSingle, bsSizeable, bsDialog,
  bsToolWindow, bsSizeToolWin) default bsSizeable;
```

Mittels der Eigenschaft *BorderStyle* lässt sich nicht nur spezifizieren, mit welchem Rahmen das Formular angezeigt wird, sondern auch, wie die Titelleiste aussieht:

- *bsNone*: Das Formular wird ohne Rahmen und ohne Titelleiste angezeigt.
- *bsSingle*: Das Formular wird mit einfachem Rahmen sowie mit Titelleiste angezeigt. Das Formular lässt sich zum Symbol verkleinern sowie auf Bildschirmgröße vergrößern, andere Größenänderungen sind jedoch nicht möglich.
- *bsSizeable*: Formular mit doppeltem Rahmen, deshalb in der Größe stufenlos änderbar.
- *bsDialog*: Formular lässt sich weder in der Größe ändern noch zum Symbol verkleinern.
- *bsToolWindow* und *bsSizeToolWin*: Fenster mit kleinerer Titelzeile. *bsSizeToolWin* lässt sich in der Größe stufenlos ändern.

- BorderIcons (Eigenschaft, veröffentlicht)

```
property BorderIcons: set of (biSystemMenu, biMinimize,
  biMaximize, biHelp);
```

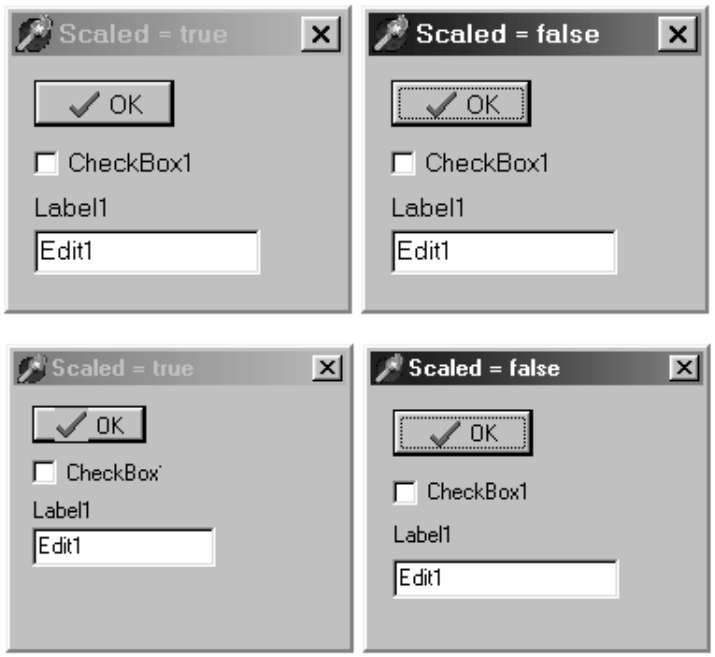
Ob ein System-Menü und / oder die Buttons zum Vergrößen und Verkleinern des Formulars angezeigt werden, hängt vor allem von der Eigenschaft *BorderStyle* ab. Mittels *BorderIcons* können jedoch Buttons von der Anzeige ausgeschlossen werden.

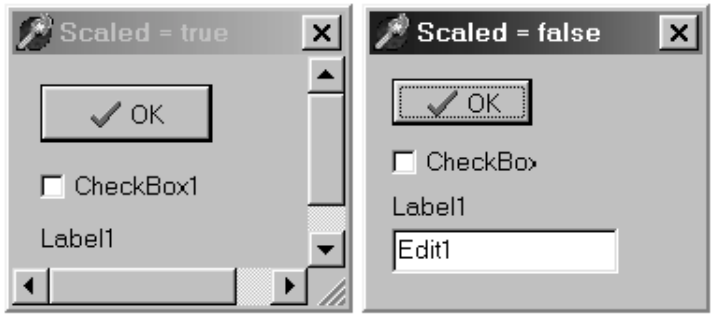
Hat *BorderStyle* den Wert *bsDialog*, dann kann mit *biHelp* ein Fragezeichen-Button hinzugefügt werden. Wird dieser Button angeklickt, dann erhält der Mauszeiger die Form eines Fragezeichens. Mit diesem Fragezeichen können nun die Komponenten des Dialogs angeklickt werden, die dann die Online-Hilfe aufrufen.

- Scaled (Eigenschaft, veröffentlicht)

```
property Scaled: boolean default true;
```

Hat *Scaled* den Wert *true*, dann wird das Formular entsprechend skaliert, wenn sich der Wert *PixelPerInch* ändert – beispielsweise bei der Einstellung einer anderen Bildschirmauflösung, oder wenn von großen auf kleine Schriften umgestellt wird. Diese an sich gute Idee führt in der Praxis fast immer zu unbrauchbaren Resultaten, so dass Sie *Scaled* stets auf *false* setzen sollten.





Vergessen Sie auch dann nicht, wenn Sie *Scaled* auf *false* gestellt haben, der Beschriftung von Checkboxen ausreichend Platz zu schaffen.

Position des Formulars

- Position (Eigenschaft, veröffentlicht)

```
property Position: (poDesigned, poDefault, poDefaultPosOnly,
  poDefaultSizeOnly, poScreenCenter, poDesktopCenter)
  default poDesigned;
```

Per Voreinstellung wird das Formular an der Position angezeigt, die es vor der Compilierung auf dem Entwicklungsrechner hatte. Wenn der Entwickler mit einer deutlich höheren Bildschirmauflösung arbeitet als der Endanwender, kann das dazu führen, dass der Endanwender das Formular dann gar nicht mehr sieht.

Wird Position auf *poDefault* gesetzt, dann wird die Größe und die Position des Formulars von Windows festgelegt. Es ist auch möglich, nur die Position (*poDefaultPosOnly*) oder nur die Größe (*poDefaultSizeOnly*) von Windows festlegen zu lassen.

Formulare lässt man meist in der Bildschirmmitte (*poScreenCenter*) anzeigen. Der Wert *poDesktopCenter* ist bei Mehrbildschirmsystemen interessant, näheres siehe Online-Hilfe.

- WindowState (Eigenschaft, veröffentlicht)

```
property WindowState: (wsNormal, wsMinimized, wsMaximized)
  default wsNormal;
```

Mit *WindowState* kann das Formular bildschirmfüllend vergrößert (*wsMaximized*) oder zum Symbol verkleinert (*wsMinimized*) werden.

MDI-Anwendungen

- FormStyle (Eigenschaft, veröffentlicht)

```
property FormStyle: (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop)
  default fsNormal;
```

Mit *FormStyle* wird spezifiziert, von welcher Art ein Formular ist. Frei verschiebbare Formulare haben als *FormStyle* den Wert *fsNormal*. Hat ein Formular den Wert *fsStayOnTop*, dann wird es nicht durch andere Formulare überlagert, es sei denn, auch diese haben den Wert *fsStayOnTop*.

Formulare, die nur innerhalb eines anderen Formulars angezeigt werden, sind so genannte Kindfenster. Sie kennen solche Kindfenster beispielsweise von einer Textverarbeitung, wo jeder Text in einem eigenen Kindfenster angezeigt wird. Um ein solches Kindfenster zu erstellen, muss *FormStyle* auf *fsMDIChild* gesetzt werden. Das Rahmenformular muss den Wert *fsMDIForm* erhalten.

Beachten Sie, dass Kindfenster nicht verborgen werden können. Ändern Sie *FormStyle* auf *fsMDIChild*, dann wird auch immer das Formular angezeigt – es bleibt auch dann angezeigt, wenn Sie *FormStyle* wieder zurück aus *fsNormal* setzen.

- ActiveMDIChild (Eigenschaft, öffentlich, nur Lesen)

```
property ActiveMDIChild: TForm;
```

Über *ActiveMDIChild* kann auf das gerade aktivierte Kindfenster zugegriffen werden.

- MDIChildCount, MDIChildren (Eigenschaften, öffentlich, nur Lesen)

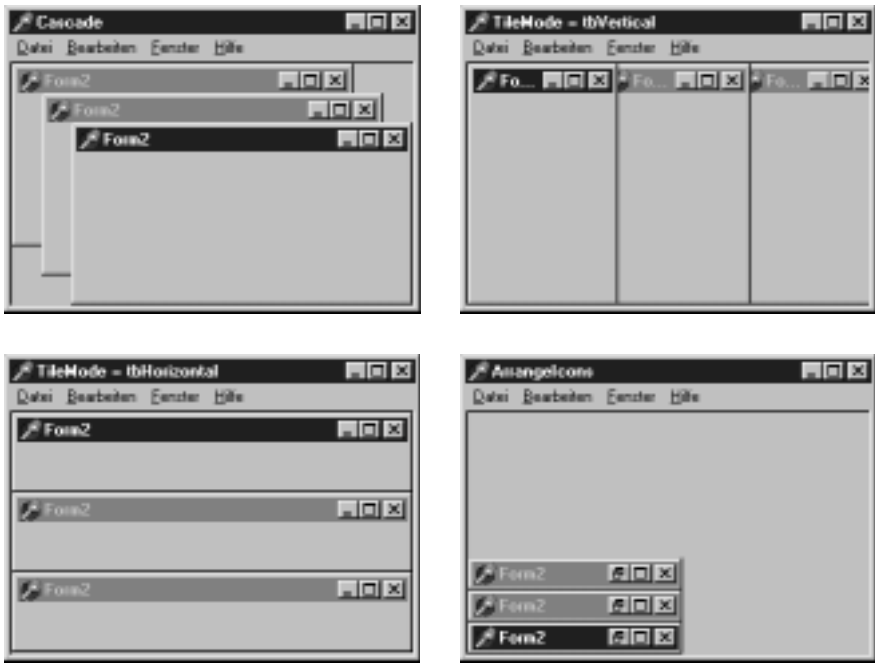
```
property MDIChildCount: Integer;
property MDIChildren[I: Integer]: TForm;
```

Die Anzahl der vorhandenen Kindfenster kann mit *MDIChildCount* ermittelt werden. Die Array-Eigenschaft *MDIChildren* stellt Zeiger auf die einzelnen Kindfenster zur Verfügung.

- ArrangeIcons, Cascade, Tile (Methoden), TileMode (Eigenschaft, öffentlich)

```
procedure ArrangeIcons;
procedure Cascade;
procedure Tile;
property TileMode: (tbHorizontal, tbVertical);
```

Mit *Cascade* werden die Kindfenster überlappend angeordnet, mit *Tile* übereinander oder nebeneinander, je nach Wert von *TileMode*. Sind die Kindfenster zu Symbolen verkleinert, so können diese mit *ArrangeIcons* angeordnet werden.



- Next, Previous (Methoden)

  ```
  procedure Next;
  procedure Previous;
  ```

  Diese Methoden werden verwendet, um das nächste (*Next*) oder das vorhergehende (*Previous*) Kindfenster zu aktivieren.

- WindowMenu (Eigenschaft, veröffentlicht)

  ```
  property WindowMenu: TMenuItem;
  ```

  Die Liste der vorhandenen Kindfenster wird gewöhnlich dem Fenster-Menü als Menüpunkt angehängt. Mit einem Klick auf einen dieser Menüpunkte kann das jeweilige Kindfenster aktiviert werden. Dafür ist kein Quelltext erforderlich, lediglich die Eigenschaft *WindowMenu* des MDI-Rahmenfensters muss auf den entsprechenden Menüpunkt (meist *Fenster1*) gesetzt werden.

  Beachten Sie bitte, dass dieses Menü nicht mit einem Trennstrich beendet werden darf, sonst passiert überhaupt nichts.

Fokus

- OnActivate, OnDeactivate (Ereignisse)

  ```
  property OnActivate(Sender: TObject);
  property OnDeactivate(Sender: TObject);
  ```

  Das Ereignis *OnActivate* tritt auf, wenn ein Formular den Fokus erhält, *OnDeaktivate*, wenn es ihn verliert. Ein Formular erhält auch dann den Fokus, wenn eines seiner Steuerelemente den Fokus erhält.

  Die *TForm*-Ereignisse *OnActivate* und *OnDeactivate* treten dann nicht auf, wenn der Fokus von einer anderen Anwendung oder zu einer anderen Anwendung wechselt. Dann treten jedoch die gleichnamigen *TApplication*-Ereignisse auf.

- Active (Eigenschaft, öffentlich, nur Lesen)

  ```
  property Active: Boolean;
  ```

  Mittels *Active* kann ermittelt werden, ob ein Formular den Fokus hat.

- ActiveControl (Eigenschaft, veröffentlicht)

  ```
  property ActiveControl: TWinControl;
  ```

  Mit *ActiveControl* kann bestimmt werden, welche untergeordnete Komponente den Fokus hat. Diese Komponente muss ein *TWinControl*-Nachfolger sein, da andere Komponenten keinen Fokus erhalten können.

- FocusControl, DefocusControl (Methoden)

  ```
  procedure FocusControl(Control: TWinControl);
  procedure DefocusControl(Control: TWinControl; Removing: Boolean);
  ```

  Mit *FocusControl* wird einer untergeordneten Komponente der Fokus zugewiesen, mit *DefokusControl* wird er ihr entzogen.

Scrollbalken

- AutoScroll (Eigenschaft, veröffentlicht)

  ```
  property AutoScroll: Boolean default true;
  ```

  Hat *AutoScroll* den Wert *true*, dann werden bei Bedarf automatisch Scrollbalken angezeigt.

- ScrollInView (Methode)

  ```
  procedure ScrollInView(AControl: TControl)
  ```

  Die Methode *ScrollIn View* verschiebt den angezeigten Ausschnitt so, dass die als Parameter übergebene Komponente – nach Möglichkeit vollständig – angezeigt wird.

- HorzScrollBar, VertScrollBar (Eigenschaften, veröffentlicht)

```
property HorzScrollBar: TControlScrollBar;
property VertScrollBar: TControlScrollBar;
```

Mittels der Objekt-Eigenschaften *HorzScrollBar* und *VertScrollBar* kann auf den waagerechten und senkrechten Scrollbalken zugegriffen werden. Der Typ *TControlScrollBar* ist in Kapitel 3.4.10 bei *TScrollBox* ausführlich beschrieben.

Sonstiges

- Icon, HelpFile (Eigenschaften, veröffentlicht)

```
property Icon: TIcon;
property HelpFile: string;
```

Für gewöhnlich übernimmt jedes Formular das Icon und die Hilfedatei der Anwendung. Mittels dieser beiden Eigenschaften kann jedoch einem Formular ein eigenes Icon und eine eigene Hilfedatei zugewiesen werden.

- Menu (Eigenschaft, veröffentlicht)

```
property Menu: TMainMenu;
```

Mit *Menu* wird eingestellt, welches Hauptmenü das Formular verwenden soll. Sie können zur Laufzeit zwischen verschiedenen Menüs wechseln. Wie Sie verschiedene Menüs mischen, ist bei *TMainMenu* in Kapitel 3.3.2 beschrieben.

- Parent (Eigenschaft, öffentlich)

```
property Parent: TWinControl;
```

Im Gegensatz zu anderen *TWinControl*-Nachfolgern werden Formulare auch dann angezeigt, wenn *Parent* den Wert *nil* hat – sie sind dann frei auf dem gesamten Bildschirm verschiebbar.

- Canvas (Eigenschaft, öffentlich, nur Lesen), OnPaint (Ereignis)

```
property Canvas: TCanvas;
property OnPaint(Sender: TObject)
```

Mittels des Objectes *Canvas* kann direkt auf das Formular gezeichnet werden. Beachten Sie dabei, dass diese Zeichnungen nicht gespeichert werden und daher wiederholt werden müssen, wenn das Formular neu gezeichnet werden muss – beispielsweise, wenn es zwischenzeitlich von einem anderen Formular verdeckt war.

Muss das Formular neu gezeichnet werden, tritt das Ereignis *OnPaint* auf. Verwenden Sie *OnPaint* nur für Zeichenoperationen, die Sie selbst auf dem Canvas ausführen.

- OnShortCut (Ereignis)

  ```
  property OnShortCut(var Msg: TWMKey; var Handled: Boolean);
  ```

  Wird eine Taste gedrückt, solange das Formular aktiviert ist, dann wird *OnShortCut* aufgerufen, bevor die Windows-Botschaft zum fokusierten Steuerelement weitergeleitet wird. Mittels *Msg* können Sie das Zeichen ermitteln und auch ändern. Setzen Sie *Handled* auf *true*, dann wird die Windows-Botschaft nicht zum fokusierten Steuerelement weitergereicht.

- OnCreate, OnDestroy (Ereignisse)

  ```
  property OnCreate(Sender: TObject);
  property OnDestroy(Sender: TObject);
  ```

  Nach dem Erstellen des Formulars wird das Ereignis *OnCreate*, vor dem Zerstören das Ereignis *OnDestroy* ausgelöst.

  Verwenden Sie *OnCreate*, um Objekte zu erzeugen, die vom Formular benötigt werden, beispielsweise Stringlisten oder Bitmaps, aber auch Komponenten, die Sie nicht mit dem Designer einfügen möchten.

  Um solche Objekte wieder freizugeben, verwenden Sie das Ereignis *OnDestroy*. Beachten Sie, dass Sie Komponenten nicht selbst freigeben müssen, solange Sie die Eigenschaft *Owner* entsprechend gesetzt haben. (*Owner* wird als Parameter dem Konstruktor *Create* übergeben.)

- Print, GetFormImage (Methoden), PrintScale (Eigenschaft, veröffentlicht)

  ```
  procedure Print;
  function GetFormImage: TBitmap;
  property PrintScale: (poNone, poProportional, poPrintToFit)
    default poProportional;
  ```

  Mit *Print* wird ein Bild des Formulars ausgedruckt. Die Skalierung kann mit *PrintScale* eingestellt werden, mehr dazu in der Online-Hilfe.

  Mit *GetFormImage* kann ein Bild des Formulars als *Bitmap* angefordert werden.