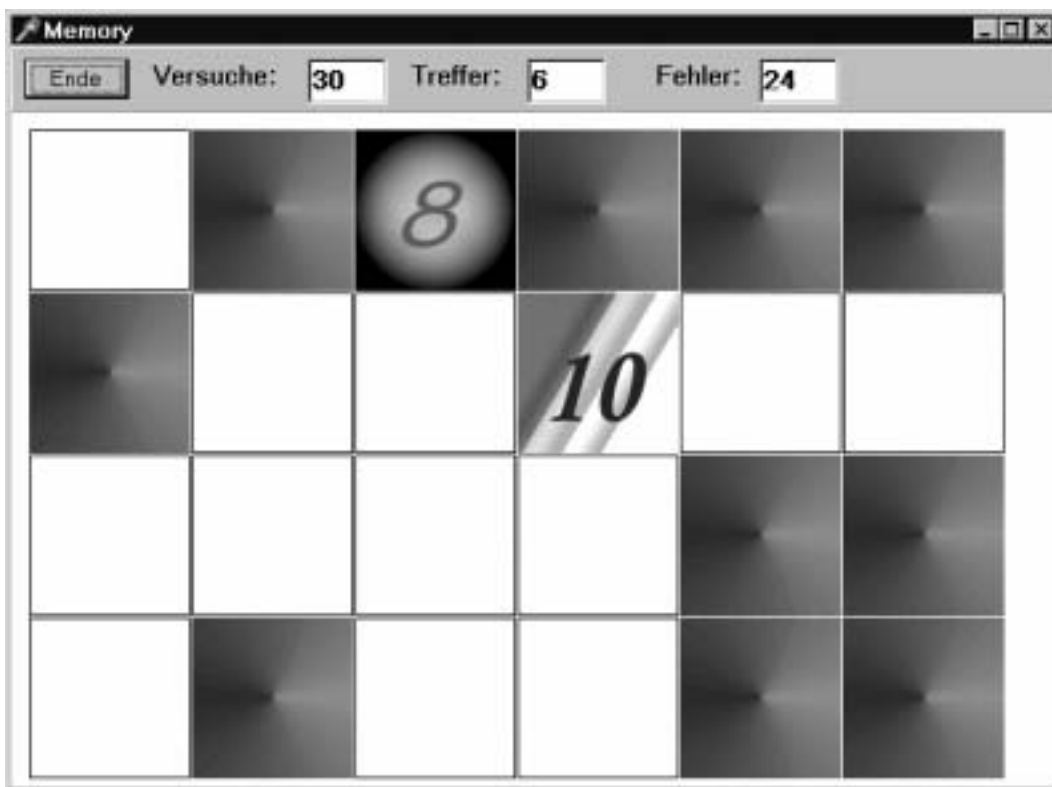


1.2 Memory

Der Umgang mit selbst definierten Objektklassen lässt sich am einfachsten an einem Beispiel verstehen, bei dem auftretende Fehler am Bildschirm direkt zu sehen sind, z. B. dadurch, dass gewünschte Grafiken nicht erscheinen. Wir wollen uns deshalb mit dem bekannten Kinderspiel *Memory* befassen, bei dem man sich an die Position verdeckter Karten erinnern muss, wobei immer zwei paarweise gleiche Karten vorhanden sind. Anfangs sind alle Karten verdeckt. Man deckt zwei frei gewählte Karten auf (die meist verschieden sind) und merkt sich deren Position. Durch wiederholtes Aufdecken (und ein gutes Gedächtnis) steigt die Wahrscheinlichkeit, zwei gleiche Karten zu erwischen. Diese „gewinnt“ man dann und entfernt sie vom Spielfeld. Bei Spielende sind keine Karten mehr vorhanden. Der Spieler mit den meisten Karten hat gewonnen.

Wir wollen das Spielfeld auf den Computerbildschirm verlegen und alleine spielen. Dann ist das Spielziel, möglichst wenig Fehler bis zum Abräumen des Spielfeldes zu machen. Nach einigen Zügen könnte der Bildschirm etwa so aussehen:



Für das Spiel benötigen wir also neben den schon bekannten Delphi-Komponenten wie Buttons, Panels, Editierfeldern, ... verschiedene *Spielkarten*, die beim „Anklicken“ richtig reagieren und die verschiedene „Kartenseiten“ am Bildschirm darstellen können. Damit müssen sie über einige Eigenschaften verfügen, die von unseren schon oft benutzten *Image*-Komponenten her bekannt sind, und über einige neue, die wir hinzufügen. Wir kommen damit zu der üblichen Arbeitsweise in der *objektorientierten Programmierung (OOP)*, eigene Objektklassen von vorhandenen abzuleiten, so dass einige der benötigten Eigenschaften *vererbt* werden und nur die wirklich neuen selbst zu implementieren sind.

1.2.1 Objekte und Klassen

Bei einer Objektklasse handelt es sich um einen *Typ*, also eine Art Schablone, die angibt, über welche Eigenschaften *Instanzen* der Klasse, die *Objekte*, verfügen. Bei unseren Bällen aus dem ersten Band beschrieb die Klasse *tBall* die Eigenschaften der dort als Variablen erzeugten Objekte *Ball1*, *Ball2*, ... Ebenso wie die anderen Objektklassen des ersten Bandes hatten unsere Bälle aber (scheinbar) keine direkten Vorfahren, von denen sie Eigenschaften erben konnten.

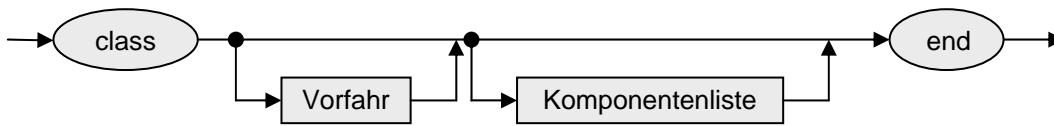
Eine Objektklasse kann *Datenfelder* und *Methoden* enthalten. (Auf *Eigenschaften* gehen wir nicht weiter ein.) Die Datenfelder enthalten in Analogie zu den Verbunden (*record*) Informationen, die ein bestimmtes Objekt beschreiben: den Namen, das Alter, die Farbe, die Größe,... Sie werden wie beim Verbund mit Angabe eines beliebigen Typs deklariert. Die Methoden sind Funktionen und/oder Prozeduren, die normalerweise mit den Daten des Objekts arbeiten. Sie können aber ohne Einschränkung auch andere Aufgaben übernehmen – wie andere Unterprogramme auch. Programmtechnisch werden Methodendeklarationen wie *forward*-vereinbarte Unterprogramme behandelt. Da der Methodenkopf vollständig angegeben wird, kann der Compiler Aufrufe dieser Methoden auf syntaktische Richtigkeit überprüfen. Der Methodenrumpf wird später an beliebiger Stelle innerhalb des *Implementation*-Teils der benutzten Unit platziert.

Neu bei den Klassentypen sind Informationen, die sich auf die *Sichtbarkeit* von Daten und Methoden beziehen. Auch hier gibt es zahlreiche Möglichkeiten, die Elemente zu unterteilen. Wir werden uns auf die *public*- und *private*-*Sichtbarkeitszuweisungen* beschränken:

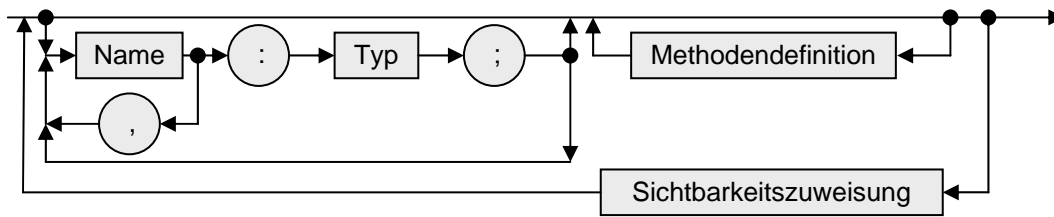
- Als *public* deklarierte Elemente unterliegen keinen Beschränkungen bzgl. ihrer Sichtbarkeit: Jeder andere Programmteil, der Zugriff auf ein Objekt dieser Klasse hat, kann auch auf diese Elemente (Daten und Methoden) zugreifen.
- Als *private* deklarierte Elemente sind für Zugriffe „von außen“, also von Programmteilen, die nicht zur selben Unit gehören, unsichtbar. Sie können also nur „von innen“, normalerweise von Methoden der eigenen Objektklasse, manipuliert werden.

Wie üblich wird die Syntax einer Klassendeklaration in einem (hier nicht ganz vollständigen) Syntaxdiagramm beschrieben:

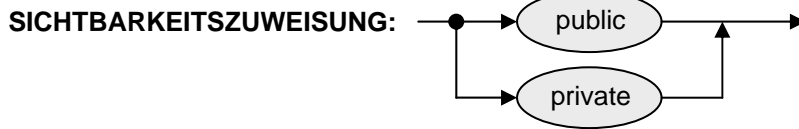
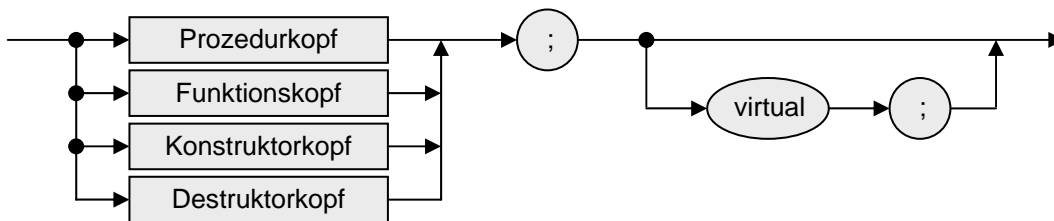
KLASSENTYP:



KOMPONENTENLISTE:



METHODENDEFINITION:



Eine Objektklasse erbt von einem Vorfahr alle Datenfelder und Methoden. Werden keine neuen eingeführt, dann sind „Nachkomme“ und „Vorfahr“ bis auf den Namen identisch. Zusätzlich kann der Nachkomme neue Datenfelder und/oder Methoden einführen. Er kann auch vorhandene Methoden *überschreiben*, also unter dem gleichen Namen eine andere Methode einführen. Da so verschiedene Nachkommen unter dem gleichen Namen unterschiedliche, auf sie angepasste Methoden ausführen können, spricht man von *Poly-morphie*. Eine besonders effiziente Art, polymorphe Methoden zu realisieren, sind *virtuelle Methoden*, auf die wir später eingehen.

Delphi-Objekte existieren in der Windows-Umgebung und müssen mit dieser kommunizieren. Insbesondere müssen sie die *Nachrichten* des Windows-Systems wie Mausklicks, Tastendrucke, Ändern der Fenstergröße, ... erhalten und darauf reagieren. Solche *Ereignisse* (*events*) werden in Windows durch eine Fülle von Konstanten übermittelt, auf die jeweils detailliert eingegangen werden muss. Viel einfacher ist es, die *Ereignisbehandlungsroutinen* von Delphi zu benutzen, in denen das Verhalten auf bestimmte Ereignisse „gekapselt“ ist. Diese liegen teilweise schon als Methoden der „Urklasse“ *tObject* vor, von der alle Klassen abgeleitet werden (auch wenn diese Vererbung nicht explizit angegeben wird). Die „Mutter aller Klassen“ reagiert auf zahlreiche Ereignisse – durch Nichtstun. Diese leeren Methoden werden von Tochterklassen bei Bedarf geeignet ersetzt, so dass von diesen nur auf die wirklich benötigte Ereignisse eingegangen werden muss.

Als Beispiel können die anfangs leeren Fensterobjekte von Delphi dienen, die von der Mutterklasse *tForm* abgeleitet werden. Diese Fenster enthalten zwar noch keine Daten und speziellen Methoden, sie können aber sehr viel: man kann sie verkleinern, vergrößern, verschieben und schließen. Sie verfügen über Methoden, die auf Ereignisse reagieren (*OnCreate*, *OnClick*, *OnKeyDown*, ...). Während unserer Arbeit ergänzt die Delphi-Entwicklungsumgebung die automatisch erzeugte Tochterklasse *tForm1* um Datenfelder (Buttons, Panels, ...) und stellt uns bei Bedarf (Doppelklick auf das entsprechende Feld des Objektinspektors) den leeren Rumpf einer Ereignisbehandlungsroutine zur Verfügung, den wir mit Pascal-Anweisungen ausfüllen.

```

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

var Form1: TForm1;

```

tForm ist Mutterklasse von TForm1

eine Instanz der Klasse TForm1 wird deklariert

Die mithilfe der *var*-Vereinbarung erzeugte Instanz, also das Objekt, ist in Delphi ein *Zeiger* der uns schon bekannten Art. Ein Objekt zeigt also auf einen Speicherbereich, in dem sich die benutzten Datenfelder befinden. Zusätzlich (bei virtuellen Methoden) können

sich dort Informationen über die für dieses Objekt gültigen Methoden befinden. Ein neu vereinbartes Objekt enthält wie jeder neue Delphi-Zeiger den Wert *nil* und zeigt deshalb ins Leere. Der Zugriff auf Daten oder Methoden des Objekts muss zu diesem Zeitpunkt scheitern, weil es das gesuchte Objekt noch gar nicht gibt. (Deshalb führt z. B. der Versuch, den Elementen eines Delphi-Fensters im *Initialization*-Teil der Unit Werte zuzuweisen, zu einem Fehler.)

Ein Delphi-Objekt erhält einen konkreten Wert durch den Aufruf einer (oft geerbten) *Konstruktor*-Methode, die einem *new*-Befehl entspricht. Meist heißt dieser Konstruktor *create*. Erst *nach* diesem Aufruf existiert das gesuchte Objekt. (Bei Delphi-Fenstern wird durch die *create*-Methode das *OnCreate*-Ereignis ausgelöst. In der entsprechenden Ereignisbehandlungsmethode haben wir dann Anfangswerte wie Farben etc. der Elemente des Fensters gesetzt.) Im Unterschied zu „normalen“ Zeigern braucht man Objekt(zeiger) nicht zu dereferenzieren. Man greift also auf das Objekt einfach über den Objektnamen zu, und nicht über *Objektname*[^] (mit nachgestelltem *^-Operator*). Der Speicherbereich von nicht mehr benötigten Objekten kann durch den Aufruf der geerbten *free*- oder *destroy*-Methoden wieder freigegeben werden (die einem *dispose*-Aufruf entsprechen).

Vereinbarung einer Klasse:

```
type tBeispiel = class
    Name: string;
    ...
end;
```

Vereinbarung einer Instanz dieser Klasse:

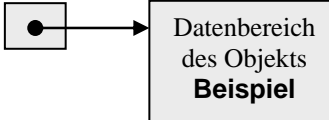
```
var Beispiel: tBeispiel;
```



(der Zeiger zeigt ins Leere)

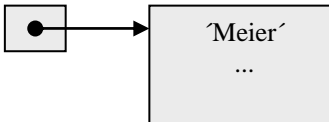
Aufruf des Konstruktors *Create*:

```
Beispiel := tBeispiel.create;
```




Zuweisung eines Werts an ein Datenfeld:

```
Beispiel.Name := 'Meier';
```



Aufruf der *free*-Methode:

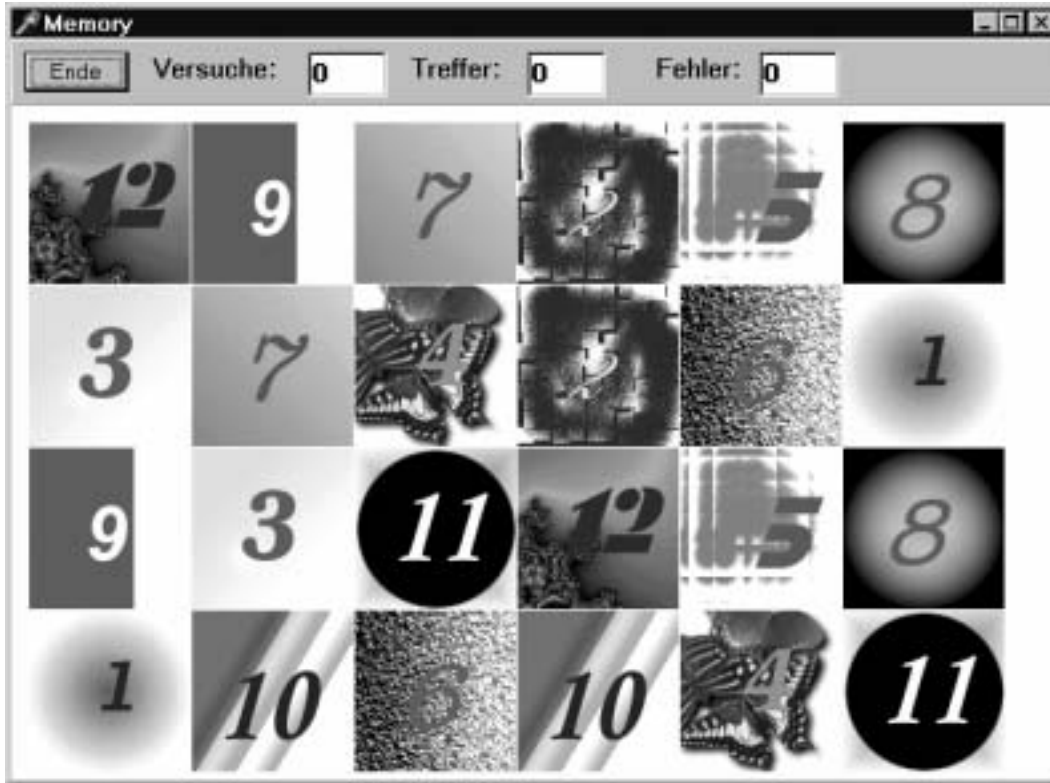
```
Beispiel.free;
```



(der Zeiger zeigt wieder ins Leere)

1.2.2 Die Klasse tKarte und ihre Methoden

Die Karten unseres Memory-Spiels sollen unterschiedliche Bilder enthalten, wenn sie „aufgedeckt“ sind. Andernfalls zeigen sie alle dieselbe „Rückseite“. Bilder und Rückseite können wir mit einem Grafikprogramm erzeugen und als Bitmap-Datei speichern (Dateiendung *BMP*). Decken wir alle Karten auf, dann erhalten wir – je nach Mühe, die wir uns mit den Kartenzeichnungen machen – z. B. das folgende Bild.



Unsere Karten enthalten als Daten also eine Nummer und die boolesche Größe *verdeckt*, die angibt, mit welcher Seite die Karte „oben“ liegt. Zusätzlich wollen wir die Position der Karte durch Angabe der Zeile *ni* und der Spalte *nj* vermerken. Als Methoden der Kartenklasse führen wir *ZeigeDich* ein, die aus der Kartennummer und der Lage das richtige Bild ermittelt und am Bildschirm anzeigt, sowie eine Ereignisbehandlungsroutine *BeiClick*, die auf das *OnClick*-Ereignis reagieren soll. Die Möglichkeit, Bilder zu laden und anzuzeigen, sollen unsere Karten von den *Image*-Komponenten erben. Die Klasse *tImage* wird deshalb als Mutterklasse gewählt.

```

tKarte = class(tImage)
  procedure BeiClick(Sender: TObject);
  public
  Nummer,ni,nj: integer;
  verdeckt    : boolean;
  procedure ZeigeDich;
  end;

```

öffentlich zugängliche
Felder und Methoden

Zum Anzeigen des richtigen Bildes benutzen wir die geerbten Fähigkeiten von *tImage*. Dieses enthält eine *Picture*-Komponente, die eine Bitmap-Datei einlesen und auf einer *Canvas* darstellen kann.

```

procedure tKarte.ZeigeDich;
begin
  if verdeckt
  then picture.LoadFromFile('rueck.bmp')
  else picture.LoadFromFile(Dateiname(Nummer));
  canvas.draw(0,0,picture.graphic);
end;

```

je nach Zustand unter-
schiedliche Bilder laden ...

... und anzeigen

Mithilfe der Nummer einer Karte lässt sich die richtige Bilddatei leicht finden.

```

function Dateiname(n: integer): string;
begin
  case n of
    1: Dateiname := 'eins.bmp';
    2: Dateiname := 'zwei.bmp';
    3: Dateiname := 'drei.bmp';
    4: Dateiname := 'vier.bmp';
    5: Dateiname := 'fuenf.bmp';
    6: Dateiname := 'sechs.bmp';
    7: Dateiname := 'sieben.bmp';
    8: Dateiname := 'acht.bmp';
    9: Dateiname := 'neun.bmp';
    10: Dateiname := 'zehn.bmp';
    11: Dateiname := 'elf.bmp';
    12: Dateiname := 'zwoelf.bmp';
  end
end;

```

aus der Nummer *n*
den Namen der
Bilddatei bestimmen

Es bleibt noch die Ereignisbehandlungsmethode der Kartenklasse zu definieren. In dieser wird (nach den Spielregeln von *Memory*) zwischen dem Ziehen der ersten bzw. der zweiten Karte unterschieden. Wird die erste Karte gezogen, dann wird in ein entsprechendes Unterprogramm verzweigt, das u. a. die Karte anzeigt. Beim Ziehen der zweiten Karte wird auch diese Karte aufgedeckt. Danach wird ein *Timer* gestartet, um das Bild für eine kurze Zeit zu bewahren. In der *Timer-Methode* des Timers wird (nach der Wartezeit)

das Unterprogramm zur Behandlung der zweiten Karte aufgerufen. Zu Beginn wird einer Variablen *Karte* vom Typ *tKarte* (also einem Zeiger) für spätere Weiterbearbeitung der Wert der gerade gezogenen Karte zugewiesen. Dieses geschieht mithilfe des *self*-Parameters, über den jede Methode des Objekts verfügt. *Self* liefert einen Zeiger auf das Objekt, das die Methode aufgerufen hat. Über diesen erhält die Methode Zugang zu den Daten des Objekts.

```

procedure tKarte.BeiClick(Sender: TObject);
var i,j: integer;
begin
Karte := self;
if ErsteKarte then BehandleErsteKarte
  else begin
    verdeckt := false;
    ZeigeDich;
    Memory.Timer1.enabled := true;
  end;
end;

procedure TMemory.Timer1Timer(Sender: TObject);
begin
BehandleZweiteKarte;
Timer1.enabled := false;
end;

```

aktuelle Karte mithilfe von *self* merken

Timer aktivieren

Timer deaktivieren

1.2.3 Karten als Fensterelemente erzeugen und wieder löschen

Unser Fenster enthält nicht einzelne Karten, sondern ein ganzes Kartenfeld, dessen Dimensionen wir mithilfe von Konstanten festlegen können. In unserem Fall wollen wir vier Zeilen mit jeweils sechs Spalten benutzen.

```
const nMax = 6; mMax = 4;
```

Damit können wir einen geeigneten Feldtyp vereinbaren, ein Feld von Karten:

```
tKarten = array[1..nMax,1..mMax] of tKarte;
```

Nachdem unser Fenster (mit dem Namen *Memory*) erzeugt wurde (also in der Methode, die auf das *OnCreate*-Ereignis des Fensters reagiert), können wir die 24 benötigten Karten instantiiieren und anzeigen. Dazu gehen wir in der folgenden Reihenfolge vor:

1. **Das Objekt erzeugen** (mit der *create*-Methode). Dabei wird als „Besitzer“ (*owner*) des Objekts das aktuelle Fenster mithilfe des *self*-Parameters angegeben. Der Besitzer kontrolliert u. a. die Existenzdauer eines Objekts: wird er gelöscht, dann wird auch der Speicherbereich des Objekts freigegeben.

```
Karten[i,j] := tKarte.create(self);
```


2. Die **parent**-Eigenschaft setzen, und dadurch dem erzeugten Objekt mitteilen, wer das „Elternobjekt“ ist, auf dem es angezeigt wird.
`Karten[i,j].parent := self;`
3. Den Datenfeldern des Objekts Werte zuweisen.
`Karten[i,j].verdeckt := true;`
4. Die (leeren) **Ereignisbehandlungsroutinen** ggf. durch eigene **ersetzen**. (hier: dem *OnClick*-Ereignis die *BeiKlick*-Methode zuweisen.)
`Karten[i,j].OnClick := BeiKlick;`

Insgesamt ergibt sich die *FormCreate*-Methode des Memory-Fensters wie folgt:

```

procedure TMemory.FormCreate(Sender: TObject);
var i,j,n : integer;
    Nummern: set of 1..24;
begin
    nummern := [1..24];
    randomize;
    Timer1.enabled := false;
    for i := 1 to nMax do
        for j := 1 to mMax do
            begin
                repeat n := random(24)+1;
                until (n in Nummern);
                Nummern := Nummern - [n];
                if n > 12 then n := n - 12;
                Karten[i,j] := tKarte.create(self);
                with karten[i,j] do
                    begin
                        parent := self;
                        name := 'Karte'+IntToStr(i)+IntToStr(j);
                        left := 10+(i-1)*100+i;
                        top := panell.height+10 + (j-1)*100+j;
                        width := 100; height := 100;
                        Nummer := n;
                        ni := i; nj := j;
                        verdeckt := true;
                        OnClick := BeiKlick;
                        ZeigeDich;
                    end
                end;
            end;
        end;
    ErsteKarte := True;
    v := 0; t := 0; f := 0;
    SchreibeNachricht;
end;

```

The diagram consists of several callout boxes pointing to specific lines of code in the `FormCreate` procedure:

- Timer deaktivieren**: Points to `Timer1.enabled := false;`
- alle Feldelemente durchlaufen**: Points to the `for i := 1 to nMax do` loop header.
- Kartennummern zufällig bestimmen, so dass jede zweimal auftritt**: Points to the `repeat n := random(24)+1; until (n in Nummern);` block.
- Objekt erzeugen**: Points to `Karten[i,j] := tKarte.create(self);`
- Eltern bestimmen**: Points to `parent := self;` inside the `with` block.
- Position und Größe festlegen**: Points to the lines `left := 10+(i-1)*100+i;` and `top := panell.height+10 + (j-1)*100+j;`
- Position und Nummer merken**: Points to `ni := i; nj := j;`
- Ereignisbehandlungsmethode zuweisen**: Points to `OnClick := BeiKlick;`
- Anzahl der Versuche, Fehler und Treffer auf Null setzen und anzeigen**: Points to the final initialization lines `v := 0; t := 0; f := 0;` and `SchreibeNachricht;`

Beim Schließen des Fensters können wir den Speicherbereich der erzeugten Karten wieder freigeben. Das geschieht innerhalb zweier geschachtelter Schleifen.

```

procedure TMemory.FormClose(Sender: TObject;
                                var Action: TCloseAction);
var i,j: integer;
begin
  for i := 1 to nMax do
    for j := 1 to mMax do Karten[i,j].free
end;

```

Speicher freigeben

1.2.4 Memory spielen

Der zeitliche Ablauf des Memoryspiels wird durch die *BeiKlick*-Methode unserer Kartenobjekte festgelegt. Wir müssen jetzt nur noch vereinbaren, was beim „Umdrehen“ der Karten zu geschehen hat. Mithilfe der booleschen Variablen *ErsteKarte* merken wir uns, welche Karte gerade „dran“ ist. Da die Variable *Karte* einen Zeiger auf die angeklickte Karte enthält, müssen wir uns nur noch die aktuelle Position (für spätere die Bearbeitung) und die Kartennummer merken. Danach wird die Karte „aufgedeckt“.

```

procedure HandleErsteKarte;
begin
  if Karte.Nummer = 0 then exit;
  i1 := Karte.ni;
  j1 := Karte.nj;
  N1 := Karte.Nummer;
  Karte.verdeckt := false;
  Karte.ZeigeDich;
  ErsteKarte := false;
end;

```

Karten mit der Nummer „0“ sind schon vergeben!

Position und Nummer merken

Karte zeigen

danach kommt die zweite Karte!

Die Behandlung der zweiten Karte ist etwas komplizierter. Das Bild wurde schon gezeigt, und etwas gewartet wurde auch schon. Jetzt muss je nach Kartennummer reagiert werden:

Position und Nummer merken	
Die Kartennummern sind gleich	
wahr	falsch
Beide Karten auf dem Bildschirm löschen und die Kartennummern auf „0“ setzen	Beide Karten verdeckt zeigen
ErsteKarte ← TRUE	
Ergebnisse anzeigen	

```

procedure BehandleZweiteKarte;
begin
if Karte.Nummer = 0 then exit;
i2 := Karte.ni;
j2 := Karte.nj;
N2 := Karte.Nummer;
v := v + 1;
if N1=N2 then
begin
with Memory.Karten[i1,j1] do
begin
t := t + 1;
canvas.brush.style := bsSolid;
canvas.brush.color := clWhite;
canvas.rectangle(0,0,100,100);
Nummer := 0;
end;
with Memory.Karten[i2,j2] do
begin
canvas.brush.style := bsSolid;
canvas.brush.color := clWhite;
canvas.rectangle(0,0,100,100);
Nummer := 0;
end;
end
else begin
f := f + 1;
with Memory.Karten[i1,j1] do
begin
verdeckt := true;
ZeigeDich
end;
with Memory.Karten[i2,j2] do
begin
verdeckt := true;
ZeigeDich
end;
end;
ErsteKarte := true;
SchreibeNachricht;
end;

```

Karten mit der Nummer "0" sind schon vergeben!

Position und Nummer merken

Nummernvergleich

erste Karte löschen

zweite Karte löschen

erste Karte verdecken

zweite Karte verdecken

Ergebnisse zeigen

Wenn wir drei Editierfelder *nVersuche*, *nTreffer* und *nFehler* eingefügt haben, so können wir die Werte der Variablen *v* (Versuche), *t* (Treffer) und *f* (Fehler) wie üblich darstellen.

```

procedure SchreibeNachricht;
begin
with Memory do
begin
nVersuche.text := IntToStr(v);
nTreffer.text  := IntToStr(t);
nFehler.text   := IntToStr(f);
end
end;

```

Die gesamte Unit ergibt sich aus den Ergebnissen des am Bildschirm zusammengestellten Fensters mit Knöpfen, Editierfeldern usw. und braucht deshalb nicht extra angegeben zu werden.

1.2.5 Aufgaben

1. Ergänzen Sie das *Memoryspiel* so, dass
 - a: nach einem Spiel ein weiteres stattfinden kann.
 - b: zwei Spieler spielen können.
 - c: „gemogelt“ werden kann. Nach Anklicken einer Karte mit der rechten Maustaste erscheint die Rückseite kurz am Bildschirm, ohne dass der Versuch als Fehler gezählt wird.
 - d: der Computer „mogelt“. Manchmal rückt er heimlich eine Karte an eine andere Stelle.
2. a: Führen Sie *Kartenstapel* ein, in denen die Karten (verdeckt oder offen) in bestimmten Abständen angeordnet werden.
 - b: Ändern Sie die Klasse *tKarte* so, dass Karten mit der Maus verschoben werden können. Schiebt man sie auf einen Stapel, dann werden sie in diesen eingeordnet.
 - c: Entwickeln Sie mithilfe eines Grafikprogramms einen vollständigen *Satz von Spielkarten*.
 - d: Erfinden Sie jetzt Kartenspiele nach dem Vorbild der *Patiencen*. Nach unterschiedlichen Spielregeln werden Karten aufgedeckt und verschoben. Ziel des Spieles ist es meist, die Karten in sortierten Kartenstapeln anzuordnen.

3. a: Scannen Sie eine detailreiche Bildvorlage und bringen Sie das Bild in ein Grafikprogramm. Zerlegen Sie es dort in quadratische Teile, die Sie einzeln abspeichern.
 - b: Füllen Sie Karten (nach dem Vorbild des Memoryspiels) mit diesen Teilbildern und stellen Sie die Karten zufällig verteilt am Bildschirm dar.
 - c: Fügen Sie aus diesen **Puzzle**-Steinen ein vollständiges Bild zusammen, indem Sie die Karten durch Anklicken von ihrer Position in ein Raster bringen. Dort müssen die Karten natürlich auch noch verschiebbar sein.

4. a: Erzeugen Sie nach dem Vorbild der **elektronischen Bausteinkästen** der Physiksammlung unterschiedliche Karten mit Schaltsymbolen (Widerständen, Transistoren, Leitungen, ...).
 - b: Erzeugen Sie verkleinerte Abbilder („Icons“) dieser Schaltsymbolkarten am Bildschirm.
 - c: Schaffen Sie eine Möglichkeit, durch Anklicken der Icons entsprechende Karten an vorgegebenen Plätzen in einem Raster zu erzeugen. Weshalb sind „Kartenlisten“ hierfür geeignet?
 - d: Erzeugen Sie auf diese Weise Schaltpläne.