Types

TheShadow 2003

www.blitzbase.de

Type-Befehlssatz ist eine sehr wichtige Erweiterung des BlitzBasic-Dialekts. Jedoch ist es ein sehr komplexes Thema und besonders Anfänger haben große Schwierigkeiten bei Verwendung von Types. Oft liegt es an anderer Vorstellung wie Types überhaupt funktionieren. Ich selbst war zuerst etwas verwirrt, da Types in anderen Programmiersprachen ganz anders funktionierten. Selbst jetzt gibt es wahrscheinlich nur wenige BB-Programmierer, die alle Tricks kennen. Dieses Tutorial soll endlich Schluss damit machen und alle Möglichkeiten erklären.

Zuerst soll noch gesagt werden, dass Types keine Universallösung darstellen. Es gibt einige Bereiche wo Types vollkommen ungeeignet sind. Meistens kann man etwas sogar ganz ohne Types lösen. Manchmal sind Types jedoch unvermeidbar und manchmal machen Types die Programmierung deutlich einfacher.

Wozu überhaupt Types? Wenn man große Projekte hat, dann benötigt man auch sehr viele Variablen. Oft hat man sogar viele ähnliche Variablengruppen, die man bei großen Programmen aber nur schwer in DIM-Feldern zusammenhalten kann. Types fassen wiederholende Variablen zusammen.

Hier wird die Benutzung empfohlen:

1) Die Anzahl der Objekte ist nicht bekannt oder variiert ständig. Beispielhaft dafür ist z.B. ein Alien-Shooter, wo dauernd Aliens abgeknallt werden. Manchmal sind nur wenige Aliens auf dem Bildschirm, manchmal kommen viele dazu.

2) Bei Verwendung von Objekten, wo man direkten Zugriff auf Eigenschaften haben möchte. Beispielhaft hierfür ist ein GUI mit Fenstern, Buttons, Checkboxen. Dabei möchte man z.B. den Status von Checkbox abfragen oder die Position der Objekte.

Hier wird die Benutzung nicht empfohlen:

1) Bei hoher Objektanzahl (über 10.000) wird die Benutzung nicht empfohlen. Dies wirkt sich sehr negativ auf den Speicher aus (dazu später mehr). Hier wird entweder DIM oder BANK empfohlen.

2) Bei statischer Objektanzahl, wo während des Programmablaufs niemals die Anzahl geändert wird. Hier wird auch entweder DIM oder BANK empfohlen. Ein großer Fehler ist z.B. die Verwaltung von Karteninformationen (Map) mit Types.

1. Definition

Der Befehl TYPE definiert eine neue Typekollektion mit

einem bestimmten Namen. Mit END TYPE wird diese Definition abgeschlossen. So sieht eine leere Definition aus, die in BlitzBasic sogar zugelassen wird:

```
TYPE person
END TYPE
```

Zwischen diesen Zeilen lassen sich beliebig viele unterschiedliche Variablen (Eigenschaften) mit FIELD notieren. Es können auch beliebige Variablenarten sein (Integer, Float, String). Dies könnte so aussehen:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE
```

Achtung! Diese Typedefinition kann nur im Hauptprogramm stehen. Dabei spielt es aber keine Rolle wo. Man kann es sogar in eine Includedatei auslagern. Zugriffe auf diese Typekollektion sind auch aus Funktionen erlaubt. Besser gesagt: die Typekollektion ist immer global verfügbar.

2. Variablen

Bisher haben wir nur eine Typedefinition. Damit lässt sich aber noch gar nichts machen. Um Zugriff auf Objekte zu erhalten, werden ganz spezielle Variablen (Containervariablen) benötigt. Ich kann nur empfehlen solche Variablen mit LOCAL/GLOBAL zu deklarieren. Diese Variablen unterscheiden sich nur wenig von normalen Variablen - es wird lediglich ein Punkt mit Typebezeichnung dazugeschrieben:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL mutter.person
GLOBAL vater.person
```

Dieser Anhang (Punkt mit Typebezeichnung) ist sehr wichtig. BB muss ja wissen zu welchem Type so eine Variable zugeordnet wurde. Übrigens kann dieser Anhang bei späterer Verwendung sogar ganz entfallen. Dies funktioniert ja auch mit normalen Variablen. So kann man in dem Beispiel ja auch den Anhang # weglassen:

```
LOCAL float#
float=0.1
```

Alle Type-Variablen haben direkt nach der Definition keinen Wert. Dies kann man sich schwer vorstellen, wenn man z.B. mit QuickBasic programmiert hat. Dort konnte man direkt nach der Definition bereits auf TYPE- Eigenschaften zugreifen - es existierte bereits ein Objekt. In BlitzBasic wurde aber lediglich eine Variable ohne Inhalt angelegt. Ich hoffe, dieses Beispiel kann das etwas verständlicher machen:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL integer% ;Wert ist gleich 0
LOCAL   vater.person   ;Wert   ist
gleich NULL
```

In BlitzBasic gibt es keinen Zwang eine Variable mit LOCAL/GLOBAL du deklarieren. So kann man ja "on-the- fly" beliebige Variablen anlegen. Dies funktioniert aber auch mit Type-Variablen:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

integer%=0
vater.person=NULL
```

Achtung! Während Zugriffe auf eine Typekollektion auch aus Funktionen erlaubt sind, kann eine Type- Variable global oder lokal sein. Somit kann man Zugriffe von Funktionen aus, auf solche Variablen erlauben oder verweigern.

3. Erstellen

Bisher hat sich noch gar nichts getan. Wir haben eine Typedefinition und eine Variable für einen Zugriff angelegt. Jetzt fangen wir endlich an Objekte anzulegen. Dazu gibt es den Befehl NEW. Damit reservieren wir Speicher und legen somit ein Objekt an:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person
LOCAL mutter.person

vater.person=NEW person
mutter.person=NEW person
```

Dieser Code ist auch erlaubt (wir erinnern uns: Anhang kann nach einer Definition ausgelassen werden):

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person
LOCAL mutter.person

vater=NEW person
mutter=NEW person
```

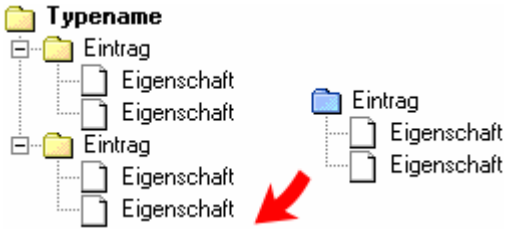
Dieser Code funktioniert auch (man kann ja Werte direkt zuweisen):

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person=NEW person
LOCAL mutter.person=NEW person
```

Das geht ebenfalls (ist ja standardmäßig lokal):

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

vater.person=NEW person
mutter.person=NEW person
```

Beachte: Jedes Mal, wenn ein Objekt erstellt wird, wird dieses Objekt ganz hinten an unsere Typekollektion angehängt.



4. Verlinkungen

Endlich existiert ein Objekt. Dieses Objekt wird in eine Liste der Typekollektion eingefügt. Die Type-Variable "vater" wurde mit einem Objekt verbunden und die Type-Variable "mutter" wurde mit einem zweiten Objekt verbunden. Diese Verbindungen können aber auch wieder aufgehoben werden:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person
LOCAL mutter.person

vater=NEW person
mutter=NEW person
mutter=NULL
```

In diesem oberen Beispiel wurden zwei Objekte erstellt und Type-Variable "mutter" wurde wieder von dem Objekt abgekoppelt. Das eigentliche Objekt existiert aber weiterhin in der Liste. Es wurde nicht gelöscht!!! Man hat nur keinen Zugriff mehr auf dieses Objekt. Wie man dennoch eine Verlinkung zu einem Objekt aufbaut, werde ich etwas später erklären. Ich möchte euch jetzt besser auf eine kleine Falle hinweisen, die bei Funktionen auftreten kann:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

FUNCTION test()
   LOCAL vater.person
   vater=NEW person
END FUNCTION

test()
```

Hier wird ein neues Objekt in der Funktion erstellt. Es wird beim Beenden aber nicht freigegeben. Ist ja auch klar: Typekollektionen sind ja immer global. Lediglich die Type-Variable "vater" wird beim Beenden vernichtet. Wenn ihr jetzt die Funktionsweise verstanden habt, dann sollte dieser Code für euch einleuchtend sein:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person
LOCAL mutter.person

vater=NEW person
mutter=NEW person
mutter=vater
```

Ich glaube, jetzt werden einige aber einen Aufstand machen: Mutter soll gleich Vater sein??? Ja, denn wir verlinken Mutter-Variable einfach mit dem Objekt von Vater-Variable. So wie bei dem oberen NULL-Beispiel geht auch hier ein direkter Zugriff auf Mutter-Objekt verloren. Mutter-Objekt bleibt aber weiterhin bestehen - nur haben wir wieder keinen Zugriff drauf. Für alle, die das schwer verdauen können: Auf Vater-Objekt kann man jetzt mit zwei Variablen zugreifen, auf Mutter- Objekt aber überhaupt nicht mehr. Somit kann man auch viele Spielereien machen - Z.B. kann man die Verlinkungen zwischen Vater und Mutter vertauschen:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person
LOCAL mutter.person
LOCAL tmp.person
```

```
vater=NEW person
mutter=NEW person

tmp=vater
vater=mutter
mutter=tmp
```

5. Zugriff

Jetzt können wir neue Objekte erstellen und damit wie die Profis hantieren. Allerdings konnten wir bisher noch nicht auf Eigenschaften zugreifen und Werte manipulieren. Das ist aber kinderleicht - es funktioniert fast wie in zich anderen Programmiersprachen auch:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person

vater=NEW person
vater\name$="Homer"
vater\adresse$="Springfield"
vater\alter%=40
vater\einkommen#=2345.67
```

Wie man sieht, wird lediglich Type-Variable angegeben. Nach einem Schrägstrich folgt Name der Eigenschaft und Wertzuweisung. In vielen anderen Programmiersprachen wird statt Schrägstrich ein Punkt verwendet. Warum wurde das nicht auch in BB so gemacht? Dies kommt daher, weil unser Punkt schon für die Kennzeichnung einer Type-Variable vergeben wurde. Besser gesagt: Der obere Code müsste eigentlich so aussehen (dies ist aber länger und unüblich):

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person

vater.person=NEW person
vater.person\name$="Homer"
vater.person\adresse$="Springfi
eld"
vater.person\alter%=40
vater.person\einkommen#=2345.67
```

In dem oberen Beispiel haben wir Werte zugewiesen. Irgendwie müssen wir diese Werte nun auslesen. Das geht ebenfalls sehr einfach:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person

vater=NEW person
vater\name$="Homer"
vater\adresse$="Springfield"
vater\alter%=40
vater\einkommen#=2345.67

PRINT vater\name$
PRINT vater\adresse$
PRINT vater\alter%
PRINT vater\einkommen#
```

Beachte: Man kann nur Werte der Eigenschaften ausgeben. Jedoch kann man nicht einfach "PRINT vater" schreiben. Dies wäre ein Fehler und Programm würde nicht starten.

Achtung! Unsere Variable "vater" muss natürlich existieren (z.B. LOCAL vater.person). Zudem muss diese Variable mit einem Objekt verbunden sein (hier passierte das über NEW). In diesem Code sind darum gleich zwei Fehler:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

vater\name$="Homer"
vater\adresse$="Springfield"
vater\alter%=40
vater\einkommen#=2345.67
```

Wer seine Programme auf maximale Sicherheit trimmen möchte, der prüft vorher, ob eine Variable mit einem Objekt verbunden ist. Dies geschieht so (Ausschnitt):

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE
```

```
LOCAL vater.person

...

IF vater<>NULL THEN
   vater\name$="Homer"
   vater\adresse$="Springfield"
   vater\alter%=40
   vater\einkommen#=2345.67
ENDIF
```

6. Löschen

Nun wissen wir endlich wie wir ein Objekt erzeugen können. Aber es war bisher nicht entfernbar. Tja, dazu gibt es wieder einen neuen Befehl DELETE. Als Parameter wird lediglich eine Variable angegeben:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person

vater=NEW person
DELETE vater
```

In diesem Beispiel wurde ein Objekt aus unserer Typekollektion komplett entfernt. Die Variable "vater" bleibt aber weiterhin bestehen. Diese bekommt jetzt aber den Wert NULL. Somit haben wir auch den Zugriff auf unser Objekt verloren.

Oft möchte man jedoch gleich alle Objekte einer Typekollektion löschen. Dies geht ebenfalls sehr sehr einfach. Beachte: Jetzt wird aber nicht mehr eine Variable, sondern Typename als Parameter mit DELETE angegeben:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person
LOCAL mutter.person

vater=NEW person
mutter=NEW person
DELETE EACH person
```

Achtung! Es gibt eine große Falle, die recht unbekannt ist. Ein Objekt wird nicht wirklich aus dem Speicher gelöscht. Intern wird es lediglich als ungültig markiert und der Speicher nicht freigegeben.

Es basiert an der Annahme: Wenn X Objekte verwendet wurden, dann werden irgendwann später erneut mindestens so viele Objekte benötigt. Die Erstellung von vielen Objekten auf einen Schlag dauert lange, dafür bleibt es später aber konstant. Der einzige Grund war die Erhöhung der Geschwindigkeit. Nach dem Beenden des Programms wird aber natürlich der komplette Speicher freigegeben.

Falls dieser Effekt zu negativ auf den Speicher auswirkt (bei vielen Objekten), dann schlage ich vor entweder DIM oder BANK zu benutzen. Nach der Verwendung von DIM oder BANK wird der Speicher natürlich wieder freigegeben.

7. Linked List

Bisher hatten wir nur einzelne Objekte. BlitzBasic-Types sind aber viel trickreicher. Wir erinnern uns: Einzelne Objekte werden immer ganz hinten in eine globale Typekollektion eingefügt. Dies kann man sich wie eine einfache Liste vorstellen. Warum fügen wir dann nicht einfach zich solche Objekte ein?

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=RAND(0,640)
   a\y=RAND(0,480)
NEXT
```

Die Variable "a" wurde nach dem Befehl NEW immer wieder mit einem neuen Objekt verlinkt. Und dann wurde den Eigenschaften "x" und "y" ein Zufallswert zugewiesen. Alle 10 Objekte bleiben aber erhalten - nur wir haben keinen direkten Zugriff mehr auf alle Objekte. Jetzt kommt aber der Trick: Alle Objekte in dieser Typekollektion sind über unsichtbare Fäden nacheinander miteinander verbunden. Und mit FOR...EACH-Schleife kann man abwechselnd alle Objekte abarbeiten:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=RAND(0,640)
   a\y=RAND(0,480)
NEXT
```

```
FOR a=EACH alien
   PRINT a\x
   PRINT a\y
NEXT
```

Beachte: Zuerst wird das erste Objekt ausgewählt, die Schleife abgearbeitet und dann das nächste Objekt aus der Liste ausgewählt. Dies dauert solange, bis keine Objekte mehr vorhanden sind. Wenn die Schleife abgearbeitet wurde, dann bekommt unsere Variable "a" den Wert NULL zugewiesen. Wir können aber auch jederzeit die Schleife mit EXIT verlassen (dann bleibt die letzte Verlinkung erhalten):

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=RAND(0,640)
   a\y=RAND(0,480)
NEXT

FOR a=EACH alien
   IF a\x<10 THEN EXIT
NEXT
```

Achtung! In BlitzBasic gibt es keine Type-Instanzen. Es gibt nur eine einzige Type-Kollektion, auch wenn man viele verschiedene Type-Variablen verwendet. Darum funktioniert dieser Code nicht so wie man das auf den ersten Blick vermuten möchte:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien
LOCAL b.alien

FOR i=1 TO 10
   a=NEW alien
NEXT

FOR i=1 TO 10
   b=NEW alien
NEXT

FOR a=EACH alien
   ...
NEXT

FOR b=EACH alien
   ...
NEXT
```

In dem Beispiel wurden 20 Objekte erstellt. Egal über welche Variable man nun alle Objekte abarbeiten möchte, es werden immer die 20 Objekte abgearbeitet. Ich denke das ist auch logisch, da unsere Variable dauernd nur zu einem Objekt gelinkt werden kann.

8. Selektion

Wow - diese neuen Möglichkeiten ermöglichen jetzt schon so viel. Aber es ist noch nicht vorbei! Wir haben bisher nur automatisch alle Objekte mit FOR...EACH abgearbeitet. Kurz zur Erinnerung: Wir konnten ja eine Verbindung zu einem Objekt mit NULL aufheben:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

LOCAL vater.person
LOCAL mutter.person

vater=NEW person
mutter=NEW person
mutter=NULL
```

Wir können aber auch mit speziellen Befehlen erneut Verbindungen aufbauen und sogar eine Typekollektion rauf und runterlaufen. Zuerst die einfacheren Befehle:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=RAND(0,640)
   a\y=RAND(0,480)
NEXT

a=FIRST alien
```

In diesem Beispiel wurden 10 Objekte erstellt. Danach wurde aber eine Verbindung mit dem ersten Objekt in der Liste "alien" aufgebaut. Wir haben also manuell unsere Variable "a" mit dem ersten Objekt verbunden. Dies funktioniert aber auch umgekehrt - man kann den letzten Eintrag auswählen:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=RAND(0,640)
   a\y=RAND(0,480)
NEXT

a=LAST alien
```

Hier gibt es aber auch eine Falle! Was ist, wenn wir gar keine Objekte in der Liste haben - zu was bauen wir dann eine Verbindung auf? Ganz einfach: sollten keine Objekte existieren, dann bekommt unsere Variable "a" den Wert NULL zugewiesen. Dies kann man auch ausnutzen:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

IF FIRST alien=NULL THEN keineobjekte
```

Erste oder letzte Objekte auswählen ist ja nicht gerade spannend. Darum kann man noch manuell vorherige oder nachfolgende Objekte auswählen. Hier ein Beispiel, um einen nachfolgenden Eintrag auszuwählen (AFTER) und einen vorherigen Eintrag auszuwählen (BEFORE):

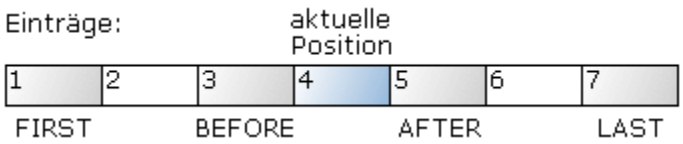
```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien
LOCAl b.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=RAND(0,640)
   a\y=RAND(0,480)
NEXT

a=FIRST alien
a=AFTER a

b=LAST alien
b=BEFORE b
```

Hier gibt es schon kleine Unterschiede. Anders als bei LAST/FIRST wird nicht mehr Typename, sondern Variablename bei AFTER/BEFORE angegeben. Es gibt aber auch noch paar Tricks:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien
LOCAl b.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=RAND(0,640)
   a\y=RAND(0,480)
NEXT

a=AFTER FIRST alien
a=AFTER AFTER a

b=BEFORE LAST alien
b=BEFORE BEFORE b
```

Bei BEFORE und AFTER muss man ebenfalls aufpassen! Was ist, wenn man bereits den ersten Eintrag ausgewählt hat und nun plötzlich einen vorherigen auswählen will. Oder was ist, wenn man bereits den letzten Eintrag ausgewählt hat und nun plötzlich einen nachfolgenden auswählen will. Tja, in dem Fall wird unser Variable der Wert NULL zugewiesen. Vielleicht werden sich jetzt viele fragen: Was bringen diese Befehle? Tja, deshalb habe ich ein kleines Beispiel, in dem alle Objekte in umgekehrter Reihenfolge abgearbeitet werden:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
   a\x=i
   a\y=i
NEXT

a=LAST alien
WHILE a<>NULL
   PRINT a\x
   a=BEFORE a
WEND
```



9. Verschieben

Wie wir wissen, werden alle neue Objekte direkt hinten in eine Liste eingefügt. Wir können aber auch die Reihenfolge der Objekte beeinflussen. Dies ist aber nicht so einfach und bequem. Dazu muss man den Befehl INSERT zusammen mit FIRST, LAST, AFTER oder BEFORE verwenden:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
NEXT

a=FIRST alien
INSERT a AFTER LAST alien

a=LAST alien
INSERT a BEFORE FIRST alien
```

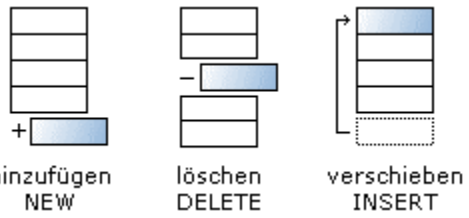
Wie man sieht wird das alles verdammt kompliziert. Das untere Beispiel verschiebt das erste Objekt einen Schritt zurück und das letzte Objekt einen Schritt vor:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien

FOR i=1 TO 10
   a=NEW alien
NEXT

a=FIRST alien
INSERT a AFTER AFTER a
a=LAST alien
INSERT a BEFORE BEFORE a
```

Es gibt noch einige Tricks. Z.B. kann man Objekt "a" hinter oder vor Objekt "b" setzten:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien
LOCAL b.alien
```

```
FOR i=1 TO 10
   a=NEW alien
NEXT

a=AFTER FIRST alien
b=BEFORE LAST alien

INSERT a AFTER b
INSERT b BEFORE a
```



10. DIM/Blitzarray

Bisher haben wir lediglich Variablen für einen Zugriff benutzt. DIM-Felder lassen sich aber auch für den selben Zweck verwenden:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

DIM a.alien(9)

FOR i=0 TO 9
   a(i)=NEW alien
   a(i)\x=i
   a(i)\y=i
NEXT
```

Genau wie Variablen, werden auch DIM-Felder mit Objekten verbunden. Diese Verbindungen lassen sich genauso mit NULL aufheben oder mit FIRST, LAST, AFTER oder BEFORE erneut aufbauen. Hier gibt es wieder einen kleinen Insider-Trick. Man kann ja beliebig oft ein DIM-Feld mit DIM redimensionieren. Dabei geben aber alle Werte verloren. Diese Werte wären bei Types unsere Verlinkungen. Möchtest du kurzzeitig direkte Zugriffe auf viele Objekte gleichzeitig haben, dann geht es auch jederzeit über erneute Verlinkungen - vorausgesetzt man kennt die Anzahl der Objekte:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL tmp.alien
```

```
FOR i=0 TO 9
   tmp=NEW alien
   tmp\x=i
   tmp\y=i
NEXT

DIM a.alien(9)

i=0
FOR tmp=EACH alien
   a(i)=tmp
   i=i+1
NEXT
```

Und woher bekommt man die Objektanzahl? Das geht nicht direkt - man muss also manuell mit FOR...EACH- Schleife zählen. Dieser Hinweis sollte wohl nicht fehlen. Neben DIM-Feldern gibt es in BlitzBasic noch Blitzarrays. Das funktioniert aber gleich, nur die Neudimensionierung gibt es bei Blitzarrays nicht:

```
TYPE alien
   FIELD x
   FIELD y
END TYPE

LOCAL a.alien[9]

FOR i=0 TO 9
   a[i]=NEW alien
   a[i]\x=i
   a[i]\y=1
NEXT
```

Anders als DIM-Felder, können Blitzarrays noch für einen weiteren Trick verwendet werden. Man kann ein Blitzarray als Eigenschaft verwenden. Dieses Feature ermöglicht wieder neue Möglichkeiten:

```
TYPE alien
   FIELD x
   FIELD y
   FIELD etwas[99]
END TYPE

LOCAL a.alien

a=NEW alien
a\etwas[0]=1
a\etwas[1]=2
a\etwas[99]=777
```

11. Linked List 2

Wir konnten im letzten Tutorial Blitzarrays als Eigenschaften benutzen. Können wir auch Type-Variablen als Eigenschaften verwenden? Ja, das ist ebenfalls problemlos möglich:

```
TYPE person
   FIELD name$
   FIELD vater.person
   FIELD mutter.person
END TYPE

vater.person=NEW person
mutter.person=NEW person
bruder.person=NEW person
schwester.person=NEW person

vater\name$="Homer"
mutter\name$="Marge"
bruder\name$="Bart"
schwester\name$="Lisa"

bruder\vater=vater
bruder\mutter=mutter
schwester\vater=vater
schwester\mutter=mutter

PRINT vater\name$
PRINT mutter\name$
PRINT bruder\name$
PRINT schwester\name$
PRINT
PRINT bruder\vater\name$
PRINT bruder\mutter\name$
PRINT schwester\vater\name$
PRINT schwester\mutter\name$
```

Wir haben jetzt Bruder/Schwester-Eigenschaften mit Vater/Mutter-Objekten verbunden. Die ersten vier PRINT-Zeilen geben die eigentlichen Namen der Personen aus. Die letzten vier Zeilen geben dafür die Namen der Eltern von "bruder" und "schwester" aus. So wie es aussieht, entsteht ein richtiges Netz aus Verbindungen. Dies kann die Programmierung komplizierter machen - manchmal ist es aber auch nützlich. Diese Verbindungen lassen sich auch auf verschiedene Typekollektionen ausbauen. Übrigens wird in diesem Beispiel alles über eine Wurzelvariable "fabrik" verarbeitet:

```
TYPE fabrik
   FIELD preis
   FIELD flaeche
   FIELD stromkosten
   FIELD papier.papier
END TYPE

TYPE papier
   FIELD baemeverbrauch
   FIELD papiermenge
END TYPE
```

```
LOCAL fabrik.fabrik

fabrik=NEW fabrik
fabrik\preis=1000000
fabrik\flaeche=1000
fabrik\stromkosten=9999
fabrik\papier=NEW papier
fabrik\papier\baemeverbrauch=11
11
fabrik\papier\papiermenge=123
```

Besonders hier möchte ich aber darauf hinweisen, dass man bei solchen hochgradigen Verbindungen stets die Existenz eines Objekts überprüfen sollte. Ansonsten könnte es Fehlermeldungen hageln. Versuche es selbst: Lösch die Zeile "fabrik\papier=NEW papier" im oberen Code. Dies Problem kann man mit der NULL-Prüfung umgehen:

```
TYPE fabrik
   FIELD preis
   FIELD flaeche
   FIELD stromkosten
   FIELD papier.papier
END TYPE

TYPE papier
   FIELD baemeverbrauch
   FIELD papiermenge
END TYPE

LOCAL fabrik.fabrik

fabrik=NEW fabrik
fabrik\preis=1000000
fabrik\flaeche=1000
fabrik\stromkosten=9999
fabrik\papier=NEW papier
fabrik\papier\baemeverbrauch=1111
fabrik\papier\papiermenge=123

IF fabrik\papier<>NULL THEN PRINT "Papierfabrik"
```

Jetzt werden einige sagen: Was passiert, wenn man die Eigenschaft eines Objekts wieder auf das selbe Objekt legt? Auch wenn das für einige vielleicht unglaublich klingen wird: es ist möglich (wie dieses Beispiel beweist):

```
TYPE test
   FIELD test.test
   FIELD wert
END TYPE

LOCAL test.test
test=NEW test
test\test=test
test\wert=1
```

```
PRINT test\wert
PRINT test\test\wert
PRINT test\test\test\wert
PRINT test\test\test\test\wert
```

12. Funktionen

Wie bereits erwähnt, werden alle Types im Hauptprogramm definiert und sind global verfügbar. Die Variablen selbst können lokal oder global sein. Dieses Beispiel funktioniert nicht, weil lokale Variable "vater" in der Funktion nicht verfügbar ist. Anders als bei normalen Variablen, wird also der Start verweigert:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

FUNCTION test()
   PRINT vater\name$
   PRINT mutter\name$
END FUNCTION

LOCAL vater.person
GLOBAL mutter.person

vater.person=NEW person
mutter.person=NEW person

test()
```

Wie wir wissen, können Type-Variablen auch in Funktionen definiert werden:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

FUNCTION test()
   LOCAL vater.person
   vater=NEW person
END FUNCTION

test()
```

Meistens möchte man eher Type-Variablen an eine Funktion übergeben. Type-Variablen lassen sich problemlos als Parameter verwenden:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

FUNCTION
ausgeben(person.person)
   PRINT person\name$
END FUNCTION

vater.person=NEW person
vater\name$="Homer"
ausgeben(vater)
```

Es gibt jedoch eine Einschränkung. Man kann solchen Parametern keine Standardwerte zuweisen, die benutzt werden, wenn man einen solchen Parameter auslässt (siehe FUNCTION in der Onlinehilfe für mehr Details). Willst du trotzdem keinen Wert übergeben, dann bleibt nur die Benutzung von NULL. Auch eine zusätzliche Prüfung in der Funktion auf Existenz des Objekts ist sinnvoll:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE

FUNCTION
ausgeben(person.person)
   IF person=NULL THEN RETURN
   PRINT person\name$
END FUNCTION

ausgeben(NULL)
```

Viel wichtiger ist jedoch die Erstellung eines Objekts in einer Funktion. Danach soll eine Type-Variable zurückgeliefert werden. Dies geht ebenfalls - schreibe hinter Funktionsnamen einfach einen Punkt mit Typename und gebe eine Type-Variable mit RETURN zurück:

```
TYPE person
   FIELD name$
   FIELD adresse$
   FIELD alter%
   FIELD einkommen#
END TYPE
```

```
FUNCTION
create.person(name$,adresse$,alter%,einkommen#)
   person.person=NEW person
   person\name$=name$
   person\adresse$=adresse$
   person\alter%=alter%
   person\einkommen#=einkommen#
   RETURN person
END FUNCTION

vater.person=create("Homer","Springfield",40,2345.67)
```

13. Handle/Object

Die geheimen Befehle HANDLE und OBJECT sind manchmal sehr nützlich. HANDLE liefert dabei die Identität eines Objekts (und nicht Type-Variable!) als Integerwert zurück. Hier eine kleine Funktion:

```
TYPE window
   FIELD x
   FIELD y
   FIELD w
   FIELD h
END TYPE

FUNCTION window_create(x,y,w,h)
   window.window=NEW window
   window\x=x
   window\y=y
   window\w=w
   window\h=h
   RETURN HANDLE(window)
END FUNCTION

mywin=window_create(50,50,300,200)
```

Jetzt haben wir aber ein Problem: Wenn wir nur einen Integerwert haben, wie sollen wir dann auf unser Objekt zugreifen? Genau dafür gibt es einen Umkehrbefehl: OBJECT. Die Verwendung ist etwas komplizierter, denn neben Integerwert, muss man auch noch Typename angeben. Dies sieht dann so aus:

```
TYPE window
   FIELD x
   FIELD y
   FIELD w
   FIELD h
END TYPE
```

```
FUNCTION window_create(x,y,w,h)
   window.window=NEW window
   window\x=x
   window\y=y
   window\w=w
   window\h=h
   RETURN HANDLE(window)
END FUNCTION

FUNCTION window_draw(id)
   window.window=OBJECT.window(id)
   RECT window\x,window\y,window\w,window\h
END FUNCTION

mywin=window_create(50,50,300,200)
window_draw(mywin)
```

Sollte man zufällig die Identität eines Objekts übergeben, das nicht mit Typename in OBJECT übereinstimmt, dann wird NULL zurückgeliefert. Also, wenn ein Integerwert zu Button gehört und man möchte eine Variable mit Windows-Objekt verbinden, dann wird logischerweise (oder glücklicherweise) NULL zurückgeliefert. Naja, viel nützlicher ist HANDLE/OBJECT bei "Polymorphismus":

```
TYPE window
END TYPE

TYPE button
END TYPE

FUNCTION window_create()
   window.window=NEW window
   RETURN HANDLE(window)
END FUNCTION

FUNCTION button_create()
   button.button=NEW button
   RETURN HANDLE(button)
END FUNCTION

FUNCTION object_delete(id)
   SELECT TRUE
   CASE OBJECT.window(id)<>NULL
      DELETE OBJECT.window(id)
   CASE OBJECT.button(id)<>NULL
      DELETE OBJECT.button(id)
   DEFAULT
      RETURN
   END SELECT
END FUNCTION
```

ACHTUNG! Die Befehle HANDLE/OBJECT sollen überlegt eingesetzt werden. Diese sind in engl. Onlinehilfe nicht dokumentiert, weil diese wahrscheinlich noch geändert werden können. Wenn ihr auf diese Befehle verzichten könnt, dann macht es auch so.

Hinweis: Die Handles werden nicht nach Erstellung eines Objekts, sondern erst nach Verwendung des Befehls HANDLE vergeben.