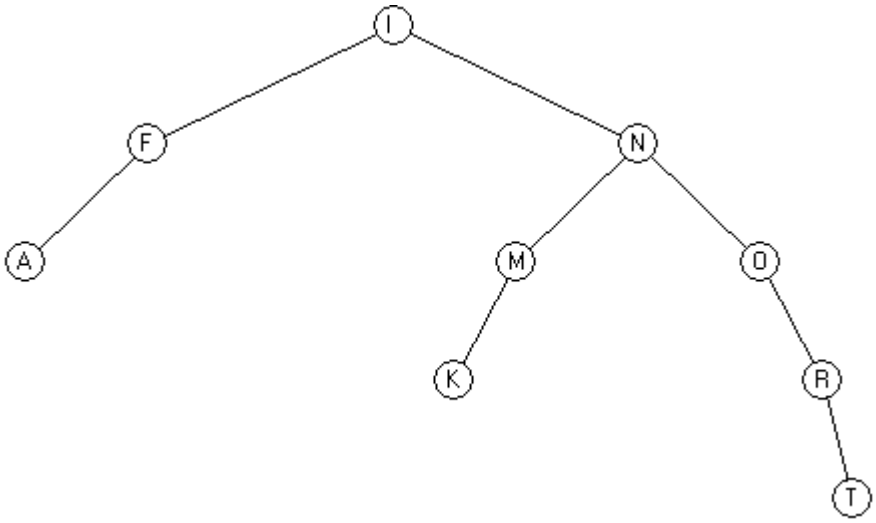


Dynamische

Strukturen

objektorientierte Programmierung

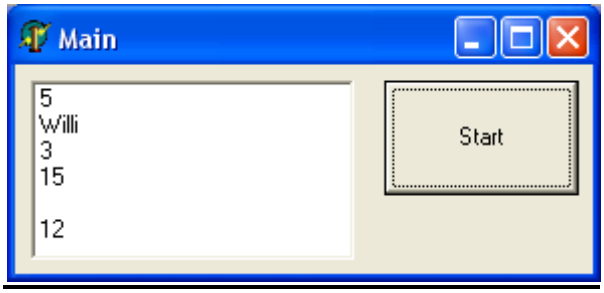
Version 2012

Autor: Dieter Lindenberg

Inhaltsverzeichnis

Objekt-Variablen

Das folgende kleine Demoprogramm zeigt den Unterschied zwischen „normalen“ Variablen und Objektvariablen.




```
unit mHaupt;
.....
type
  Testklasse = class(TObject)
    public
      Inhalt: INTEGER;
  end;

  TMain = class(TForm)
.....
  end;

var
  Main: TMain;
  Test1, Test2: Testklasse;
  x,y : INTEGER;
  wort1, wort2: STRING;
  A, B: ARRAY[1..3] of INTEGER;


implementation
{$R *.dfm}
```

```
procedure TMain.BtStartClick(Sender: TObject);
begin
  x := 5;
  y := x;
  x := 1;
  Listbox1.Items.add(IntToStr(y));

  wort1 := 'Willi';
  wort2 := wort1;
  wort1 := 'Anton';
  Listbox1.Items.add(wort2);

  A[1]:=3; A[2]:=3; A[3]:=3;
  B := A;
  A[1]:=5; A[2]:=5; A[3]:=5;
  Listbox1.Items.add(IntToStr(B[1]));

  Test1 := Testklasse.Create;
  Test1.Inhalt := 30;
  //Test2 := Testklasse.Create;
  Test2 := Test1;
  Test1.Inhalt := 15;
  Listbox1.Items.Add(IntToStr(Test2.Inhalt));

  Listbox1.Items.Add('');

  Showmessage('Achtung unsinnige Ausgabe!');
  Test1.Destroy;
  Listbox1.Items.Add(IntToStr(Test1.Inhalt));

  Showmessage('Achtung Absturz!');
  Test2.Inhalt := 10
end;

end.
```

wichtiger Programmierhinweis :
Sei X eine Objektvariable. Dann wird der Befehl *X.destroy* den für den Inhalt von X reservierten Speicherbereich wieder freigeben. Wird dieser Befehl direkt anschließend noch einmal gegeben, d.h. soll ein bereits freier Speicherbereich ein zweites Mal freigegeben werden, so stürzt das Programm ab! Dieser Programmierfehler kommt leider recht häufig vor!

Im obigen Beispielprogramm wird an einer Stelle versucht, in einem bereits freigegebenen Speicherbereich etwas zu lesen. Leider stürzt das Programm dabei nicht ab. Allerdings liest es völlig unsinnige Werte. Das sind eigentlich die schlimmsten Fehler, die passieren können. Das Delphi-System meldet nicht den unerlaubten Leseversuch, sondern gibt irgendeine harmlose, aber völlig falsche Zahl als Ergebnis zurück. Das ist natürlich auch ein gravierender Fehler des Delphi-Systems (Delphi 6 im Jahre 2010).

Im obigen Programm wird direkt nach diesem unerlaubten Leseversuch ein ebenso unerlaubter Schreibversuch im bereits freigegebenen Speicherbereich durchgeführt. Glücklicherweise stürzt das Programm dabei ab.

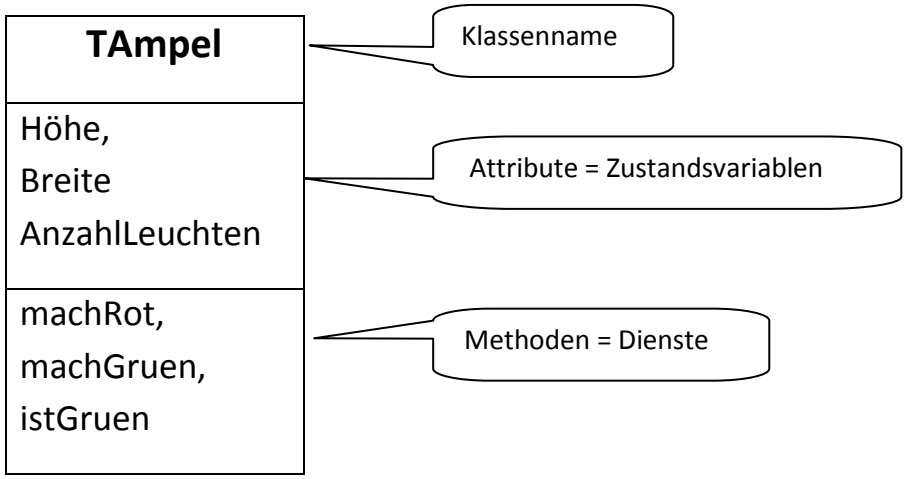
Klassen und Objekte

Grundlagen

Wir werden in diesem Kurs Programme erstellen, die mehrere Units und teilweise sogar mehrere Formblätter enthalten. Außerdem werden die Programme durchaus ziemlich umfangreich sein. Damit alle Beteiligten zusammenarbeiten können, sind u.a. auch gemeinsame Regeln für Namensgebungen notwendig.

Das Hauptformblatt lautet ***Main***, falls kein besserer Name gefunden wird. Die zugehörige ***Unit***, bzw. das zugehörige ***Modul***, wird unter dem Namen ***mMain.pas*** gespeichert. Alle *Unitnamen* beginnen mit dem Kleinbuchstaben *m*. Der Quelltext für das Hauptprogramm erhält automatisch den Namen der gespeicherten Datei. Projekte werden unter dem Namen ***pMain.dpr*** gespeichert.

Eine Klasse wird durch ein sog. UML-Klassendiagramm (**U**nified **M**odelling **L**anguage) dargestellt. Dieses ist dreigeteilt und enthält den Namen, die Attribute und die Dienste der Klasse. UML-Diagramme enthalten keine Typbezeichnungen (weil diese von der gewählten Programmiersprache abhängig wären).



Eine Klasse besitzt Attribute und Methoden (ein anderer Name für *Methoden* ist *Dienste*). Ein Objekt einer Klasse wird auch Instanz genannt.

Attribute sind Variablen, die zu jedem Objekt einer Klasse gehören. Verschiedene Objekte derselben Klasse haben üblicherweise unterschiedliche Attributwerte. Aus diesem Grund nennt man die Attribute auch *Zustandsvariablen*.

Eine Methode ist eine Prozedur oder Funktion, die zu einer Klasse gehört und von allen Klassenobjekten gemeinsam genutzt werden kann.

In Delphi gibt es sehr viele schon vordefinierte Klassen, z.B. *TForm, TEdit, TButton, TImage, TTimer*. In diesem Kurs werden wir eigene neue Klassen definieren. Alle Klassennamen sollten mit dem Buchstaben T (für Type) beginnen.

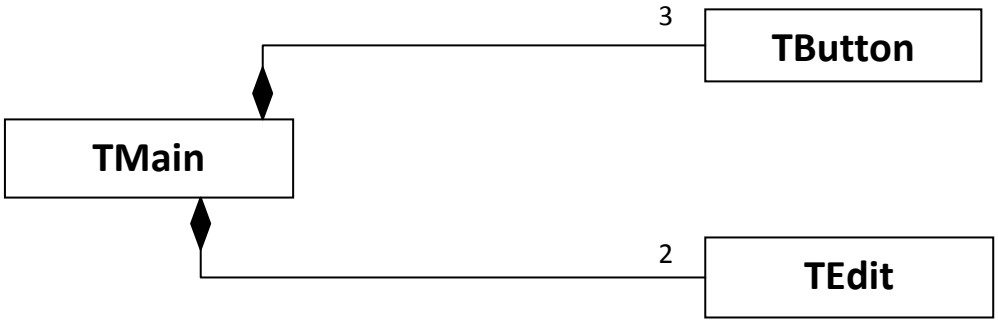
Die Namen der von Delphi zur Verfügung gestellten Objekte sollen alle mit einem oder mehreren kennzeichnenden Buchstaben beginnen:
EdName: Editfeld, RbName: Radiobutton, BtName: Button, LbName: Listbox, ImName: Bilder, MemName: Memofeld, LName: Label
Alle gewählten Namen müssen sinnvoll sein.

Durch die Existenz des Namens einer Objektvariablen gibt es aber das Objekt selbst noch nicht. Dieses wird entweder durch die Delphi-Programmier-umgebung schon aufgrund der Gestaltung des Formblattes automatisch erzeugt, oder aber erst im laufenden Programm durch Aufruf eines entsprechenden Konstruktorbefehls:

```
procedure Beispiel;
VAR A: TAmpel;
BEGIN
  A := TAmpel.create;
  IF A.istGruen THEN ….……
END;
```

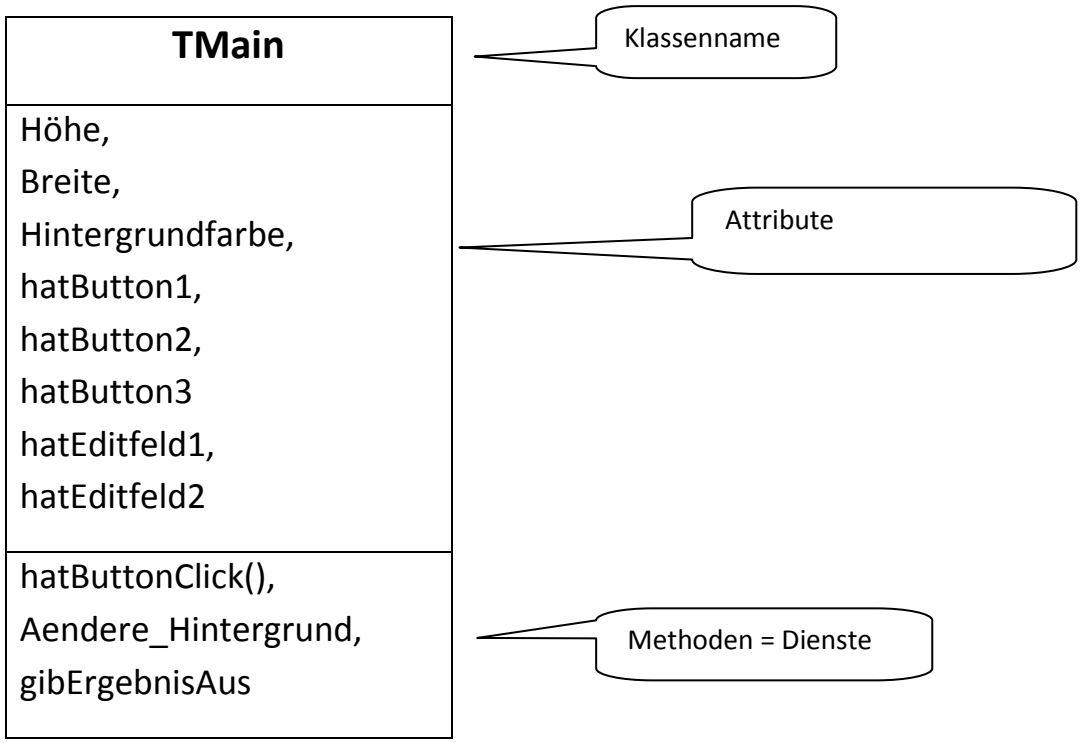
Sehr oft „haben“ Objekte einer Klasse wiederum andere Objekte. Zum Beispiel besitzt üblicherweise unser Hauptformblatt ein oder mehrere Objekte vom Typ *TButton, TEdit, TLabel, TTimer* usw.

Diese sog. ***Hat-Beziehung*** wird dann so dargestellt:



Oft werden, wenn die Beziehungen zwischen mehreren Klassen dargestellt werden sollen, die jeweiligen Attribute und Dienste nicht nochmal dargestellt.

Innerhalb einer einzigen Klasse wird diese Hat-Beziehung durch ein oder mehrere entsprechende Attribute realisiert:



Schutzklassen

In größeren Programmen werden Teilmodule üblicherweise von vielen unterschiedlichen Programmierern erstellt. Ein fertig gestelltes Modul wird anschließend allen anderen Programmierern, die es gebrauchen könnten, zur Verfügung gestellt. Dabei darf es natürlich nicht erlaubt sein, dass jeder Programmierer für seine Zwecke kleine Änderungen an diesem Modul vornimmt. Ansonsten gäbe es in kürzester Zeit viele unterschiedliche Versionen dieses Moduls und niemand wüsste mehr, was dieses Modul denn nun wirklich macht.

Der Ersteller des Moduls bestimmt nun, welche Methoden des Moduls **benutzt** werden dürfen (durch Aufruf der entsprechenden Dienste). Diese Methoden werden unter der Schutzklasse ***public*** eingetragen. Ein Benutzer dieses Moduls (üblicherweise ein anderer Programmierer) kann nur diese Methoden aufrufen. An deren Programmierung kann er auch nichts ändern, weil er natürlich nur (im Gegensatz zu unserem Unterricht!) den bereits compilierten Code und selbstverständlich nicht den Delphi-Quelltext erhält.

Davon abgesehen benötigt das Modul auch Prozeduren, Funktionen und Variablen, die von einem anderen Programmierer nicht benutzt werden dürfen, bzw. von deren Existenz ein anderer Programmierer überhaupt nichts wissen muss (weil es ihn normalerweise auch nicht interessiert, wie das Modul intern funktioniert). Derartige Attribute und Methoden werden unter der Schutzklasse ***private*** eingetragen.

Im Prinzip sollten alle Attribute der Schutzklasse *private* (oder der Schutzklasse *protected*) angehören.
Beispiel: Angenommen, man hätte eine Klasse *TRechteck* mit den Attributen *Länge, Breite* und *Flächeninhalt*. Wenn jemand willkürlich nur das Attribut *Länge* ändern würde, so hätte dies Auswirkungen auf das Attribut *Flächeninhalt*. Eine mögliche Änderung des Attributes *Länge* darf deshalb nur über eine öffentlich zur Verfügung gestellte Methode *setzeLaenge(n:INTEGER)* vorgenommen werden. Diese vom Ersteller der Klasse *TRechteck* implementierte Methode berücksichtigt gleichzeitige Änderungen des Attributes *Flächeninhalt*.

Oft möchte man von gegebenen Klassen weitere Unterklassen erstellen, welche größtenteils der Originalklasse ähneln aber noch zusätzliche oder auch weniger Attribute und Methoden haben als die Oberklasse. Für diesen Fall gibt es noch die Schutzklasse ***protected***. Alle hier deklarierten Attribute und Methoden sind auch in allen abgeleiteten Unterklassen sichtbar.

Unter *private* deklarierte Attribute und Methoden werden auch nicht an Unterklassen vererbt.

Das sinnvolle Konzept der Schutzklassen *private* und *public* (und *protected*) ist in Delphi leider nicht konsequent umgesetzt worden. Der Grund dafür liegt in der beabsichtigten, möglichen Kompatibilität bzw. Übertragbarkeit von und zu den älteren Pascal-Programmen.

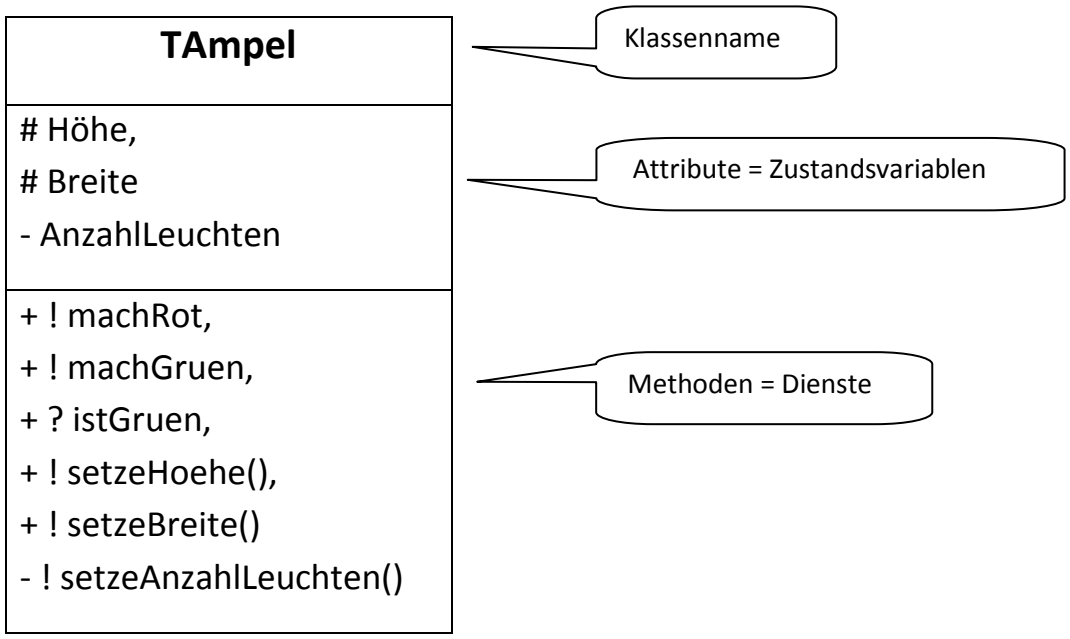
Leider gilt deshalb folgende Einschränkung in Delphi:
Innerhalb einer Unit kann man immer auf alle, insbesondere auch auf private Bestandteile einer Klasse zugreifen, auch wenn sich diese in einer anderen Klasse (in derselben Unit) befinden.
Die Schutzklassen private bzw. protected schützen also immer nur dann, wenn versucht wird, auf Attribute oder Methoden einer Klasse zuzugreifen, die in einer anderen Unit definiert ist.

Benutzt man die Delphi-Programmierumgebung, so werden alle automatisch erzeugten Attribute und Methoden oberhalb des ***private***-Teils eingetragen, also scheinbar außerhalb der Schutzklassen. Dies ist eine Besonderheit der Programmierumgebung von Delphi. Aber alle diese automatisch vorgenommenen Eintragungen sind leider öffentlich (Schutzklasse *public*).
Es stellt sich natürlich die Frage, warum diese Eintragungen nicht unter *public* gemacht werden. Das liegt daran, dass Delphi keine rein objektorientierte Sprache ist, sondern eine Erweiterung der Sprache Pascal mit der Möglichkeit, auch objektorientiert programmieren zu können.

Im Klassendiagramm wird die Schutzklasse hinzugefügt. Dabei gilt:
„+“ = public, “–“ = private, „#“=protected

Zusätzlich unterscheidet man bei den Diensten zwischen Aufträgen (in Delphi: Prozeduren), gekennzeichnet durch ein Ausrufungszeichen und Anfragen (in Delphi: Funktionen), die mit einem Fragezeichen gekennzeichnet werden.

Übergabeparameter werden oft nur durch zwei Klammern angedeutet.



Wenn man Objekte erzeugen will, so muß man dafür sorgen, dass diejenigen Module (Units), in denen die zugehörigen Klassen definiert sind, unter USES eingebunden werden

Statische Methoden

Methoden sind standardmäßig statisch. Statische Methoden werden vom Compiler direkt übersetzt. Sämtliche **statische** Methoden vom Typ *public* und *protected* werden von Unterklassen **völlig identisch** übernommen. *Private* Methoden werden nicht vererbt!

Beim Aufruf einer statischen Methode bestimmt der zur Compilerzeit durch die Deklaration festgelegte Typ der im Aufruf verwendeten Variablen, welche Implementierung (falls es 2 statische Methoden desselben Namens in Unter- und Oberklasse gibt) verwendet wird.

Am deutlichsten wird dies an Hand eines Beispiels:

```
type
  TFigur = class(TObject)
    procedure zeichne;
  end;

  TRechteck = class(TFigur)
    procedure zeichne;
  end;
```

Das wären also die beiden Klassen. Was geschieht nun bei Aufrufen:

```
procedure TForm1.Button1Click(Sender: TObject);
VAR  figur: TFigur;
         rechteck: TRechteck;
BEGIN
  figur := TFigur.create;
  figur.zeichne;    // ruft TFigur.zeichne auf

  figur := TRechteck.create;
  figur.zeichne;    // ruft TFigur.zeichne auf !!!
  // weil figur als TFigur deklariert ist
  figur := TFigur.create;
  TRechteck(figur).zeichne;    // ruft TRechteck.zeichne auf
  (figur as TRechteck).zeichne;    // äquivalenter Befehl
  rechteck := TRechteck.create;
  rechteck.zeichne   // ruft natürlich TRechteck.zeichne auf
END;
```

Virtuelle Methoden

Es gibt einige sehr wenige Klassenmethoden, die aufgerufen werden können, ohne dass ein Objekt der Klasse existiert. Die wichtigste davon ist die sog. Konstruktormethode ***create***. Mit ihr wird ein Objekt erzeugt. Logischerweise muss man sie also schon vor der Existenz eines Objektes aufrufen können.

Wird in einer Klasse kein Konstruktor implementiert, so wird der Konstruktor der übergeordneten Klasse aufgerufen. Alle Klassen sind Unterklassen von der obersten Delphiklasse namens *TObject*. Obwohl die Deklaration keinen Rückgabewert enthält, gibt ein Konstruktor immer einen Verweis (einen sog. Zeiger) auf das Objekt, das er erstellt, zurück.

Der Konstruktor reserviert zunächst Speicherplatz für das neue Objekt. Anschließend werden alle Attribute initialisiert. Ordinalvariablen (Integer, Real usw.) werden mit dem Wert Null, alle Zeiger mit NIL und alle Strings mit einem Leerstring initialisiert. Aus diesem Grund braucht der Programmierer in dieser Create-Methode nur noch denjenigen Attributen einen Wert zuweisen, die einen bestimmten Anfangswert haben sollen.

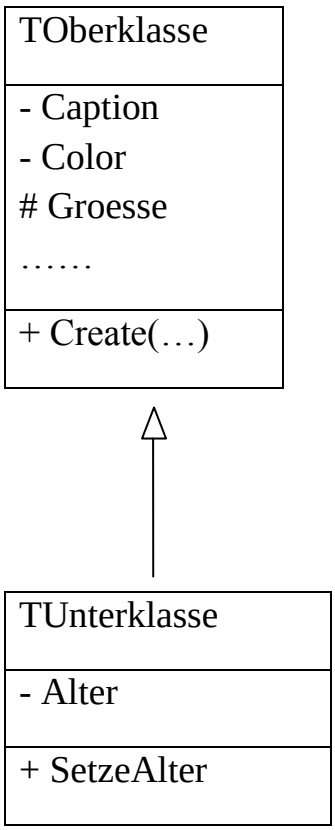
Wenn man in der Unterklasse eine Methode (neu) definiert, die denselben Namen wie eine Methode der Oberklasse besitzt, so verdeckt die neue Methode in der Unterklasse die alte Methode aus der Oberklasse (ähnlich verhält es sich z.B. mit lokalen und globalen Variablen gleichen Namens in Prozeduren). Üblicherweise gibt es dann beim Compilieren eine entsprechende Warnung:
[Warning] Method 'Create' hides virtual method of base type'
Auf die Methode der Oberklasse lässt sich allerdings immer noch mit dem Stichwort ***inherited*** zugreifen.

Das Erzeugen dieser Warnung lässt sich allerdings unterdrücken, wenn man hinter dem Namen der neu definierten Methode den Zusatz ***reintroduce;*** angibt.
Beispiel: *procedure Methodenname(.....); reintroduce; virtual;*

Oft will man jedoch bewußt die alte Methode der Oberklasse überschreiben (also nicht nur verdecken). Dies ist möglich, indem man der Methode in der Unterklasse das Attribut ***override*** gibt. Man unterscheidet zwischen statischen und virtuellen Methoden. Ist schon bei der Implementierung einer Klasse

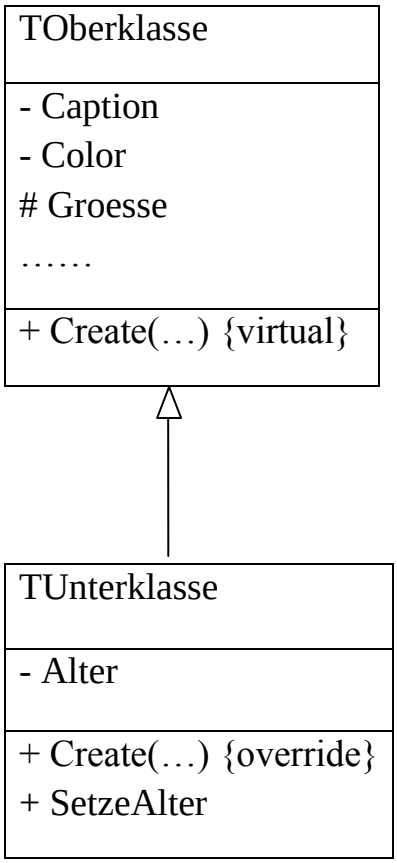
abzusehen, dass eine Methode in späteren Unterklassen geändert oder ganz ersetzt werden wird, so gibt man dieser Methode in der Oberklasse die Eigenschaft ***virtual***. Die entsprechende Methode in der Unterklasse erhält dann die zusätzliche Eigenschaft ***override***. Methoden mit der Eigenschaft *override* ersetzen praktisch die entsprechende Methode der Oberklasse. Override-Methoden können in weiteren, tiefer liegenden Unterklassen dort wiederum mit *override* ersetzt werden. Mit *override* kann man nur Methoden ersetzen, die in der Oberklasse das Attribut *virtual* oder *override* besitzen.

Im Folgenden betrachten wir nun die beiden Klassen *TOberklasse* und *TUnterklasse*. Die Unterklasse leitet sich von der Oberklasse ab. Sie beinhaltet noch eine (*private*) Eigenschaft *Alter: INTEGER*, die sich also nicht noch weiter vererben kann.



Private Eigenschaften und Methoden der Oberklasse werden **nicht** auf die Unterklasse vererbt. Jedes Objekt der Unterklasse besitzt also nur diejenigen Eigenschaften und Methoden der Oberklasse, die entweder *public* oder *protected* sind (beachte oben genannte Einschränkung in Delphi).
Zusätzlich gibt es aber noch weitere, nur in der Unterklasse definierte Eigenschaften und Methoden.

Falls die Objekte der Unterklasse nur **zusätzliche** Attribute haben sollen (und nicht schon bei der Erzeugung andere Attributswerte als ein Objekt der Oberklasse), so genügt derselbe Konstruktor. Das Delphisystem hängt die zusätzlichen Attribute und Methoden automatisch bei der Erzeugung mit an.



Falls die Objekte der Unterklasse schon bei der Erzeugung andere Attreibutswerte haben sollen, so muss man den Konstruktor *Create(...)* der Oberklasse überschreiben.

Dies geschieht dadurch, dass man in der Unterklasse hinter der Deklaration des Konstruktors *Create(...)* den Zusatz *override* angibt. Der Konstruktor der Oberklasse lässt sich allerdings nur überschreiben, wenn bei dessen Deklaration der Zusatz *virtual* angegeben wird.

Die Benutzung von *override* setzt voraus, dass die Methode in der Unterklasse exakt dieselben Parameter besitzt wie die Methode in der Oberklasse.
Benötigt man in der Unterklasse mehr oder andere Parameter, so muß man die entsprechende Methode auch in der Unterklasse als *virtual* deklarieren!

Virtuelle Methoden der Oberklasse, die in der Unterklasse überschrieben werden (die also in der Unterklasse hinter dem Namen den Zusatz *override* haben), bleiben in der Unterklasse virtuelle Methoden. Sie können also von (in der Ableitungshierarchie) noch tiefer liegenden Unterklassen wiederum überschrieben werden!

Interessanterweise lässt sich in der Unterklasse auch dann noch auf den ererbten Konstruktor zugreifen, wenn man diesen mit *override* überschrieben hat. Das liegt daran, dass jede Klasse intern auch einen Zeiger besitzt, der auf die direkte Oberklasse verweist; und dort kann man auf die vererbte Methode zugreifen.

Eine typische Create-Methode sieht so aus:

```
constructor TUnterklasse.Create;   //override
BEGIN
  Inherited Create;
  Alter := 70
END;
```

Das Schlüsselwort *inherited* ist von großer Bedeutung. Folgt auf *inherited* ein Name, so entspricht dies einem normalen Methodenaufruf. Der einzige Unterschied besteht darin, dass die Suche nach der entsprechenden Methode bei der direkten Oberklasse beginnt und gegebenenfalls in noch höheren Oberklassen fortgesetzt wird.

Die Anweisung *inherited* ohne zusätzlichen Namen verweist auf die geerbte Methode mit demselben Namen wie die aufrufende Methode, in welcher dieser Befehl steht. Haben die Oberklassen keine Methode mit demselben Namen, ignoriert Delphi den Aufruf von *inherited*.

Virtuelle Methoden-Tabelle (VMT)

Für **jede** im Programm benutzte Klasse wird eine Tabelle der virtuellen Methoden angelegt. Diese Tabelle ordnet den Methodennamen die zugehörige Speicheradresse zu. Außerdem werden die virtuellen Methoden durchnummeriert, sodass man mithilfe eines Indexes auf sie zugreifen kann.

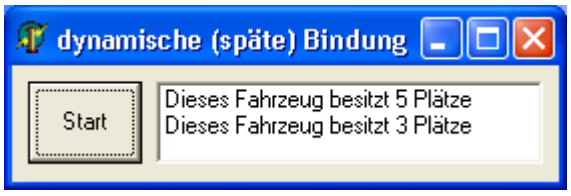
Diese ***VMT*** enthält zunächst sämtliche von allen Oberklassen übernommenen virtuellen Methoden und zusätzlich deren Adressen. Falls eine dieser virtuellen Methoden mit *override* überschrieben wurde, so enthält die VMT die dadurch aktualisierte Adresse. Das bedeutet, dass die Indizes (nicht die Adressen!) aller virtuellen Methoden, die schon in den Oberklassen existieren, in Ober- und Unterklasse übereinstimmen. Erst danach, am Schluß der Tabelle, werden in der aktuellen Unterklasse neu hinzukommende virtuelle Methoden angehängt.

Beim Aufruf einer virtuellen Methode bestimmt nicht der bei der Compilierung deklarierte Typ einer Variablen (wie bei statischen Methoden) sondern der zur Laufzeit aktuelle Typ der im Aufruf verwendeten Variablen, welche der beiden virtuellen Methoden (in Ober- oder Unterklasse) aktiviert wird.

Beispiel:
Vu sei ein Objekt der Unterklasse, *statMeth* sei eine statische Methode (vom Typ *public* oder *protected*) der Oberklasse (die also in der Unterklasse geerbt wurde), welche u.a. die virtuelle Methode mit dem Index 17 aufruft. Dann wird durch die Anweisung *Vu.statMeth* die in der *VMT* der Unterklasse stehende Methode mit dem Index 17 gesucht und ausgeführt. DiesesVerhalten nennt man auch ***Polymorphie***.

Sollte in der Unterklasse hinter einem Methodennamen nicht *override* sondern wiederum *virtual* stehen, so wird in der VMT eine zweite gleichnamige Methode angehängt (die dann natürlich einen anderen Index erhält).

Welche virtuelle Methode nun gegebenenfalls ausgeführt wird, hängt in jedem Fall davon ab, ob der Methodenname von einem Objekt der Ober- oder von einem Objekt der Unterklasse aufgerufen wird:




```
unit mMain;

.........
type
  TMain = class(TForm)
    BtStart: TButton;
    ListBox1: TListBox;
    procedure BtStartClick(Sender: TObject);
  end;

  TAuto = class(TObject)
    public
      procedure InformationsBereitstellung;
      function SitzplatzAnzahl: INTEGER; virtual;
  end;

  TLKW = class(TAuto)
    public
      function SitzplatzAnzahl: INTEGER; override;
  end;

var  Main: TMain;
     Auto: TAuto;
     LKW: TLKW;

Implementation.......
{$R *.dfm}

procedure TAuto.InformationsBereitstellung;
BEGIN
  Main.ListBox1.Items.Add('Dieses Fahrzeug besitzt '
             + IntToStr(Sitzplatzanzahl) + ' Plätze')
  // nur möglich, weil alle Klassen in derselben Unit stehen
END;
```

```
function TAuto.SitzplatzAnzahl: INTEGER;
BEGIN
  RESULT := 5
END;


function TLKW.SitzplatzAnzahl: INTEGER;
BEGIN
  RESULT := 3
END;

procedure TMain.BtStartClick(Sender: TObject);
begin
  Auto := TAuto.Create;
  LKW := TLKW.Create;
  Auto.InformationsBereitstellung;
  LKW.InformationsBereitstellung
end;

end.
```

Beachte: Die statische Methode *Informationsbereitstellung* ist in der Unterklasse *TLKW* identisch mit der gleichnamigen Methode *Informationsbereitstellung* in der Oberklasse *TAuto*.
Wenn allerdings innerhalb dieser statischen Methode *Informationsbereitstellung* die virtuelle Hilfsmethode *Sitzplatzanzahl* aufgerufen wird, dann wird erst geprüft, welcher Objekttyp gerade vorliegt. In Abhängigkeit vom Ergebnis dieser Prüfung wird entschieden, ob die Hilfsmethode *Sitzplatzanzahl* aus der Ober- oder aus der Unterklasse verwendet wird.

Der Delphi-Quelltext wird also so compiliert, dass erst zur Laufzeit des Programms die richtige virtuelle Methodenauswahl getroffen wird. Diese Programm- oder Compilereigenschaft nennt man ***dynamische*** oder ***späte Bindung***.

Erst zur Laufzeit des Programms wird geprüft, welcher Objekttyp vorliegt und in Abhängigkeit von dieser Prüfung wird entschieden, welche der beiden gleichnamigen Methoden ausgeführt wird.
Diese Prüfung findet nur für virtuelle und überschriebene (override) Methoden statt. Das Ergebnis dieser Prüfung ist übrigens nicht abhängig von der Deklaration der Typvariablen (bekanntlich kann eine Variable der Oberklasse auch ein Objekt einer Unterklasse beinhalten).
Zur Erinnerung: bei statischen Methoden wird nicht geprüft. Dort wird grundsätzlich die zum Deklarationstyp gehörende Methode ausgeführt.

Aus der Sicht des Programmbenutzers zeigen die verschiedenen Objekte ein ***polymorphes*** (=vielgestaltiges) Verhalten: Objekte verschiedener Klassen reagieren unterschiedlich auf identische Befehle (*Informationsbereitstellung*).

Hinweis: Man kann einer Variablen vom Typ einer bestimmten Klasse beliebige Objekte von abgeleiteten Klassen zuordnen. Das Umgekehrte geht nicht!
Von dieser Möglichkeit werden wir im Schulunterricht im Allgemeinen jedoch keinen Gebrauch machen.

Interessant ist jedenfalls folgende zusätzliche Prozedur im Hauptprogramm:

```
procedure TMain.Zufall;
VAR Karre: TAuto;
    zahl: INTEGER;
begin
  zahl := RANDOM(2);
  CASE zahl of
    0: Karre := TAuto.create;
    1: Karre := TLKW.create
  END;
  IF Karre is TLKW
    THEN Showmessage('ich bin ein LKW')
  ELSE Showmessage('ich bin ein normales Auto');
  Karre.InformationsBereitstellung
end;
```

Methoden überladen

Man kann auch Methoden mehrfach deklarieren mit jeweils unterschiedlichen Parametern. Öfters benutzt wird dies bei der Create-Methode, aber es funktioniert bei allen Methoden, unabhängig davon, ob sie statisch oder virtuell sind:

```
Type
  T1 = class(TObject)
    ……..
    procedure Test(n: INTEGER); overload;
    procedure Test(s: STRING); overload;
    procedure Test(x: REAL; n: INTEGER); overload;
    …….
  end;
```

Der Compiler benutzt den Typ und die Anzahl der Parameter, um zu entscheiden, welche überladene Methode aufgerufen werden soll.

Alle in **einer** Klasse überladene Methoden müssen das Attribut ***overload*** haben. Wenn man eine Methode der Oberklasse in der Unterklasse überladen will, so ist es nicht notwendig, dass die Methode in der Oberklasse auch das Attribut *overload* erhält.

Das Schlüsselwort *overload* muss gegebenenfalls vor den Schlüsselworten *virtual* oder *override* stehen.

Lineare Strukturen

Die Klasse *TQueue*

Objekte der Klasse *TQueue* (Schlange) verwalten beliebige Objekte nach dem ***FIFO***-Prinzip (First-In-First-Out), d.h. das zuerst abgelegte Element wird als erstes wieder entnommen.

Die Syntax der Methoden der Klasse *TQueue* wurde im Jahre 2007 zumindest für alle Schulen in NRW gemeinsam folgendermaßen festgelegt:

```
constructor create
function isEmpty: boolean
procedure enqueue(pObject: TObject)
procedure dequeue
function front: TObject
destructor destroy
```

Der Typ *TQueue* ist im Prinzip eine Liste von Objekten. Es können zum Beispiel mehrere Listen existieren, die teilweise dieselben Objekte enthalten. Beispiel: Eine erste Liste der noch nicht volljährigen, und eine zweite Liste aller männlichen Schüler in der JgSt. 12. Wenn man hier ein Objekt aus der ersten Liste entfernt, darf das natürlich keinen Einfluss auf die zweite Liste haben. Aus diesem Grund sollte bei der Methode *dequeue* das Element zwar aus der Liste, aber nicht aus dem Speicher gelöscht werden.

Dokumentation der Methoden der Klasse TQueue

Konstruktor	**create**
Nachher	Eine leere Schlange ist erzeugt.
Anfrage	**isEmpty: boolean**
Nachher	Die Anfrage liefert den Wert true, wenn die Schlange keine Elemente enthält, sonst liefert sie den Wert false.
Auftrag	**enqueue(pObject: TObject)**
Vorher	Die Schlange ist erzeugt.
Nachher	pObject ist als letztes Element in der Schlange abgelegt.
Auftrag	**dequeue**
Vorher	Die Schlange ist nicht leer.
Nachher	Das vorderste Element ist aus der Schlange entfernt (aber nicht aus dem Speicher gelöscht!)
Anfrage	**front: TObject**
Vorher	Die Schlange ist nicht leer.
Nachher	Die Anfrage liefert das vorderste Element der Schlange. Die Schlange ist unverändert.
Destruktor	**destroy**
Nachher	Die Schlange existiert nicht mehr.

Bemerkungen:

Bei der nun folgenden ersten Version der Klasse *TQueue* wird die Schlange aus Vereinfachungsgründen noch keine beliebigen, allgemeinen Objekte, sondern nur spezielle Objekte verwalten.
Außerdem können diese speziellen Objekte noch nicht in mehreren Schlangen enthalten sein. Beide Nachteile werden in der zweiten Version behoben.

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

Fehlt in einer Klasse eine eigene Methode *constructor*, so wird automatisch der Konstruktor der Oberklasse ausgeführt. Es wird also eine Instanz der Oberklasse erzeugt und die zusätzlichen Attribute der eigenen Klasse werden dieser Instanz angefügt. Damit wird also ein Element der aktuellen Klasse erzeugt.

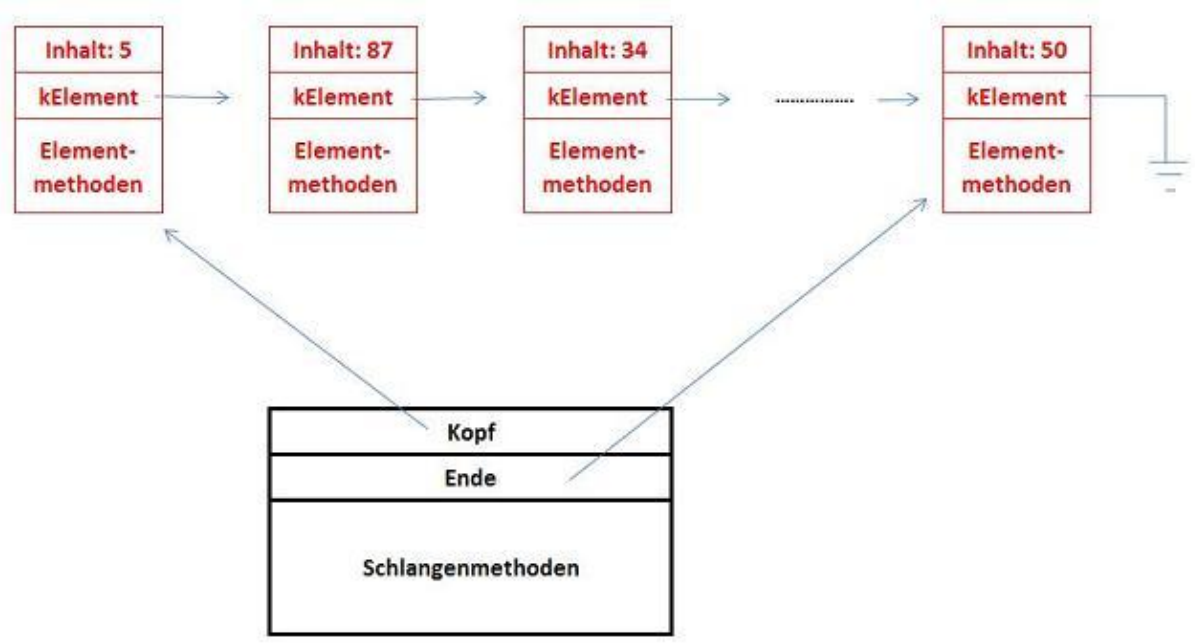
Beim Erzeugen eines Klassenobjektes belegt die Delphi-Programmierumgebung automatisch die jeweils vorhandenen Attribute mit folgenden Startwerten:

- Alle Datenfelder mit einem ganzzahligen Datentyp (z.B. Integer) werden mit 0 initialisiert. Wichtig: Dies geschieht nicht bei normalen INTEGER-Variablen außerhalb eines Klassenobjektes!
- Alle Datenfelder mit einem String-Typ werden durch eine leere Zeichenkette initialisiert.
- Alle Datenfelder mit einem Zeigertyp werden mit dem Wert *NIL* initialisiert.
- Alle anderen Datenfelder bleiben undefiniert!

Analoges gilt für die Methode *destroy*. Wird diese in der aktuellen Unterklasse nicht extra implementiert, so wird die entsprechende Methode der Oberklasse ausgeführt.

Die Klasse *TQueue*, Version 1

Obwohl unsere Schlange völlig beliebige Elemente enthalten kann (z.B. Bilder, Musikstücke, Texte), arbeiten wir zunächst mit einfachen Elementen, deren eigentlicher Inhalt nur eine Integer-Zahl ist. Allerdings kennt jedes Element seinen Nachfolger in der Schlange. Und weil es nur einen Nachfolger kennt, kann dieses Element auch nur in einer einzigen Schlange enthalten sein.



Nur das in der obigen Skizze schwarz Gezeichnete ist ein Objekt der Klasse *Schlange* bzw. *TQueue*. Jede Schlange belegt also nur sehr wenig Speicherplatz bzw. alle Schlangen belegen einen Speicherplatz gleicher Größe.
Man benötigt jeweils nur Speicherplatz für die Methoden der Schlange sowie für die zwei Variablennamen *Kopf* und *Ende*.
Allerdings benötigt man für jedes einzelne Element, welches zur Schlange gehört, entsprechend viel Speicherplatz.
Die Elemente der Schlange müssen untereinander selbst verkettet sein.
Die Schlange selbst hat nur Zugriff auf das erste und letzte Element, welches zur Schlange gehört.

```
unit mElement;

interface
Type
  TElement = class(TObject)
    private
     Inhalt: INTEGER;
     kElement: TElement;
    public
       // Beachte, dass hier ein eigener Konstruktor fehlt ! Deswegen wird
       // automatisch Inhalt mit 0 und kElement mit NIL initialisiert.
       procedure setInhalt(zahl: INTEGER);
       function getInhalt: INTEGER;
       procedure setNachfolger(pElement: TElement);
       function getNachfolger: TElement;
  End;


implementation

   procedure TElement.setInhalt(zahl:INTEGER);
    BEGIN
      Inhalt := zahl
    END;



   function TElement.getInhalt: INTEGER;
     BEGIN
       Result := Inhalt
     END;


   function TElement.getNachfolger: TElement;
   BEGIN
     Result := kElement;
   END;
```

```
    procedure TElement.setNachfolger(pElement:TElement);
    BEGIN
      kElement := pElement;
    END;

end.
```

Beachte, dass in der nun folgenden, noch vorläufigen Version 1 der Klasse *TQueue* die Methode *enqueue* als Parameter und die Methode *front* als Ergebnis noch keine allgemeinen Objekte enthalten!

```
unit mQueue;

interface
USES mElement;

Type
 TQueue = class(TObject)
   private
     kopf, ende: TElement;
   public
     // Beachte, dass hier ein Konstruktor fehlt ! Deshalb werden
     // automatisch kopf und ende mit NIL initialisiert.
     function isEmpty: boolean;
     procedure enqueue(pElement: TElement);
     procedure dequeue;
     function front: TElement;
 END;

implementation

function TQueue.isEmpty: boolean;
BEGIN
  Result := (kopf = NIL);
END;
```

```
procedure TQueue.enqueue(pElement: TElement);
BEGIN
  If kopf = NIL THEN BEGIN
     kopf := pElement;
     ende := pElement
  END
  ELSE BEGIN
        ende.setNachfolger(pElement);
        ende := pElement
       END
END;



procedure TQueue.dequeue;
BEGIN
  IF NOT isEmpty THEN BEGIN
    IF (kopf = ende) THEN BEGIN
      ende := NIL;
      kopf := NIL
    END
    ELSE kopf := kopf.getNachfolger;
  END
END;



function TQueue.front: TElement;
BEGIN
  Result := kopf
END;
```

Das folgende Hauptprogramm dient nur dazu, mit der Klasse *TQueue* etwas zu arbeiten. Die Befehle zur Heap-Belegung sind abhängig vom Betriebssystem.


```
unit mHaupt;
interface
uses  ….mElement, mQueue;

type
  TMain = class(TForm)
    BtElementErzeugen, BtSchlangenAusgabe: TButton;
    AusgabeBox: TListBox;
    BtHeapbelegung, BtLoeschen: TButton;
    Label1, Label2: TLabel;
    EdFrei, EdBelegt: TEdit;
    procedure BtElementErzeugenClick(Sender:Object);
    procedure FormCreate(Sender: TObject);
    procedure BtSchlangenAusgabeClick(Sender:Object);
    procedure BtHeapbelegungClick(Sender: TObject);
    procedure BtLoeschenClick(Sender: TObject);
  end;

var
  Main: TMain;
  Schlange: TQueue;

implementation
{$R *.dfm}
```

```
procedure TMain.BtElementErzeugenClick(Sender:Object);
VAR e: TElement;
begin
  e := TElement.create;
  e.setInhalt(random(1000));
  Schlange.enqueue(e);
  AusgabeBox.Items.Add(IntToStr(e.getInhalt))
end;


procedure TMain.FormCreate(Sender: TObject);
begin
  Schlange := TQueue.create;
  randomize
end;


procedure TMain.BtSchlangenAusgabeClick(Sender:Object);
begin
  IF Schlange <> NIL THEN
    While Not Schlange.isEmpty DO BEGIN

AusgabeBox.Items.Add(IntToStr(Schlange.front.getInhalt));
      Schlange.dequeue
    END
end;


procedure TMain.BtHeapbelegungClick(Sender: TObject);
VAR frei, belegt: Cardinal;
    HSt: THeapStatus;
begin
  HST := GetHeapStatus;
  frei := HSt.TotalFree;
  belegt := HSt.TotalAllocated;
  EdBelegt.Text := IntToStr(belegt);
  EdFrei.Text := IntToStr(frei)
end;
```

```
procedure TMain.BtLoeschenClick(Sender: TObject);
begin
  IF Schlange <> NIL THEN BEGIN
    Schlange.destroy;
    Schlange := NIL
  END
end;

end.
```

Aufgaben zur Struktur der Schlange

1. Obiges Programm enthält folgenden Fehler: Wenn man den Button *Schlange löschen* betätigt und anschließend den Button *Element erzeugen* anklickt, so stürzt das Programm ab. Beseitige den Fehler!

2. Erstelle für das obige Programm die beiden Klassendiagramme der Klassen *TElement* und *TQueue*!

3. Wie viel Speicherplatz wird frei, wenn man die Schlange löscht? Ist das abhängig davon, wie viele Elemente die Schlange enthält? Macht es Sinn, vor dem Löschen der Schlange alle Elemente aus ihr zu entfernen?

4. Wie viel zusätzlicher Speicherplatz wird belegt, wenn man ein weiteres Element an die Schlange anhängt? Untersuche den Speicherplatzbedarf für die Variable *kElement*, indem du z.B. eine Variable *kElement2* hinzufügst. Wie viel Speicherplatz belegt die Integervariable *Inhalt*?

5. Erweitere die Klasse *TQueue* durch ein weiteres Attribut namens *anzahl* und einer entsprechenden Methode *getAnzahl*, welche die Anzahl der Elemente der Schlange angibt. Natürlich müssen auch die Einfüge- und Lösch-methoden angepasst werden.

6. Erweitere die Klasse *TQueue* durch eine Methode namens *append(s: TQueue)*, welche an die bereits vorhandene Schlange die Schlange ***s*** anhängt.

Anwendungsaufgaben mit der Klasse TQueue

7. Im obigen Hauptprogramm wird leider bei der Ausgabe der Schlange in der Listbox die gesamte Schlange gelöscht. Ändere die Ausgabeprozedur so um, dass die Originalschlange gleichzeitig mit der Ausgabe zusätzliche in eine Hilfsschlange kopiert wird. Diese Hilfsschlange erhält anschließend den Namen der Originalschlange.

8. Das Hauptprogramm soll mit Hilfe einer Hilfsschlange die Anzahl der Elemente der Originalschlange feststellen und ausgeben können.

9. Simuliere folgendes Spiel: Ein Kartenspiel mit 32 Karten soll gemischt werden. Erzeuge dafür eine Originalschlange, welche als Elemente alle natürlichen Zahlen von 1 bis 32 enthält. Die Inhalte dieser Originalschlange werden nun nach dem Zufallsprinzip auf vier Hilfsschlangen verteilt. Die Originalschlange ist danach leer. Dann werden die vier Hilfsschlangen aneinander gehängt zu einer neuen Originalschlange. Das Ganze wird mehrmals gemacht. Gib anschließend die Originalschlange aus!

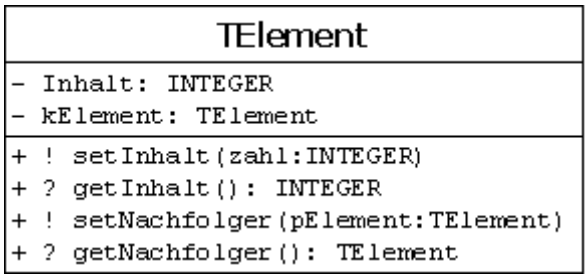
10. Zwei bereits sortierte Zahlenschlangen A und B sollen zu einer einzigen sortierten Schlange C verschmolzen werden.

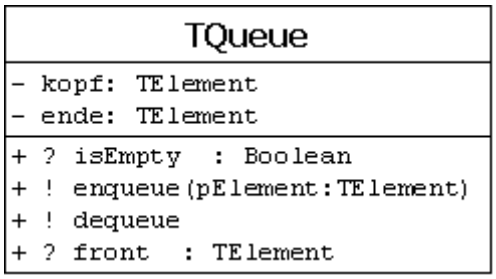
11. Eine Schlange enthält zufällige, ungeordnete, natürliche Zahlen. Mit Hilfe zweier Hilfsschlangen sollen die Zahlen so geordnet werden, dass anschließend wieder alle Zahlen in der Originalschlange stehen, aber alle geraden Zahlen (durchaus ungeordnet) am Anfang und alle ungeraden Zahlen am Schluß der Schlange.

12. Eine Schlange enthält 100 zufällige, ungeordnete, natürliche Zahlen, die alle kleiner als 10 000 sind. Mit Hilfe von zehn Hilfsschlangen Hilf0, Hilf1, Hilf2, ... , Hilf9 sollen die Zahlen folgendermaßen geordnet werden: Zuerst ordnet man alle Zahlen aufgrund ihrer Einerziffer in die zehn Hilfsschlangen ein. Danach werden alle Hilfsschlangen nacheinander wieder an die nun leere Originalschlange angehängt. Anschließend ordnet man aufgrund der Zehnerziffern usw.

Lösungen

Aufgabe 2

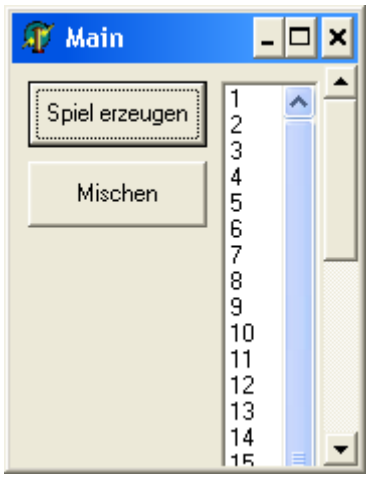




Aufgabe 6

```
procedure TQueue.append(s:TQueue);
BEGIN
  IF s <> NIL THEN BEGIN
    IF NOT self.isEmpty THEN //alternativ: if kopf <> NIL THEN ...
      ende.setNachfolger(s.front)
    ELSE BEGIN
      kopf := s.front;
      ende := kopf
    END;
    WHILE ende.getnachfolger <> NIL DO
        ende := ende.getnachfolger
  END // of  if s<>NIL
END;
```

Aufgabe 9

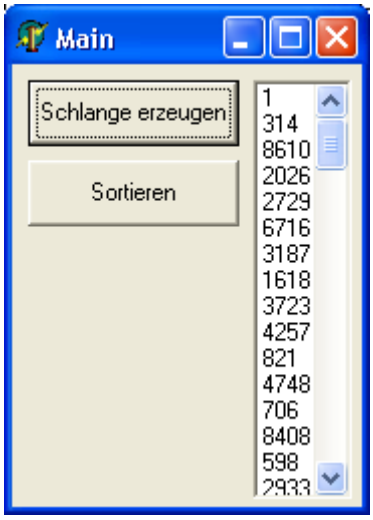



```
unit mHaupt;
interface
uses  ……, mQueue, mElement;

type  TMain = class(TForm)
           ……................
var  Main: TMain;
     Spiel, HilfA, HilfB, HilfC, HilfD: TQueue;

implementation
{$R *.dfm}

procedure TMain.BtSpielErzeugenClick(Sender: TObject);
VAR i: INTEGER;  e: TElement;
begin
  Spiel := TQueue.create;
  FOR i := 1 TO 32 DO BEGIN
    e := TElement.create;
    e.setInhalt(i);
    Spiel.enqueue(e)
  END;
  Ausgabe
end;
```

```
procedure TMain.Ausgabe;
VAR zahl: INTEGER;
    e: TElement;
    hilf: TQueue;
begin
  LbAusgabe.clear;
  hilf := TQueue.create;
  WHILE Not Spiel.isEmpty DO BEGIN
    e := Spiel.front;
    zahl := e.getInhalt;
    LbAusgabe.Items.Add(IntToStr(zahl));
    hilf.enqueue(e);
    Spiel.dequeue
  END;

  Spiel.destroy;
  Spiel:= hilf
end;


procedure TMain.BtMischenClick(Sender: TObject);
VAR zufall: INTEGER;
    e: TElement;
begin
  HilfA := TQueue.create;
  HilfB := TQueue.create;
  HilfC := TQueue.create;
  HilfD := TQueue.create;
  WHILE NOT Spiel.isEmpty DO BEGIN
    e := Spiel.front;
    zufall := Random(4)+1;
    CASE zufall of
      1: HilfA.enqueue(e);
      2: HilfB.enqueue(e);
      3: HilfC.enqueue(e);
      4: HilfD.enqueue(e)
    END;  // of CASE
    Spiel.dequeue;
```

```
  END;
  Anhaengen(HilfA);
  Anhaengen(HilfB);
  Anhaengen(HilfC);
  Anhaengen(HilfD);
  HilfA.destroy;
  HilfB.destroy;
  HilfC.destroy;
  HilfD.destroy;
  Ausgabe
end;


procedure TMain.Anhaengen(s: TQueue);
VAR e: TElement;
begin
  WHILE NOT s.isEmpty DO BEGIN
    e := s.front;
    Spiel.enqueue(e);
    s.dequeue;
  END
end;

end.
```

Aufgabe 12




```
unit mHaupt;
interface
uses ……..mQueue, mElement;

type
  TMain = class(TForm)
    BtSchlangeErzeugen: TButton;
    LbAusgabe: TListBox;
    BtSortieren: TButton;
    procedure BtSchlangeErzeugenClick(Sender: TObject);
    procedure Ausgabe;
    procedure BtSortierenClick(Sender: TObject);
    procedure Anhaengen(s:TQueue);
  end;

var  Main: TMain;

     Schlange, Hilf0, Hilf1, …. Hilf9: TQueue;
implementation
{$R *.dfm}

procedure TMain.BtSchlangeErzeugenClick(Sender:TObject);
…….
```

```
procedure TMain.Ausgabe;
……

procedure TMain.BtSortierenClick(Sender: TObject);
VAR Divisor, ziffer: INTEGER;
    e: TElement;
begin
  Hilf0 := TQueue.create;
  ……
  Hilf9 := TQueue.create;
  WHILE NOT Schlange.isEmpty DO BEGIN
    e := Schlange.front;
    ziffer := e.getInhalt MOD 10;
    CASE ziffer of
      0: Hilf0.enqueue(e);
      …….
      9: Hilf9.enqueue(e)
    END;  // of CASE
    Schlange.dequeue;
  END;

  Anhaengen(Hilf0);
  …….
  Anhaengen(Hilf9);

 WHILE NOT Schlange.isEmpty DO BEGIN
    e := Schlange.front;
    ziffer := (e.getInhalt DIV 10) MOD 10;
    CASE ziffer of
      0: Hilf0.enqueue(e);
      ……
      9: Hilf9.enqueue(e)
    END;  // of CASE
    Schlange.dequeue;
  END;
  Anhaengen(Hilf0);
  ……
  Anhaengen(Hilf9);
```

```
WHILE NOT Schlange.isEmpty DO BEGIN
    e := Schlange.front;
    ziffer := (e.getInhalt DIV 100) MOD 10;
    CASE ziffer of
      0: Hilf0.enqueue(e);
      ……..
      9: Hilf9.enqueue(e)
    END;  // of CASE
    Schlange.dequeue;
 END;
 Anhaengen(Hilf0);
 ……
 Anhaengen(Hilf9);

 WHILE NOT Schlange.isEmpty DO BEGIN
    e := Schlange.front;
    ziffer := e.getInhalt DIV 1000;
    CASE ziffer of
      0: Hilf0.enqueue(e);
      ……..
      9: Hilf9.enqueue(e)
    END;  // of CASE
    Schlange.dequeue;
  END;
  Anhaengen(Hilf0);
  ……..
  Anhaengen(Hilf9);

  Hilf0.destroy;
  ……..
  Hilf9.destroy;
  Ausgabe
end;

procedure TMain.Anhaengen(s: TQueue);
…….
```

Die Klasse *TQueue*, Version 2

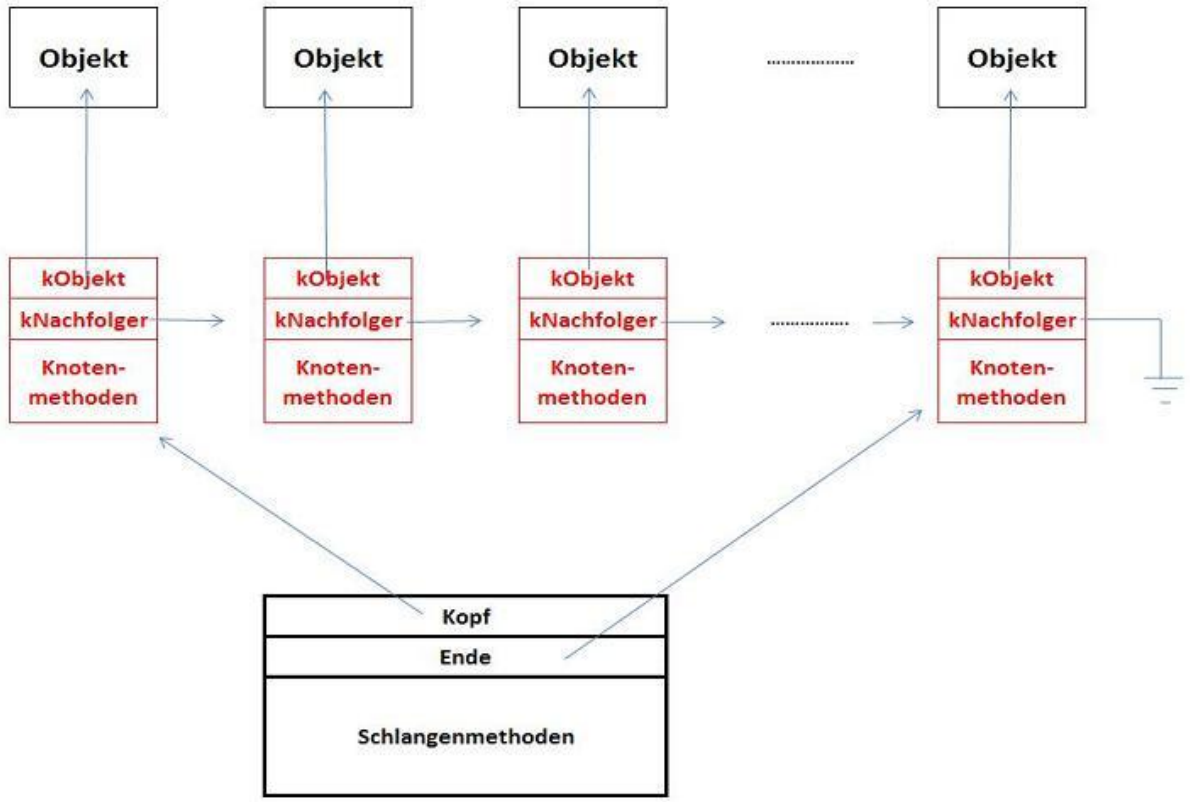
Nachteilig an der vorherigen Version war, dass die Schlange nur ganz spezielle Elemente enthalten konnte. Außerdem war die Information darüber, welches das nächste Element war, in jedem Element selbst gespeichert. Das hatte den weiteren Nachteil, dass jedes Element nur in einer einzigen Schlange enthalten sein konnte. Im Folgenden soll die Klasse *TQueue* so verfeinert werden, dass man beliebige Objekte (man beachte: jedes beliebige Objekt ist auch ein Objekt der Oberklasse *TObject*) in beliebig vielen Schlangen speichern kann.

Aus diesem Grund muss die Information über das jeweils nächste Objekt *„in der Schlange selbst“* und nicht mehr in den Objekten gespeichert werden. Man denke etwa an eine Dia-Show oder eine Abfolge von Musikstücken. Dabei ist die Information über das nächste Bild bzw. Musikstück auch nicht in den Bildern oder Musikstücken gespeichert.

Das ganze Problem bzw. die Verbesserung der alten Version läßt sich im Prinzip durch eine einzige Änderung lösen: In der Skizze auf Seite 9 dieses Skriptes wird in den dort gezeichneten Elementen die Information *Inhalt* ersetzt durch einen Zeiger auf den eigentlichen Inhalt. Damit ergibt sich die nachfolgend gezeichnete Struktur.

Die alten Schlangenelemente sollen in Zukunft als *Knoten* bezeichnet werden, um eine Verwechselung mit den eigentlich zu speichernden Objekten zu vermeiden. Jeder Knoten enthält also zwei Verweise: einen auf das eigentliche Objekt, und einen auf den nächsten Knoten.

Mit der nun folgenden Version lassen sich beliebige Objekte in der Schlange verwalten.



Man kann in die Schlange jedes beliebige Objekt einfügen, da jede Klasse eine Unterklasse der Klasse *TObject* ist. Beim Lesen eines Schlangenelements allerdings **muss eine Typkonvertierung stattfinden**, da die Klasse *TQueue* nur Objekte der allgemeinen Klasse *TObject* zurückgibt. Die Typkonvertierung wird erreicht, indem der gewünschte Typ mit der sog. ***as-Anweisung*** angegeben wird.

Beispiel: `e := Schlange.front as TIntegerElement`
Alternativ kann man auch schreiben:
e := TIntegerElement(Schlange.front)

Begründung: Der Befehl `e := Schlange.front` alleine würde nur ein beliebiges Objekt liefern. Der Rechner wüßte nicht, ob **e** ein Bild oder ein Musikstück oder sonst etwas ist.

Wichtig: An den Methodennamen der Klasse *TQueue* ändert sich nur der Übergabeparameter bei *procedure enqueue(pObjekt: TObject)* und das Funktionsergebnis bei *function front: TObject*

Im folgenden kleinen Demo-Programm sollen Zahlen mit einer Schlange verwaltet werden. Nachteilig ist allerdings, dass z.B. die primitiven Datentypen

INTEGER, BOOLEAN, CHAR, STRING usw. in DELPHI noch nicht als Objekte zählen. Um mit dem Programm trotzdem einfach arbeiten zu können, wird deshalb wieder eine ziemlich einfache Klasse *TIntegerElement* konstruiert. Allerdings hat diesmal die Klasse *TIntegerElement* nichts mit der Klasse *TQueue* zu tun. Insbesondere besitzt ein Objekt dieser Klasse *TIntegerElement* keine Zeiger auf nachfolgende Objekte.

```
unit mIntegerElement;
interface
Type
  TIntegerElement = class(TObject)
    private
     Inhalt: INTEGER;
    public
      constructor create(pZahl: INTEGER);
      function getInhalt: INTEGER;
  End;

implementation

  constructor TIntegerElement.create(pZahl: INTEGER);
   BEGIN
     inherited create;
     {Wenn in der Oberklasse die create-Methode dieselben Parameter hätte
      wie hier, dann hätte auch der Aufruf  inherited  alleine (ohne Angabe
      des Konstruktornamens „create“) genügt.}
     Inhalt := pZahl
   END;

   function TIntegerElement.getInhalt: INTEGER;
     BEGIN
       Result := Inhalt
     END;
end.
```

```
unit mQueue;
interface

Type
 TKnoten = class(TObject)
   private
     kObjekt: TObject;
     kNachfolger: TKnoten;
   public
     constructor create(pObjekt: TObject);
     function getObjekt: TObject;
     function getNachfolger: TKnoten;
     procedure setNachfolger(pNachfolger: TKnoten);
 END;


 TQueue = class(TObject)
   private
     kopf, ende: TKnoten;
   public
     function isEmpty: boolean;
     procedure enqueue(pObjekt: TObject);
     procedure dequeue;
     function front: TObject;
     destructor destroy; override;
 END;

implementation

constructor TKnoten.create(pObjekt: TObject);
BEGIN
  inherited create; // überflüssig
  kObjekt := pObjekt;
  kNachfolger := NIL // nicht notwendig, weil DELPHI Zeiger
                        standardmäßig mit NIL initialisiert werden.
END;
```

```
function TKnoten.getObjekt: TObject;
BEGIN
  Result := kObjekt
END;


function TKnoten.getNachfolger: TKnoten;
BEGIN
  Result := kNachfolger
END;


procedure TKnoten.setNachfolger(pNachfolger: TKnoten);
BEGIN
  kNachfolger := pNachfolger
END;


function TQueue.isEmpty: boolean;
BEGIN
  Result := (kopf = NIL);
END;


procedure TQueue.enqueue(pObjekt: TObject);
VAR hilf: TKnoten;
BEGIN
  hilf := TKnoten.create(pObjekt);
  If kopf = NIL THEN kopf := hilf
  ELSE ende.setNachfolger(hilf);
  ende := hilf
END;


procedure TQueue.dequeue;
VAR hilf: TKnoten;
BEGIN
  IF NOT isEmpty THEN BEGIN
```

```
    IF (kopf = ende) THEN BEGIN
      ende.destroy;
      ende := NIL;
      kopf := NIL
    END
    ELSE BEGIN
           hilf := kopf;
           kopf := kopf.getNachfolger;
           hilf.Destroy;
         END
  END
END;


function TQueue.front: TObject;
BEGIN
  IF kopf <> NIL THEN Result := kopf.getObjekt
  ELSE Result := NIL
END;


destructor TQueue.destroy;
BEGIN
  While NOT isEmpty Do dequeue;
  inherited destroy
END;

end.
```

```
unit mHaupt;
interface
uses  ….mIntegerElement, mQueue;

type
  TMain = class(TForm)
      // Deklaration wie in Version 1
  end;

var
  Main: TMain;
  Schlange: TQueue;

implementation
{$R *.dfm}

procedure TMain.BtElementErzeugenClick(Sender:
TObject);
VAR e: TIntegerElement;
begin
  e := TIntegerElement.create(random(1000));
  If Schlange = NIL THEN Schlange := TQueue.create;
  Schlange.enqueue(e);
  AusgabeBox.Items.Add(IntToStr(e.getInhalt))
end;


procedure TMain.FormCreate(Sender: TObject);
begin
  Schlange := TQueue.create;
  randomize
end;
```

```
procedure TMain.BtSchlangenAusgabeClick(Sender: TObject);
VAR e: TIntegerElement;
begin
  IF Schlange <> NIL THEN
    While Not Schlange.isEmpty DO BEGIN
      e := Schlange.front as TIntegerElement;
      // alternative Möglichkeit:     e := TIntegerElement(Schlange.front)
      AusgabeBox.Items.Add(IntToStr(e.getInhalt));
      Schlange.dequeue
    END
end;


procedure TMain.BtHeapbelegungClick(Sender: TObject);
// siehe Version 1

procedure TMain.BtLoeschenClick(Sender: TObject);
begin
  IF Schlange <> NIL THEN BEGIN
    Schlange.destroy;
    Schlange := NIL
  END
end;


end.
```

Aufgaben

1. Das Hauptprogramm soll eine Schlange B an eine schon bestehende Schlange A anhängen. Die Schlange A wird dadurch länger. Die Schlange B verschwindet.

2. Schreibe ein Programm zur Simulation einer Tanzschule. In der Tanzschule treffen neue Personen ein, die entweder in die Schlange der Damen oder die Schlange der Herren eingereiht werden. Anschließend werden Paare gebildet, die in eine dritte Schlange, die Tanzschlange, eingereiht werden. Paare verlassen die Tanzschlange in der Reihenfolge ihrer Ankunft.

3. Simulation eines Skat-Kartenspiels

a) Erzeuge eine Klasse *TSpielkarte* mit den Attributen *Farbe* und *Wert*! Es gibt die Farben Karo, Herz, Pik und Kreuz und die Werte 7, 8, 9, 10, Bube, Dame, König, Ass.

b) Erzeuge eine Schlange namens *Spiel*, welche alle 32 Skat-Karten speichert.

c) alle Karten der Schlange *Spiel* sollen nacheinander in einer Listbox ausgegeben werden.

d) Die Karten sollen gemischt werden. Dazu werden alle Karten nach dem Zufallsprinzip auf vier unterschiedliche Schlangen verteilt. Anschließend werden sie wieder in die Originalschlange *Spiel* zurückgeschrieben. Die Karten der Schlange werden anschließend in der Listbox ausgegeben.

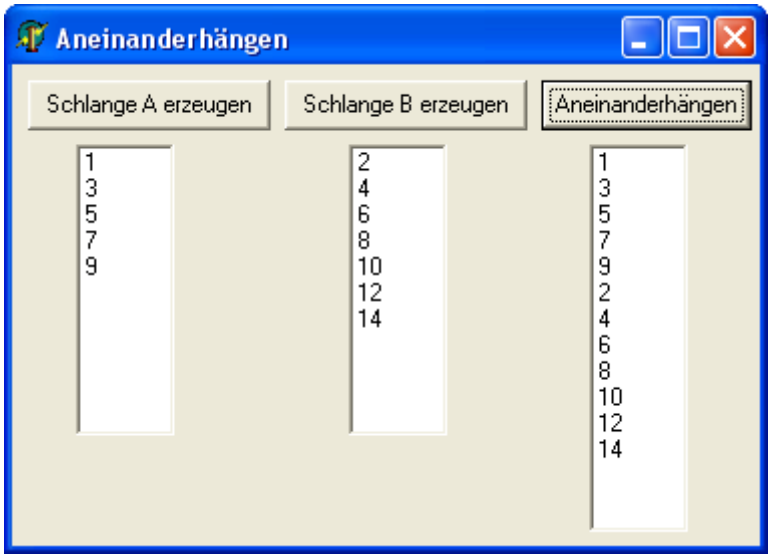
e) Das *Spiel* soll nach Farben sortiert werden. Die Karten der Schlange werden anschließend in der Listbox ausgegeben.

4. Sei X vom Typ *TObject*. Wie reagiert dein Hauptprogramm auf folgende Befehlssequenz? Warum?

```
X := Schlange.front;
X.destroy;
X := Schlange.front;
```

Lösungen

Aufgabe 1




```
unit mHaupt;
interface
uses ……  Windows, mIntegerElement, mQueue;
type  TMain = class(TForm)
      ……..
var Main: TMain;
    SchlangeA, SchlangeB: TQueue;

Implementation
{$R *.dfm}

procedure TMain.BtAErzeugenClick(Sender: TObject);
VAR e: TIntegerElement;
    i: INTEGER;
begin
  SchlangeA := TQueue.create;
  AusgabeboxA.clear;
  FOR i := 1 TO 5 DO BEGIN
    e := TIntegerElement.create(2*i-1);
    SchlangeA.enqueue(e);
    AusgabeBoxA.Items.Add(IntToStr(e.getInhalt))
  END
end;
```

```
procedure TMain.BtBErzeugenClick(Sender: TObject);
```
//analog

```
procedure TMain.BtAneinanderHaengenClick(Sender:
TObject);
VAR e: TIntegerElement;
begin
  AusgabeboxC.clear;
  IF SchlangeB <> NIL THEN BEGIN
    WHILE NOT SchlangeB.isEmpty DO BEGIN
      e := SchlangeB.front as TIntegerElement;
      SchlangeA.enqueue(e);
      SchlangeB.dequeue;
    END;

    WHILE NOT SchlangeA.isEmpty DO BEGIN
      e := SchlangeA.front as TIntegerElement;
      SchlangeB.enqueue(e);
      SchlangeA.dequeue;
      AusgabeBoxC.Items.Add(IntToStr(e.getInhalt))
    END;
    SchlangeA.destroy;
    SchlangeA := SchlangeB;
    SchlangeB := NIL
  END
end;

end.
```

Aufgabe 3


```
unit mSpielkarte;
interface
Type
  Farbe = (Karo, Herz, Pik, Kreuz);
  Wert  = (Sieben, Acht, Neun, Zehn, Bube, Dame,
           Koenig, Ass);

  TSpielkarte = class(TObject)
    private
     f: Farbe;
     w: Wert;
    public
      constructor create(pf: Farbe; pw: Wert);
      function getFarbe: Farbe;
      function getWert: Wert;
  End;
```

```
Implementation

constructor TSpielkarte.create(pf: Farbe; pw: Wert);
BEGIN
  inherited create;
  f:= pf;
  w := pw
END;

function TSpielkarte.getFarbe: Farbe;
BEGIN
  RESULT := f
END;

function TSpielkarte.getWert: Wert;
BEGIN
  RESULT := w
END;

end.


unit mHaupt;
interface
uses       Windows, mSpielkarte, mQueue;
type
  TMain = class(TForm)
    BtSpielErzeugen: TButton;
    AusgabeBox: TListBox;
    BtMischen: TButton;
    procedure BtSpielErzeugenClick(Sender: TObject);
    procedure BtMischenClick(Sender: TObject);
    private
      procedure Ausgabe(e: TSpielkarte);
      procedure Spielausgabe;
      procedure Anhaengen(s: TQueue);
  end;
```

```
var  Main: TMain;
     Spiel: TQueue;
implementation
{$R *.dfm}

procedure TMain.BtSpielErzeugenClick(Sender:
TObject);
VAR f: Farbe;
    w: Wert;
    e: TSpielkarte;
begin
  Spiel := TQueue.create;
  Ausgabebox.clear;
  FOR f := Karo TO Kreuz DO
    FOR w := Sieben TO Ass DO BEGIN
      e := TSpielkarte.create(f,w);
      Spiel.enqueue(e);
    END;
  Spielausgabe
end;


procedure TMain.Ausgabe(e: TSpielkarte);
VAR f: Farbe;
    w: Wert;
    wort: STRING;
BEGIN
  f:= e.getFarbe;
  w:= e.getWert;
  CASE f of
    Karo:  wort := 'Karo ';
    Herz:  wort := 'Herz ';
    Pik:   wort := 'Pik ';
    Kreuz: wort := 'Kreuz '
  END;
  CASE w of
    Sieben: wort := wort + '7';
    Acht:   wort := wort + '8';
```

```
    Neun:   wort := wort + '9';
    Zehn:   wort := wort + '10';
    Bube:   wort := wort + 'Bube';
    Dame:   wort := wort + 'Dame';
    Koenig: wort := wort + 'König';
    Ass:    wort := wort + 'Ass';
  END;
  Ausgabebox.Items.Add(wort)
END;

procedure TMain.Spielausgabe;
VAR hilf: TQueue;
    e: TSpielkarte;
BEGIN
  AusgabeBox.clear;
  hilf := TQueue.create;
  While Not Spiel.isEmpty DO BEGIN
    e := Spiel.front as TSpielkarte;
    Ausgabe(e);
    hilf.enqueue(e);
    Spiel.dequeue
  END;
  Spiel.destroy;
  Spiel := hilf
END;


procedure TMain.BtMischenClick(Sender: TObject);
{Das Mischen geschieht (nahezu) völlig identisch wie auf den Seiten 21 und 22
 in diesem Skript dargestellt.}
```

Die Klasse *TStack*

Objekte der Klasse *TStack* (Keller, Stapel) verwalten beliebige Objekte nach dem ***LIFO***-Prinzip (Last-In-First-Out), d.h. das zuletzt abgelegte Element wird als erstes wieder entnommen.

Die Syntax der Methoden der Klasse *TStack* wurde im Jahre 2007 zumindest für alle Schulen in NRW gemeinsam folgendermaßen festgelegt:

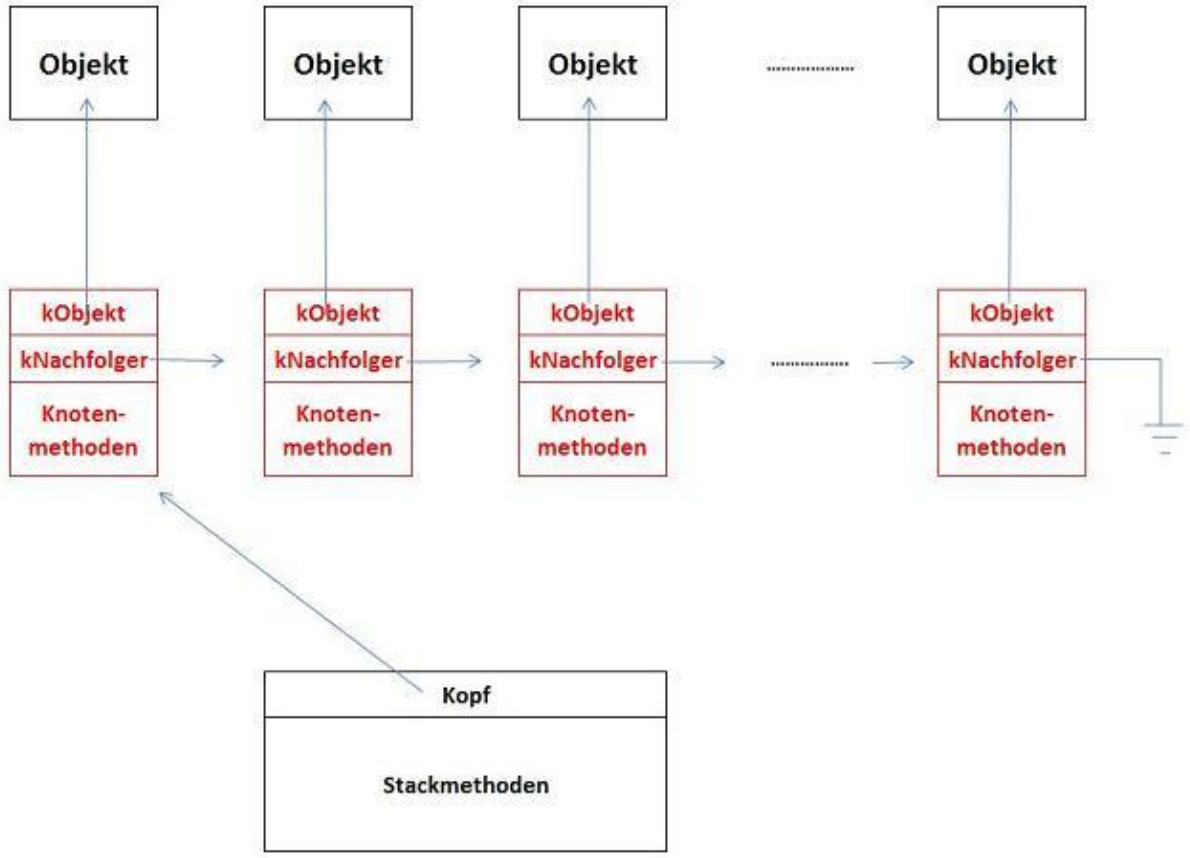
```
constructor create
function isEmpty: boolean
procedure push(pObject: TObject)
procedure pop
function top: TObject
destructor destroy
```

Auch der Typ *TStack* ist im Prinzip eine Liste von Objekten. Es können zum Beispiel mehrere Listen existieren, die teilweise dieselben Objekte enthalten. Aus diesem Grund sollte bei der Methode *pop* das Element zwar aus der Liste, aber nicht aus dem Speicher gelöscht werden.

Dokumentation der Methoden der Klasse *TStack*

Konstruktor	**create**
Nachher	Ein leerer Stapel ist erzeugt.
Anfrage	**isEmpty: boolean**
Nachher	Die Anfrage liefert den Wert true, wenn der Stapel keine Elemente enthält, sonst liefert sie den Wert false.
Auftrag	**push(pObject: TObject)**
Vorher	Der Stapel ist erzeugt.
Nachher	*pObject* liegt oben auf dem Stapel.
Auftrag	**pop**
Vorher	Der Stapel ist nicht leer.
Nachher	Das zuletzt eingefügte Element ist von dem Stapel entfernt (aber nicht aus dem dem Speicher gelöscht).
Anfrage	**top: TObject**
Vorher	Der Stapel ist nicht leer.
Nachher	Die Anfrage liefert das oberste Stapelelement. Der Stapel ist unverändert.
Destruktor	**destroy**
Nachher	Der Stapel existiert nicht mehr.

Die Implementation der Klasse ***TStack*** unterscheidet sich nur wenig von der Implementation der Klasse *TQueue*. Deshalb werden hier nur die Änderungen dargestellt.



Die von uns zwecks Lieferung von einfachen Zahlobjekten benutzte Klasse *TIntegerElement* wird identisch übernommen.

```
unit mStack;
interface
Type
 TKnoten = class(TObject)
   // Diese Klasse wird auch völlig identisch übernommen
 END;

 TStack = class(TObject)
   private
     kopf: TKnoten;
   public
     // es gibt keinen eigenen Konstruktor !
     function isEmpty: boolean;
     procedure push(pObjekt: TObject);
     procedure pop;
     function top: TObject;
     destructor destroy; override;
 END;
```

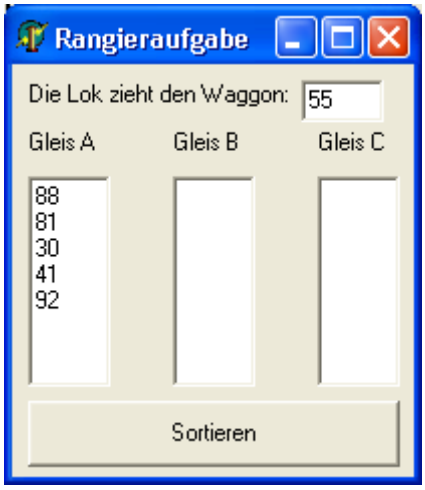
```
implementation
```
// Die Methoden der Klasse TKnoten wurden völlig identisch übernommen.

Ein eigener Konstruktor ist nicht notwendig, weil der Zeiger kopf *automatisch mit NIL initialisiert wird. Bei fehlendem Konstruktor ruft die Delphi-Umgebung den Konstruktor der Oberklasse auf.*

```
function TStack.isEmpty: boolean;
BEGIN
  Result := (kopf = NIL);
END;


procedure TStack.push(pObjekt: TObject);
VAR hilf: TKnoten;
BEGIN
  hilf := TKnoten.create(pObjekt);
  hilf.setNachfolger(kopf);
  kopf := hilf
END;


procedure TStack.pop;
VAR hilf: TKnoten;
BEGIN
  IF NOT isEmpty THEN BEGIN
     hilf := kopf;
     kopf := kopf.getNachfolger;
     hilf.Destroy;
  END
END;
```

```
function TStack.top: TObject;
BEGIN
  IF kopf <> NIL THEN Result := kopf.getObjekt
  ELSE Result := NIL
END;


destructor TStack.destroy;
BEGIN
  While NOT isEmpty Do pop;
  inherited destroy
END;

end.
```

Rangieraufgabe



Auf einem Bahnhof soll eine Lok die einzelnen Waggons eines Zuges sortieren. Die Waggons sind nummeriert und stehen auf dem Gleis A in ungeordneter Reihenfolge hintereinander. Nach getaner Arbeit sollen die einzelnen Waggons auf dem Gleis C in sortierter Reihenfolge stehen. Als Hilfsgleis steht das Gleis B zur Verfügung. Die Lok kann immer nur einen einzelnen Waggon verschieben. Die Lok „weiß“, welche Waggonnummer auf welchem Gleis vorne steht.

Lösung der Rangieraufgabe

Der Befehl Listenausgabe im folgenden Programm behindert ein wenig das Verständnis des Lösungsweges, aber er verdeutlicht beim Programmablauf den Lösungsweg.

```
unit MMain;

interface
uses  ….. mIntegerElement, mStack;

type
  TMain = class(TForm)
    Label1, Label2, Label3, Label4: TLabel;
    MemoA, MemoB, MemoC: TMemo;
    BtSortieren: TButton;
    Edit1: TEdit;
    procedure Listenausgabe;
    procedure FormActivate(Sender: TObject);
    procedure BtSortierenClick(Sender: TObject);
    private
      StackA, StackB, StackC: TStack;
  end;

var  Main: TMain;

implementation
{$R *.dfm}

procedure TMain.Listenausgabe;
 procedure Ausgabe(Stapel:TStack; Memo: TMemo);
 VAR e: TIntegerElement;
     hilf: TStack;
 begin
  IF Stapel <> NIL THEN BEGIN
    hilf := TStack.create;
    While Not Stapel.isEmpty DO BEGIN
      e := Stapel.top as TIntegerElement;
```

```
      Memo.Lines.Add(IntToStr(e.getInhalt));
      hilf.push(e);
      Stapel.pop
    END;
    While Not hilf.isEmpty DO BEGIN
      e := hilf.top as TIntegerElement;
      Stapel.push(e);
      hilf.pop
    END;
    hilf.destroy;
  END // of IF
 end;

 BEGIN
  MemoA.Clear;
  MemoB.Clear;
  MemoC.Clear;
  If NOT StackA.isEmpty THEN Ausgabe(StackA,MemoA);
  If NOT StackB.isEmpty THEN Ausgabe(StackB,MemoB);
  If NOT StackC.isEmpty THEN Ausgabe(StackC,MemoC);
  Application.ProcessMessages;       // Die Reihenfolge ist wichtig
  Sleep(1500);
 END;

procedure TMain.FormActivate(Sender: TObject);
VAR i, n :INTEGER;
    Nummern : set of 1..255;
    e: TIntegerElement;
begin
 MemoA.Clear;
 MemoB.Clear;
 MemoC.Clear;
 StackA := TStack.create;
 StackB := TStack.create;
 StackC := TStack.create;
 Nummern := [];
 RANDOMIZE;
```

```
 FOR i := 1 TO 5 DO Begin
   REPEAT n := RANDOM(100) UNTIL NOT (n in Nummern);
   Nummern := Nummern + [n];
   e := TIntegerElement.create(n);
   StackA.push(e)
 End;  // of for
 Listenausgabe;
end;

procedure TMain.SortierenClick(Sender: TObject);
VAR eA, eB, eC : TIntegerElement;
BEGIN
 WHILE NOT StackA.isEmpty DO BEGIN
   eA := StackA.top as TIntegerElement;
   IF StackC.isEmpty THEN BEGIN
     Edit1.Text:= IntToStr(eA.getInhalt);
     StackA.pop;
     Listenausgabe;
     StackC.push(eA);
     Edit1.Text:= '';
     Listenausgabe;
   END
   ELSE BEGIN  //ELSE Nummer 1
          eC:= StackC.top as TIntegerElement;
          IF eC.getInhalt < eA.getInhalt THEN BEGIN
           Edit1.Text:= IntToStr(eA.getInhalt);
           StackA.pop;
           Listenausgabe;
           StackC.push(eA);
           Edit1.Text:= '';
           Listenausgabe;
          END
          ELSE BEGIN  //ELSE Nummer 2
                 REPEAT
                   Edit1.Text:= IntToStr(eC.getInhalt);
                   StackC.pop;
                   Listenausgabe;
```

```
                    Edit1.Text:= '';
                    StackB.push(eC);
                    Listenausgabe;
                    IF NOT StackC.isEmpty THEN
                     eC:= StackC.top as TIntegerElement;
                  UNTIL StackC.isEmpty OR
                       (eC.getInhalt <= eA.getInhalt);
                  Edit1.Text:= IntToStr(eA.getInhalt);
                  StackA.pop;
                  Listenausgabe;
                  StackC.push(eA);
                  Edit1.Text:= '';
                  Listenausgabe;
                  REPEAT
                    eB:= StackB.top as TIntegerElement;
                    StackB.pop;
                    Edit1.Text:= IntToStr(eB.getInhalt);
                    Listenausgabe;
                    Edit1.Text:= '';
                    StackC.push(eB);
                    Listenausgabe;
                  UNTIL StackB.isEmpty
              END // von ELSE Nummer 2
        END       // von ELSE Nummer 1
 END;
 Showmessage('Fertig')
END;

end.
```

Die Klasse *TLangzahl*

Dokumentation

Objekte der Klasse *TLangzahl* stellen beliebig große, natürliche Zahlen einschließlich 0 dar. Ein Objekt wird durch einen Stack dargestellt. Jedes Stackelement enthält nur eine einzige Dezimalziffer. Oben auf dem Stack liegt die Einerstelle der Zahl.

Oberklasse: TStack

Konstruktor **create**
Nachher Eine leeres Objekt ist erzeugt. Insbesondere besitzt diese Langzahl noch keinen Wert.

Auftrag **StrToLangzahl(s:String)**
vorher s stellt eine gültige natürliche Zahl einschließlich 0 dar.
nachher das Objekt hat den entsprechenden Wert.

Anfrage **LangzahlToStr: String**
nachher der Wert des Objektes wird als String geliefert. Das Objekt selbst wird nicht verändert.

Auftrag **eliminiereFuehrendeNullen**
nachher das Objekt besitzt die kürzest mögliche Darstellung.

Anfrage **plus(lz: TLangzahl): TLangzahl**
nachher es wird die Summe Objekt + lz geliefert. Das Objekt selbst und auch lz werden nicht verändert.

Anfrage	**minus(lz: TLangzahl): TLangzahl**
vorher	das Objekt muss größer oder gleich LZ sein, so dass die Differenz nicht negativ wird.
Nachher	es wird die Differenz (Objekt – lz) geliefert. Das Objekt selbst und auch lz werden nicht verändert.

Anfrage	**mal(faktor: INTEGER): TLangzahl**
vorher	faktor darf nicht negativ sein.
nachher	das Produkt wird geliefert. Das Objekt selbst und auch faktor werden nicht verändert.

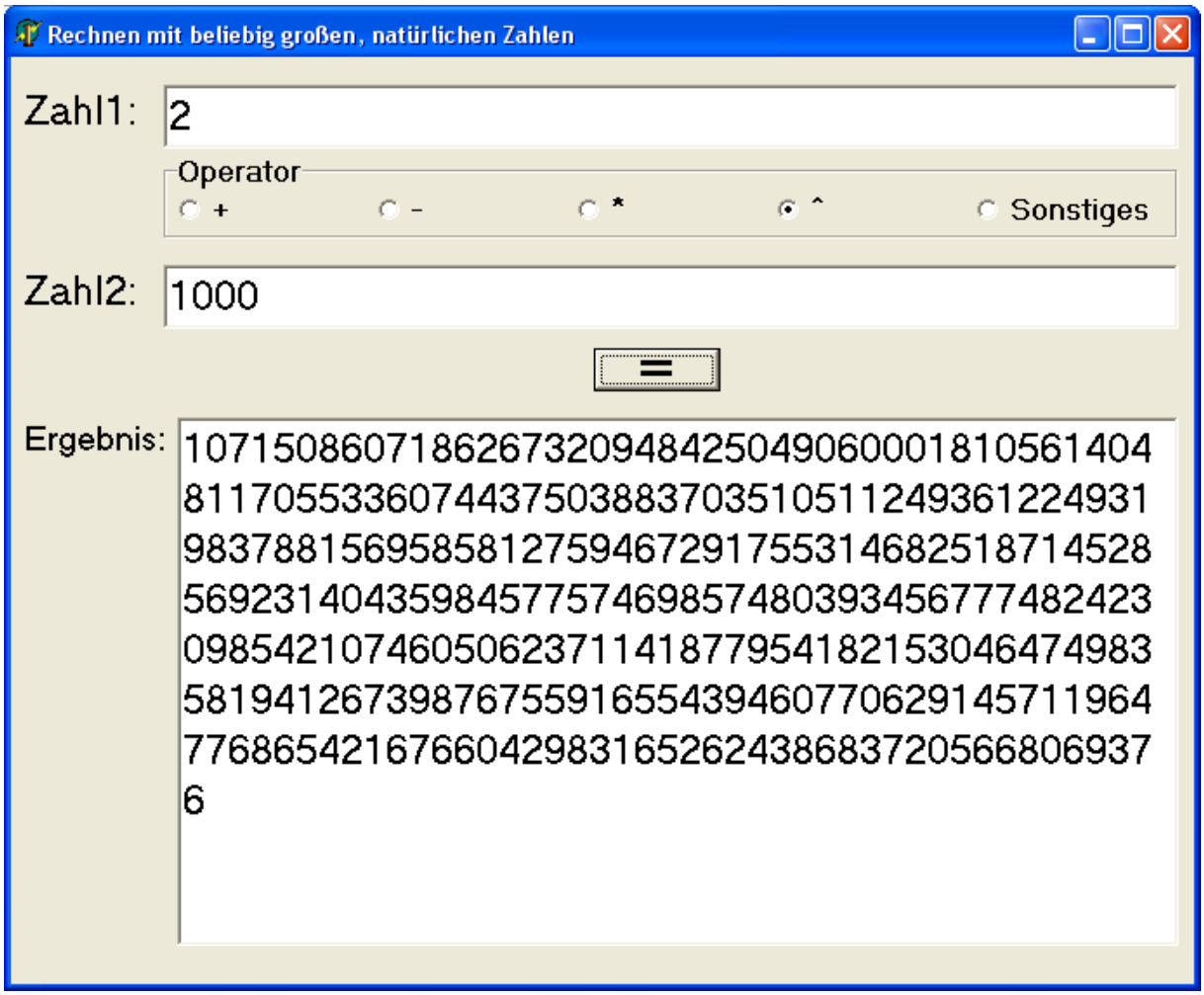
Anfrage	**hoch(exponent: INTEGER): TLangzahl**
vorher	exponent darf nicht negativ sein. Der Wert des Objektes (self) liegt noch im Integerbereich.
nachher	die Potenz wird geliefert. Das Objekt (self) selbst und auch exponent werden nicht verändert.

Anfrage	**stellenzahl: INTEGER**
nachher	die Anzahl der Dezimalstellen wird geliefert.

Anfrage	**istGroesserAls(LZ: TLangzahl):Boolean**
nachher	liefert Wahrheitswert von (Objekt > LZ).

Anfrage	**istKleinerAls(LZ: TLangzahl):Boolean**
nachher	liefert Wahrheitswert von (Objekt < LZ).

Anfrage	**istGleich(LZ: TLangzahl):Boolean**
nachher	liefert den Wahrheitswert von (Objektwert = LZ).


Rechnen mit beliebig großen, natürlichen Zahlen
Zahl1:
2
Operator
+
-
*
^
Sonstiges
Zahl2:
1000
=
Ergebnis:
1071508607186267320948425049060001810561404
8117055336074437503883703510511249361224931
9837881569585812759467291755314682518714528
5692314043598457757469857480393456777482423
0985421074605062371141877954182153046474983
5819412673987675591655439460770629145711964
7768654216766042983165262438683720566806937
6

```
unit mLangzahl;

interface
USES mStack, mIntegerElement, Dialogs, SysUtils;
Type
  TLangzahl = class(TStack)
    private
      function multiplikation(ziffer: Byte): TLangzahl;
      function vergleich(x,y: TLangzahl):INTEGER;
    public
      procedure StrToLangzahl(s:String);
      function LangzahlToStr: String;
      procedure eliminiereFuehrendeNullen;
      function plus(lz: TLangzahl): TLangzahl;
      function mal(faktor: INTEGER): TLangzahl;
      function hoch(exponent: INTEGER): TLangzahl;
      function minus(lz: TLangzahl): TLangzahl;
      function stellenzahl: INTEGER;
      function istGroesserAls(LZ: TLangzahl):Boolean;
      function istKleinerAls(LZ: TLangzahl): Boolean;
      function istGleich(LZ: TLangzahl): Boolean;
  End;

implementation

procedure TLangzahl.StrToLangzahl(s:String);
CONST ziffernZeichen=['0'..'9'];
VAR e:TIntegerElement;
    i:INTEGER;
    ziffernstring: Boolean;
begin
  ziffernstring := true;
  WHILE (1<length(s)) AND (s[1]='0') DO Delete(s,1,1);
  For i := length(s) DOWNTO 1 DO
    IF NOT (s[i] in ziffernZeichen) THEN
                                ziffernstring := false;
  IF NOT ziffernstring THEN
     Showmessage('String enthält nicht nur Ziffern!')
```

```
  ELSE BEGIN
         While NOT self.isEmpty Do pop;
         For i := 1 TO length(s) DO BEGIN
           e := TIntegerElement.create(ORD(s[i])-48);
           push(e)
         End;
       END;
end;


function TLangzahl.LangzahlToStr: String;
VAR hilf: TStack;
    e: TIntegerElement;
    s: String;
Begin
  s := '';
  hilf := TStack.create;
  WHILE NOT self.isEmpty DO BEGIN
     e := self.top as TIntegerElement;
     s := IntToStr(e.getInhalt) + s;
     hilf.push(e);
     self.pop
  END;
  WHILE NOT hilf.isempty DO BEGIN
    self.push(hilf.top);
    hilf.pop
  END;
  hilf.destroy;
  RESULT := s
End;


procedure TLangzahl.eliminiereFuehrendeNullen;
VAR hilf: TStack;
    e: TIntegerElement;
BEGIN
  hilf := TStack.create;
  WHILE NOT self.isEmpty DO BEGIN
```

```
      hilf.push(self.top);
      self.pop
  END;

  while (NOT hilf.isEmpty) AND
  ((hilf.top as TIntegerElement).getInhalt = 0)
  DO hilf.pop;
  IF hilf.isEmpty THEN begin
    e := TIntegerElement.create(0);
    hilf.push(e)
  end;

  WHILE NOT hilf.isempty DO BEGIN
    self.push(hilf.top);
    hilf.pop
  END;
  hilf.destroy;
END;

function TLangzahl.stellenzahl: INTEGER;
VAR zaehler: INTEGER;
    hilf: TStack;
begin
  eliminiereFuehrendeNullen;
  zaehler := 0;
  hilf := TStack.create;
  WHILE NOT self.isEmpty DO BEGIN
     hilf.push(self.top);
     INC(zaehler);
     self.pop
  END;
  WHILE NOT hilf.isEmpty DO BEGIN
    self.push(hilf.top);
    hilf.pop
  END;
  hilf.destroy;
  RESULT := zaehler;
end;
```

```
function TLangzahl.plus(LZ: TLangzahl): TLangzahl;
VAR uebertrag, Ziffer: INTEGER;
    e1, e2, e3: TIntegerElement;
    hilf, zahl1, zahl2, ergebnis: TLangzahl;
Begin
 hilf := TLangzahl.create;
 zahl1 := TLangzahl.create;
 zahl2 := TLangzahl.create;
 WHILE NOT self.isEmpty DO BEGIN
   hilf.push(self.top);
   self.pop
 END;
 WHILE NOT hilf.isEmpty DO BEGIN
   self.push(hilf.top);
   zahl1.push(hilf.top);
   hilf.pop
 END;

 WHILE NOT lz.isEmpty DO BEGIN
   hilf.push(lz.top);
   lz.pop
 END;
 WHILE NOT hilf.isEmpty DO BEGIN
   lz.push(hilf.top);
   zahl2.push(hilf.top);
   hilf.pop
 END;

 uebertrag := 0;
 WHILE (Not zahl1.isEmpty) AND (Not zahl2.isEmpty) DO
 Begin
   e1 := zahl1.top as TIntegerElement;
   e2 := zahl2.top as TIntegerElement;
   ziffer := (e1.getInhalt + e2.getInhalt +
                                    uebertrag) MOD 10;
   uebertrag := (e1.getInhalt + e2.getInhalt +
                                    uebertrag) DIV 10;
   zahl1.pop;
```

```
    zahl2.pop;
    e3 := TIntegerElement.create(ziffer);
    hilf.push(e3)
  END;
  IF zahl1.isEmpty THEN
    WHILE NOT zahl2.isEmpty DO Begin
      e2:= zahl2.top as TIntegerElement;
      ziffer := (e2.getInhalt + uebertrag) MOD 10;
      uebertrag := (e2.getInhalt + uebertrag) DIV 10;
      e3 := TIntegerElement.create(ziffer);
      hilf.push(e3);
      zahl2.pop
    END
  ELSE
    WHILE NOT zahl1.isEmpty DO Begin
      e1:= zahl1.top as TIntegerElement;
      ziffer := (e1.getInhalt + uebertrag) MOD 10;
      uebertrag := (e1.getInhalt + uebertrag) DIV 10;
      e3 := TIntegerElement.create(ziffer);
      hilf.push(e3);
      zahl1.pop
    END;
  IF uebertrag > 0 THEN BEGIN
    e3 := TIntegerElement.create(uebertrag);
    hilf.push(e3);
  END;
  zahl1.destroy;
  zahl2.destroy;

  ergebnis := TLangzahl.create;
  WHILE NOT hilf.isEmpty DO BEGIN
    ergebnis.push(hilf.top);
    hilf.pop
  END;

  hilf.destroy;
  RESULT := ergebnis
End; // of plus
```

```
function TLangzahl.minus(lz: TLangzahl): TLangzahl;
VAR uebertrag, Ziffer, Ziffer1, Ziffer2: INTEGER;
    e: TIntegerElement;
    hilf, zahl1, zahl2, ergebnis: TLangzahl;
BEGIN
 hilf := TLangzahl.create;
 zahl1 := TLangzahl.create;
 zahl2 := TLangzahl.create;
 WHILE NOT self.isEmpty DO BEGIN
   hilf.push(self.top);
   self.pop
 END;
 WHILE NOT hilf.isEmpty DO BEGIN
   self.push(hilf.top);
   zahl1.push(hilf.top);
   hilf.pop
 END;

 WHILE NOT lz.isEmpty DO BEGIN
   hilf.push(lz.top);
   lz.pop
 END;
 WHILE NOT hilf.isEmpty DO BEGIN
   lz.push(hilf.top);
   zahl2.push(hilf.top);
   hilf.pop
 END;

  uebertrag := 0;
  WHILE Not zahl2.isEmpty  DO Begin
   Ziffer1 := (zahl1.top as TIntegerElement).getInhalt;
   Ziffer2 := (zahl2.top as TIntegerElement).getInhalt;
   IF Ziffer1 >= (Ziffer2 + uebertrag) THEN BEGIN
      ziffer := Ziffer1 - (Ziffer2 + uebertrag);
      uebertrag := 0
   END
   ELSE BEGIN
          ziffer:= Ziffer1 + 10 - (Ziffer2 + uebertrag);
```

```
          uebertrag := 1
        END;
    zahl1.pop;
    zahl2.pop;
    e := TIntegerElement.create(ziffer);
    hilf.push(e)
  END;
  zahl2.destroy;

  WHILE NOT zahl1.isEmpty DO Begin
     Ziffer1:= (zahl1.top as TIntegerElement).getInhalt;
     IF Ziffer1 >= uebertrag THEN BEGIN
       ziffer := Ziffer1 - uebertrag;
       uebertrag := 0
     END
     ELSE BEGIN
            ziffer := Ziffer1 + 10 - uebertrag;
            uebertrag := 1
          END;
     zahl1.pop;
     e := TIntegerElement.create(ziffer);
     hilf.push(e)
  End; // of while

 zahl1.destroy;

 ergebnis := TLangzahl.create;
 WHILE NOT hilf.isEmpty DO BEGIN
   ergebnis.push(hilf.top);
   hilf.pop
 END;
 hilf.destroy;
 ergebnis.eliminiereFuehrendeNullen;
 RESULT := ergebnis
END;
```

```
function TLangzahl.multiplikation(ziffer: Byte): TLangzahl;
VAR hilf, zahl1, ergebnis: TLangzahl;
    e, e2: TIntegerElement;
    nummer, uebertrag: Byte;
BEGIN
 hilf := TLangzahl.create;
 zahl1 := TLangzahl.create;
 WHILE NOT self.isEmpty DO BEGIN
   hilf.push(self.top);
   self.pop
 END;
 WHILE NOT hilf.isEmpty DO BEGIN
   self.push(hilf.top);
   zahl1.push(hilf.top);
   hilf.pop
 END;

 uebertrag := 0;
 WHILE Not zahl1.isEmpty DO Begin
   e := zahl1.top as TIntegerElement;
   nummer:= (e.getInhalt* ziffer + uebertrag) MOD 10;
   uebertrag:= (e.getInhalt* ziffer + uebertrag) DIV 10;
   zahl1.pop;

   e2 := TIntegerElement.create(nummer);
   hilf.push(e2)
 END;
 IF uebertrag > 0 THEN BEGIN
    e2 := TIntegerElement.create(uebertrag);
    hilf.push(e2);
 END;

 zahl1.destroy;
 ergebnis := TLangzahl.create;
 WHILE NOT hilf.isEmpty DO BEGIN
   ergebnis.push(hilf.top);
   hilf.pop
 END;
```

```
 RESULT := ergebnis
END;


function TLangzahl.mal(faktor: INTEGER): TLangzahl;
VAR rest: Byte;
    hilf, selfKopie, erg: TLangzahl;
    e: TIntegerElement;
BEGIN
 hilf := TLangzahl.create;
 selfKopie := TLangzahl.create;
 WHILE NOT self.isEmpty DO BEGIN
   hilf.push(self.top);
   self.pop
 END;
 WHILE NOT hilf.isEmpty DO BEGIN
   self.push(hilf.top);
   selfKopie.push(hilf.top);
   hilf.pop
 END;

 erg := TLangzahl.create;
 e := TIntegerElement.create(0);
 erg.push(e);
 WHILE faktor > 0 DO BEGIN
    rest := faktor MOD 10;
    faktor := faktor DIV 10;
    hilf := selfKopie.multiplikation(rest);
    erg:= erg.plus(hilf);
    IF faktor <> 0 THEN BEGIN
      e := TIntegerElement.create(0);
      selfKopie.push(e)
    END
 END;
 RESULT := erg
END;
end.
```

```
function TLangzahl.hoch(Exponent: INTEGER): TLangzahl;
VAR i, intZahl: INTEGER;
    hilf, selfKopie, erg: TLangzahl;
    e: TIntegerElement;
BEGIN
 hilf := TLangzahl.create;
 selfKopie := TLangzahl.create;
 WHILE NOT self.isEmpty DO BEGIN
   hilf.push(self.top);
   self.pop
 END;
 WHILE NOT hilf.isEmpty DO BEGIN
   self.push(hilf.top);
   selfKopie.push(hilf.top);
   hilf.pop
 END;

 erg := TLangzahl.create;
 IF exponent = 0 THEN BEGIN
    e := TIntegerElement.create(1);
    erg.push(e)
 END
 ELSE BEGIN
   erg := selfKopie;
   intZahl := StrToInt(selfKopie.langzahlToStr);
   FOR i := 2 TO Exponent DO erg := erg.mal(intZahl);
 END;
 RESULT := erg
END;
```

```
function TLangzahl.vergleich(x,y: TLangzahl):INTEGER;
VAR entscheidung, ziffer1, ziffer2, ergebnis:INTEGER;
    hilf, zahl1, zahl2: TLangzahl;
BEGIN
  x.eliminiereFuehrendeNullen;
  y.eliminiereFuehrendeNullen;
  IF x.stellenzahl > y.stellenzahl THEN ergebnis := 1
  ELSE IF y.stellenzahl > x.stellenzahl THEN ergebnis:= 2
       ELSE BEGIN
         hilf := TLangzahl.create;
         zahl1 := TLangzahl.create;

         WHILE NOT x.isEmpty DO BEGIN
           hilf.push(x.top);
           zahl1.push(x.top);
           x.pop
         END;
         WHILE NOT hilf.isEmpty DO BEGIN
           x.push(hilf.top);
           hilf.pop
         END;

         zahl2 := TLangzahl.create;
         WHILE NOT y.isEmpty DO BEGIN
           hilf.push(y.top);
           zahl2.push(y.top);
           y.pop
         END;
         WHILE NOT hilf.isEmpty DO BEGIN
           y.push(hilf.top);
           hilf.pop
         END;

         entscheidung := 0;
         WHILE NOT (zahl1.isEmpty) AND
                (entscheidung = 0) DO BEGIN
           ziffer1 := (zahl1.top as TIntegerElement)
                                          .getInhalt;
```

```
            ziffer2 := (zahl2.top as TIntegerElement)
                                              .getInhalt;
            IF ziffer1 > ziffer2 THEN
               entscheidung := 1
            ELSE IF ziffer1 < ziffer2 THEN
                    entscheidung := 2;
            zahl1.pop;
            zahl2.pop
          END;
          ergebnis := entscheidung
       END;
  RESULT := ergebnis
END;


function TLangzahl.istGroesserAls(LZ:TLangzahl): Boolean;
BEGIN
  Result := (vergleich(self, LZ) = 1)
END;


function TLangzahl.istKleinerAls(LZ:TLangzahl): Boolean;
BEGIN
  Result := (vergleich(self, LZ) = 2)
END;


function TLangzahl.istGleich(LZ:TLangzahl): Boolean;
BEGIN
  Result := (vergleich(self, LZ) = 0)
END;
```

```
unit MMain;

interface
uses  ….Dialogs, StdCtrls, mLangzahl;
type
  TMain = class(TForm)
    Label1, Label2, Label3: TLabel;
    EZahl1, EZahl2: TEdit;
    BtBerechne: TButton;
    RadioGroupOperator: TRadioGroup;
    MErgebnis: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure BtBerechneClick(Sender: TObject);
    private
     zahl1, zahl2, ergebnis: TLangzahl;
  end;

var  Main: TMain;

implementation
{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
 Zahl1 := TLangzahl.Create;
 Zahl2 := TLangzahl.Create;
 Ergebnis := TLangzahl.Create;
 EZahl1.Text := '';
 EZahl2.Text:='';
 MErgebnis.Text := '';
end;


procedure TMain.BtBerechneClick(Sender: TObject);
VAR wahl, faktor, exponent: INTEGER;
begin
  MErgebnis.Text := '';
  Zahl1.StrToLangzahl(EZahl1.Text);
```

```
  wahl := RadioGroupOperator.ItemIndex;
  CASE wahl of
    0: BEGIN
         Zahl2.StrToLangzahl(EZahl2.Text);
         ergebnis := zahl1.plus(zahl2);
         MErgebnis.Text:= ergebnis.LangzahlToStr
       END;
    1: BEGIN
         Zahl2.StrToLangzahl(EZahl2.Text);
         ergebnis := zahl1.minus(Zahl2);
         MErgebnis.Text:= ergebnis.LangzahlToStr
       END;
    2: BEGIN
         faktor := StrToInt(EZahl2.Text);
         ergebnis := zahl1.mal(faktor);
         MErgebnis.Text:= ergebnis.LangzahlToStr
       END;
    3: BEGIN
         exponent := StrToInt(EZahl2.Text);
         ergebnis := zahl1.hoch(exponent);
         MErgebnis.Text:= ergebnis.LangzahlToStr
       END;
    4: BEGIN
         Zahl2.StrToLangzahl(EZahl2.Text);
         IF zahl1.istGroesserAls(zahl2) THEN
            MErgebnis.Text := 'zahl1 ist größer'
         ELSE IF zahl1.istKleinerAls(zahl2) THEN
                MErgebnis.Text := 'zahl1 ist kleiner'
              ELSE MErgebnis.Text := 'zahl1 = zahl2'
       END
  END; // of Case
 end;

end.
```

Aufgaben

1. Berechne die Fakultät einer natürlichen Zahl n als Langzahl! Die Zahl n sei vom Typ INTEGER, deren Fakultät sei eine Langzahl. Achte darauf, dass der Speicherplatz für nicht mehr benötigte Zwischenergebnisse wieder frei gegeben wird.

2. Die **Fibonacci-Folge** ist eine unendliche Folge von Zahlen (den Fibonacci-Zahlen), bei der sich die jeweils folgende Zahl durch Addition ihrer beiden vorherigen Zahlen ergibt: 0, 1, 1, 2, 3, 5, 8, 13, … Benannt ist sie nach *Leonardo Fibonacci*, der damit 1202 das Wachstum einer Kaninchenpopulation beschrieb. Die Fibonacci-Folge $f_0, f_1, f_2, \ldots$ist durch das rekursive Bildungsgesetz $f_n = f_{n-1} + f_{n-2}$ für $n \geq 2$ mit den Anfangswerten $f_0 = 0$ und $f_1 = 1$ definiert.

Erstelle eine Liste der ersten 200 Fibonacci-Zahlen! Achte darauf, dass der Speicherplatz für nicht mehr benötigte Zwischenergebnisse wieder frei gegeben wird.
Hinweis: mit der Anweisung *MErgebnis.Lines.Add('Hallo');* wird in das Memofeld namens *Mergebnis* eine entsprechend neue Zeile hinzugefügt.

3. Eine Klasse ***TVZLangzahl***, welche beliebig große (also auch negative) ganze Zahlen enthalten soll, ist relativ leicht zu erstellen, wenn man auf die Klasse *TLangzahl* zurückgreifen kann. Implementiere diese neue Klasse ***TVZLangzahl*** und erstelle ein entsprechendes Hauptprogramm, um die entsprechenden Methoden zu testen!

```
TVZLangzahl = class(TObject)
  public
    VZ: INTEGER;  // enthält den Wert  1 oder -1
    Betrag: TLangzahl;
    constructor create;
    procedure StrToVZLangzahl(s:String);
    function VZLangzahlToStr: String;
    procedure eliminiereFuehrendeNullen;
    function plus(vzlz: TVZLangzahl): TVZLangzahl;
    function minus(vzlz: TVZLangzahl): TVZLangzahl;
    function stellenzahl: INTEGER;
    function istGroesserAls(vzlz: TVZLangzahl): Boolean;
    function istKleinerAls(vzlz:TVZLangzahl): Boolean;
    function istGleich(vzlz:TVZLangzahl): Boolean;
    function mal(faktor: INTEGER): TVZLangzahl;
    function hoch(exponent: INTEGER): TVZLangzahl;
End;
```

Für die Erstellung der folgenden Klasse ***LangDezimalBruch*** benötigt man wesentlich mehr Zeit und Arbeit. Deren Objekte sind Dezimalzahlen mit genau einer Stelle vor dem Komma und einer vorher festgelegten Anzahl von Stellen hinter dem Komma. Erstelle diese Klasse und löse damit die folgenden Aufgaben:

4. Berechne den Kehrwert einer natürlichen Zahl.

5. Berechne die Kreiszahl π. Es gilt: $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - + \cdots$

6. Berechne die Eulersche Zahl e. Es gilt: $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$

Lösungen

Aufgabe 1 (Fakultät)

```
.............
BEGIN
  hilf := StrToInt(EZahl1.Text);
  ergebnis.StrToLangzahl('1');
  FOR i := hilf-1 DOWNTO 1 DO Begin
    help := ergebnis;
    ergebnis :=ergebnis.mal(i);
    help.destroy
  End;
  MErgebnis.Text:= ergebnis.LangzahlToStr
END;
```

Aufgabe 2 (Fibonacci-Folge)

```
.................
BEGIN
  hilf := StrToInt(EZahl1.Text);
  f1 := TLangzahl.create;
  f1.StrToLangzahl('0');
  MErgebnis.Clear;
  MErgebnis.Text:= '1: ' + f1.LangzahlToStr;
  f2 := TLangzahl.create;
  f2.StrToLangzahl('1');
  MErgebnis.Lines.Add('2: ' + f2.LangzahlToStr);
  FOR i := 3 TO hilf DO BEGIN
    f3 := f2.plus(f1);
    MErgebnis.Lines.Add(IntToStr(i) + ': ' +
                                      f3.LangzahlToStr);
    help := f1;
    f1:= f2;
    f2 := f3;
    help.destroy
  END
END;
```

Lösung der Aufgabe 3: Langzahlen mit Vorzeichen

```
unit mVZLangzahl;
interface
USES mLangzahl, Dialogs;
Type
  TVZLangzahl = class(TObject)
    public
      VZ: INTEGER;  // enthält den Wert  1 oder -1
      Betrag: TLangzahl;
      constructor create;
      procedure StrToVZLangzahl(s:String);
      function VZLangzahlToStr: String;
      procedure eliminiereFuehrendeNullen;
      function plus(vzlz: TVZLangzahl): TVZLangzahl;
      function minus(vzlz: TVZLangzahl): TVZLangzahl;
      function stellenzahl: INTEGER;
      function istGroesserAls(vzlz: TVZLangzahl): Boolean;
      function istKleinerAls(vzlz:TVZLangzahl): Boolean;
      function istGleich(vzlz:TVZLangzahl): Boolean;
      function mal(faktor: INTEGER): TVZLangzahl;
      function hoch(exponent: INTEGER): TVZLangzahl;
End;

implementation

constructor TVZLangzahl.create;
BEGIN
  inherited;
  VZ := 1;
  Betrag := TLangzahl.create
END;


procedure TVZLangzahl.StrToVZLangzahl(s:String);
BEGIN
  IF s[1]='-' THEN BEGIN
    self.VZ := -1;
    Delete(s,1,1)
  END
  ELSE IF s[1]='+' THEN BEGIN
         self.VZ := 1;
```

```
        Delete(s,1,1)
      END
      ELSE VZ := 1;
  Betrag.StrToLangzahl(s);
END;


function TVZLangzahl.VZLangzahlToStr: String;
VAR s: STRING;
BEGIN
  s := '';
  IF self.VZ = -1 THEN s := '-';
  s := s + Betrag.LangzahlToStr;
  RESULT := s
END;


procedure TVZLangzahl.eliminiereFuehrendeNullen;
BEGIN
  Betrag.eliminiereFuehrendeNullen
END;


function TVZLangzahl.plus(vzlz: TVZLangzahl):
TVZLangzahl;
VAR ergebnis: TVZLangzahl;
BEGIN
  ergebnis := TVZLangzahl.create;
  IF self.VZ = vzlz.VZ THEN BEGIN
    ergebnis.Betrag := self.Betrag.plus(vzlz.Betrag);
    ergebnis.VZ := self.VZ
  END
  ELSE IF self.Betrag.istGroesserAls(vzlz.Betrag)
       THEN begin
          ergebnis.Betrag :=
                      self.Betrag.minus(vzlz.Betrag);
          ergebnis.VZ := self.VZ
       end
       ELSE begin
              ergebnis.Betrag :=
                      vzlz.Betrag.minus(self.Betrag);
              ergebnis.VZ := vzlz.VZ
            end;
```

```
  RESULT := ergebnis
END;


function TVZLangzahl.minus(vzlz: TVZLangzahl):
TVZLangzahl;
VAR hilf: TVZLangzahl;
BEGIN
  hilf := vzlz.mal(-1);
  RESULT := self.plus(hilf);
  hilf.destroy
END;


function TVZLangzahl.stellenzahl: INTEGER;
BEGIN
  RESULT := self.Betrag.Stellenzahl
END;


function TVZLangzahl.istGroesserAls(vzlz:
TVZLangzahl): Boolean;
BEGIN
  IF (self.VZ=1) AND (vzlz.VZ=1) THEN
    RESULT := self.Betrag.istGroesserAls(vzlz.Betrag)
  ELSE IF (self.VZ=1) AND (vzlz.VZ=-1) THEN RESULT := TRUE
       ELSE IF (self.VZ=-1) AND (vzlz.VZ=1) THEN
               RESULT := FALSE
            ELSE RESULT :=
              self.Betrag.istKleinerAls(vzlz.Betrag);
END;


function TVZLangzahl.istKleinerAls(vzlz: TVZLangzahl):
                                                  Boolean;
BEGIN
  Result := vzlz.istGroesserAls(self)
END;


function TVZLangzahl.istGleich(vzlz:TVZLangzahl):
                                                  Boolean;
BEGIN
```

```
  RESULT := NOT self.istGroesserAls(vzlz) AND
            NOT self.istKleinerAls(vzlz)
END;


function TVZLangzahl.mal(faktor: INTEGER):
TVZLangzahl;
VAR ergebnis: TVZLangzahl;
    Vorzeichen: INTEGER;
BEGIN
  If faktor >= 0 THEN Vorzeichen := 1
  ELSE Vorzeichen := -1;
  ergebnis := TVZLangzahl.create;
  ergebnis.VZ := self.VZ * Vorzeichen;
  ergebnis.Betrag := self.Betrag.mal(ABS(faktor));
  RESULT := ergebnis
END;


function  TVZLangzahl.hoch(exponent: INTEGER):
TVZLangzahl;
VAR ergebnis: TVZLangzahl;
    Basistext: String;
BEGIN
  IF exponent < 0 THEN
    Showmessage('Der Exponent darf nicht negativ sein!')
  ELSE BEGIN
        ergebnis := TVZLangzahl.create;
        ergebnis.Betrag:= self.Betrag.hoch(exponent);
        IF exponent MOD 2 = 0 THEN ergebnis.VZ := 1
        ELSE ergebnis.VZ := self.VZ
       END;
  RESULT := ergebnis
END;

end.
```

Das zugehörige Hauptprogramm lautet:

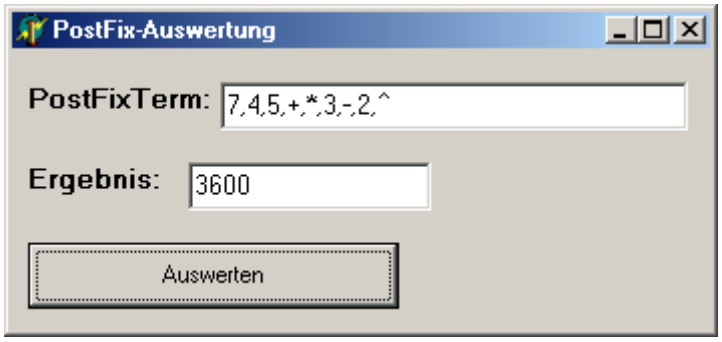
```
unit MMain;
interface
uses ……mVZLangzahl;

type
  TMain = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    EZahl1: TEdit;
    EZahl2: TEdit;
    Label3: TLabel;
    BtBerechne: TButton;
    RadioGroupOperator: TRadioGroup;
    MErgebnis: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure BtBerechneClick(Sender: TObject);
    private
     zahl1, zahl2, ergebnis: TVZLangzahl;
  end;

var  Main: TMain;
implementation
{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
 Zahl1 := TVZLangzahl.Create;
 Zahl2 := TVZLangzahl.Create;
 Ergebnis := TVZLangzahl.Create;
 EZahl1.Text := '';
 EZahl2.Text:='';
 MErgebnis.Text := '';
end;


procedure TMain.BtBerechneClick(Sender: TObject);
VAR wahl, faktor, exponent: INTEGER;
begin
  MErgebnis.Text := '';
  Zahl1.StrToVZLangzahl(EZahl1.Text);
  wahl := RadioGroupOperator.ItemIndex;
```

```
  CASE wahl of
    0: BEGIN
         Zahl2.StrToVZLangzahl(EZahl2.Text);
         ergebnis := zahl1.plus(zahl2);
         MErgebnis.Text:= ergebnis.VZLangzahlToStr
       END;
    1: BEGIN
         Zahl2.StrToVZLangzahl(EZahl2.Text);
         ergebnis := zahl1.minus(Zahl2);
         MErgebnis.Text:= ergebnis.VZLangzahlToStr
       END;
    2: BEGIN
         faktor := StrToInt(EZahl2.Text);
         ergebnis := zahl1.mal(faktor);
         MErgebnis.Text:= ergebnis.VZLangzahlToStr
       END;
    3: BEGIN
         exponent := StrToInt(EZahl2.Text);
         ergebnis := zahl1.hoch(exponent);
         MErgebnis.Text:= ergebnis.VZLangzahlToStr
       END;
    4: BEGIN
         Zahl2.StrToVZLangzahl(EZahl2.Text);
         IF zahl1.istGroesserAls(zahl2) THEN
                 MErgebnis.Text := 'zahl1 ist größer'
         ELSE IF zahl1.istKleinerAls(zahl2) THEN
                MErgebnis.Text := 'zahl1 ist kleiner'
              ELSE MErgebnis.Text := 'zahl1 = zahl2'
       END
  END; // of Case
 end;

end.
```

Aufgabe:
Erstelle eine Dokumentation für die Klasse TVZLangzahl

Umgekehrte polnische Notation

Auswertung mit Hilfe eines Stacks



Die polnische Notation wurde nach dem polnischen Mathematiker *Jan Lukasiewicz* benannt. Ihre wichtigste Eigenschaft besteht darin, dass die Reihenfolge, in der die Operationen durchzuführen sind, vollständig durch die Reihenfolge der Operatoren und Operanden bestimmt ist und keinerlei Klammern oder Prioritätenregeln notwendig sind. *Lukasiewicz* schrieb den Operator vor den Operanden (in sog. *Präfixnotation*), also statt A+B wie in der üblichen sog. *Infixnotation* schrieb er +AB. Bei der umgekehrten polnischen Notation wird der Operator hinter den beiden Operanden geschrieben (deswegen auch *Postfixnotation* genannt): AB+
Der obige Beispielterm entspricht der *Infixnotation* $[7 \cdot (4 + 5) - 3]^2$

Zum Lösungsalgorithmus: Als Trennzeichen fungiert das Komma. Als Endezeichen wird an den Term eine schließende Klammer angehängt.
Der Term wird nun von links nach rechts abgearbeitet.

Wiederhole
Wird eine Zahl gefunden, so wird diese auf den Stack gelegt.
Wird ein Operator ⊗ gefunden, dann
- entnehme dem Stack die oberste Zahl A und die zweitoberste Zahl B,
- berechne B ⊗ A,
- lege das Ergebnis wieder auf den Stack.

Bis das Endezeichen erreicht wird
Gib als Endergebnis die einzige, noch auf dem Stack vorhandene Zahl aus.

Einschränkungen: Es werden nur binäre Operatoren zugelassen (also keine Vorzeichen). Das nachfolgende Programm beschränkt sich zunächst auf einstellige Zahlen. Dadurch wird das Einlesen der Zahlen wesentlich einfacher. Hinweis: Die Zahlelemente müssen vom Typ REAL sein, weil Divisionen vorkommen dürfen.

```
unit mHaupt;
interface
uses  ..... mRealElement, mStack;

CONST Ziffernmenge = ['0'..'9'];
      Operatorenmenge = ['+','-','*','/','^'];
type
  TMain = class(TForm)
    BtAuswerten: TButton;
    EdEingabe, EdErgebnis: TEdit;
    Label1, Label2: TLabel;
    procedure FormActivate(Sender: TObject);
    function naechstesZeichen: CHAR;
    procedure Rechne(ch:CHAR);
    procedure BtAuswertenClick(Sender: TObject);
  private
    Stapel : TStack;
    Term: STRING;
  end;
var  Main: TMain;

implementation
{$R *.dfm}

procedure TMain.FormActivate(Sender: TObject);
begin
 Stapel := TStack.create;
end;
```

```
function TMain.naechstesZeichen: CHAR;
VAR hilf : CHAR;
BEGIN
 hilf := Term[1];
 IF hilf <> ')' THEN Delete(Term,1,1);
 IF Length(Term) > 1 THEN Delete(Term,1,1);
 // nächstes Trennzeichen löschen
 naechstesZeichen := hilf
END;

procedure TMain.Rechne(ch:CHAR);
VAR ergebnis, zahl1, zahl2: REAL;
    e: TRealElement;
BEGIN
 e := Stapel.Top as TRealElement;
 zahl1 := e.getInhalt;
 Stapel.Pop;
 e.Destroy;
 e := Stapel.Top as TRealElement;
 zahl2 := e.getInhalt;
 Stapel.Pop;
 e.Destroy;
 CASE ch of
  '+' : ergebnis := zahl2 + zahl1;
  '-' : ergebnis := zahl2 - zahl1;
  '*' : ergebnis := zahl2 * zahl1;
  '/' : ergebnis := zahl2 / zahl1;
  '^' : IF zahl2 = 0 THEN ergebnis := 0
          ELSE ergebnis := exp(zahl1*ln(zahl2))
 END;
 e := TRealElement.create(ergebnis);
 Stapel.push(e);
END;
```

```
procedure TMain.BtAuswertenClick(Sender: TObject);
VAR ch : CHAR;
    e: TRealElement;
begin
 Term := EdEingabe.Text + ')';
 REPEAT
  ch := naechstesZeichen;
  IF ch in Ziffernmenge THEN BEGIN
     e := TRealElement.create(StrToFloat(ch));
     Stapel.push(e);
  END
  ELSE IF ch in Operatorenmenge THEN Rechne(ch);
 UNTIL ch = ')';
 e := Stapel.Top as TRealElement;
 Stapel.pop;
 EdErgebnis.Text := FloatToStr(e.getInhalt);
 e.Destroy
end;

end.
```

Aufgaben

1. Erweitere das Programm so, dass im Eingabeterm nicht nur einzelne Ziffern, sondern mehrstellige, natürliche Zahlen stehen können. Dabei soll allerdings der positive INTEGER-Bereich nicht überschritten werden.
Hinweis: prüfe zunächst das erste Zeichen in der Variable *Term* im obigen Programm. Ist dieses Zeichen eine Ziffer, so bilden die ersten Zeichen von *Term* eine Zahl. Diese wird erst durch ein Komma oder durch das Endezeichen „)“ beendet.

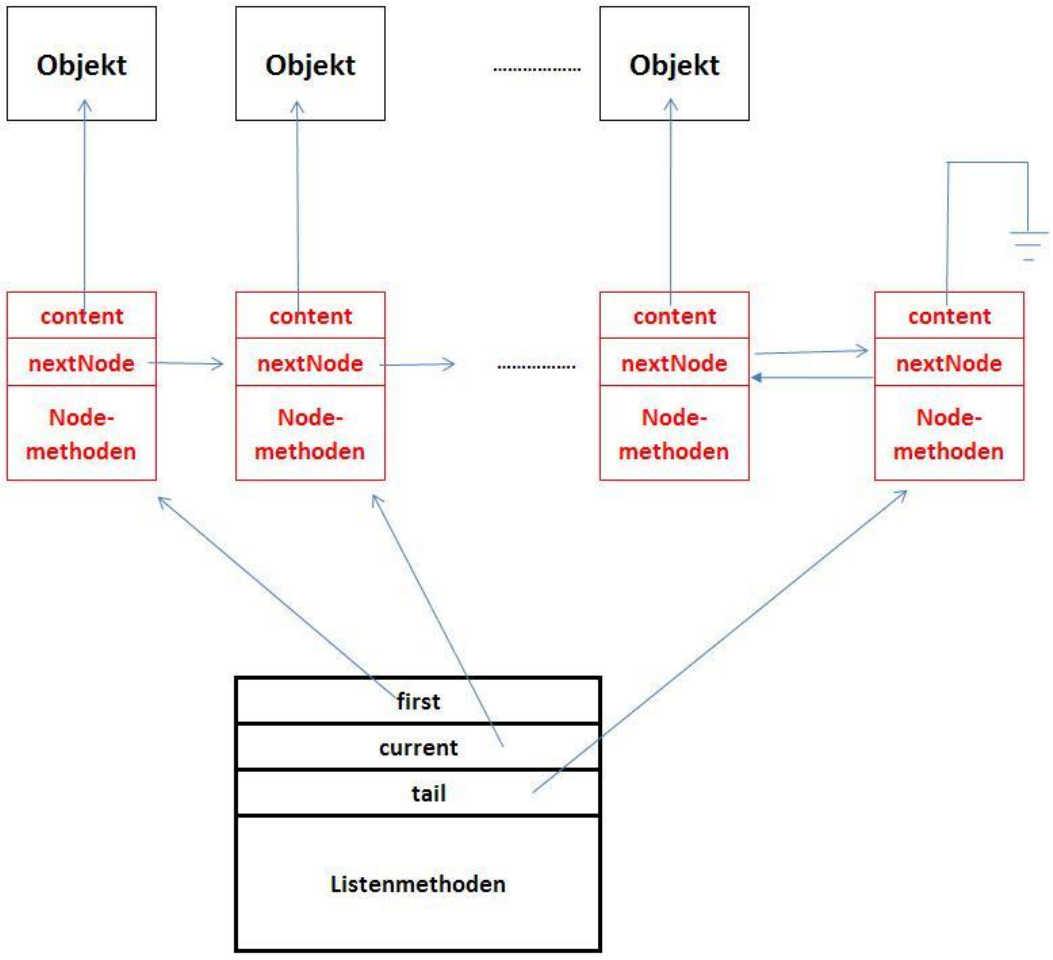
2. Der Eingabeterm soll nun auch ganze Zahlen (mit Vorzeichen) enthalten können.

Die Klasse *TList*

Eine lineare Liste ist ähnlich aufgebaut wie eine Schlange oder ein Stack. Der wesentliche Unterschied besteht darin, dass man bei einer Liste auch auf ein beliebiges Listenelement zugreifen kann. Es können auch Elemente mitten in der Liste gelöscht oder eingefügt werden.

Objekte der Klasse *TList* verwalten beliebige Objekte nach einem Listenprinzip. Ein interner Positionszeiger wird durch die Listenstruktur bewegt, seine Position markiert ein aktuelles Objekt. Die Lage des Positionszeigers kann abgefragt, verändert und die Objektinhalte an den Positionen können gelesen oder verändert werden.

Es gibt mehrere gute Methoden, eine Liste zu implementieren. Für die zentralen Abiturklausuren in NRW ab dem Jahr 2012 wurde nachfolgend beschriebene Implementation eingeführt:

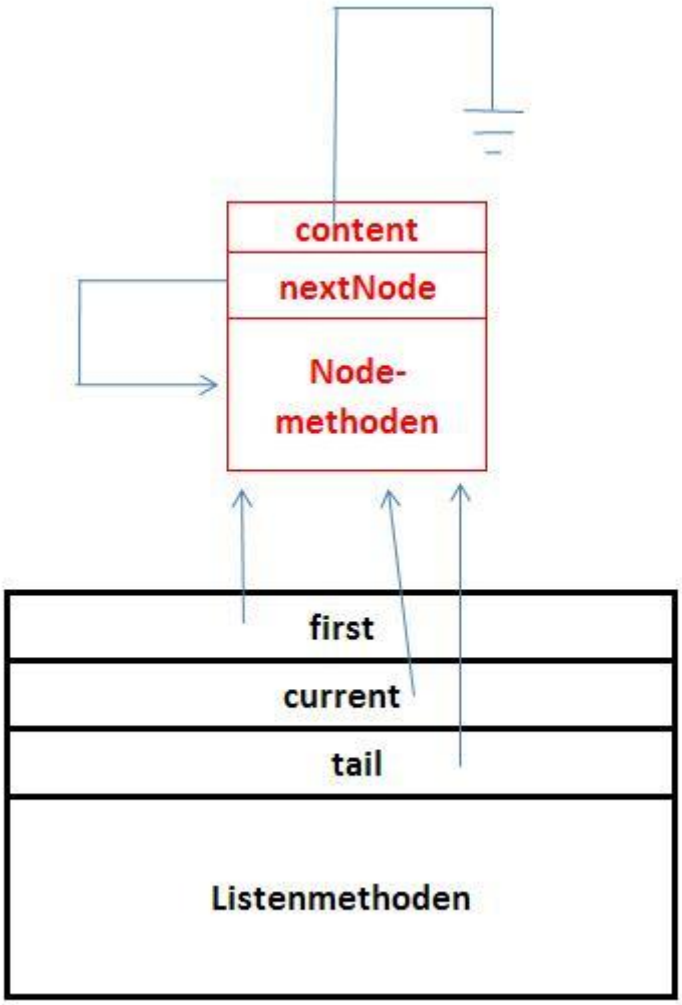


Wenn man ein Element an das Ende einer Liste anfügt, ist dies normalerweise programmiertechnisch deutlich umständlicher als beim Einfügen in der Mitte der Liste. Daher ist es üblich, ein sog. Dummy-Element zu benutzen. Dieser Knoten verweist auf keinen speziellen Inhalt, er dient nur zur Markierung des Listenendes. In dieser Implementation wird er mit *tail* bezeichnet.

Der Zeiger *tail* zeigt grundsätzlich immer nur auf das angefügte Dummy-Element der Liste. Bei nichtleerer Liste zeigt der Zeiger *first* immer auf das erste „wirkliche“ Listenelement. Der interne Positionszeiger *current* hingegen kann auf ein beliebiges Element (einschließlich Dummy-Element) zeigen. Zum Beispiel zeigt *current* nach dem Durchlaufen der Liste auf das Dummy-Element.

Das „wirklich existierende“ letzte Element einer Liste erreicht man durch *tail.nextNode* .

Eine existierende, aber leere Liste sieht so aus:



```
TNode = class    // damit ist TNode eine direkte Unterklasse von TObject
  private    { weil alles private ist, können die beiden Attribute und die
               sechs Methoden nur deshalb  von TList benutzt werden, weil
               TList in derselben Unit steht}
    content: TObject;
    nextNode: TNode;
    constructor create(pObject: TObject);
    procedure setContent(pObject: TObject);
    procedure setNext(pNext: TNode);
    function getContent: TObject;
    function getNext: TNode;
    destructor destroy; override;
  end;


implementation


constructor TNode.create(pObject: TObject);
begin
  content := pObject;
  nextNode := nil;   // überflüssig, weil Default-Einstellung
end;


procedure TNode.setContent(pObject: TObject);
{wird nur benötigt, wenn an der aktuellen Position das Inhaltsobjekt
  ausgetauscht werden soll   }
begin
  content := pObject;
end;


procedure TNode.setNext(pNext: TNode);
begin
  nextNode := pNext;
end;
```

```
function TNode.getContent: TObject;
begin
  result := content;
end;


function TNode.getNext: TNode;
begin
  result := nextNode;
end;


destructor TNode.destroy;
begin
  inherited destroy; // eigentlich überflüssig in DELPHI
end;
```

Objekte der Klasse ***TList*** verwalten beliebig viele, linear angeordnete Objekte. Auf höchstens ein Listenobjekt, aktuelles Objekt genannt, kann jeweils zugegriffen werden. Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt. Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden. Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt oder ein Listenobjekt an das Ende der Liste angefügt werden.

Dokumentation der Klasse TList

Konstruktor **`create`**
Eine leere Liste wird erzeugt.

Anfrage **`isEmpty: boolean`**
Die Anfrage liefert den Wert true, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert false.

Anfrage **`hasAccess: boolean`**
Die Anfrage liefert den Wert true, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert false.

Auftrag **`next`**
Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d.h. hasAccess liefert den Wert false.

Auftrag **`toFirst`**
Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Auftrag **`toLast`**
Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Anfrage **`getObject: TObject`**
Falls es ein aktuelles Objekt gibt (hasAccess = true), wird das aktuelle Objekt zurückgegeben, andernfalls (hasAccess = false) gibt die Anfrage den Wert nil zurück.

Auftrag **`setObject(pObject: TObject)`**
Falls es ein aktuelles Objekt gibt (hasAccess = true) und pObject ungleich nil ist, wird das aktuelle Objekt durch pObject ersetzt. Sonst bleibt die Liste unverändert.

Auftrag **`append(pObject: TObject)`**
Ein neues Objekt pObject wird am Ende der Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt pObject in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (hasAccess = false). Falls pObject gleich nil ist, bleibt die Liste unverändert.

Auftrag **`insert(pObject: TObject)`**
Falls es ein aktuelles Objekt gibt (hasAccess = true), wird ein neues Objekt vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Falls die Liste leer ist und es somit kein aktuelles Objekt gibt (hasAccess = false), wird pObject in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls es kein aktuelles Objekt gibt (hasAccess = false) und die Liste nicht leer ist oder pObject gleich nil ist, bleibt die Liste unverändert.

Auftrag **`concat(pList: TList)`**
Die Liste pList wird an die Liste angehängt. Anschließend wird pList eine leere Liste. Das aktuelle Objekt bleibt unverändert. Falls pList nil oder eine leere Liste ist, bleibt die Liste unverändert.

Auftrag **`remove`**
Falls es ein aktuelles Objekt gibt (hasAccess = true), wird das aktuelle Objekt aus der Liste gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt. Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr (hasAccess = false). Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (hasAccess = false), bleibt die Liste unverändert.

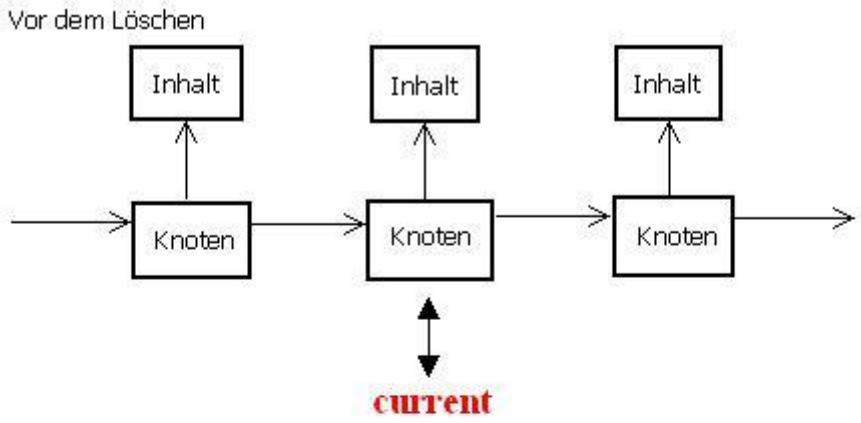
Destruktor **`destroy`**
Das Objekt der Klasse TList wird entfernt und der von dem Objekt verwendete Speicher wird wieder frei gegeben.

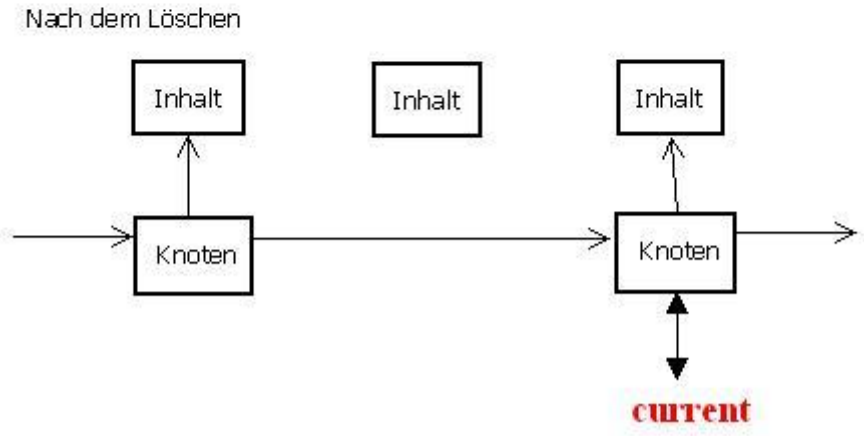
```
TList = class   // damit ist TList eine direkte Unterklasse von TObject
   protected
     first: TNode;
     tail: TNode;
     current: TNode;
   public
     constructor create; virtual;
     function isEmpty: boolean; virtual;
     function hasAccess: boolean; virtual;
     procedure next; virtual;
     procedure toFirst; virtual;
     procedure toLast; virtual;
     function getObject: TObject; virtual;
     procedure setObject(pObject: TObject); virtual;
     procedure append(pObject: TObject); virtual;
     procedure insert(pObject: TObject); virtual;
     procedure concat(pList: TList); virtual;
     procedure remove; virtual;
     destructor destroy; override;
   end;

implementation

constructor TList.create;
begin
  tail := TNode.create(nil); // dummy
  first := tail;
  tail.setNext(tail); // dummy.next toLast
  current := first;
end;

function TList.isEmpty: boolean;
begin
  result := first = tail;
end;
```

```
function TList.hasAccess: boolean;
begin
  result := (current <> tail);
end;

procedure TList.next;
begin
  if self.hasAccess then current := current.getNext;
end;

procedure TList.toFirst;
begin
  current := first;
end;

procedure TList.toLast;
begin
  current := tail.getNext;
end;

function TList.getObject: tObject;
begin
  result := current.getContent
end;

procedure TList.setObject(pObject: TObject);
begin
  if (pObject <> nil) and self.hasAccess then
    current.setContent(pObject)
end;

procedure TList.insert(pObject: TObject);
var  currentPos, aktPos, newNode, frontNode:TNode;
begin
  if pObject <> nil then begin
    if self.isEmpty then self.append(pObject)
    else begin
      if self.hasAccess then begin
        currentPos := current;
        aktPos := current;
        newNode := TNode.create(pObject);
        newNode.setNext(current);
        if aktPos = first then first := newNode
```

```
        else begin
          self.toFirst;
          frontNode := current;
          while self.hasAccess and
                          (current <> aktPos) do begin
            frontNode := current;
            self.next;
          end;
          frontNode.setNext(newNode);
        end;
        current := currentPos;
      end;
    end;
  end;
end;


procedure TList.append(pObject: TObject);
var  newNode, previousNode: TNode;
begin
  if pObject <> nil then begin
    newNode := TNode.create(pObject);
    newNode.setNext(tail);
    if self.isEmpty then first := newNode
    else begin
      previousNode := tail.getNext;
      previousNode.setNext(newNode);
    end;
    tail.setNext(newNode);
  end;
end;


procedure TList.concat(pList: TList);
var  lastNode,dummy: TNode;
begin
  if (pList <> nil) and not pList.isEmpty then begin
    if self.isEmpty then begin
      first := pList.first;
      dummy := tail;
      tail := pList.tail;
      current := tail;
    end
```

```
    else begin
      dummy := tail;
      lastNode := tail.getNext;
      tail := pList.tail;
      lastNode.setNext(pList.first);
      if current = dummy then current := tail
    end;
    dummy.destroy;

    pList.tail := TNode.create(nil);
    pList.first := pList.tail;
    pList.tail.setNext(pList.tail);
    pList.current := pList.first;
  end;
end;
```





```
procedure TList.remove;
var  currentPos, markPos, frontPos: TNode;
begin
  if self.hasAccess then begin
    currentPos := current;
    if current = first then begin
      first := current.getNext;
      if current.getNext = tail then
                                    tail.setNext(first);
      current := first;
    end
    else begin
      markPos := current;
      self.toFirst;
      frontPos := current;
      while self.hasAccess and (current <> markPos)
      do begin
        frontPos := current;
        self.next;
      end;
      frontPos.setNext(markPos.getNext);
      current := frontPos.getNext;
      if current = tail then tail.setNext(frontPos);
    end;
    currentPos.destroy;
  end;
end;


destructor TList.destroy;
begin
  while not self.isEmpty do begin
    self.toFirst;
    self.remove;
  end;
  tail.destroy;
  inherited destroy;
end;

end.
```

Aufgaben

Die nachfolgenden Aufgaben sollen nur mit den öffentlichen Listenmethoden gelöst werden! Es sollen keine weiteren Methoden implementiert werden!

1. Stelle von einer Liste A eine unabhängige Kopie B her! A und B sollen also nicht nur zwei unterschiedliche Namen für dieselbe Liste sein! Wenn man eine der beiden Listen löscht, soll die andere weiterhin existieren!

2. Schreibe das Programm für die im Folgenden abgebildete Anwendung!

Hinweis: Speichere die Liste in der externen Datei

Zahlendatei = file of INTEGER;

Problem ist hier, dass die Elemente der externen Datei eine feste Speichergröße haben müssen. Ein *file of TObject* gibt es also leider nicht.

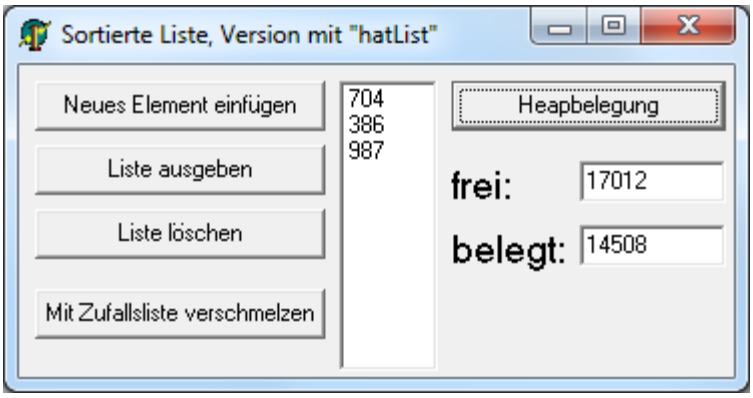
3. Geschwindigkeitsmessung

Erzeuge ein ARRAY mit 1000 Zufallszahlen. Kopiere dieses ARRAY in eine lineare Liste (und in eine externe Datei)! Sortiere nun jeweils und vergleiche die benötigten Zeiten!

4. Implementiere die Klasse *TQueue* als direkte Unterklasse von *TObject* derart, dass sie als privates Attribut ein Objekt vom Typ *TList* hat!

5. Implementiere die Klasse *TStack* als direkte Unterklasse von *TObject* derart, dass sie als privates Attribut ein Objekt vom Typ *TList* hat!

6. Implementiere eine Klasse *TsortierteListe* als direkte Unterklasse von *TObject* derart, dass sie als privates Attribut ein Objekt vom Typ *TList* hat! Die alten Methoden *setObject* und *append* gibt es nicht mehr und *insert* wird neu angepasst. Die Methode *concat* wird ersetzt durch die neue Methode *meltwith(pList: TsortierteListe)* . Diese Klasse TsortierteListe soll nur Integer-Zahlen verwalten können. Teste deine Implementation mit folgendem Hauptprogramm:



7. Implementiere die Klasse *TQueue* als Unterklasse von *TList* !

8. Implementiere die Klasse *TStack* als Unterklasse von *TList* !

9. Implementiere wie in Aufgabe 6 eine Klasse *TsortierteListe,* diesmal aber als direkte Unterklasse von *TList* !

Lösungen

Aufgabe 2

```
unit mHaupt;
interface
uses  ……..mIntegerElement, mList;

type
  TMain = class(TForm)
    BtZufallsListeErzeugen, BtSortieren: TButton;
    AusgabeBox: TListBox;
    BtListenAusgabe, BtSuchen: TButton;
    BtLoeschen, BtSpeichern, BtLaden: TButton;
    EdAnzahl, EdSuch, EdLoesch: TEdit;
    BtLoesch, BtNteZahl: TButton;
    EdNteZahlEingabe, EdNteZahlAusgabe: TEdit;
    EdVornEinfuegen, BtVornEinfuegen: TButton;
    EdHintenAnhaengen: TEdit;
    BtHintenAnhaengen, BtElementeanzahl: TButton;

    procedure BtZufallsListeErzeugenClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure BtListenAusgabeClick(Sender: TObject);
    procedure BtLoeschenClick(Sender: TObject);
    procedure BtSpeichernClick(Sender: TObject);
    procedure BtLadenClick(Sender: TObject);
    procedure BtSortierenClick(Sender: TObject);
    procedure BtVornEinfuegenClick(Sender: TObject);
    procedure BtHintenAnhaengenClick(Sender: TObject);
    procedure BtElementeanzahlClick(Sender: TObject);
    procedure BtSuchenClick(Sender: TObject);
    procedure BtLoeschClick(Sender: TObject);
    procedure BtNteZahlClick(Sender: TObject);
  end;

var  Main: TMain;
```

```
    Liste: TList;
    Zahlendatei: File of INTEGER;


Implementation
{$R *.dfm}

procedure TMain.BtZufallsListeErzeugenClick(Sender: TObject);
VAR e: TIntegerElement;
    i:  INTEGER;
begin
  If Liste = NIL THEN Liste := TList.create;
  FOR i := 1 TO 7 DO BEGIN
    e := TIntegerElement.create(random(100));
    Liste.append(e)
  END
end;


procedure TMain.FormCreate(Sender: TObject);
begin
  Liste := NIL;
  randomize;
  AssignFile(Zahlendatei, 'Integerdatei');
end;


procedure TMain.BtListenAusgabeClick(Sender: TObject);
VAR e: TElement;
begin
  Ausgabebox.Clear;
  IF Liste <> NIL THEN BEGIN
    Liste.toFirst;
    While Liste.hasAccess DO BEGIN
      e := Liste.getObject as TElement;
      AusgabeBox.Items.Add(IntToStr(e.getInhalt));
      Liste.next
    END
```

```
  END
end;



procedure TMain.BtLoeschenClick(Sender: TObject);
begin
  IF Liste <> NIL THEN BEGIN
    Liste.destroy;
    Liste := NIL
  END;
  AusgabeBox.Clear
end;



procedure TMain.BtSpeichernClick(Sender: TObject);
VAR e: TIntegerElement;
    zahl: INTEGER;
begin
  IF Liste <> NIL THEN BEGIN
    Rewrite(Zahlendatei);
    Liste.toFirst;
    While Liste.hasAccess DO BEGIN
      e := Liste.getObject as TIntegerElement;
      zahl := e.getInhalt;
      write(Zahlendatei, zahl);
      Liste.next
    END;
    CloseFile(Zahlendatei)
  END
end;



procedure TMain.BtLadenClick(Sender: TObject);
VAR e: TIntegerElement;
    zahl: INTEGER;
begin
  If Liste <> NIL THEN Liste.destroy;
  Liste := TList.create;
  IF NOT FileExists('Integerdatei') THEN
              Showmessage('Es gibt noch keine Datei')
```

```
  ELSE BEGIN
    Reset(Zahlendatei);
    WHILE NOT EOF(Zahlendatei) DO BEGIN
      Read(Zahlendatei, zahl);
      e := TIntegerElement.create(zahl);
      Liste.append(e)
    END;
    CloseFile(Zahlendatei)
  END // of Else
end;


procedure TMain.BtSortierenClick(Sender: TObject);
VAR hilfliste: TList;
    e: TIntegerElement;
begin
  IF Liste = NIL THEN showmessage('Liste ist leer')
  ELSE BEGIN
    hilfliste := TList.create;
    Liste.toFirst;
    WHILE Liste.hasAccess DO BEGIN
       e := Liste.getObject as TIntegerElement;
       IF hilfliste.isEmpty THEN hilfliste.Append(e)
       ELSE BEGIN
         hilfliste.toFirst;
         WHILE hilfliste.hasAccess AND (e.getInhalt >
         (hilfliste.getObject AS TElement).getInhalt)
          DO hilfliste.Next;
         IF hilfliste.hasAccess THEN hilfliste.Insert(e)
         ELSE hilfliste.Append(e);
       END;
       Liste.Remove
    END; // of WHILE
    Liste.destroy;
    Liste := hilfliste
  END // of Else
end;
```

```
procedure TMain.BtVornEinfuegenClick(Sender: TObject);
VAR e: TIntegerElement;
    zahl: INTEGER;
begin
  zahl := StrToInt(EdVornEinfuegen.Text);
  e := TIntegerElement.create(Zahl);
  Liste.toFirst;
  Liste.insert(e);
end;

procedure TMain.BtHintenAnhaengenClick(Sender: TObject);
VAR e: TIntegerElement;
    zahl: INTEGER;
begin
  zahl := StrToInt(EdHintenAnhaengen.Text);
  e := TIntegerElement.create(Zahl);
  Liste.append(e);
end;


procedure TMain.BtElementeanzahlClick(Sender: TObject);
VAR count: INTEGER;
begin
  count := 0;
  IF Liste <> NIL THEN BEGIN
    Liste.toFirst;
    WHILE Liste.hasAccess DO BEGIN
      INC(count);
      Liste.next
    END
  END;
  EdAnzahl.Text := IntToStr(count)
end;


procedure TMain.BtSuchenClick(Sender: TObject);
VAR e: TIntegerELement;
```

```
    zahl: INTEGER;
    gefunden: BOOLEAN;
begin
  zahl := StrToInt(EdSuch.Text);
  IF Liste <> NIL THEN BEGIN
    Liste.toFirst;
    gefunden := False;
    While Liste.hasAccess AND NOT gefunden DO
    BEGIN
       e := Liste.getObject as TIntegerElement;
       IF e.getInhalt = zahl THEN gefunden := TRUE;
       Liste.next
    END;
    If gefunden THEN Showmessage('Die Zahl '
                    + EdSuch.text + ' ist enthalten')
    ELSE Showmessage('Die Zahl ' + EdSuch.text +
                                ' ist nicht enthalten')
  END // of IF Liste <> NIL
end;


procedure TMain.BtLoeschClick(Sender: TObject);
VAR e: TIntegerELement;
    zahl: INTEGER;
begin
  zahl := StrToInt(EdLoesch.Text);
  IF Liste <> NIL THEN BEGIN
    Liste.toFirst;
    While Liste.hasAccess DO BEGIN
       e := Liste.getObject as TIntegerElement;
       IF e.getInhalt = zahl THEN Liste.remove;
       Liste.next
    END;
  END // of IF Liste <> NIL
end;
```

```
procedure TMain.BtNteZahlClick(Sender: TObject);
VAR zahl, i: INTEGER;
    e: TIntegerElement;
begin
  IF Liste = NIL THEN Showmessage('Die Liste ist leer')
  ELSE BEGIN
       zahl := StrToInt(EdNteZahlEingabe.Text);
       Liste.toFirst;
       {Vorsicht: zahl <= Elementeanzahl?}
       FOR i := 1 TO zahl -1 Do Liste.next;
       e := Liste.getObject as TIntegerElement;
       EdNteZahlAusgabe.Text := IntToStr(e.getInhalt)
  END  // of ELSE
end;

end.
```

Aufgabe 4

```
unit mSchlange;
interface
USES mList;

type
  TQueue = class   // damit ist TQueue eine direkte Unterklasse von TObject
  private
    hatList: TList;
  public
    constructor create;
    procedure enqueue(pObject: TObject);
    procedure dequeue;
    function front: TObject;
    function isEmpty: boolean;
    destructor destroy; override;
  end;

implementation

constructor TQueue.create;
begin
  inherited create;   // eigentlich überflüssig
  hatList:= TList.create;
end;


procedure TQueue.enqueue(pObject:TObject);
BEGIN
  hatList.append(pObject)
END;

procedure TQueue.dequeue;
BEGIN
  hatList.toFirst;
  hatList.remove
END;
```

```
function TQueue.front: TObject;
BEGIN
  hatList.toFirst;
  Result := hatList.getObject
END;


function TQueue.isEmpty: boolean;
begin
  Result := hatList.isEmpty;
end;


destructor TQueue.destroy;
BEGIN
  hatList.destroy;
  inherited destroy
END;

end.
```

Aufgabe 5

```
unit mStack;
interface
USES mList;

type
  TStack = class  // damit ist TStack eine direkte Unterklasse von TObject
  private
    hatList: TList;
  public
    constructor create;
    procedure push(pObject: TObject);
    procedure pop;
    function top: TObject;
    function isEmpty: boolean;
    destructor destroy; override;
  end;

implementation

constructor TStack.create;
begin
  inherited create;  // eigentlich überflüssig
  hatList:= TList.create;
end;


procedure TStack.push(pObject:TObject);
BEGIN
  hatList.toFirst;
  hatList.insert(pObject)
END;
```

```
procedure TStack.pop;
BEGIN
  hatList.toFirst;
  hatList.remove
END;


function TStack.top: TObject;
BEGIN
  hatList.toFirst;
  Result := hatList.getObject
END;


function TStack.isEmpty: boolean;
begin
  Result := hatList.isEmpty;
end;


destructor TStack.destroy;
BEGIN
  hatList.destroy;
  inherited destroy
END;

end.
```

Aufgabe 6

```
unit mSortierteListe;
interface
USES mList, mElement;

type
  TsortierteListe = class(TObject)
  private
    hatList: TList;
  public
     constructor create; virtual;
     function isEmpty: boolean; virtual;
     function hasAccess: boolean; virtual;
     procedure next; virtual;
     procedure toFirst; virtual;
     procedure toLast; virtual;
     function getObject: TObject; virtual;
     procedure insert(pObject: TObject); virtual;
     procedure remove; virtual;
     procedure meltWith(pList: TsortierteListe);
     destructor destroy; override;
  end;


implementation

constructor TsortierteListe.create;
BEGIN
  inherited create;
  hatList := TList.create
END;


function TsortierteListe.isEmpty: boolean;
BEGIN
  Result := hatList.isEmpty
END;
```

```
function TsortierteListe.hasAccess: boolean;
BEGIN
  Result := hatList.hasAccess
END;


procedure TsortierteListe.next;
BEGIN
  hatList.next
END;


procedure TsortierteListe.toFirst;
BEGIN
  hatList.toFirst
END;


procedure TsortierteListe.toLast;
BEGIN
  hatList.toLast
END;


function TsortierteListe.getObject: TObject;
BEGIN
  RESULT := hatList.getObject
END;


procedure TsortierteListe.insert(pObject: TObject);
Var zahl, neueZahl: INTEGER;
BEGIN
  IF hatList.isEmpty THEN hatList.insert(pObject)
  ELSE BEGIN
    hatList.toFirst;
    neueZahl := (pObject as TElement).getInhalt;
    REPEAT
```

```
      zahl := (hatList.GetObject as TElement).getInhalt;
      IF neueZahl <= zahl THEN hatList.insert(pObject)
      ELSE hatList.next
    UNTIL (neueZahl <= zahl) OR NOT hatList.hasAccess;
    IF NOT hatList.hasAccess THEN hatList.append(pObject)
  END // of ELSE
END;


procedure TSortierteListe.meltWith(pList:
TsortierteListe);
BEGIN
  pList.toFirst;
  WHILE pList.hasAccess DO BEGIN
     self.insert(pList.getObject);
     pList.next
  END
END;


procedure TsortierteListe.remove;
BEGIN
  hatList.remove
END;


destructor TsortierteListe.destroy;
BEGIN
  hatList.destroy;
  inherited destroy
END;


end.
```

Aufgabe 8

```
unit mStack;
interface
USES mList;

type
  TStack = class(TList)
   public
    procedure push(pObject: TObject);
    procedure pop;
    function top: TObject;

    // folgende Methoden müssen noch deaktiviert werden:
    function hasAccess: boolean;
    procedure next;
    procedure toFirst;
    procedure toLast;
    function getObject: TObject;
    procedure setObject(pObject: TObject);
    procedure append(pObject: TObject);
    procedure insert(pObject: TObject);
    procedure concat(pList: TList);
    procedure remove;
  end;


implementation

procedure TStack.push(pObject:TObject);
BEGIN
  inherited toFirst;
  inherited insert(pObject)
END;
```

```
procedure TStack.pop;
BEGIN
  inherited toFirst;
  inherited remove
END;


function TStack.top: TObject;
BEGIN
  inherited toFirst;
  Result := inherited getObject
END;


function TStack.hasAccess: boolean;
Begin
  // wird hiermit deaktiviert
END;


procedure TStack.next;
Begin
  // wird hiermit deaktiviert
END;


procedure TStack.toFirst;
Begin
  // wird hiermit deaktiviert
END;


procedure TStack.toLast;
Begin
  // wird hiermit deaktiviert
END;
```

```
function TStack.getObject: TObject;
Begin
   // wird hiermit deaktiviert
END;


procedure TStack.setObject(pObject: TObject);
Begin
   // wird hiermit deaktiviert
END;


procedure TStack.append(pObject: TObject);
Begin
   // wird hiermit deaktiviert
END;


procedure TStack.insert(pObject: TObject);
Begin
   // wird hiermit deaktiviert
END;


procedure TStack.concat(pList: TList);
Begin
   // wird hiermit deaktiviert
END;


procedure TStack.remove;
Begin
   // wird hiermit deaktiviert
END;

end.
```

Aufgabe 9

```
unit mSortierteListe;

interface
USES mList, mElement;
type
  TsortierteListe = class(TList)
  public
    procedure insert(pObject: TObject); override;
    procedure meltWith(pList: TsortierteListe);

    // Folgende Methoden müssen deaktiviert werden:
    procedure setObject(pObject: TObject); override;
    procedure append(pObject: TObject); override;
    procedure concat(pList: TList); override;
  end;

implementation

procedure TsortierteListe.insert(pObject: TObject);
Var zahl, neueZahl: INTEGER;
BEGIN
  IF self.isEmpty THEN inherited append(pObject)
  {wichtig: hier darf nicht inherited insert aufgerufen werden, weil diese
   Methode wiederum die Methode append aufruft, welche hier in dieser
   Unterklasse deaktiviert wurde.}
  ELSE BEGIN
     toFirst;
     neueZahl := (pObject as TElement).getInhalt;
     REPEAT
       zahl:= (self.getObject as TElement).getInhalt;
       IF neueZahl <= zahl THEN inherited insert(pObject)
       ELSE self.next
     UNTIL (neueZahl <= zahl) OR NOT self.hasAccess;
     IF NOT self.hasAccess THEN inherited append(pObject)
  END // of ELSE
END;
```

```
procedure TsortierteListe.meltWith(pList:TsortierteListe);
BEGIN
  pList.toFirst;
  WHILE pList.hasAccess DO BEGIN
     self.insert(pList.getObject);
     pList.next
  END
END;


procedure TsortierteListe.setObject(pObject: TObject);
BEGIN
  // wird hiermit deaktiviert
END;

procedure TsortierteListe.append(pObject: TObject);
BEGIN
  // wird hiermit deaktiviert
END;


procedure TsortierteListe.concat(pList: TList);
BEGIN
  // wird hiermit deaktiviert
END;

end.
```

Projekt Vokabelliste

Erstelle folgendes Projekt, welches eine Liste von Vokabeln verwaltet!

Aufgaben

1. Erweitere das Programm so, dass man nicht nur nach englischen sondern auch nach deutschen Begriffen Begriffen suchen kann.

2. Die Vokabelliste soll auch nach deutschen Begriffen sortiert werden.

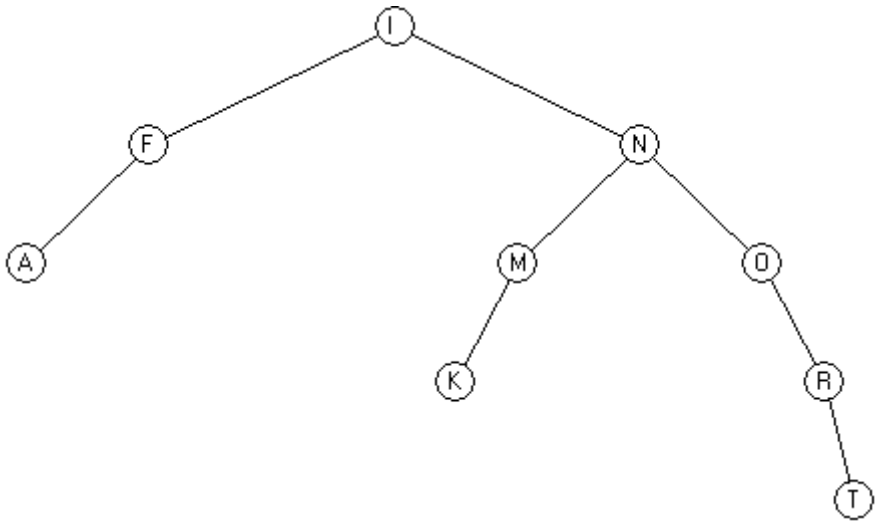
3. Beim Beenden des Programms (Schließen des Hauptformulars) wird zwar üblicherweise die Liste gelöscht, aber die einzelnen Elemente der Liste werden nicht aus dem Speicher entfernt. Sinnvollerweise sollte man dies unter Verwendung des *OnDestroy*-Ereignisses des Hauptformulars tun.

4. Die Vokabelliste soll
   a) in eine externe Datei namens *Vokabel.dat* gespeichert werden
   b) aus der externen Datei *Vokabel.dat* wieder eingelesen werden können

Nicht-Lineare Strukturen

Binärbäume

Die einzelnen Elemente eines Baumes heißen ***Knoten***, wobei die untersten Knoten keine Nachfolger haben und als ***Blätter*** bezeichnet werden.
Der oberste Knoten heißt auch ***Wurzel*** des Baumes.
Jeder ***innere Knoten*** eines Baumes hat somit mindestens einen Nachfolger.
Die Tiefe bzw. Höhe eines Baumes wird in der Literatur leider nicht immer identisch festgelegt. Wir definieren: Unter der ***Tiefe eines Knotens*** versteht man die Anzahl der ***Kanten*** von diesem Knoten bis hoch zur Wurzel. Die Wurzel selbst besitzt die Tiefe 0.
Als ***Höhe des Baumes*** bezeichnet man die Tiefe des Blattes mit der größten Tiefe. Die Höhe eines leeren Baumes soll **-1** sein.
Jeder Knoten besitzt einen (evtl. leeren) linken und rechten ***Teilbaum***.



Der obige Beispielsbaum besitzt 9 Knoten und 3 Blätter. Er besitzt die Höhe 4. Die Wurzel enthält den Buchstaben „I“.
Dieser Beispielsbaum ist ein sog. ***geordneter Baum*** oder ***Suchbaum***: Wenn man irgendeinen beliebigen Knoten betrachtet, so sind alle Elemente des linken Teilbaumes kleiner, und alle Elemente des rechten Teilbaumes größer als der Knoten selbst.
Der obige Beispielsbaum ist ein Suchbaum und entsteht übrigens, wenn man das Wort *„INFORMATIK“* buchstabenweise in einen zunächst leeren Binärbaum einordnet. Ein Suchbaum enthält jede Information (z.B. den Buchstaben „I“) nur einmal.

Aufgaben

1. Zeichne alle möglichen Binärbaumstrukturen mit drei Knoten!

2. Gegeben sei ein Binärbaum der Höhe *n*. Erinnerung: ein Binärbaum, welcher nur aus der Wurzel besteht, hat die Höhe 0.
Wie viele Knoten besitzt dieser Baum mindestens und höchstens?

An obigem Binärbaum kann man auch schon unsere nächsten zu lösenden Aufgaben im Unterricht erkennen:

- Wie stellt man einen Binärbaum graphisch dar?
- Wie kann man feststellen, ob der Binärbaum ein bestimmtes Element enthält (Suchen)?
- Wie kann man ein Element löschen?
- Wie kann man ein neues Element einfügen?
- Wie kann man einen Binärbaum extern speichern?
- In welcher Reihenfolge soll man einen Binärbaum durchlaufen?
- Wie kann man einen Binärbaum optimieren? (Das Wort *Informatik* lässt sich sicherlich auch in einem kleineren Binärbaum darstellen!)

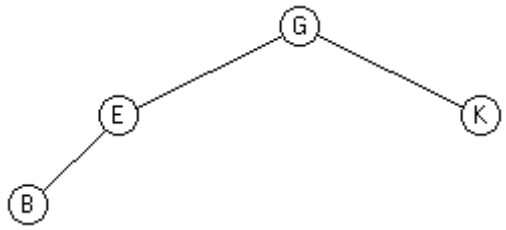
Definitionen:

(Adelson, Velskii und Landis):
Ein Baum ist genau dann *ausgeglichen* oder *ausgewogen*, wenn sich für jeden Knoten die Tiefen der beiden Teilbäume um höchstens 1 unterscheiden.

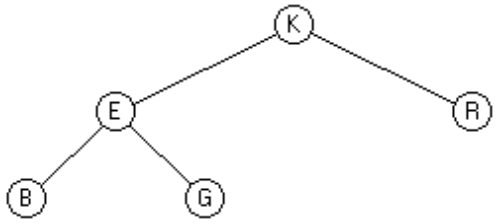
Derartige Binärbäume werden ***AVL-Bäume*** genannt.

Ein Binärbaum heißt *vollständig ausgeglichen* bzw. *vollständig ausgewogen*, wenn sich für jeden Knoten die Zahl der Knoten in seinem linken und rechten Teilbaum um höchstens 1 unterscheiden.

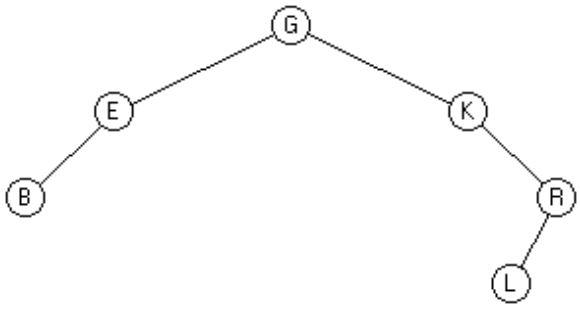
Beispiel a)



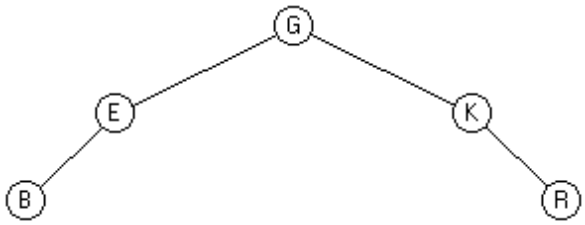
Beispiel b)



Beispiel c)



Beispiel d)

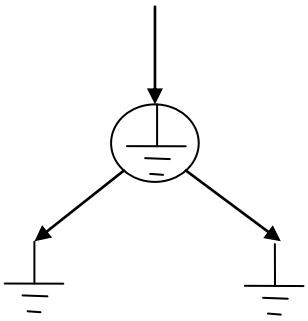


Aufgabe: Zeichne einen Binärbaum mit 11 Knoten, der
a) ausgewogen, aber nicht vollständig ausgewogen ist.
b) vollständig ausgewogen ist.

Im Folgenden wollen wir uns zunächst mit ungeordneten Binärbäumen beschäftigen.

Die Klasse *TBinaryTree*

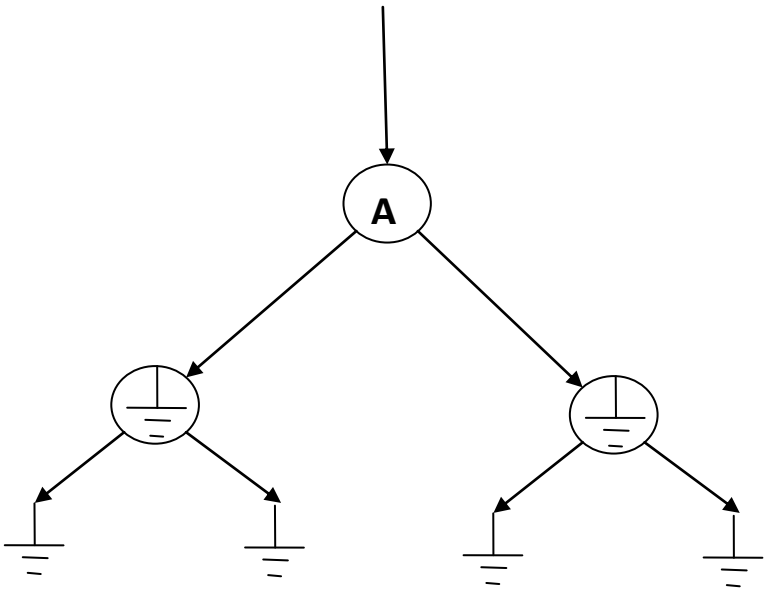
Bemerkung: Die folgende Dokumentation und Implementation entspricht den Abiturvorgaben NRW für das Jahr 2012. Leider entspricht sie nicht dem auf dem Softwaremarkt Üblichen. Dies hat auch Folgen für die Lösungswege der anschließenden Übungsaufgaben.



Ein leerer Binärbaum wird in dieser Implementation wie nebenstehend dargestellt realisiert. Sowohl der Inhalt als auch die beiden Unterbäume zeigen auf *nil*. Das zugehörige Objekt für diesen leeren Baum ist allerdings nicht NIL. Ein leerer Baum ist auch ein Baum!

Ein Baum wird als leer erkannt, wenn sein Inhalt *nil* ist. Es gibt keine Knoten, deren Inhalt *nil* ist und deren Unterbäume ungleich *nil* sind!

Alle Blätter eines Baumes haben einen **leeren** linken und einen **leeren** rechten Unterbaum. Insbesondere sind diese beiden Teilbäume also nicht *nil*.



Mithilfe der Klasse *TBinaryTree* können beliebig viele Inhaltsobjekte in einem Binärbaum verwaltet werden. Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse *TBinaryTree* sind.

Dokumentation der Klasse TBinaryTree

Konstruktor **create**
Nach dem Aufruf existiert ein leerer Binärbaum.

Konstruktor **create (pObject: TObject)**
Wenn der Parameter pObject ungleich *nil* ist, existiert nach dem Aufruf der Binärbaum und hat pObject als Inhaltsobjekt und zwei leere Teilbäume. Falls der Parameter *nil* ist, wird ein leerer Binärbaum erzeugt.

Konstruktor **create(pObject: TObject; pLeftTree, pRightTree: TBinaryTree)**
Wenn der Parameter pObject ungleich *nil* ist, wird ein Binärbaum mit pObject als Inhaltsobjekt und den beiden Teilbäume pLeftTree und pRightTree erzeugt. Sind pLeftTree oder pRightTree gleich *nil*, wird der entsprechende Teilbaum als leerer Binärbaum eingefügt. Wenn der Parameter pObject gleich *nil* ist, wird ein leerer Binärbaum erzeugt.

Anfrage **isEmpty**
Diese Anfrage liefert den Wahrheitswert ***true***, wenn der Binärbaum leer ist, sonst liefert sie den Wert ***false***.

Auftrag **setObject(pObject: TObject)**
Wenn der Binärbaum leer ist, wird der Parameter pObject als Inhaltsobjekt sowie ein leerer linker und rechter Teilbaum eingefügt. Ist der Binärbaum nicht leer, wird das Inhaltsobjekt durch pObject ersetzt. Die Teilbäume werden dann nicht geändert. Wenn pObject *nil* ist, bleibt der ganze Binärbaum unverändert.

Anfrage **getObject: TObject**
Diese Anfrage liefert das Inhaltsobjekt des Binärbaums. Wenn der Binärbaum leer ist, wird *nil* zurückgegeben.

Auftrag **setLeftTree (pTree: TBinaryTree)**
Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als linken Teilbaum. Falls der Parameter *nil* ist, ändert sich nichts.

Auftrag **setRightTree (pTree: TBinaryTree)**
Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als rechten Teilbaum. Falls der Parameter *nil* ist, ändert sich nichts.

Anfrage **getLeftTree: TBinaryTree**
Diese Anfrage liefert den linken Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird *nil* zurückgegeben.

Anfrage **getRightTree: TBinaryTree**
Diese Anfrage liefert den rechten Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird *nil* zurückgegeben.

Destruktor **destroy**
Das Objekt der Klasse TBinaryTree wird entfernt und der von dem Objekt verwendete Speicher wird wieder frei gegeben.

Materialien zu den zentralen Abiturpruefungen im Fach Informatik ab 2012 in Nordrhein-Westfalen.
Klasse TBinaryTree
NW-Arbeitsgruppe:
Materialentwicklung zum Zentralabitur im Fach Informatik
Version 2010-12-28

```
unit mBinaryTree;

interface

type
  TBinaryTree = class
    private
      content: TObject;
      leftTree, rightTree: TBinaryTree;
    public
      constructor create; overload; virtual;
      constructor create(pObject: TObject); overload;
                                              virtual;
      constructor create(pObject: TObject; pLeftTree,
         pRightTree: TBinaryTree); overload; virtual;
      function isEmpty: boolean; virtual;
      procedure setObject(pObject: TObject); virtual;
      function getObject: TObject; virtual;
      procedure setLeftTree(pTree: TBinaryTree);virtual;
      procedure setRightTree(pTree: TBinaryTree);virtual;
      function getLeftTree: TBinarytree; virtual;
      function getRightTree: TBinaryTree; virtual;
      procedure removeTree(pTree: TBinaryTree);//nicht in der
      procedure removecompletely;   //Dokumentation vorhanden
      destructor destroy; override;
  end;
```

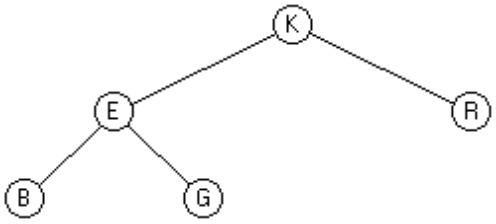
```
implementation

  constructor TBinaryTree.create;
  begin
   content := nil;     // eigentlich überflüssig, weil  das
   leftTree := nil;    // automatisch gemacht wird
   rightTree := nil;
  end;


  constructor TBinaryTree.create(pObject: TObject);
  begin
    if pObject = nil then begin
     content := nil;
     leftTree := nil;
     rightTree := nil;
    end
    else begin  // die Unterbäume werden als leer definiert
           content := pObject;
           leftTree := TBinaryTree.create;
           rightTree := TBinaryTree.create;
         end;
  end;


  constructor TBinaryTree.create
    (pObject: TObject; pLeftTree,pRightTree:TBinaryTree);
  begin
    if pObject = nil then begin
      content := nil;
      leftTree := nil;
      rightTree := nil;
    end
    else begin
           content := pObject;
           if pLeftTree <> nil
             then leftTree := pLeftTree
```

```
          else leftTree := TBinaryTree.create;
          if pRightTree <> nil
            then rightTree := pRightTree
          else rightTree := TBinaryTree.create;
    end;
  end;


  function TBinaryTree.isEmpty: boolean;
  begin
    result := content = nil;
  end;


  procedure TBinaryTree.setObject(pObject: TObject);
  begin
    if pObject <> nil then begin
      if self.isEmpty then begin
         leftTree := TBinaryTree.create;
         rightTree := TBinaryTree.create;
      end;
      content := pObject;
    end;
  end;


procedure TBinaryTree.setEmpty;
//wird nicht in der Dokumentation erwähnt
  begin
   content := nil;
   if leftTree <> nil then leftTree.destroy;
   leftTree := nil;
   if rightTree <> nil then rightTree.destroy;
   rightTree := nil;
  end;
```

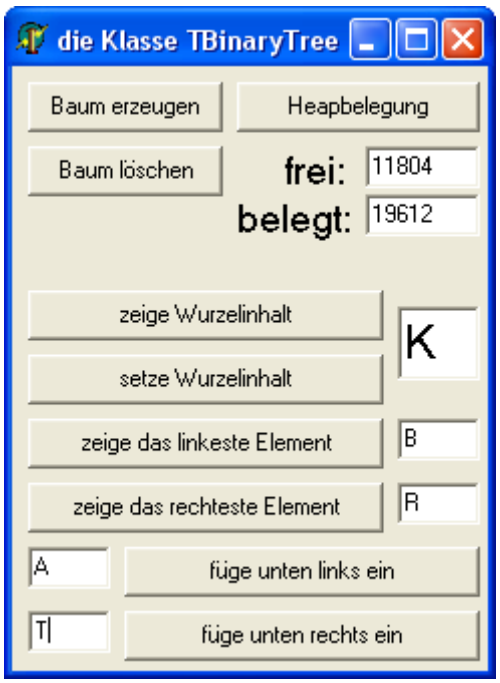
```
function TBinaryTree.getObject: TObject;
begin
  result := content;
end;


procedure TBinaryTree.setLeftTree(pTree: TBinaryTree);
begin
  if not self.isEmpty and (pTree <> nil)
    then leftTree := pTree;
end;


procedure TBinaryTree.setRightTree(pTree:TBinaryTree);
begin
  if not self.isEmpty and (pTree <> nil)
    then rightTree := pTree;
end;


function TBinaryTree.getLeftTree: TBinaryTree;
begin
  if not self.isEmpty then result := leftTree
  else result := nil;
end;


function TBinaryTree.getRightTree: TBinaryTree;
begin
  if not self.isEmpty then result := rightTree
  else result := nil;
end;


procedure TBinaryTree.removeTree(pTree: TBinaryTree);
begin
  if pTree <> nil then begin
    if not pTree.isEmpty then begin
```

```
        removeTree(pTree.getLeftTree);
        removeTree(pTree.getRightTree);
        if pTree.getLeftTree.isEmpty and
                      pTree.getRightTree.isEmpty then
        pTree.setEmpty;
      end;
    end;
  end;


  procedure TBinaryTree.removeCompletely;
  begin
    self.removeTree(self);
  end;


  destructor TBinaryTree.destroy;
  begin
   if leftTree <> NIL THEN leftTree.destroy;
   if rightTree <> NIL THEN rightTree.destroy;
   inherited destroy
  end;

end.
```

Um die Methoden der Klasse *TBinaryTree* kennenzulernen, implementieren wir eine neue Klasse *TZeichenElement*, welche einzelne Buchstaben enthalten kann. Da die Klasse *TBinaryTree* noch keine Ordnung kennt, gibt es auch noch keine sinnvolle Einfügemethode. Deshalb muss jeder Baum noch umständlich zusammengesetzt werden. Im folgenden Beispielprogramm soll mit dem nachstehend dargestellten Baum gearbeitet werden:



Die Konstruktion dieses Baumes geschieht so:
Erzeuge zuerst die beiden Einzelbäume B und G, danach den Baum E! Analog wird dann der Baum R erzeugt und zuletzt der Baum K.



Um zu überprüfen, ob die implementierte Klasse *TBinaryTree* auch wie gewünscht funktioniert, programmiere das nebenstehende Hauptprogramm!

Die beiden letzten Button sollen jeweils einen weiteren Knoten in den Baum einfügen. Ob das geklappt hat, kann man anschließend z.B. mit dem Button „zeige das linkeste Element“ überprüfen.

```
unit mHaupt;

interface
uses  ……, mElement, mBinaryTree;

type
  TMain = class(TForm)
    BtBaumErzeugen, BtHeapbelegung: TButton;
    Label1, Label2: TLabel;
    EdFrei, EdBelegt, EdWurzel, EdMin, EdMax: TEdit;
    EdUL, EdUR: TEdit;
    BtLoeschen, BtZeigeWurzel, BtClear: TButton;
    BtSetzeWurzel, BtLinkestesElement: TButton;
    BtRechtestesElement: TButton;
    BtFuegeUntenLinksEin, BtFuegeUntenRechtsEin: TButton;
    procedure BtBaumErzeugenClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure BtHeapbelegungClick(Sender: TObject);
    procedure BtLoeschenClick(Sender: TObject);
    procedure BtZeigeWurzelClick(Sender: TObject);
    procedure BtSetzeWurzelClick(Sender: TObject);
    procedure BtLinkestesElementClick(Sender: TObject);
    procedure BtRechtestesElementClick(Sender: TObject);
    procedure BtFuegeUntenLinksEinClick(Sender:TObject);
    procedure BtFuegeUntenRechtsEinClick(Sender:TObject);
  end;

var
  Main: TMain;
  Baum: TBinaryTree;

implementation
{$R *.dfm}
```

```
procedure TMain.BtBaumErzeugenClick(Sender: TObject);
VAR e: TZeichenElement;
    baumB, baumG, baumE, baumR: TBinaryTree;
begin
  e := TZeichenElement.create('B');
  baumB := TBinaryTree.create(e);
  e := TZeichenElement.create('G');
  baumG := TBinaryTree.create(e);
  e := TZeichenElement.create('E');
  baumE := TBinaryTree.create(e, baumB, baumG);
  e := TZeichenElement.create('R');
  baumR := TBinaryTree.create(e);
  e := TZeichenElement.create('K');
  baum := TBinaryTree.create(e, baumE, baumR);
end;


procedure TMain.FormCreate(Sender: TObject);
begin
  baum := NIL;
end;


procedure TMain.BtHeapbelegungClick(Sender: TObject);
VAR frei, belegt: Cardinal;
    HSt: THeapStatus;
begin
  HST := GetHeapStatus;
  frei := HSt.TotalFree;
  belegt := HSt.TotalAllocated;
  EdBelegt.Text := IntToStr(belegt);
  EdFrei.Text := IntToStr(frei)
end;
```

```
procedure TMain.BtLoeschenClick(Sender: TObject);
begin
  IF baum <> NIL THEN BEGIN
    baum.destroy;
    baum := NIL
  END
end;


procedure TMain.BtZeigeWurzelClick(Sender: TObject);
VAR e: TZeichenelement;
begin
  e := baum.getObject as TZeichenelement;
  EdWurzel.Text := e.getInhalt
end;


procedure TMain.BtSetzeWurzelClick(Sender: TObject);
VAR e: TZeichenelement;
begin
  e := TZeichenElement.create(EdWurzel.Text[1]);
  baum.setObject(e);
end;


procedure TMain.BtLinkestesElementClick(Sender:TObject);
VAR Knoten: TBinaryTree;
    e: TZeichenElement;
begin
  knoten := baum;
  If NOT knoten.isEmpty THEN BEGIN
    while NOT knoten.getLeftTree.isEmpty DO knoten :=
                                    knoten.getLeftTree;
    e := knoten.getObject as TZeichenElement;
    if e <> NIL THEN EdMin.Text := e.getInhalt
  END
end;
```

```
procedure TMain.BtRechtestesElementClick(Sender:TObject);
VAR Knoten: TBinaryTree;
    e: TZeichenElement;
begin
  knoten := baum;
  If NOT knoten.isEmpty THEN BEGIN
    while NOT knoten.getRightTree.isEmpty DO
                        knoten := knoten.getRightTree;
    e := knoten.getObject as TZeichenElement;
    if e <> NIL THEN EdMax.Text := e.getInhalt
  END
end;
```

```
procedure TMain.BtFuegeUntenLinksEinClick(Sender:TObject);
VAR Knoten, neuKnoten: TBinaryTree;
    e: TZeichenElement;
begin
  knoten := baum;
  If NOT knoten.isEmpty THEN BEGIN
    while NOT knoten.getLeftTree.isEmpty DO knoten :=
                                  knoten.getLeftTree;
    e := TZeichenElement.create(EdUL.Text[1]);
    //Vorsicht, wenn Editfeld leer ist!
    neuKnoten := TBinaryTree.create(e);
    knoten.setLeftTree(neuKnoten);
  END
end;
```

```
procedure TMain.BtFuegeUntenRechtsEinClick(Sender:TObject);
VAR Knoten, neuKnoten: TBinaryTree;
    e: TZeichenElement;
begin
  knoten := baum;
  If NOT knoten.isEmpty THEN BEGIN
    while NOT knoten.getRightTree.isEmpty DO
                        knoten := knoten.getRightTree;
    e := TZeichenElement.create(EdUR.Text[1]);
    //Vorsicht, wenn Editfeld leer ist!
    neuKnoten := TBinaryTree.create(e);
    knoten.setRightTree(neuKnoten);
  END
end;

end.
```

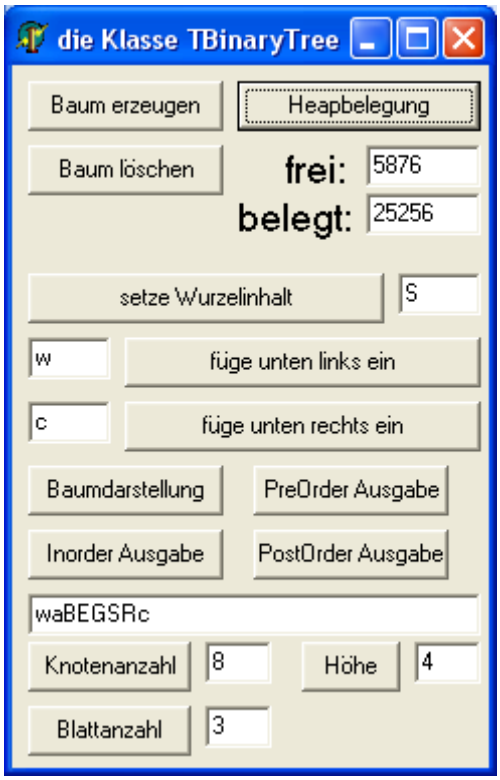
Aufgabe: Programmiere die letzten vier Methoden mithilfe von rekursiven Unterprozeduren!

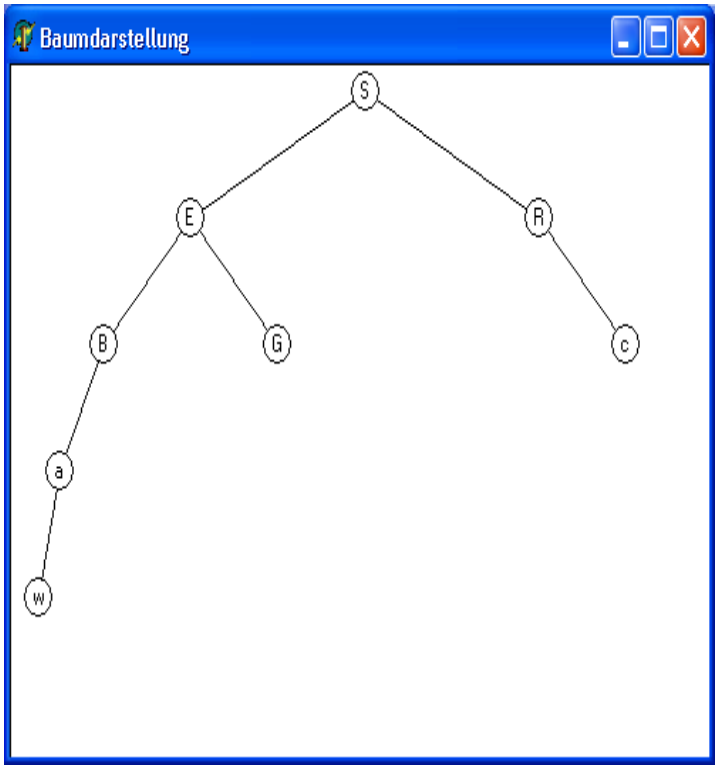
Lösungen:

```
procedure TMain.BtLinkestesElementClick(Sender: TObject);
  // rekursive Version

  procedure geheLinks(T:TBinaryTree);
  BEGIN
    IF NOT T.getLeftTree.isEmpty
       THEN geheLinks(T.getLeftTree)
    ELSE EdMin.Text := (T.getObject as
                          TZeichenElement).getInhalt
  END;

begin
   If NOT baum.isEmpty THEN geheLinks(baum)
end;
```

```
procedure TMain.BtFuegeUntenLinksEinClick(Sender: TObject);
// rekursive Version
Var e:TZeichenelement;
    neuKnoten: TBinaryTree;

  procedure geheLinks(T:TBinaryTree);
  BEGIN
    IF NOT T.getLeftTree.isEmpty
        THEN geheLinks(T.getLeftTree)
    ELSE BEGIN       //Vorsicht, wenn Editfeld leer ist!
           e := TZeichenElement.create(EdUL.Text[1]);
           neuKnoten := TBinaryTree.create(e);
           T.setLeftTree(neuKnoten);
         END
  END;

begin
   If NOT baum.isEmpty THEN geheLinks(baum)
end;
```

Im Folgenden wird unser Hauptprogramm durch einige sehr wichtige, rekursive Methoden ergänzt.






```
procedure
TMain.Zeichne(pBaum:TBinaryTree;xl,xr,y:INTEGER);
  VAR xm: INTEGER;
      e: TZeichenelement;
  Begin
   IF NOT pBaum.isEmpty THEN BEGIN
     xm := (xl + xr) DIV 2;
     With Plan.Canvas DO BEGIN
        IF NOT pBaum.getLeftTree.isEmpty THEN begin
          // Die Kanten werden anschließend teilweise von den Kreisen
          // überzeichnet.
          MoveTo(xm,y);
          LineTo((xl+xm)DIV 2,y+deltay)
        end;
```

```
        IF NOT pBaum.getRightTree.isEmpty THEN begin
          MoveTo(xm,y);
          LineTo((xr+xm)DIV 2,y+deltay)
        end;
        Ellipse(xm-10,y-10,xm+10,y+10);
        e := pBaum.getObject as TZeichenelement;
        IF e <> NIL THEN textout(xm-4,y-7, e.getInhalt);
       // Der Buchstabe soll in den Kreis
      END;
      Zeichne(pBaum.getLeftTree, xl, xm, y + deltay);
      Zeichne(pBaum.getRightTree, xm, xr, y + deltay)
    END;
  End;


procedure TMain.BtZeigeBaumClick(Sender: TObject);
VAR xl, xr: INTEGER;
begin
  With Plan DO BEGIN
    Canvas.Rectangle(0,0,Width,Height);
    xl := 5;         // Die Koordinaten werden etwas angepasst, so dass
    xr := Width - 5;   // es am Bildrand keine Probleme gibt.
    deltay := Height DIV 6 ; // willkürliche Tiefenbegrenzung
  END;
  Zeichne(baum, xl, xr, 14)
end;
```

```
procedure TMain.BtInorderAusgabeClick(Sender: TObject);
VAR e: TZeichenelement;

  procedure Inorder(pBaum: TBinaryTree);
    begin
      IF NOT pBaum.isEmpty THEN Begin
        Inorder(pBaum.getLeftTree);
        e := pBaum.getObject as TZeichenelement;
        IF e <> NIL THEN
          EdAusgabe.Text:=EdAusgabe.Text+e.getInhalt;
        Inorder(pBaum.getRightTree)
      End
    end;


begin
 EdAusgabe.Text := '';
 IF baum <> NIL THEN Inorder(baum)
end;


procedure TMain.BtPreOrderAusgabeClick(Sender: TObject);
VAR e: TZeichenelement;
  procedure Preorder(pBaum: TBinaryTree);
    begin
      IF NOT pBaum.isEmpty THEN Begin
        e := pBaum.getObject as TZeichenelement;
        IF e <> NIL THEN
          EdAusgabe.Text:=EdAusgabe.Text+e.getInhalt;
        Preorder(pBaum.getLeftTree);
        Preorder(pBaum.getRightTree)
      End
    end;

begin
 EdAusgabe.Text := '';
 IF baum <> NIL THEN Preorder(baum)
end;
```

```
procedure TMain.BtPostOrderAusgabeClick(Sender: TObject);
VAR e: TZeichenelement;

  procedure Postorder(pBaum: TBinaryTree);
    begin
      IF NOT pBaum.isEmpty THEN Begin
        Postorder(pBaum.getLeftTree);
        Postorder(pBaum.getRightTree);
        e := pBaum.getObject as TZeichenelement;
        IF e <> NIL THEN
          EdAusgabe.Text:=EdAusgabe.Text+e.getInhalt;
      End
    end;

begin
 EdAusgabe.Text := '';
 IF baum <> NIL THEN Postorder(baum)
end;


procedure TMain.BtHoeheClick(Sender: TObject);
 function Maximum(a, b: INTEGER): INTEGER;
  BEGIN
   IF b > a THEN RESULT := b ELSE RESULT := a
  END;

 function Hoehe(pBaum: TBinaryTree):Integer;
  begin
   IF pBaum.isEmpty THEN Result := -1
   ELSE Result := Maximum(Hoehe(pBaum.getLeftTree),
                          Hoehe(pBaum.getRightTree)) +1
  end;

BEGIN
 EdHoehe.Text := IntToStr(Hoehe(baum))
END;
```

```
procedure TMain.BtKnotenanzahlClick(Sender: TObject);

  function Anzahl(pBaum: TBinaryTree):Integer;
  begin
   IF pBaum.isEmpty THEN Result := 0
   ELSE Result := Anzahl(pBaum.getLeftTree) +
                        Anzahl(pBaum.getRightTree) +1
  end;

begin
 IF baum <> NIL
 THEN EdKnoten.Text := IntToStr(Anzahl(baum))
 ELSE EdKnoten.Text := '0'
end;


procedure TMain.BtBlattAnzahlClick(Sender: TObject);
VAR count: INTEGER;

  procedure BlattAnzahl(pBaum: TBinaryTree);
  begin
    IF (pBaum.getLeftTree.isEmpty) AND
                          pBaum.getRightTree.isEmpty)
      THEN INC(count)
    ELSE BEGIN
          IF NOT pBaum.getLeftTree.isEmpty
            THEN BlattAnzahl(pBaum.getLeftTree);
          IF NOT pBaum.getRightTree.isEmpty
            THEN BlattAnzahl(pBaum.getRightTree)
        END
    end;

begin
 count := 0;
 IF baum <> NIL THEN
   IF NOT baum.isEmpty THEN BlattAnzahl(baum);
 EdBlatt.Text := IntToStr(count)
end;
```

```
procedure TMain.FormCreate(Sender: TObject);
begin
  baum := NIL;
  Left := 0;
  Plan := TForm.create(self); // Parameter = Owner
  With Plan DO BEGIN
    Top := self.Top;
    Left := self.Width;
    Width := 500;
    Height := self.Height;
    Caption := 'Baumdarstellung';
    Color := clWhite;
    Show
  END;
end;

end.
```

Aufgaben

1. Gegeben sei ein nicht geordneter Binärbaum mit 9 Knoten.
Inorderausgabe: E A C K F H D B G
Preorderausgabe: F A E K C D H G B
Zeichne den Baum!

2. Gib (in Inorder-Reihenfolge) jedes Element eines Binärbaumes zusammen mit seiner Ebenennummer aus! Für den Binärbaum auf Seite 99 sähe die Ausgabe so aus: w5, a4, B3, E2, G3, S1, R2, c3

3. Ein Binärbaum mit höchstens 9 Ebenen soll ebenenweise ausgegeben werden. Für den Binärbaum auf Seite 147 sähe die Ausgabe so aus:

Ebene1: S
Ebene2: E, R
Ebene3: B, G, c
Ebene4: a
Ebene5: w

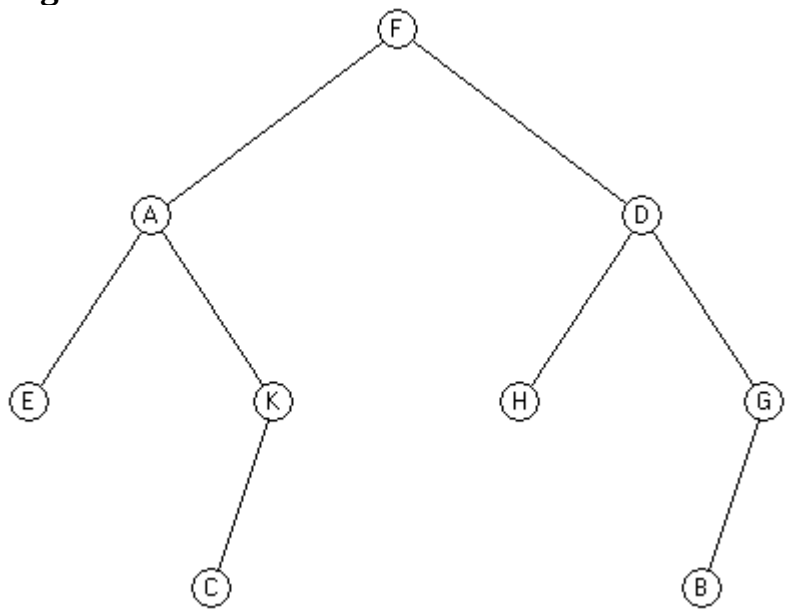
Lösungsweg: Erzeuge 9 Schlangen, traversiere den Binärbaum und schreibe dabei jedes Element in die entsprechende Schlange. Gib anschließend die Schlangen aus.

4. Gib (in Inorder-Reihenfolge) alle Elemente eines Binärbaumes zusammen mit ihren Pfadnamen aus. Dabei steht **L** für links und **R** für rechts. Für den Binärbaum auf Seite 147 sähe die Ausgabe so aus:

w	LLLL
a	LLL
B	LL
E	L
G	LR
S	
R	R
c	RR

Lösungen

Aufgabe 1



Aufgabe 2

```
procedure TMain.BtInorderEbeneClick(Sender: TObject);
VAR e: TZeichenelement;

 procedure Ausgabe(pbaum:TBinaryTree; n: INTEGER);
 BEGIN
  IF NOT pBaum.isEmpty THEN Begin
        Ausgabe(pBaum.getLeftTree, n+1);
        e := pBaum.getObject as TZeichenelement;
        IF e <> NIL THEN
          EdAusgabe.Text:=EdAusgabe.Text+e.getInhalt
                                        +IntToStr(n)+', ';
        Ausgabe(pBaum.getRightTree,n+1)
      End
 END;

Begin
 EdAusgabe.Text:= '';
 Ausgabe(baum,1)
end;
```

Aufgabe 3

```
procedure TMain.BtEbenenAusgabeClick(Sender:TObject);
VAR e: TZeichenelement;
    A: ARRAY[1..9] of TQueue;
    i: INTEGER;
    zeile: STRING;

procedure InSchlangen(pbaum:TBinaryTree; n: INTEGER);
BEGIN
  IF NOT pBaum.isEmpty THEN Begin
        InSchlangen(pBaum.getLeftTree, n+1);
        e := pBaum.getObject as TZeichenelement;
        IF e <> NIL THEN A[n].enqueue(e);
        InSchlangen(pBaum.getRightTree,n+1)
      End
END;

begin
 Listbox1.Clear;
 FOR i := 1 TO 9 DO A[i] := TQueue.create;
 InSchlangen(baum,1);
 FOR i := 1 TO 9 DO BEGIN
    zeile:= '';
    WHILE NOT A[i].isEmpty DO BEGIN
      zeile := zeile +
           (A[i].front as TZeichenelement).getInhalt + ',  ';
      A[i].dequeue;
    END;
    zeile := 'Ebene ' + IntToStr(i) + ': ' + zeile;
    Listbox1.Items.Add(zeile)
 END;
 FOR i := 1 TO 9 DO A[i].destroy;
end;
```

Aufgabe 4

```
procedure TMain.BtPfadnamenClick(Sender: TObject);
VAR e: TZeichenelement;

 procedure Pfadausgabe(pbaum:TBinaryTree; s:STRING);
 BEGIN
   IF NOT pBaum.isEmpty THEN Begin
     Pfadausgabe(pBaum.getLeftTree, s+'L');
     e := pBaum.getObject as TZeichenelement;
     IF e <> NIL THEN
              Listbox1.Items.Add(e.getInhalt+'  '+s);
     Pfadausgabe(pBaum.getRightTree,s+'R')
   End
 END;

begin
  Listbox1.Clear;
  Pfadausgabe(baum,'')
end;
```

Morsecode-Aufgabe

Lateinische Buchstaben	
Buchstabe	**Code**
A	· —
B	— · · ·
C	— · — ·
D	— · ·
E	·
F	· · — ·
G	— — ·
H	· · · ·
I	· ·
J	· — — —
K	— · —
L	· — · ·
M	— —
N	— ·
O	— — —
P	· — — ·
Q	— — · —
R	· — ·
S	· · ·
T	—
U	· · —
V	· · · —
W	· — —
X	— · · —
Y	— · — —
Z	— — · ·

Der Morsecode kann in einem Binärbaum dargestellt werden. Dabei stehen in den Knoten die Buchstaben. Die Wurzel enthält keinen Buchstaben. Ein Punkt im Morsecode entspricht der Verzweigung nach links, ein Strich der Verzweigung nach rechts.

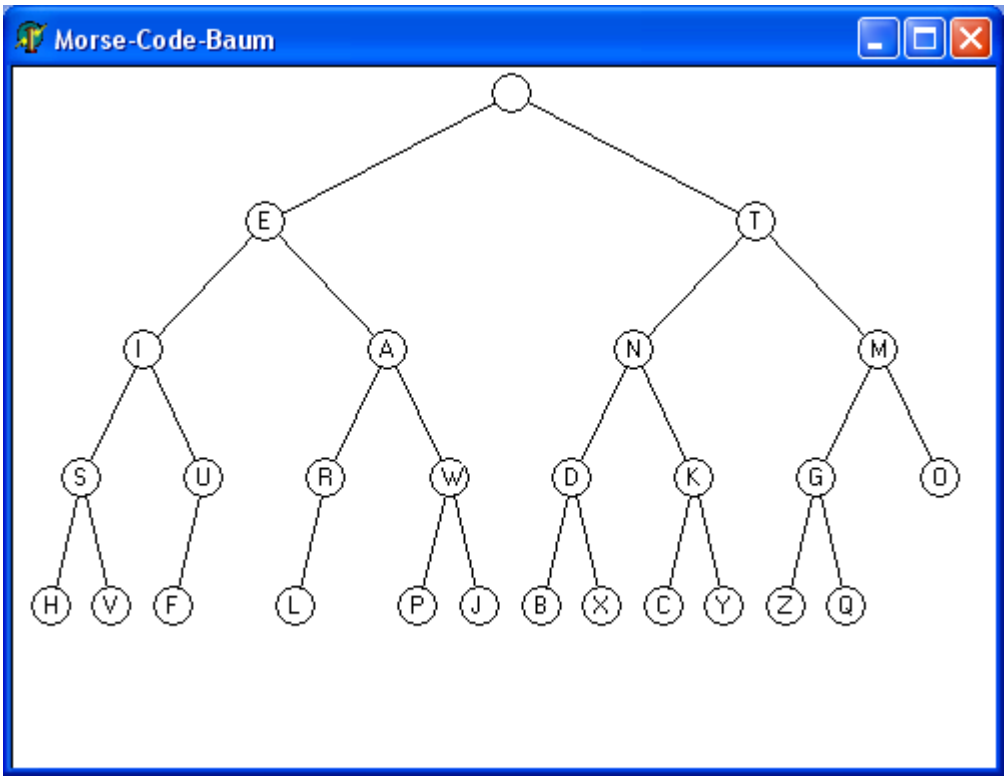
a) Zeichne zunächst den Morsecode-Baum auf Papier!

b) Erzeuge anschließend diesen Morsecode-Baum so, wie es im letzten Hauptprogramm vorgestellt wurde!
Beachte die `procedure TMain.BtBaumErzeugenClick;`

c) Schreibe ein Programm, welches mit Hilfe des Morsecode-Baumes den in einem Memofeld stehenden Morsecode, z.B.
. – – . / . . . / – . – – / . . . / . . / – . – /
dekodiert. Als Trennzeichen dient dabei der Querstrich (oder ein Leerzeichen).

d) Der in einem Memofeld stehende Originaltext soll in den Morsecode codiert werden.

Lösung der Morse-Code-Aufgabe:



procedure TMain.BtBaumErzeugenClick(Sender: TObject);

```
VAR e: TZeichenElement;
    baumL, baumR, baum3, baum4, baum5, baumE,
    baumT: TBinaryTree;
begin
  e := TZeichenElement.create('H');
  baumL := TBinaryTree.create(e);
  e := TZeichenElement.create('V');
  baumR := TBinaryTree.create(e);
  e := TZeichenElement.create('S');
  baum3 := TBinaryTree.create(e, baumL, baumR);
  e := TZeichenElement.create('F');
  baumL := TBinaryTree.create(e);
  e := TZeichenElement.create('U');
  baum4 := TBinaryTree.create(e, baumL, NIL);
  e := TZeichenElement.create('I');
  baum5 := TBinaryTree.create(e, baum3, baum4);
```

```
e  := TZeichenElement.create('L');
baumL := TBinaryTree.create(e);
e := TZeichenElement.create('R');
baum3 := TBinaryTree.create(e, baumL, NIL);
e := TZeichenElement.create('P');
baumL := TBinaryTree.create(e);
e := TZeichenElement.create('J');
baumR := TBinaryTree.create(e);
e := TZeichenElement.create('W');


baum4 := TBinaryTree.create(e, baumL, baumR);
e := TZeichenElement.create('A');
baum4 := TBinaryTree.create(e, baum3, baum4);
e := TZeichenElement.create('E');
baumE := TBinaryTree.create(e, baum5, baum4);


e := TZeichenElement.create('B');
baumL := TBinaryTree.create(e);
e := TZeichenElement.create('X');
baumR := TBinaryTree.create(e);
e := TZeichenElement.create('D');
baum3 := TBinaryTree.create(e, baumL, baumR);
e := TZeichenElement.create('C');
baumL := TBinaryTree.create(e);
e := TZeichenElement.create('Y');
baumR := TBinaryTree.create(e);
e := TZeichenElement.create('K');
baum4 := TBinaryTree.create(e, baumL, baumR);
e := TZeichenElement.create('N');
baum5 := TBinaryTree.create(e, baum3, baum4);


e  := TZeichenElement.create('Q');
baumR := TBinaryTree.create(e);
e  := TZeichenElement.create('Z');
baumL := TBinaryTree.create(e);
e := TZeichenElement.create('G');
```

```
  baum3 := TBinaryTree.create(e, baumL, baumR);
  e := TZeichenElement.create('O');
  baum4 := TBinaryTree.create(e);
  e := TZeichenElement.create('M');
  baum4 := TBinaryTree.create(e, baum3, baum4);
  e := TZeichenElement.create('T');
  baumT := TBinaryTree.create(e, baum5, baum4);

  e := TZeichenElement.create(' '); // Leerzeichen
  baum := TBinTree.create(e, baumE, baumT);
end;
```

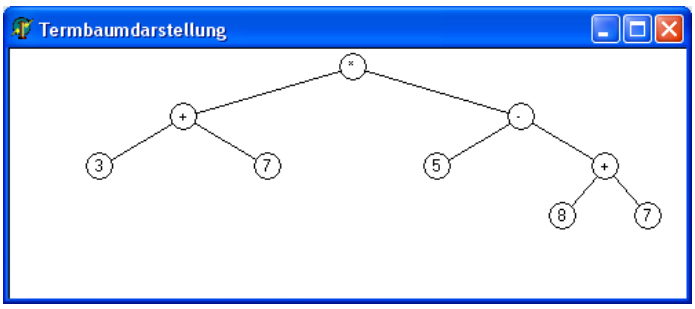

```
procedure TMain.BtDekodierenClick(Sender: TObject);
VAR CodeText: STRING;

  function dekodierterText(pCode: String):String;
  VAR zaehler: INTEGER;
      hilfsbaum: TBinaryTree;
      e: TZeichenelement;
      ergebnis: String;
  BEGIN
    zaehler := 1;
    hilfsbaum := baum;
    ergebnis := '';
```

```
    while zaehler <= length(pCode) DO BEGIN
      if pCode[zaehler] ='.'
            THEN hilfsbaum := hilfsbaum.getLeftTree
      ELSE if pCode[zaehler] = '-'
           THEN hilfsbaum := hilfsbaum.getRightTree
{ Wenn der Benutzer Eingaben ins Memofeld schreibt, werden leider auch
 zusätzlich die ASCII-Codes von Return (13) und Linefeed (10) gespeichert.}
          ELSE BEGIN
            e := hilfsbaum.getObject as
                                    TZeichenelement;
            IF e <> NIL THEN
              IF e.getInhalt in ['A'..'Z'] THEN
                ergebnis := ergebnis + e.getInhalt;
            hilfsbaum := baum
          END;
      INC(zaehler)
    END;
    result := ergebnis
  END;

begin
  CodeText := MCode.Text;
  CodeText := CodeText + '$';
// zusätzliches Endezeichen erleichtert die Auswertung.
  MText.text := dekodierterText(CodeText)
end;
```

```
procedure TMain.BtCodierenClick(Sender: TObject);
VAR OriginalText: String;
    i: INTEGER;
    e: TZeichenelement;

  function Zeichencode(hilfsBaum: TBinaryTree;
                   ch: CHAR; weg: String): String;
  VAR ergebnis: STRING;
  BEGIN
    ergebnis := '';
    IF hilfsBaum <> NIL THEN Begin
      e := hilfsBaum.getObject as TZeichenelement;
      IF e <> NIL THEN IF e.getInhalt = ch
                        THEN ergebnis:= weg;
      IF ergebnis = '' THEN ergebnis :=
 Zeichencode(hilfsBaum.getLeftTree, ch, weg + '.');
      IF ergebnis = '' THEN ergebnis :=
 Zeichencode(hilfsBaum.getRightTree, ch, weg + '-')
    End;
    RESULT := ergebnis
  END;

begin
  OriginalText := MText.text;
  MCode.Text := '';
  FOR i := 1 TO Length(OriginalText) DO
    IF OriginalText[i] in ['A'..'Z'] THEN
      MCode.Text := MCode.Text +
    ZeichenCode(baum, OriginalText[i], '') + ' / ';
end;
```

Termbäume

Termbäume enthalten sowohl Operatoren (+,-, * , /, ^) als auch Operanden (Ziffern bzw. Integerzahlen).
Dafür werden die beiden Klassen *TOperatorObjekt* und *TOperandObjekt* benötigt.



Dokumentation der Klasse *TOperatorObjekt*

```
Konstruktor create(s: char);
vorher:     Das Zeichen s entspricht dem mathematischen Operator.
nachher     Ein dem mathematischen Operator entsprechendes Objekt
            existiert.

Anfrage     getInhalt: char;
nachher     Das enthaltene Operatorzeichen wird zurückgegeben.
```

Dokumentation der Klasse *TOperandObjekt*

```
Konstruktor create(s: INTEGER);
vorher:     Die Ziffer s entspricht der mathematischen Zahl.
nachher     Ein der mathematischen Zahl entsprechendes Objekt existiert.

Anfrage     getInhalt: INTEGER;
nachher     Die enthaltene Ziffer wird zurückgegeben.
```

Dokumentation der Klasse *TTermbaum*

Objekte dieser Klasse enthalten einfache Ausdrücke mit **einziffrigen** INTEGER-Operanden. Die Divisionen sind INTEGER-Divisionen.
In einem Termbaum sind zwei unterschiedliche Objekttypen (Operatoren und Operanden) gespeichert. Die Operanden sind grundsätzlich in den Blättern enthalten.

Oberklasse: *TBinaryTree*

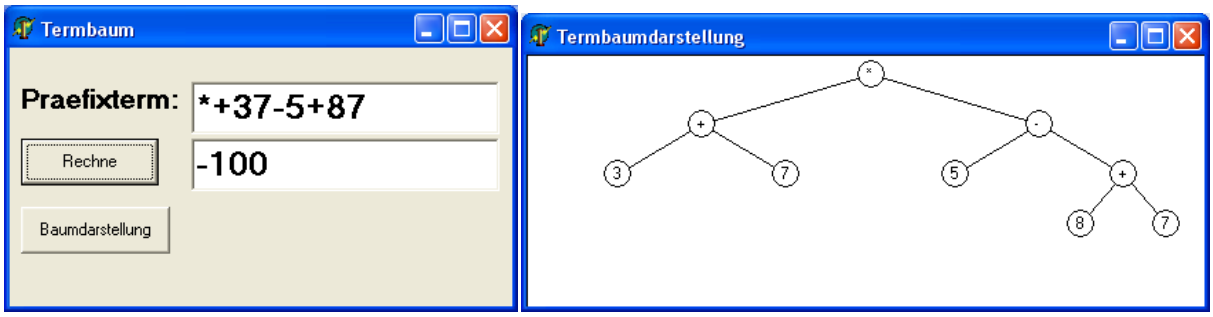
`Konstruktor`	**`create`**`(VAR Zeichenkette: String);`
`vorher:`	Die Zeichenkette enthält den mathematischen Term in Praefix-Notation. Bemerkung: Terme in Praefixnotation benötigen keine Klammern (im Gegensatz zu der sonst üblichen Infixnotation).
`nachher`	Ein der Zeichenkette entsprechender Termbaum existiert. Die Variable Zeichenkette ist anschließend leer.
Bemerkung:	Die Konstruktoren der übergeordneten Klasse existieren immer noch.

`Anfrage`	**`Auswertung:`** `INTEGER`
`nachher`	Diese Anfrage liefert den mathematischen Wert des Terms.

```
unit mTermbaum;
interface
uses mBinaryTree, SysUtils;

const Operatoren = ['+','-','*','/','^'];

type TOperatorObjekt = class
      private
        Operator: char;
      public
        constructor create(s: char);
        function getInhalt: char;
     end;

     TOperandObjekt = class
      private
        Operand: integer;
      public
        constructor create(s: integer);
        function getInhalt: integer;
     end;


     TTermbaum = class(TBinaryTree)
        constructor create(VAR Zeichenkette: String);
                                    overload; virtual;
        function Auswertung: Integer;
     end;


implementation

constructor TOperatorObjekt.create(s: char);
begin
  Operator:= s;
end;
```

```
function TOperatorObjekt.getInhalt: char;
begin
  result:=Operator;
end;


constructor TOperandObjekt.create(s: integer);
begin
  Operand:= s;
end;

function TOperandObjekt.getInhalt: integer;
begin
  result:= Operand;
end;


constructor TTermbaum.create(VAR Zeichenkette: String);
 var Zeichen: char;

 function NaechstesZeichen(var Zeichenkette: string): char;
   begin
     RESULT := Zeichenkette[1];
     Delete(Zeichenkette,1,1);
   end;

 begin
   Zeichen:= NaechstesZeichen(Zeichenkette);
   if Zeichen in Operatoren
   then begin
     self.setObject(TOperatorObjekt.create(Zeichen));
     setLeftTree(TTermbaum.create(Zeichenkette));
     setRightTree(TTermbaum.create(Zeichenkette));
   end
   else
     self.setObject(TOperandObjekt.create(StrToInt(Zeichen)));
 end;
```

```
function TTermbaum.Auswertung: Integer;
var Operand1, Operand2: Integer;
    Ergebnis: Integer;
    Teilbaum: TTermBaum;
begin
  if self.isEmpty then Ergebnis:= 0
  else if self.getObject is TOperandObjekt then
         Ergebnis:=TOperandObjekt(self.getObject).getInhalt
       else if self.getObject is TOperatorObjekt then
              begin
                Teilbaum:= TTermbaum(self.getLeftTree);
                Operand1:= Teilbaum.Auswertung;
                Teilbaum:= TTermBaum(self.getRightTree);
                Operand2:= Teilbaum.Auswertung;
                case
          TOperatorObjekt(self.getObject).getInhalt of
            '+': Ergebnis:= Operand1 + Operand2;
            '-': Ergebnis:= Operand1 - Operand2;
            '*': Ergebnis:= Operand1 * Operand2;
            '/': if Operand2<>0 then Ergebnis:=
                                  Operand1 div Operand2
                 else Ergebnis:= maxint;
            '^': begin
                   Ergebnis:= 1;
                   while Operand2 > 0 do begin
                     Ergebnis:= Ergebnis * Operand1;
                     dec(Operand2);
                   end;
                 end;
         end;// of case
     end; // of else
 RESULT := Ergebnis;
end;

end.
```




```
unit Termbaum;
interface

uses
  Dialogs, mTermbaum, mBinaryTree, StdCtrls;

type
  TMain = class(TForm)
    EdPraefix: TEdit;
    L1: TLabel;
    BtRechne: TButton;
    EdErgebnis: TEdit;
    BtBaumdarstellung: TButton;
    procedure FormCreate(Sender: TObject);
    procedure BtRechneClick(Sender: TObject);
    procedure BtBaumdarstellungClick(Sender: Object);
  end;


var

  Main: TMain;
  Zeichenkette: STRING;
  TB: TTermbaum;
  Plan: TForm;
```

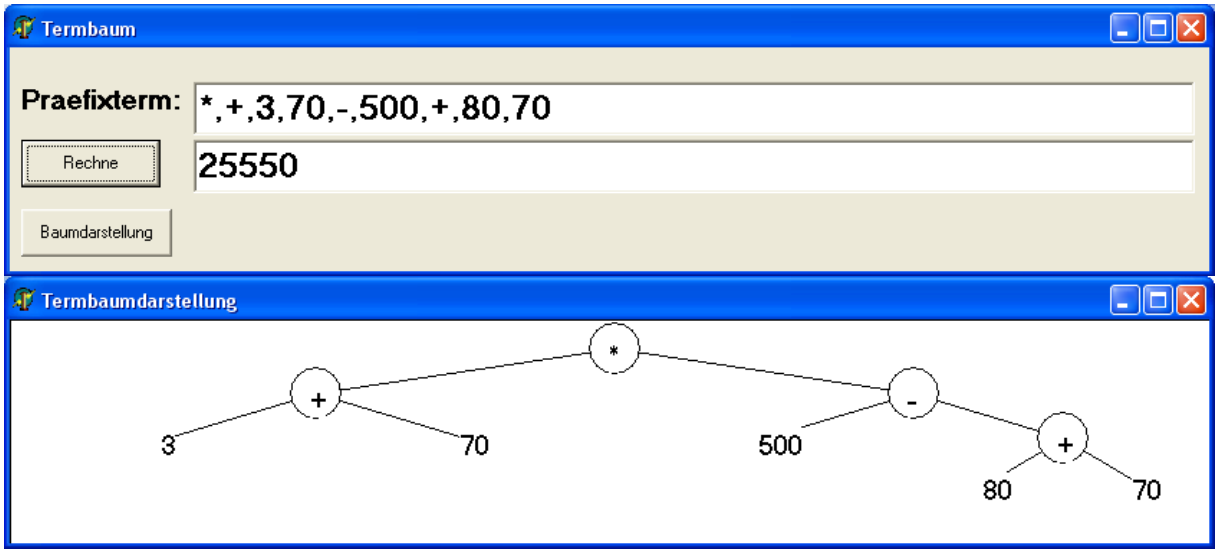
```
Implementation
{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
 self.Left := 0;
 self.Top := 0;
 EdPraefix.text := '*+37-5+87';
 Plan := TForm.Create(self);
 Plan.Color := clWhite;
 Plan.Width := 500;
 Plan.Height := self.Height;
 Plan.Left := self.Left+self.Width+10;
 Plan.Caption := 'Termbaumdarstellung';
 Plan.show
end;

procedure TMain.BtRechneClick(Sender: TObject);
begin
  Zeichenkette := EdPraefix.Text;
  IF TB <> NIL THEN TB.destroy;
  TB := TTermbaum.create(Zeichenkette);
  EdErgebnis.Text := IntToStr(TB.Auswertung);
end;

procedure TMain.BtBaumdarstellungClick(Sender: TObject);
VAR deltay: INTEGER;

  procedure Zeichne(pBaum:TBinaryTree; xl,xr,y:INTEGER);
  VAR xm: INTEGER;
      e: TObject;
  Begin
   IF NOT pBaum.isEmpty THEN BEGIN
     xm := (xl + xr) DIV 2;
     With Plan.Canvas DO BEGIN
        IF NOT pBaum.getLeftTree.isEmpty THEN begin
        // Die Kanten werden anschließend teilweise von den Kreisen
```

```
          // überzeichnet.
            MoveTo(xm,y);
            LineTo((xl+xm)DIV 2,y+deltay)
          end; // of IF

          IF NOT pBaum.getRightTree.isEmpty THEN begin
            MoveTo(xm,y);
            LineTo((xr+xm)DIV 2,y+deltay)
          end; // of IF
          Ellipse(xm-10,y-10,xm+10,y+10);
          e := pBaum.getObject;
          IF e <> NIL THEN
             IF e is TOperandObjekt THEN
                textout(xm-4,y-7,
             IntToStr((e as TOperandObjekt).getInhalt))
             ELSE textout(xm-4,y-7,
                       (e as TOperatorObjekt).getInhalt)
        // der Buchstabe soll in den Kreis
      END; // of With
      Zeichne(pBaum.getLeftTree, xl, xm, y + deltay);
      Zeichne(pBaum.getRightTree, xm, xr, y + deltay)
    END;
   End; // of procedure Zeichne

 begin
    Zeichenkette := EdPraefix.Text;
    IF TB <> NIL THEN TB.destroy;
    TB := TTermbaum.create(Zeichenkette);
    With Plan DO BEGIN
      Canvas.Rectangle(0,0,Width,Height);
      deltay := Height DIV 6 ; // willkürliche Tiefenbegrenzung
    END;
    IF TB <> NIL THEN Zeichne(TB, 5, Plan.width-5, 14)
    // Die Koordinaten werden etwas angepasst, so dass es am Bildrand keine
    //Probleme gibt.
 end;
end.
```

Aufgaben zu Termbäumen

1. Zeichne folgende Termbäume:

$3 \cdot 4 + 5, \quad 3 + 4 \cdot 5, \quad 3 - 4 \cdot \frac{5}{6}, \quad 3 + 4 \cdot 5^6 \ , \quad (3 \cdot 4 + 5) \cdot (6 \cdot 7 - 8)^9$



2. Ändere im obigen Programm die Eingabe des Praefixtermes so, dass immer ein Komma als Trennzeichen benutzt wird!

3. Ändere das Programm so, dass auch mehrstellige INTEGER-Zahlen verarbeitet werden.

Lösung

Es muss hauptsächlich nur die *function NaechstesZeichen(var Zeichenkette: String)* geändert werden:

```
constructor TTermbaum.create(VAR Zeichenkette: String);
 var Zeichen: String;


function NaechstesZeichen(var Zeichenkette:String):STRING;
  VAR ergebnis:STRING;
      ch: CHAR;
   begin
     ergebnis := '';
     ch := Zeichenkette[1];
     IF ch in Operatoren THEN BEGIN
       Delete(Zeichenkette,1,2);
       // nach einem Operator muss immer noch ein Komma stehen
       ergebnis := ch;
     END
     ELSE WHILE ch in Ziffern DO BEGIN
            ergebnis := ergebnis + ch;
            Delete(Zeichenkette,1,1);
            IF Length(Zeichenkette) > 0 THEN
                                 ch :=Zeichenkette[1];
            IF ch=',' THEN  Delete(Zeichenkette,1,1);
            IF Length(Zeichenkette) = 0 THEN
                                    ch := 'A' //Dummy
          END;
     Result := ergebnis;
   end;


 begin //von create
   Zeichen:= NaechstesZeichen(Zeichenkette);
   if Zeichen[1] in Operatoren
   then begin
     self.setObject(TOperatorObjekt.create(Zeichen[1]));
     setLeftTree(TTermbaum.create(Zeichenkette));
     setRightTree(TTermbaum.create(Zeichenkette));
   end
   else
     self.setObject(TOperandObjekt.create(StrToInt(Zeichen)));
 end;
```

Suchbäume

Ein *Suchbaum* oder *geordneter Baum* ist ein spezieller Binärbaum, in dem die Daten geordnet abgelegt sind. Dabei gilt, dass die Daten im linken Teilbaum kleiner als die Daten in der Wurzel und die Daten im rechten Teilbaum größer als in der Wurzel sein müssen. Dies gilt dann auch wieder für alle Teilbäume. Daraus folgt, dass ein Suchbaum jedes Element nur einmal enthalten kann. Alle Teilbäume sind ebenfalls Suchbäume.

Voraussetzung ist natürlich, dass alle Elemente im Baum überhaupt der Größe nach geordnet werden können:

Die abstrakte Klasse *TItem*

Die Klasse *TItem* ist Oberklasse aller Klassen, deren Objekte in einen der Größe nach sortierten Baum eingefügt werden sollen. Dabei ist die *„Größe“* eines Objektes nicht immer von vornherein klar. Was ist zum Beispiel die Größe eines Morsezeichens? Oder die Größe eines Menschen (Körpergröße, Alter, IQ)? Oder die Größe einer mathematischen Matrix (Wert der Determinante)? Damit die Methoden eines Suchbaumes für alle diese Fälle identisch programmiert werden können, verwenden wir die Oberklasse *TItem*.

Die Ordnungsrelation wird in den Unterklassen von *TItem* durch Überschreiben der drei ***abstrakten Methoden*** *isEqual*, *isGreater* und *isLess* festgelegt.

Die Klasse TItem stellt Methoden mit folgender Syntax zur Verfügung:

function isEqual(pItem: TItem): boolean; virtual; abstract;

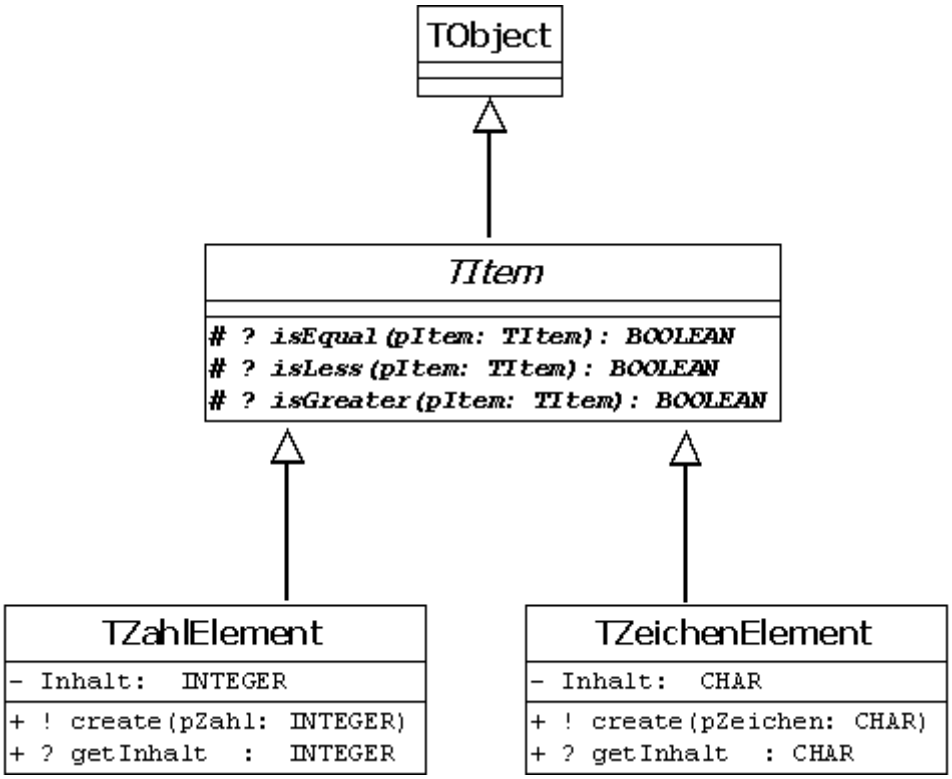
function isLess(pItem: TItem): boolean; virtual; abstract;

function isGreater(pItem: TItem): boolean; virtual; abstract;

Bemerkung: Von einer abstrakten Klasse kann man direkt keine Objekte bilden, weil die abstrakten Methoden noch gar nicht definiert sind. Erst von den Unterklassen lassen sich Objekte bilden.

Um weiterhin mit einfachen Zahlen oder Buchstaben auch in Suchbäumen arbeiten zu können, passen wir im Folgenden die entsprechenden Datentypen an.

Beachte: Abstrakte Klassen- und Methodennamen werden üblicherweise kursiv geschrieben.



```
unit mOrderedElement;


Interface
Type
  TItem = class(TObject)
    public
      function isEqual(pItem:TItem):boolean;virtual;abstract;
      function isLess(pItem:TItem): boolean;virtual;abstract;
      function isGreater(pItem:TItem):boolean;virtual;abstract;
  END;


  TZahlElement = class(TItem)
    private
      Inhalt: INTEGER;
    public
      constructor create(pZahl: INTEGER); virtual;
      function getInhalt: INTEGER;
      function isEqual(pItem:TItem):boolean;override;
      function isLess(pItem:TItem): boolean;override;
      function isGreater(pItem: TItem): boolean;override;
  End;


  TZeichenElement = class(TItem)
    private
      Inhalt: CHAR;
    public
      constructor create(pZeichen: CHAR); virtual;
      function getInhalt: CHAR;
      function isEqual(pItem:TItem):boolean;override;
      function isLess(pItem:TItem): boolean;override;
      function isGreater(pItem:TItem):boolean;override;
  End;
```

```
implementation

  constructor TZahlElement.create(pZahl: INTEGER);
   BEGIN
     inherited create;
     Inhalt := pZahl
   END;

  function TZahlElement.getInhalt: INTEGER;
    BEGIN
      Result := Inhalt
    END;

  function TZahlElement.isEqual(pItem:TItem):boolean;
    VAR hilf: TZahlelement;
    BEGIN
      hilf := pItem as TZahlelement;
      Result := (Inhalt = hilf.getInhalt)
    END;

  function TZahlElement.isLess(pItem:TItem): boolean;
    VAR hilf: TZahlelement;
    BEGIN
      hilf := pItem as TZahlelement;
      Result := (Inhalt < hilf.getInhalt)
    END;

  function TZahlElement.isGreater(pItem:TItem):boolean;
    VAR hilf: TZahlelement;
    BEGIN
      hilf := pItem as TZahlelement;
      Result := (Inhalt > hilf.getInhalt)
    END;
```

```
  constructor TZeichenElement.create(pZeichen: CHAR);
   BEGIN
     inherited create;
     Inhalt := pZeichen
   END;

   function TZeichenElement.getInhalt: CHAR;
     BEGIN
       Result := Inhalt
     END;

  function TZeichenElement.isEqual(pItem:TItem):boolean;
    VAR hilf: TZeichenElement;
    BEGIN
      hilf := pItem as TZeichenElement;
      Result := (Inhalt = hilf.getInhalt)
    END;

  function TZeichenElement.isLess(pItem:TItem):boolean;
    VAR hilf: TZeichenElement;
    BEGIN
      hilf := pItem as TZeichenElement;
      Result := (Inhalt < hilf.getInhalt)
    END;

  function TZeichenElement.isGreater(pItem:TItem):boolean;
    VAR hilf: TZeichenElement;
    BEGIN
      hilf := pItem as TZeichenElement;
      Result := (Inhalt > hilf.getInhalt)
    END;


end.
```

Die Klasse *TBinarySearchTree*

Materialien zu den zentralen Abiturpruefungen im Fach Informatik ab 2012 in NRW
NW-Arbeitsgruppe:
Materialentwicklung zum Zentralabitur im Fach Informatik
Version 2011-01-21

In einem Objekt der Klasse *TBinarySearchTree* werden beliebig viele Objekte in einem Binärbaum (binärer Suchbaum) entsprechend einer Ordnungsrelation verwaltet. Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse *TBinarySearchTree* sind. Dabei gilt:

Die Inhaltsobjekte sind Objekte einer Unterklasse von *TItem*, in der durch Überschreiben der drei Vergleichsmethoden *isLess*, *isEqual*, *isGreater* eine eindeutige Ordnungsrelation festgelegt sein muss.
Alle Objekte im linken Teilbaum sind kleiner als das Inhaltsobjekt des Binärbaumes. Alle Objekte im rechten Teilbaum sind größer als das Inhaltsobjekt des Binärbaumes.
Diese Bedingung gilt auch in beiden Teilbäumen.
Die Klasse *TBinarySearchTree* ist keine Unterklasse der Klasse *TBinaryTree*, sodass deren Methoden nicht zur Verfügung stehen.

Dokumentation der Methoden der Klasse *TBinarySearchTree*

Konstruktor	**`create`**
nachher	Der geordnete Binärbaum existiert und ist leer.
Anfrage	**`isEmpty: boolean`**
nachher	Diese Anfrage liefert den Wahrheitswert *true*, wenn der Suchbaum leer ist, sonst liefert sie den Wert *false*.
Auftrag	**`insertItem(pItem: TItem)`**
nachher	Falls ein mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, passiert nichts. Andernfalls wird das Objekt *pItem* entsprechend der vorgegebenen Ordnungsrelation in den Baum eingeordnet. Falls der Parameter *nil* ist, ändert sich nichts.
Anfrage	**`search(pItem: TItem): TItem`**
nachher	Falls ein bezüglich der verwendeten Ordnungsrelation mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, liefert die Anfrage dieses, ansonsten wird *nil* zurückgegeben. Falls der Parameter *nil* ist, wird *nil* zurückgegeben.
Auftrag	**`remove(pItem: TItem)`**
nachher	Falls ein bezüglich der verwendeten Ordnungsrelation mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, wird dieses entfernt. Das Inhaltsobjekt des gelöschten Knotens wurde nicht freigegeben. Falls der Parameter *nil* ist, ändert sich nichts.
Anfrage	**`getItem: TItem`**
nachher	Diese Anfrage liefert das Inhaltsobjekt des Suchbaumes. Wenn der Suchbaum leer ist, wird *nil* zurückgegeben.
Anfrage	**`getLeftTree: TBinarySearchTree`**
nachher	Diese Anfrage liefert den linken Teilbaum des Suchbaumes.

Der Suchbaum ändert sich nicht. Wenn er leer ist, wird *nil* zurückgegeben.

Anfrage	**getRightTree: TBinarySearchTree**
nachher	Diese Anfrage liefert den rechten Teilbaum des Suchbaumes. Der Suchbaum ändert sich nicht. Wenn er leer ist, wird *nil* zurückgegeben.

Destruktor	**destroy**
nachher	Das Objekt der Klasse TBinarySearchTree wird entfernt und der vom Objekt verwendete Speicher wird wieder freigegeben.

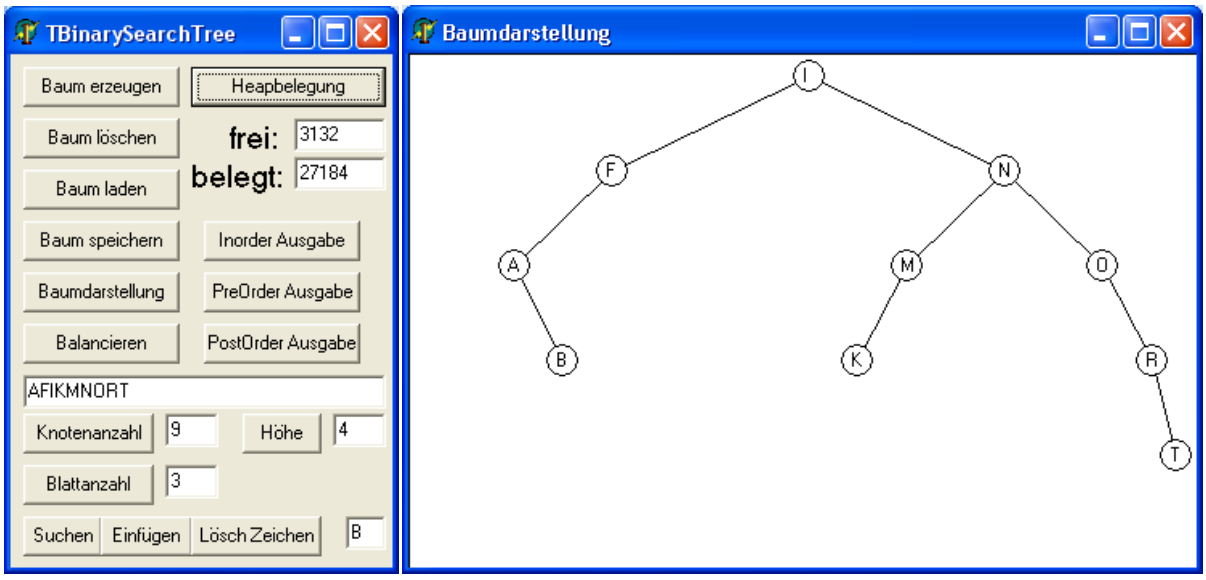
unit mBinarySearchTree;

```
interface
uses mBinaryTree, mList;
type
  TBinarySearchTree = class
  {fehlt die Angabe der Oberklasse, so wird automatisch TObject als Oberklasse gewählt }
  private
    binaryTree: TBinaryTree;
  public
    constructor create; virtual;
    function isEmpty: boolean; virtual;
    procedure insert(pItem: TItem); virtual;
    function search(pItem: TItem): TItem; virtual;
    procedure remove(pItem: TItem); virtual;
    function getItem : TItem; virtual;
    function getLeftTree: TBinarySearchTree; virtual;
    function getRightTree: TBinarySearchTree; virtual;
    destructor destroy; override;
  end;

implementation

  constructor TBinarySearchTree.create;
  begin
    binaryTree := TBinaryTree.create;
  end;
```

```
function TBinarySearchTree.isEmpty: boolean;
begin
  result := binaryTree.isEmpty;
end;

procedure TBinarySearchTree.insert(pItem: TItem);
var lItem: TItem;
    lTree,rTree: TBinarySearchTree;
begin
  if pItem <> nil then begin
    if binaryTree.isEmpty then binaryTree.setObject(pItem)
    else begin
      lItem := TItem(binaryTree.getObject);
      if pItem.isLess(lItem) then begin
        lTree:=self.getLeftTree;
        lTree.insert(pItem);
        self.binaryTree.setLeftTree(lTree.binaryTree);
      end   {sehr interessant, weil binarytree private ist und lTree ≠ self }
      else begin
        if pItem.isGreater(lItem)then begin
          rTree := self.getRightTree;
          rTree.insert(pItem);
          self.binaryTree.setRightTree(rTree.binaryTree);
        end;
      end; // else
    end; // wenn Baum nicht leer ist
  end; // wenn pItem <> nil
end; // of procedure

function TBinarySearchTree.search(pItem: TItem): TItem;
var lItem: TItem;
begin
  if binaryTree.isEmpty or (pItem = nil) then result := nil
  else begin
    lItem := TItem(binaryTree.getObject);
    if pItem.isLess(lItem) then
      result := self.getLeftTree.search(pItem)
    else
      if pItem.isGreater(lItem) then
        result:=self.getRightTree.search(pItem)
      else result:=lItem;
  end;
end;
```

```
procedure TBinarySearchTree.remove(pItem: TItem);
var lKnoten,glKnoten: TBinaryTree;
 lTree,rTree: TBinarySearchTree;
 lInhalt: TItem;
begin
  if (not self.isEmpty) and (pItem <> nil)  then begin
    lInhalt := self.getItem;
    if lInhalt.isEqual(pItem) then begin
      if binaryTree.getRightTree.isEmpty and
         binaryTree.getLeftTree.isEmpty
      then binaryTree.setEmpty;
      else begin
        if binaryTree.getRightTree.isEmpty then begin
          lKnoten := binaryTree.getLeftTree;
          binaryTree.setObject(lKnoten.getObject);
          binaryTree.setLeftTree(lKnoten.getLeftTree);
          binaryTree.setRightTree(lKnoten.getRightTree);
          lKnoten.destroy;
        end
        else begin
          if binaryTree.getleftTree.isEmpty then begin
            lKnoten := binaryTree.getRightTree;
            binaryTree.setObject(lKnoten.getObject);
            binaryTree.setLeftTree(lKnoten.getLeftTree);
            binaryTree.setRightTree(lKnoten.getRightTree);
            lKnoten.destroy;
          end
          else begin
            glKnoten := binaryTree.getLeftTree;
            while not glKnoten.getRightTree.isEmpty do begin
              glKnoten := glKnoten.getRightTree;
            end;
            binaryTree.setObject(glKnoten.getObject);
           self.getLeftTree.remove(TItem(glKnoten.getObject));
          end;
        end;
      end;
    end
    else begin
      if lInhalt.isLess(pItem) then begin
        rTree := self.getRightTree;
        rTree.remove(pItem);
      end
      else begin
```

```
        lTree := self.getLeftTree;
        lTree.remove(pItem);
      end;
    end;
  end;
end;


function TBinarySearchTree.getItem: TItem;
begin
  if self.isEmpty then result := nil
  else result := TItem(binaryTree.getObject);
end;


function TBinarySearchTree.getLeftTree: TBinarySearchTree;
var lTree: TBinarySearchTree;
begin
  if self.isEmpty then result := nil
  else begin
    lTree := TBinarySearchTree.create;
    lTree.binaryTree := binarytree.getLeftTree;
    result := lTree;
  end;
end;


function TBinarySearchTree.getRightTree: TBinarySearchTree;
var lTree: TBinarySearchTree;
begin
  if self.isEmpty then result := nil
  else begin
    lTree := TBinarySearchTree.create;
    lTree.binaryTree := binaryTree.getRightTree;
    result := lTree;
  end;
end;

destructor TBinarySearchTree.destroy;
begin
  binaryTree.removeCompletely;
  inherited destroy;
end;
end.
```



{ In dieser Aufgabe wird nur der spezielle Suchbaum behandelt, der als Inhalt einzelne Zeichen enthält. }

```
unit mHaupt;

interface

uses …, mOrderedElement, mBinarySearchTree,
     mBinaryTree;

type
  TMain = class(TForm)
    ……..
    procedure BtBaumSpeichernClick(Sender: TObject);
    procedure BtBaumLadenClick(Sender: TObject);
    procedure BtSuchenClick(Sender: TObject);
    procedure BtEinfuegClick(Sender: TObject);
    procedure BtBalancierenClick(Sender: TObject);
    procedure BtLoeschZeichenClick(Sender: TObject);
    ……..
  end;

var  …….
implementation
{$R *.dfm}
```

```
procedure TMain.BtBaumErzeugenClick(Sender: TObject);
VAR e: TZeichenElement;
begin
If Baum = NIL THEN Baum := TBinarySearchTree.create;
  e := TZeichenElement.create('I');
  Baum.insert(e);
  e := TZeichenElement.create('N');
  Baum.insert(e);
  ……..
end;


procedure TMain.BtBaumSpeichernClick(Sender: TObject);
VAR Dat: File Of CHAR;
    e: TZeichenelement;
    zeichen: CHAR;

  procedure PreOrderSpeicher(pBaum: TBinarySearchTree);
   begin
     If NOT pBaum.isEmpty THEN Begin
       e := pBaum.getItem as TZeichenelement;
       zeichen := e.getInhalt;
       write(Dat, zeichen);
       PreOrderSpeicher(pBaum.getLeftTree);
       PreOrderSpeicher(pBaum.getRightTree);
     End
   end;

begin
  AssignFile(Dat, 'Baumspeicher');
  Rewrite(Dat);
  IF baum <> NIL THEN PreOrderSpeicher(Baum);
  CloseFile(Dat)
end;
```

```
procedure TMain.BtBaumLadenClick(Sender: TObject);
VAR Dat: File Of CHAR;
```
{ ein *File of TZeichenelement* ist nicht erlaubt. Derartige dynamische Objekte können in einer Datei vom Typ *File of ...* nicht gespeichert werden.}
```
    e: TZeichenelement;
    zeichen: CHAR;
begin
  If Baum = NIL THEN Baum := TBinarySearchTree.create;
  AssignFile(Dat, 'Baumspeicher');
  Reset(Dat);
  WHILE NOT EOF(Dat) DO BEGIN
    Read(Dat, zeichen);
    e := TZeichenelement.create(zeichen);
    Baum.insert(e)
  END;
  CloseFile(Dat)
end;


procedure TMain.Zeichne(pBaum: TbinarySearchTree; xl,
                                        xr, y: INTEGER);
  VAR xm: INTEGER;
      e: TZeichenelement;
  Begin
   IF Not pBaum.isEmpty  THEN BEGIN
     xm := (xl + xr) DIV 2;
     With Plan.Canvas DO BEGIN
        IF NOT pBaum.getLeftTree.isEmpty THEN begin
          // Die Kanten werden anschließend teilweise von den Kreisen überzeichnet.
          MoveTo(xm,y);
          LineTo((xl+xm)DIV 2,y+deltay)
        end;
        IF NOT pBaum.getRightTree.isEmpty THEN begin
          MoveTo(xm,y);
          LineTo((xr+xm)DIV 2,y+deltay)
        end;
        Ellipse(xm-10,y-10,xm+10,y+10);
        e := pBaum.getItem as TZeichenelement;
```

```
        IF e <> NIL THEN textout(xm-4,y-7,e.getInhalt);
      // Der Buchstabe soll in den Kreis
     END;
     Zeichne(pBaum.getLeftTree, xl, xm, y + deltay);
     Zeichne(pBaum.getRightTree, xm, xr, y + deltay)
   END;
  End;


procedure TMain.BtZeigeBaumClick(Sender: TObject);
VAR xl, xr: INTEGER;
begin
  Plan := TForm.create(self); // Parameter = Owner
  With Plan DO BEGIN
    Top := self.Top;
    Left := self.Width;
    Width := 500;
    Height := self.Height;
    Caption := 'Baumdarstellung';
    Color := clWhite;
    Show
  END;
  With Plan.Canvas DO Rectangle(0,0,Plan.Width,
                                    Plan.Height);
  xl := 5;              // Die Koordinaten werden etwas angepasst, so
  xr := Plan.Width - 5; // dass es am Bildrand keine Probleme gibt.
  deltay := Height DIV 6 ; // willkürliche Tiefenbegrenzung
  IF baum <> NIL THEN Zeichne(baum, xl, xr, 14)
end;
```

```
procedure TMain.BtSuchenClick(Sender: TObject);
VAR e, f: TZeichenElement;
begin
  e := TZeichenelement.create(EdSuchEinfueg.Text[1]);
  f := Baum.search(e) as TZeichenelement;
  If f = NIL THEN Showmessage('nicht gefunden!')
  ELSE Showmessage('gefunden')
end;

procedure TMain.BtEinfuegClick(Sender: TObject);
VAR e: TZeichenElement;
begin
  e := TZeichenelement.create(EdSuchEinfueg.Text[1]);
  If Baum = NIL THEN Baum := TBinarySearchTree.create;
  Baum.insert(e);
end;


procedure TMain.BtBalancierenClick(Sender: TObject);
VAR A: ARRAY[1..100] of CHAR;
    // Ein Array of TZeichenelement ist in Delphi nicht erlaubt.
    i, elementeAnzahl: INTEGER;

  procedure InOrderInsFeld(pBaum: TBinarySearchTree);
  VAR zeichen : CHAR;
      e: TZeichenElement;
  begin
    If NOT pBaum.isEmpty THEN BEGIN
      InOrderInsFeld(pBaum.getLeftTree);
      INC(elementeAnzahl);
      e := pbaum.getItem as TZeichenelement;
      zeichen := e.getInhalt;
      A[elementeAnzahl] := zeichen;
      InOrderInsFeld(pBaum.getRightTree);
    END
  end;
```

```
  procedure ausbalancieren(li,re: INTEGER);
  VAR mitte: INTEGER;
      e: TZeichenelement;
  BEGIN
    If li = re THEN BEGIN
      e := TZeichenelement.create(A[li]);
      baum.insert(e)
    END
    ELSE If li + 1 = re THEN BEGIN
           e := TZeichenelement.create(A[li]);
           baum.insert(e);
           e := TZeichenelement.create(A[re]);
           baum.insert(e)
         END
         ELSE BEGIN
           mitte := (li + re) DIV 2;
           e := TZeichenelement.create(A[mitte]);
           baum.insert(e);
           If mitte-1>=li THEN ausbalancieren(li,mitte-1);
           ausbalancieren(mitte+1,re)
         END
  END;

begin // of BtBalancierenClick(Sender: TObject);
  elementeAnzahl := 0;
  FOR i := 1 TO 100 DO A[i] := '#'; // Dummy-Zeichen
  If NOT baum.isEmpty NIL THEN InOrderInsFeld(baum);
  baum.destroy;
  baum := TBinarySearchTree.create;
  Ausbalancieren(1, elementeAnzahl)
end;
```

```
procedure TMain.BtLoeschZeichenClick(Sender: TObject);
VAR zeichen: CHAR;
    e: TZeichenelement;
begin
  zeichen := EdSuchEinfueg.Text[1];
  e := TZeichenelement.create(zeichen);
  baum.remove(e);
end;

end.
```

Suchbaumaufgaben

1. Gegeben sei folgende Liste von Buchstaben
J, R, D, G, T, E, M, H, P, A, F, Q, die nacheinander in einen zunächst leeren, binären Suchbaum eingeordnet werden sollen.
Konstruiere den Suchbaum!
Man kann einen inneren Knoten, der zwei Nachfolger besitzt, löschen, indem man ihn durch das kleinste Element aus seinem rechten Teilbaum ersetzt.
Zeichne obigen Suchbaum, nachdem man die Knoten M und D gelöscht hat!