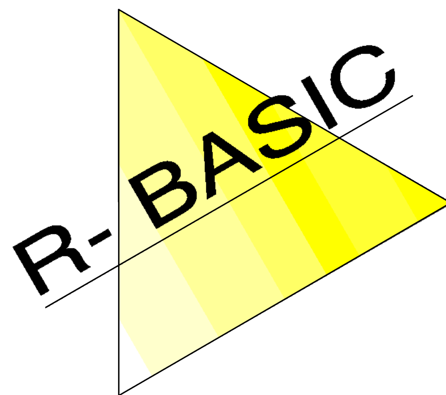
R-BASIC

Einfach unter PC/GEOS programmieren



Objekt-Handbuch

Volume 1
Überblick, Grundlegende Konzepte

Version 1.0

(Leerseite)

Inhaltsverzeichnis

1 Überblick 4

1.1 Ein Beispiel zur Einführung 4
1.2 Grundlegende Begriffe 11
1.3 Vereinbarungen für dieses Handbuch 14
1.4 Syntax für Objektzugriffe 16
1.5 Vereinbarung von Action-Handlern 19

2 Grundlegende Konzepte 21

2.1 Objekte und Objekt-Bäume (Trees) 21

2.1.1 Überblick 21
2.1.2 Arbeit mit Objekten 23
2.1.3 Verwaltung von Objektblöcken (*) 27
2.1.4 Beeinflussung der Objektblöcke im UI-Code (*) 29
2.1.5 Anlegen und Vernichten von Objekten zur Laufzeit (*) 33

(*) Kapitel für Fortgeschrittene

2.2 Ausgabe von Grafik 36

2.2.1 Objekte zur Grafikausgabe 36
2.2.2 Konzepte zur Grafikausgabe 38

2.3 Arbeit mit dem Screen 41

2.3.1 Die Screen-Variable 41
2.3.2 Clipping 43
2.3.3 Speichern und Wiederherstellen des Screen-Status 44
2.3.4 Anpassen des Koordinatensystems 46
2.3.5 Komplexe Manipulation des Koordinatensystems 49

2.4 Objekte individualisieren 51

Willkommen in der Welt der Objekte, Ereignisse und Botschaften

In diesem Handbuch wird beschrieben, wie Sie mit R-BASIC Programme erstellen, die sich vollständig ins System integrieren und sich nach außen nicht von "normalen" (mit dem PC/GEOS SDK erstellten) Programmen unterscheiden.

Zu den grundlegenden Konzepten von GEOS und damit auch von R-BASIC gehören die objektorientierte Programmierung (OOP) und die ereignisorientierte Programmierung. Selbst der "klassische" Modus von R-BASIC ist intern mit OOP realisiert. Hier erfahren Sie, welche Objekte es gibt, welche Eigenschaften und Fähigkeiten sie haben und wie Sie die nutzen können.

Verweise auf andere Kapitel beziehen sich, wenn nicht explizit anderes angegeben, immer auf das Objekt Handbuch.

Um mit diesem Handbuch arbeiten zu können **müssen Sie unbedingt die Kapitel 1.3** (Vereinbarungen für dieses Handbuch) **und 1.4** (Syntax von UI-Objekten) **lesen**. Die dort vorgestellten Sachverhalte werden in allen darauffolgenden Kapiteln vorausgesetzt.

Im Benutzerhandbuch wird erklärt, wie man das R-BASIC Oberfläche benutzt, wie man Programme in andere Sprachen übersetzt und andere Dinge, die nur indirekt mit dem eigentlichen Programmieren zu tun haben.

Grundlegenden Befehle und Konzepte, die die R-BASIC Programmiersprache ausmachen, finden Sie im R-BASIC Programmierhandbuch. Dort erfahren Sie auch alles über Variablen, Schleifen, Verzweigungen, Unterprogramme und andere grundlegende Dinge.

Das Handbuch "Spezielle Themen" widmet sich weiterführenden Themen, wie der Arbeit mit Dateien oder die Verwendung von Schriften.

1 Überblick

1.1 Ein Beispiel zur Einführung

Erstellen der grafischen Oberfläche

In den meisten Fällen beginnt man ein Programm damit, dass man die grafische Oberfläche programmiert.
Starten Sie R-BASIC mit einer neuen Datei. Links unten finden Sie die verfügbaren Code-Fenster. Wir benötigen nur die Fenster "BASIC-Code" und "UI-Objekte". Im Fenster "DIM & DATA" werden bei größeren Projekten globale Vereinbarungen und Variablen untergebracht. Die Codefenster "Init-Code" und "Tools" können wir zunächst ignorieren. Hier kann man Teile seines Programmcodes ablegen um das Programm übersichtlicher zu halten. UI steht für **U**ser **I**nterface (Benutzerschnittstelle) und bezeichnet die Objekte, mit denen der Benutzer des Programms interagieren kann. Das Fenster "UI-Objekte" enthält den **UI-Code** des Programms, der beschreibt, welche Objekte es gibt und welche Eigenschaften sie haben.

Klicken Sie nun auf "UI-Objekte", so sehen Sie ... nichts. Wir müssen zunächst die gewünschten Objekte vereinbaren. Für die ersten Versuche benötigen wir nur ein Programmfenster und eine Möglichkeit, etwas auszugeben. Kopieren Sie dazu einfach den folgenden Code in Ihr "UI-Objekte" Fenster.

```
Application  DemoApplication
  Children = DemoPrimary
END Object

Primary      DemoPrimary
  BreakButton = TRUE
  Children = DemoView
  SizeWindowAsDesired
END Object

View   DemoView
  Content = DemoBitmap
  hControl = HVC_NO_LARGER_THAN_CONTENT + \
                 HVC_NO_SMALLER_THAN_CONTENT
  vControl = HVC_NO_LARGER_THAN_CONTENT + \
                 HVC_NO_SMALLER_THAN_CONTENT
END Object

BitmapContent  DemoBitmap
  bitmapFormat = 640, 400, 8
  DefaultScreen
  defaultColor = YELLOW, LIGHT_BLUE
END Object
```

R-BASIC unterstützt das Anlegen von Objekten sehr komfortabel. Im Menü "Extras" finden Sie den Eintrag "Code Bausteine" und dort "UI-Objekt" und "Neue UI-Sequenz". Bevor wir den UI-Code besprechen sollten Sie sich etwas durch das Menü "Code Bausteine" und seine Untermenüs klicken. Die Verwendung dieser Menüs wird Ihnen sehr viel Schreibarbeit ersparen! Menüpunkte, die jetzt noch grau sind, werden aktiv, wenn Sie im Fenster "BASIC-Code" sind.

Von Klassen und Objekten

Zurück zu unserem UI Code. Das erste Objekt ist vom Typ (man sagt: der Klasse) "Application" und hat den Namen DemoApplication. Dieses Objekt stellt die Verbindung zum GEOS-System her. Jedes Programm muss genau ein solches Objekt haben. Der Namen (hier: DemoApplication) ist frei wählbar. Über diesen Namen kann man das Objekt im BASIC-Code ansprechen.

Die Zeile

```
Children = DemoPrimary
```

stellt die Verbindung zum nächsten Objekt, dem Hauptfenster (Primary-Objekt) her. Mehr zum sogenannten Objekt-Tree finden Sie im Kapitel 2.1.
"SizeWindowAsDesired" (engl: Window-Größe wie gewünscht) legt fest, dass das Primary anfangs nur so groß sein soll, dass alle Children gerade hineinpassen. Ansonsten nehmen Primaries automatisch einen relativ großen Bereich des Bildschirms ein.
Mit END Object wird angezeigt, dass die Definition des Objekts beendet ist.

Das nächste Objekt gehört der Klasse "Primary" an und hat den Namen "DemoPrimary". Man hätte es natürlich auch anders nennen können. Das Primary-Objekt nimmt uns sehr viel Arbeit ab, denn es erzeugt selbständig einige Objekte: das Dateimenü, das System-Menü links oben neben der Titelleiste und die Minimieren-Maximieren-Schalter rechts oben. Auch das Express-Menü wird hier platziert. Das Child (Kind) des Primary-Objekts ist ein View-Objekt, dass in unserem Programm gemeinsam mit seinem "Content" für alle Ausgaben auf den Bildschirm zuständig ist.

Mit der Anweisung

```
BreakButton = TRUE
```

wird festgelegt, dass das Dateimenü einen BREAK-Schalter enthält, mit dem man ein "hängendes" Programm abbrechen kann. Dieser Eintrag aktiviert auch die Tastenkombination "Strg-B" für diese Aufgabe.

Die letzten beiden Objekte sind ein View-Objekt (Objektklasse "View" mit dem Namen "DemoView") und ein Content-Objekt (der Klasse "BitmapContent" namens "DemoBitmap"). Diese beiden Objekte arbeiten eng zusammen. Während das Content ausschließlich für die Verwaltung der darzustellenden Grafik zuständig ist kümmert sich das View-Objekt um das Wie, Wann und Wo. Eine ausführliche Beschreibung dieses Zusammenspiels finden Sie im Kapitel über das View-Objekt (Kapitel 4.6). Dabei ist nur das View-Objekt als Child des Primary-Objekts in den Tree eingebunden.

Die Zeile

```
Content = DemoBitmap
```

legt das Content-Objekt fest. Mit den Zeilen

```
hControl = HVC_NO_LARGER_THAN_CONTENT + \
           HVC_NO_SMALLER_THAN_CONTENT
vControl = HVC_NO_LARGER_THAN_CONTENT + \
           HVC_NO_SMALLER_THAN_CONTENT
```

wird das View angewiesen, seine eigene Größe von der des Content-Objekts abhängig zu machen, indem es sich in horizontaler (hControl) und vertikaler Richtung (vControl) nicht größer (engl. large: groß) oder kleiner (engl. small: klein) als das Content-Objekt macht. Dass stellt sicher, dass die komplette Bitmap stets sichtbar ist. Der Vorsatz HVC_ steht für Horizontal-Vertikal-Control.

Das Objekt "DemoBitmap" verwaltet eine editierbare Bitmap, deren Größe und Farbtiefe mit der Anweisung

```
bitmapFormat = 640, 400, 8
```

festgelegt wird. Das Objekt legt damit automatisch eine Bitmap der Größe 640 x 400 Pixel mit 256 Farben (8 Bit pro Pixel) an. Sie brauchen sich nicht weiter um die Verwaltung der Bitmap zu kümmern - mit der Ausnahme etwas hineinzu-zeichnen. Das Objekt weiß z.B. selbst wann die Bitmap angelegt, gezeichnet oder am Programmende vernichtet werden muss.

Damit Sie etwas in die Bitmap zeichnen können legt die Anweisung

```
DefaultScreen
```

fest, dass alle Grafik- und Textausgaben standardmäßig an dieses Objekt gehen. Mit der Anweisung

```
defaultColor = YELLOW, LIGHT_BLUE
```

werden die Ausgabefarben auf "Gelb auf Blau" eingestellt.

Beachten Sie, dass das BitmapContent-Objekt "DemoBitmap" nirgends als Child eines Objekts auftaucht, sondern nur als "Content" des View-Objekts. Das ist für das ordnungsgemäße Zusammenspiel von View und Content erforderlich.

Wenn Sie nun das Programm starten (Menü "Programm", "Programm starten") wird es compiliert und ausgeführt. Dabei erzeugt der Compiler die im UI-Code angegeben Objekte und übersetzt den Quelltext im "BASIC-Code" Fenster in ein für R-BASIC ausführbares Format. Um zur R-BASIC IDE zurückzukehren wählen Sie aus dem Datei-Menü des laufenden BASIC-Programms den "Beenden" Eintrag.

Ein OnStartup Handler

Aktuell sehen wir nur eine blaue Fläche. Nun wollen wir etwas hineinzeichnen. Eine sehr typische Situation ist, dass beim Programmstart automatisch eine Routine abgearbeitet werden muss, die das Programm initialisiert, d.h. in den Anfangszustand versetzt. In R-BASIC ist das der OnStartup-Handler. Routinen, die von Objekten direkt aufgerufen werden heißen in R-BASIC alle "Action-Handler". Wechseln Sie in das Codefenster "BASIC-Code" und wählen Sie aus dem Menü "Extras", "Code Bausteine" den Eintrag "Action-Handler" und dort den Punkt "System-Action". OnStartup-Handler müssen als "SystemAction" vereinbart werden. Geben Sie einen möglichst selbsterklärenden Namen für den Handler, z.B. AppStart, ein. R-BASIC erstellt einen leeren Actionhandler. In den Handler können Sie nun Ihren Initialisierungscode schreiben. Wir benutzen ein Print-Befehl und zwei Grafikanweisungen.

```
SYSTEMACTION AppStart
  Print "Hallo Welt"
  Ellipse 10, 10, 200, 200
  Rectangle 30, 30, 180, 180
END ACTION
```

Starten wir das Programm jetzt (F9-Taste) so sehen wir ... wieder nichts. Das liegt daran, dass R-BASIC noch nicht weiß, wann es den Handler ausführen soll. Wir müssen noch festlegen, dass unser Handler namens "AppStart" der "OnStartup-Handler ist. Das passiert im UI-Code, im Application Objekt.

```
Application DemoApplication
    Children = DemoPrimary
    OnStartup = AppStart
END Object
```

Wenn wir jetzt wieder F9 drücken sehen wir endlich das Ergebnis unserer Bemühungen.
Nachdem der OnStartup-Handler abgearbeitet ist geht das Programm in den Wartezustand über. Damit wieder etwas passiert müssen wir etwas tun, z.B. einen Button anklicken.

Ein neues Objekt

Fügen Sie zum UI-Code die folgenden Zeilen hinzu:

```
Button TestButton
Caption$ = "Drück mich"
END Object
```

Vergessen Sie nicht, den Testbutton in die Children-Liste des Primary-Objekts aufzunehmen:

```
Children = DemoView, TestButton
```

Die Reihenfolge der Einträge bestimmt dabei ihre Anordnung im Primary-Objekt. Wenn sie jetzt F9 drücken sehen sie einen zusätzlichen Schalter mit der Aufschrift

"Drück mich". Sie können ihn anklicken - aber es passiert nichts. Das liegt daran, dass Sie ihm nicht mitgeteilt haben, was er zu tun hat.

Wenn sich etwas ereignet

Wird der Button angeklickt, so spricht man von einem **Ereignis** - in diesem Fall ein Maus-Ereignis. Der Button nimmt dieses Ereignis entgegen und "weiß" was er damit zu tun hat. Das ist der Kern der objektorientierten Programmierung: Jedes Objekt reagiert auf Ereignisse (Maus, Tastatur, Mitteilungen von anderen Objekten), indem es die Behandlung selbst vollständig übernimmt oder ein weiteres Ereignis auslöst. Auch das Starten des Programms stellt ein Ereignis dar. unser Application-Objekt reagiert auf dieses Ereignis, indem es seinen OnStartup-Handler aufruft (siehe oben). Ein Textobjekt z.B. stellt einen eingegebenen Buchstaben selbständig auf dem Bildschirm dar, für unseren Button aber müssen wir eine eigene Routine (einen eigenen Handler) schreiben, die eine Aktion ausführt.
Die Verknüpfung zwischen der Routine (dem Handler) und dem Button erfolgt wieder in UI-Code. Beim Button heißt die Instancevariable, die man dazu belegen muss, einfach ActionHandler.

Ändern Sie den UI-Code:

```
Button TestButton
  Caption$ = "Drück mich", 1
  ActionHandler = TestAction
END Object
```

Die 1 hinter dem Caption$-Text bewirkt, dass der Buchstabe mit der Nummer 1 unterstrichen wird und zur Tastaturnavigation verwendet werden kann. In unserem Fall ist das das 'r', da die Zählung bei Null beginnt.

Actionhandler von Buttons müssen als "ButtonAction" vereinbart werden. Das teilt dem Compiler mit, dass die Routine "TestAction" von einem Button aktiviert wird. Jede Objektklasse hat ihren eigenen Handlertyp. Näheres dazu finden Sie bei der Beschreibung er entsprechenden Objekte. Schreiben Sie im BASIC-Code-Fenster (nicht im UI-Code) den ActionHandler:

```
ButtonAction TestAction
  CLS
  Print "Button meldet: Bildschirm gelöscht."
END Action
```

Starten Sie das Programm mit F9. Herzlichen Glückwunsch! Sie haben Ihr erstes objektorientiertes BASIC-Programm geschrieben!

Es geht weiter...

Ändern Sie den BASIC-Code wie folgt, starten Sie das Programm erneut und klicken Sie auf den Button.

```
ButtonAction TestAction
  sender.enabled = FALSE
END Action
```

Was ist passiert? "Sender" ist ein Parameter, der jedem Action-Handler übergeben wird. Je nach Action-Handler gibt es weitere Parameter, die Sie bitte der Beschreibung der entsprechenden Objekte entnehmen. Der Parameter "sender" ist immer das Objekt, das den ActionHandler aktiviert hat. "Enabled" ist eine Eigenschaft, über die jedes Objekt verfügt. Ein Objekt ist "enabled", wenn der Nutzer damit interagieren kann, andernfalls ist es "disabled" und wird grau dargestellt. Eine weitere wichtige Eigenschaft ist "visible" (sichtbar). Versuchen Sie folgendes:

```
ButtonAction TestAction
  sender.visible = FALSE
END Action
```

Spielen Sie ruhig etwas mit den Objekten und den ActionHandlern herum! Ändern Sie zum Beispiel die Größe der Bitmap oder die Farben.

Anordnen von Objekten - das Geometriemanagement

Wir fügen zunächst einen zweiten Button ändern die ActionHandler wie folgt. Vergessen Sie nicht den neuen Button als Child des Primary einzubinden!

```
Button TestButton2
  Caption$ = "Drück mich auch", 6
  ActionHandler = TestAction2
END Object
```

```
ButtonAction TestAction
  FillEllipse  50, 50, 600, 350, LIGHT_GRAY
  Ellipse  50, 50, 600, 350
  Print "Grafik gezeichnet."
END Action

ButtonAction TestAction2
  CLS
  Print "Bildschirm ist gelöscht"
END Action
```

LIGHT_GRAY ist eine Farbkonstante, eine Zahl, die für eine Farbe steht. Näheres zur Beschreibung von Farben finden Sie im Programmierhandbuch, Kapitel 2.8.2 (Farben).
Wenn Sie das Programm jetzt starten funktioniert es zwar, aber die links unten angeordneten Buttons sehen nicht sehr schön aus. Wir wollen jetzt die beiden Buttons unter der Bitmap, aber nebeneinander anordnen.

Unter GEOS, und damit auch in R-BASIC, funktioniert die Anordnung von Objekten nicht dadurch, dass man festlegt, **wo** die Objekte platziert werden,

sondern **wie** sie angeordnet werden sollen. Man sagt nicht: platziere den Button dort, sondern man sagt: ordne beide Buttons nebeneinander an. Eine ausführliche Beschreibung dieses Konzepts finden Sie im Kapitel 3.3 (Geometriemanagement), dass Sie unbedingt lesen sollten.

Versuchen Sie zunächst folgendes:

```
Primary DemoPrimary
    BreakButton = TRUE
    Children = DemoView, TestButton, TestButton2
    orientChildren = ORIENT_HORIZONTALLY
    SizeWindowAsDesired
END Object
```

Sie werden wahrscheinlich nicht zufrieden sein, weil die Buttons jetzt neben der Bitmap sind. Die Lösung für dieses Dilemma ist, die Buttons in eine eigene Gruppe (ein Objekt der Klasse Group) zu verschieben. Die grundlegende Idee dahinter ist es, die Objekte innerhalb der Group nebeneinander anzuordnen während die Group selbst unterhalb der Bitmap bleibt. Dieses Konzept ist sehr ausführlich im Kapitel über das Geometriemanagement (siehe oben) beschrieben. Der große Vorteil dieser Technik ist, dass GEOS die Anordnung intelligent handelt. Man bekommt also auch dann ein gefälliges Aussehen, wenn der Nutzer eine andere Schriftgröße oder eine andere Bildschirmauflösung verwendet, als man selbst.

```
Primary DemoPrimary
    BreakButton = TRUE
    Children = DemoView, BottomGroup
    SizeWindowAsDesired
END Object

Group BottomGroup
    Children = TestButton, TestButton2
    orientChildren = ORIENT_HORIZONTALLY
End Object
```

Das sieht schon besser aus, ist aber noch nicht perfekt. Versuchen Sie - zunächst nacheinander und dann in beliebiger Kombination - die folgenden, fett markierten Programmzeilen. Die Reihenfolge der Codezeilen spielt dabei keine Rolle!

```
Group BottomGroup
    Children = TestButton, testButton2
    orientChildren = ORIENT_HORIZONTALLY
    ExpandWidth
    justifyChildren = J_FULL  ' oder J_CENTER
    IncludeEndsInChildSpacing
    DrawInBox
End Object
```

Probieren Sie ruhig weitere Hints aus dem Kapitel 3.3 aus! Ich wünsche Ihnen viel Spaß beim Probieren!

1.2 Grundlegende Begriffe

Hier sind einige der grundlegenden Begriffe erklärt, die im Handbuch immer wieder vorkommen. Sie müssen diese Begriffe jetzt weder vollständig verstehen noch auswendig lernen. Aber sie können dieses Kapitel zum Nachschlagen benutzen.

Objektorientierte Programmierung (OOP)

OOP ist eine Programmierphilosophie, bei der das Programm aus einzelnen Objekten besteht. Die Objekte verfügen über eine gewisse "Eigenintelligenz" (Eigenschaften und Fähigkeiten), mit denen die Programmfunktionalität realisiert wird. Zu diesem Zweck tauschen die Objekte Botschaften (Messages) aus und reagieren auf Ereignisse (Events). Unter GEOS sind die meisten Objekte sichtbar, wie z.B. Buttons oder Textobjekte. Es gibt aber auch unsichtbare Objekte, wie z.B. das Application-Objekt (vgl. Kapitel 1.1).

Klassen

Jedes Objekt hat eine bestimmten "Typ" der in der OOP als **Klasse** bezeichnet wird. Die Klassendefinition legt fest, welche Eigenschaften und Fähigkeiten die Objekte dieser Klasse haben.

Objekte

Objekte sind konkrete Manifestationen einer Klasse. Welche Eigenschaften und Fähigkeiten ein bestimmtes Objekt hat, wird durch seine Klasse bestimmt. Fachlich korrekt spricht man davon, dass ein bestimmtes Objekt eine **Instanz** einer bestimmten Klasse ist. Beispielsweise ist der Schalter "Schließen" in einer Dialogbox eine Instanz der Klasse "Button". Es hat sich jedoch eingebürgert vereinfachend zu sagen: Der "Schließen" Schalter ist ein Button-Objekt.

Window-Objekte

Als Window-Objekte werden alle Objekte bezeichnet, die ein unabhängiges Fenster (Window) auf dem Bildschirm erzeugen. Dazu gehören z.B. Dialogboxen und auch das Hauptfenster des Programms. Diese Fenster sind oft verschieblich und größenveränderlich.

Instance-Variablen

Jedes Objekt muss einen bestimmten Satz an Daten speichern, um korrekt arbeiten zu können. Beispielsweise muss jedes Objekt seine eigene Größe und Position kennen, um sich selbst korrekt darzustellen. Welche Daten das konkret sind, wird von der Klasse des Objekts bestimmt, die Datenwerte selbst sind jedoch für jedes Objekt, d.h. für jede Instanz einer Klasse, verschieden. Die von einem konkreten Objekt verwalteten Daten werden deshalb als **Instance-Variablen** (englisch für Instanz-Variablen) bezeichnet.

Hints

Hints (engl. Hilfen) sind eine besondere Art von Instance Variablen. Hints werden nur im Objekt gespeichert, wenn Sie explizit angegeben werden. Andernfalls sind sie einfach nicht vorhanden. Das spart eine Menge Platz. Fast alle Instance-Variablen zum Geometrie-Management sind als Hints imple-

mentiert. Wie der Name schon sagt sind Hints keine Befehle, sondern Hilfen. Es steht einem Objekt frei, wie es die Hints konkret umsetzt. Dazu gehört auch, dass es Hints ignorieren kann, wenn das angebracht ist.

Ereignisse (Events)

Betätigt der Nutzer eine Taste auf der Tastatur, bewegt er die Maus oder drückt eine Maustaste, so spricht man von einem "Ereignis". Ereignisse können auch der Start eines Programms oder vieles andere sein. GEOS registriert dieses Ereignis und sendet eine Message (Botschaft) an die für das entsprechende Ereignis zuständigen Objekte. Die Objekte können dann angemessen reagieren.

Ereignisorientierte Programmierung

Dieser Begriff beschreibt, dass ein GEOS-Programm nur dann etwas tut, wenn ein Ereignis auftritt. Dieses Verfahren ist viel effizienter als z.B. in einer Schleife ständig die Tastatur abzufragen. OOP und ereignisorientierte Programmierung gehen daher Hand in Hand.

Botschaften (Messages)

Die Informationen, die Objekte untereinander austauschen werden als Messages oder auf Deutsch als Botschaften bezeichnet. Botschaften sind eines der Kernkonzepte in der objektorientierten Programmierung.

Methoden

Die Routine, die ausgeführt wird, wenn ein Objekt eine Message erhält, wird als "Methode" bezeichnet. Die meisten Methoden sind R-BASIC-intern, d.h. sie sind für den R-BASIC Programmierer nicht direkt zugänglich, sondern werden automatisch ausgeführt. Es gibt davon jedoch Ausnahmen. Wenn Sie z.B. wollen, das eine Dialogbox auf dem Bildschirm erscheint, dann müssen Sie der Dialog-Objekt eine Message senden, die keine Instance-Variable setzt, sondern die Dialogbox auf den Schirm bringt. In diesem Beispiel müssen Sie schreiben

MyDialog.Open

wobei "Open" die Methode ist, die das Objekt namens "MyDialog" auszuführen hat, damit es auf dem Bildschirm erscheint.

Action-Handler

R-BASIC realisiert die OOP-Konzepte konzeptionell über Action-Handler. Registriert ein Objekt ein Ereignis, z.B. das Anklicken eines Buttons, so sendet es eine Message an den R-BASIC-Kern. Sie enthält die Information, welche Programm-Routine ausgeführt werden soll. Diese Programm-Routine wird als **ActionHandler** bezeichnet, da sie eine Aktion des Users behandelt. Der R-BASIC Kern führt den Handler (der in R-BASIC-Code geschrieben ist) aus und geht dann wieder auf "Stand-by" bis das nächste Ereignis eintritt.
ActionHandler können auch Botschaften an andere Objekte aussenden oder Methoden ausführen. Beispielsweise wird die Anweisung

EndeButton.enabled = FALSE

über eine Message an den EndeButton realisiert. Der Button disabled sich daraufhin. Die gesamte Programmfunktionalität eines BASIC-Programms steckt also in den verschiedenen ActionHandlern.

Objekt-Tree, Parents und Children

Objekte sind unter GEOS in sogenannten Bäumen (Trees) organisiert. Jedes Objekt hat genau ein **Parent** (Eltern) und kann kein, ein oder mehrere **Children** (Kinder) haben. Der Objekttree eines Programms dient sowohl der Kommunikation der Objekte untereinander (was in R-BASIC meist intern stattfindet) als auch der Organisation der Objekte auf dem Bildschirm. Details dazu finden Sie im Anschnitt 2.1 (Arbeit mit Objekt Bäumen).

Vererbung

Der Begriff beschreibt, dass die Objekt**klassen** voneinander abstammen. Eine neue Klasse wird von einer Vorgängerklasse abgeleitet. Dabei "erbt" die neue Objektklasse die Eigenschaften und Fähigkeiten seiner Vorgängerklasse. Hinzu kommen neue Eigenschaften und Fähigkeiten, die der Vorgänger nicht hat.
Ein Beispiel: Die meisten R-BASIC-Objekte können mit einer Aufschrift (**Caption$**) versehen werden. Bei Primary-Objekten erscheint sie in der Titelzeile, bei Buttons ist es "die" Aufschrift. Es macht nun keinen Sinn, diese Fähigkeit für jede Objektklasse von Grund auf neu zu implementieren, sondern sie ist in einer Klasse implementiert, die im GEOS SDK als "GenericClass" bezeichnet wird und von der die allermeisten R-BASIC-Objekte abstammen.

Folgen:

In der GenericClass ist z.B. auch das Geometrie-Management implementiert, so dass bei allen von der GenericClass abstammenden R-BASIC-Objekten das Geometrie-Management über die gleichen Befehle abgewickelt wird. Das gilt auch für viele weitere grundlegende Fähigkeiten von R-BASIC-Objekten. Das Handbuch ist deshalb so organisiert, dass die allen Objektklassen gemeinsamen Eigenschaften in separaten Kapiteln beschrieben werden. In den Kapiteln über die konkreten Klassen werden dann nur eventuelle Abweichungen ("überschriebene" Eigenschaften) und die neu hinzugekommenen Fähigkeiten, Eigenschaften und Instance-Variablen beschrieben.

1.3 Vereinbarungen für dieses Handbuch

In diesem Handbuch gelten einheitlich die folgenden Abkürzungen. Es wird empfohlen zumindest diese Seiten auszudrucken.

numWert Ein numerischer Wert allgemein z.B. 12 oder –17.4
Alle grünen und nicht weiter gekennzeichneten Elemente bedeuten einen numerischen Wert. Darunter sind z.B.

- **x**, **y**, **sizeX**, **sizeY** Positions- oder Größenangaben
- **n** Eine Anzahl oder eine Position
- **index** Der mögliche Wertebereich eines Index beginnt immer bei Null.

Im BASIC-Code kann anstelle einer Zahl auch immer ein numerischer Ausdruck (Variablen, Funktionen..) stehen.

"Text" Die Anführungszeichen ".." kennzeichnen eine beliebige String-Konstante, z.B. "Ja, ich will"
Im BASIC-Code kann anstelle eines expliziten Textes auch immer ein String-Ausdruck (Variablen, Funktionen..) stehen.

[..] Eckige Klammern kennzeichnen ein optionales Element, d.h. das Element kann angegeben werden, muss aber nicht vorhanden sein.

x I y Eine Alternative wird durch einen senkrechten Strich gekennzeichnet. Z.B. visible = TRUE I FALSE bedeutet, dass visible = TRUE oder visible = FALSE möglich ist.

TRUE R-BASIC Konstante mit dem Wert –1
Wird verwendet, wenn eine Eigenschaft "erfüllt" oder "wahr" ist. z.B. **<obj>.**visible = TRUE gibt an, dass das Objekt sichtbar ist.

FALSE R-BASIC Konstante mit dem Wert 0
Wird verwendet, wenn eine Eigenschaft "nicht erfüllt" oder "falsch" ist. z.B. **<obj>.**enabled = FASLE gibt an, dass der Nutzer nicht mit dem Objekt interagieren kann (es wird i.a. "grau" dargestellt).

My.. , Demo.., Test.. Zeigen an, dass das entsprechende Element (Objekt, Routine, ActionHandler usw.) von Programmierer selbst definiert wurde, d.h. nicht aus R-BASIC stammt.

<objVar> Objekt-Variable
Variable vom Typ OBJECT, z.B. DIM ov as OBJECT

<obj>, **<obj2>** Referenz auf ein Objekt: namentlich aufgeführtes Objekt oder Objekt-Variable. Felder und Struktur-Elemente vom Typ OBJECT sind erlaubt. In diesem Handbuch bezeichnen wir das als "Objekt Referenz".

<objektListe> Liste von namentlich aufgeführten Objekten

<numVar>	Numerische Variable (Real, Word, Integer usw.) auch z.B. **<pos>**, **<index>**, **<breite>**, usw.
<numExpr>	Numerischer Ausdruck
<stringVar>	String-Variable auch z.B. **<name$>**, **<path$>**, usw.
<stringExpr>	String Ausdruck
<handleVar>	Handle-Variable auch z.B. **<han>**, **<gsHan>**, usw.
<handleExpr>	Handle Ausdruck
<structVar>	Struktur-Variable
<structExpr>	Struktur Ausdruck
<Handler>	Der Name eines Action-Handlers

Sicher erkennen Sie das System dahinter, so dass Sie auch Elemente, die hier nicht explizit aufgeführt sind, zuordnen können.

Instancevariablen werden häufig in einer Tabelle dargestellt. Beispiel:

Variable	Syntax im UI-Code	Im BASIC-Code
Children	Children = **<objektListe>**	nur lesen
numChildren	—	nur lesen
parent	—	lesen, schreiben

Dabei bedeuten:

Variable:	Name der Instancevariablen
Syntax im UI-Code:	Belegung der Instancevariablen im UI-Code Fenster Die so festgelegten Werte stehen beim Programmstart sofort zur Verfügung.
Im BASIC Code:	Beschreibt, ob und wie man die Instancevariable vom BASIC Code aus (d.h. zur Laufzeit des Programms) ansprechen kann. Die Syntax dafür ist für alle Instance-variablen gleich und wird weiter unten, im Abschnitt "Syntax für Objektzugriffe", beschrieben. Eventuelle Aus-nahmen sind bei den entsprechenden Instancevariablen selbst beschrieben.
—	Ein Strich bedeutet, dass die Variable hier nicht verwendet werden kann.

1.4 Syntax für Objektzugriffe

Vereinbaren von Objekten

Objekte müssen im Fenster "UI-Objekte" vereinbart werden. Die Vereinbarung beginnt der Objekt-Klasse, gefolgt vom (frei wählbaren) Namen des Objekts. Dann folgen die Instance-Werte für dieses Objekt. Die Anweisung "END OBJECT" beendet die Objekt-Vereinbarung.

Beispiele:

```
Button  MyButton
  Caption$ = "Durchlauf starten"
  ActionHandler = DemoActionHandler
END Object

Primary  MyPrimary
  Children = MyButton, MyGroup, MyText
END Object
```

Zugriff auf Objekte

Objektzugriffe erfolgen entweder mit ihrem Namen oder über eine Objekt-Variable (Variable vom Typ OBJECT). Zur Vereinfachung bezeichnen wir die beiden Möglichkeiten als "Objekt-Referenz".

Beispiel:

```
DIM  oba, obb       AS OBJECT
DIM  of(10)         AS OBJECT      ' ein Feld von Objekt-Variablen
  oba = MyButton
  obb = MyPrimay
  of(1) = oba
  IF   oba = MyButton THEN ...
  IF   of(1) = MyPrimay THEN ...
```

Lesen von Instance-Variablen im BASIC-Code

Auf Instance-Variablen wird mit einer Objekt-Referenz (namentlich aufgeführte Objekte oder Objekt-Variablen, siehe oben), gefolgt von einem Punkt und dem Namen der Instance-Variablen, zugegriffen.

Beispiele:

```
DIM      obb          AS   OBJECT
DIM      x            AS   REAL
DIM      s$           AS   STRING

  s$ = MyPrimay.Caption$
  Print s$,  MyButton.Caption$

  obb = MyButton
  IF   obb.visible THEN ...   ' visible kann TRUE oder FLASE sein
```

Einige Instance-Werte (wie Children oder bitmapFormat) erwarten im UI-Code mehr als einen Wert. Gelesen kann aber immer nur ein Wert. Deshalb muss beim Lesen als Index angegeben werden, welcher Wert gelesen werden soll. Die zulässigen Werte für den Index beginnen **immer** mit Null.

Beispiel. Im UI-Code sei vereinbart:

```
Primary  MyPrimary
  Children = Group1, View1, Group2
END Object

BitmapContent      MyBitmap
  bitmapFormat = 48, 32, 8
END  Object

 < weitere Objekte..>
```

Lesen der Werte

```
DIM      x, y, c
DIM      ob, ch       AS   OBJECT

  ! Abfrage der Bitmap-Werte
  x = MyBitmap.bitmapFormat (0)            ' Breite
  y = MyBitmap.bitmapFormat (1)            ' Höhe
  Print  "Bitmap-Größe: "; x; "x"; y; "Pixel"
  Print  "Farbtiefe: "; ob.bitmapFormat(2); "Bit"

  ch = MyPrimary.Children(0)               ' Liefert Group1
  Print  ch.numChildren

  ch = MyPrimary.Children(1)               ' Liefert View1
  Print  ch.Caption$

  ob = MyPrimary                          ' Zur Demonstration
  ch = ob.Children(2)                     ' Liefert Group2
```

Hinweis: Kombinationen der Form
MyPrimary.Children(0).Caption$
oder ähnlich sind nicht zulässig.

Schreiben in Instance-Variablen im BASIC-Code

Wie beim Lesen werden die Instance-Variablen auch beim Schreiben durch einen Punkt von der Objekt-Referenz getrennt. Das Objekt kann auch hier namentlich aufgeführt oder über eine Objekt-Variable referenziert werden.
Die Syntax beim Schreiben in Instance-Variablen entspricht ansonsten genau derjenigen im UI-Code, Berechnungen mit Variablen und Funktionen sind im Gegensatz zum UI-Code aber zulässig.

Beispiele

```
DIM x    AS   WORD

  MyButton.visible = FALSE
  MyButton.Caption$ = "Beenden", 1

  ! Neuanlegen einer Bitmap
  x = 100
  MyBitmap.bitmapFormat = x,  x/2,  8
```

1.5 Vereinbarung von Action-Handlern

Action-Handler sind spezielle Unterprogramme, die aufgerufen werden, wenn ein Ereignis eintritt, z.B. ein Button gedrückt oder mit der Maus geklickt wird.

Der Typ
Jeder Typ von Ereignissen erfordert auch einen speziellen Typ von Action-Handlern: Das Drücken eines Buttons z.B. **ButtonAction**, ein Mausereignis wird von einem Handler des Typs **MouseAction** behandelt.

Parameter
Jeder Typ von Action-Handlern hat einen eigenen Satz an Parametern. Allen gemeinsam ist der erste Parameter "sender" vom Typ OBJECT. Er enthält eine Referenz auf das Objekt, dass das Ereignis ausgelöst hat. Dann folgen bis zu drei numerische Parameter, deren Bezeichnung und Bedeutung vom Typ des Handlers abhängt.

Vereinbarung des Handlers
Ein Handler wird vereinbart indem der Typ des Handlers, gefolgt von einem (frei wählbaren) Namen, angegeben wird. Die Parameter des Action-Handlers werden beim Vereinbaren des Handlers **nicht** explizit angegeben. Die Anweisung END ACTION schließt den Handler ab. **Tipp**: Verwenden Sie den Menüpunkt "Extras"-"Code Bausteine"-"Action-Handler". Damit erhalten Sie neben dem Handler-Rumpf einen Kommentarblock mit allen Parametern des Handlers.
Da Actionhandler spezielle Unterprogramme sind gelten die gleichen Regeln wie bei SUB's und FUNCTION's, d.h. es können z.B. lokale Variablen definiert, andere SUB's oder FUNCTION's gerufen oder Operationen mit Objekten durchgeführt werden. Andere ActionHandler können jedoch nicht direkt aufgerufen werden.

Beispiel: Im UI-Code sei vereinbart

```
Button   MyButton
  Caption$ = "Durchlauf starten"
  actionHandler = DemoHandler
END Object
```

Vereinbarung von "DemoHandler"

```
ButtonAction  DemoAction
DIM      x
  sender.enabled = FALSE          ' MyButton "grau" zeichnen
  FOR x = 10 TO 100 STEP 10
        LINE x, 5, x, 100 : Pause 1
        NEXT
  sender.enabled = TRUE
END ACTION
```

Zuweisung im BASIC Code
Sie können einem Objekt zur Laufzeit (im BASIC Code) einen anderen ActionHandler zuweisen. Die Zuweisung des speziellen Wertes **NoAction** bewirkt das Löschen des Actionhandlers für das entsprechende Objekt.

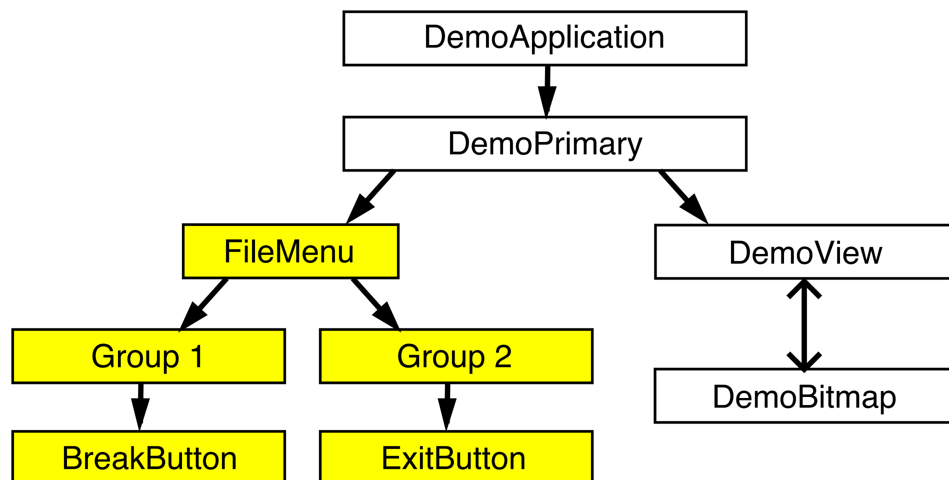
```
MyButton.ActionHandler = DemoAction
MyButton.ActionHandler = NoAction
```

2 Grundlegende Konzepte

2.1 Objekte und Objekt-Bäume (Trees)

2.1.1 Überblick

Objekte sind unter GEOS in sogenannten Bäumen (Trees) organisiert. Jedes Objekt hat genau ein **Parent** (Eltern) und kann kein, ein oder mehrere **Children** (Kinder) haben. Der Objekt-Tree eines Programms im "klassischen" BASIC-Modus sieht wie folgt aus (vergleiche UI-Code im Anschnitt 1.1):



Die gelb hinterlegten Objekte werden dabei vom Primary-Objekt automatisch angelegt und erscheinen daher nicht im UI-Code. Das Parent des Primary-Objekts ist das Application-Objekt, seine Children sind das FileMenu und das Bitmap-Objekt. Aus der Sicht des FileMenu ist sein Parent-Objekt das Primary, die Children des FileMenu sind zwei Groups (Gruppen), die jeweils den BreakButton bzw. den ExitButton enthalten. Das DemoBitmap-Objekt ist kein direkter Teil des Trees, es ist das Content-Objekt des DemoView-Objekts (siehe Kapitel 4.9).

Der Objekttree eines Programms dient sowohl der Kommunikation der Objekte untereinander (was in R-BASIC meist intern stattfindet) als auch der Anordnung der Objekte auf dem Bildschirm. Children zeichnen sich immer in den Grenzen, die ihnen das Parent vorgibt. Außerdem legt das Parent-Objekt fest, ob die Children neben- oder untereinander angeordnet werden und wie sie sich den vorhandenen Platz aufteilen. Details dazu finden Sie im Kapitel über das Geometriemanagement.

R-BASIC unterstützt die Arbeit mit Objekt-Trees sehr ausführlich. Sie können Informationen über das Parent und die Children eines Objekts erhalten, Objekte aus dem Tree entfernen und an anderer Stelle einfügen.

GenericClass Tree und VisualClass Tree

In R-BASIC Programmen gibt es zwei Arten von Objekt-Trees: den GenericClass Tree und den VisualClass Tree. Gelegentlich wird auch der allgemeine Term "UI-Tree" verwendet. Welcher Tree gemeint sein kann, ergibt sich dann aus dem Kontext.

Der **GenericClass Tree** (kurz: generic Tree) enthält die "normalen" GEOS-Objekte, wie Primary, Listen, Text-Objekte aber auch z.B. Dialogboxen. Sie stammen alle von der **GenericClass** ab. GenericClass-Objekte können nur auf dem Bildschirm erscheinen, wenn von ihnen ein vollständiger Pfad zum Application-Objekt führt. Das bedeutet, dass das Objekt selbst, sein Parent, dessen Parent oder ein anderes "übergeordnetes" (Parent-) Objekt letztlich mit dem Application-Objekt verbunden ist. Mit Objekten oder Objekt Trees, die nicht vollständig in den Tree eingebunden sind, kann man trotzdem arbeiten, d.h. man kann Instance-Variablen lesen und schreiben. Die veränderten Werte werden wirksam, sobald das Objekt vollständig in den Tree eingebunden ist und auf dem Bildschirm erscheint.

Der **VisualClass Tree** (kurz: visual Tree) enthält Objekte, von der **VisualClass** abstammen. Das Top-Objekt (oberstes Parent) ist ein VisContent oder BitmapConten-Objekt. Der Tree erschient auf den Bildschirm, sobald das Top-Objekt einem **View**-Objekt als "content" (=Inhalt) zugewiesen wird. Das View managed dann wie der visual Tree dargestellt wird. Details dazu finden Sie im Abschnitt über View- und Content-Objekte (Kapitel 4.9) sowie im Kapitel über die VisualClass (Kapitel 5).

In vielen Fällen besteht der "VisualClass Tree" ausschließlich aus dem VisContent bzw. BitmapContent-Objekt. Einfache BASIC-Programme haben oft gar keinen VisualClass Tree.
Komplexe Programme können mehrere Visual Trees haben und machen ggf. ausführlich von der Möglichkeit, Objekte anzulegen, wieder zu vernichten und im Tree zu verschieben, gebrauch.

Achtung! Es ist illegal, den GenericClass Tree und den VisualClass Tree zu mischen, d.h. in den GenericClass Tree Objekte einzufügen, die von der VisualClass abstammen und umgekehrt. Die Verbindung passiert ausschließlich über die "Content" Instance-Variable eines View-Objekts.

2.1.2 Arbeit mit Objekten

Die hier vorgestellten Möglichkeiten sind sowohl auf den GenericClass Tree als auch auf den VisualClass Tree anwendbar.

Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
Class$	—	nur lesen
Children	Children = <objektListe>	nur lesen
numChildren	—	nur lesen
parent	—	lesen, schreiben

Methoden

Syntax im BASIC-Code	Aufgabe
<numVar> = <obj>.**FindChild**(<childObj>)	Child suchen

Routinen

Syntax im BASIC-Code	Aufgabe
<stringVar> = **ObjInfo$**(<obj>)	interne Informationen anfordern

Class$

Class$ enthält den Namen der Objektklasse im Klartext, z.B. "MEMO", oder "DYNMAIC_LIST". Class$ kann nur gelesen werden. Ist das Objekt kein gültiges BASIC-Objekt, z.B. ein Null-Objekt, liefert Class$ einen leeren String.

Syntax Lesen: <stringVar> = <obj> **.Class$**

Children

Children legt im UI-Code fest, dass die aufgezählten Objekte Children des aktuellen Objekts sind.

Syntax im UI-Code **Children** = <objektListe>

Beispiel:

```
Primary MyPrimary
  Children = InfoMenu, MainGroup, BitmapArea
END Object
```

Die hinter Children auftauchenden Objekte müssen an anderer Stelle im UI-Code vereinbart sein. Jedes Objekt darf nur in maximal einer Children-Liste auftauchen. Ein Objekt nirgends als Child zu spezifizieren ist ok.

Die Anzahl der Objekte in einer einzigen Childrenliste ist auf 25 begrenzt. Wenn Sie mehr Children spezifizieren wollen können Sie mehrere Childrenanweisungen für ein Objekt verwenden.

Beispiel:

```
Group MyBigGroup
  Children = Button1, Group1
  Children = Button2, Group2
END Object
```

Beachten Sie, dass der Compiler die Children-Anweisungen nicht sofort ausführt, sondern sie zunächst auf eine Stapelspeicher (Stack) legt um sie am Ende des UI Compilevorgangs in umgekehrter Reihenfolge auszuführen. Die beiden Children-Anweisungen im obigen Beispiel sind also identisch mit:

```
Group MyBigGroup
  Children = Button2, Group2, Button1, Group1
END Object
```

Im BASIC-Code kann lesend auf die Child-Objekte eines Objekts zugegriffen verwenden:

Syntax Lesen: **<objVar>** = **<obj>.Children** (**index**)
index gibt die Nummer des Child-Objekts an
Wertebereich 0 .. numChildern – 1

Beispiel Basic-Code:

```
DIM obj, obj2 AS OBJECT

  obj = MyPrimay.Children(0)       ! erstes Child-Objekt lesen
  Print obj.Caption$
```

numChildren

NumChildren kann nur im BASIC-Code auftreten und kann nur gelesen werden. Es liefert die Anzahl der direkten Children des Objekts. Sollten die Child-Objekte eigenen Children haben, werden diese nicht mitgezählt.

Syntax Lesen: **<numVar>** = **<obj>** **.numChildren**

Beispiel

```
DIM n AS WORD
  n = MyPrimary.numChildren
  Print "MyPrimary hat";n;"Children"
```

parent

Parent kann nur im BASIC-Code auftreten. Es kann gelesen und geschrieben werden. Im UI-Code können sie das Parent nicht direkt setzen, sondern müssen über die Children-Liste gehen.

Lesen: Parent liefert das Parent-Objekt. Sollte das Objekt kein Parent haben, wird ein Null-Objekt zurückgegeben (analog der NullObj() Funktion).

Syntax Lesen: **<objVar> = <obj> .parent**

Beispiel:

```
DIM obj AS OBJECT

  obj = MyView.parent
  IF    obj = NullObj()  THEN
    Print "MyView hat kein Parent"
  ELSE
    Print "Caption$ = "; obj.Caption$
  END  IF
```

Schreiben: Parent weist einem Objekt ein neues Parent-Objekt zu. Damit wird das Objekt im UI-Tree verschoben. R-BASIC handelt alle dafür notwendigen Schritte. Sie können auch ein Null-Parent zuweisen (mit der Funktion NullObj()). Das Objekt wird versteckt, ist aber bereit an gleicher oder anderer Stelle wieder eingefügt zu werden.

Syntax Schreiben: **<obj> .parent = <obj2> , index**
index: Position (= neue ChildNr, beginnend bei Null), an der das Objekt eingefügt werden soll.
0: als erstes Child, 1: als 2. Child usw.
Sonderfall: –1 als letztes Child

Beispiel: Ausgangssituation im UI-Code

```
Group   MyGroup1
  Children = MyButton1, MyButton2, MyButton3, MyButton4
END Object

Group   MyGroup2
  Children = MyButton5
END Object
```

Beispiele BASIC-Code

```
! Button5 nach Group1 verschieben, als erstes Child
  MyButton5.Parent = MyGroup1, 0

! Button5 nach Group1 verschieben, als vorletztes Child
! Die Reihenfolge ist dann
```

```
! MyButton1, MyButton2, MyButton3, MyButton5, MyButton4
  MyButton5.Parent = MyGroup1, 3

! Button5 aus dem Tree entfernen
  MyButton5.Parent = NullObj() ,  0

! Button5 wieder zu Grpup2 hinzufügen, als letztes Child
  MyButton5.Parent = MyGroup2,  –1
```

Beachten Sie, dass sich die Child-Nummern der Objekte, die sich hinter dem eingefügten Objekt befinden, verändern.

FindChild

Die Methode FindChild untersucht, ob zwei Objekte im Child-Parent Verhältnis zueinander stehen.

Syntax: **<numVar>** = **<obj>.FindChild**(**<childObj>**)
<obj>: Variable oder Ausdruck vom Typ OBJECT
<childObj>: Variable oder Ausdruck vom Typ OBJECT
Es wird geprüft ob <childObj> ein Child von <obj> ist.
Rückgabewert:
0 .. N <childObj> ist ein Child von <obj>
Der Wert ist die Childnummer, die Zählung beginnt bei Null.
– 1 <childObj> ist **kein** Child von <obj>

ObjInfo$

Die Stringfunktion ObjInfo$ liefert interne Informationen über das Objekt. Sie können die Funktion zur Fehersuche in Objekt-Trees verwenden. Der String könnte z.B. so aussehen:
ObjReferenz=18:34, Typ=3 (BUTTON) * [Gen]
Die ObjektReferenz identifiziert ein Objekt eindeutig. Die erste Zahl beschreibt den Objektblock, in der sich das Objekt befindet.
Der Typ beschreibt die Objekt-Klasse eindeutig.
Zusätzlich ist der Klassen-Name noch im Klartext angegeben.

Syntax: **<stringVar>** = **ObjInfo$** (**<obj>**)
<obj>: Variable oder Ausdruck vom Typ OBJECT

2.1.3 Verwaltung von Objektblöcken

Diese Kapitel richtet sich an fortgeschrittene Programmierer.

R-BASIC ist daraufhin optimiert, die Verwaltung der Objektblöcke weitgehend automatisch zu erledigen. Dieses Kapitel enthält grundlegende und Hintergrundinformationen für den Fall, dass Sie ein Problem mit der Objektblockgröße haben (Kapitel 2.1.4) oder dass Sie Objekte zur Laufzeit des Programms anlegen wollen (Kapitel 2.1.5).

BASIC-Anweisungen

Syntax im BASIC-Code	Aufgabe
<handleVar> = GetObjBlockHandle(**<obj>**)	Handle eines Objektblocks lesen
<numVar> = GetObjBlockSize(**<handle>**)	Größe eines Objektblocks lesen
<handleVar> = CreateObjBlock ( )	Neuen Objektblock anlegen
DestroyObjBlock **<handleVar>**	Objektblock löschen

GEOS und damit R-BASIC verwaltet den Speicher in "Blöcken" von einigen Kilobytes Größe. Das gilt auch für Objekte. Die Instance-Daten (persönliche Daten) eines Objekts nehmen einige 10 bis einige 100 Byte ein. Es werden daher immer mehrere Objekte gemeinsam in einem Speicherblock abgelegt. Einen solchen Speicherblock nennt man "Objektblock". Jedes Mal, wenn auf die Instance-Daten eines Objekts zugegriffen werden muss, z.B. weil das Objekt sich auf dem Schirm darstellt oder weil Sie es angeklickt haben und es auf den Mausklick reagiert, wird der gesamte Objektblock in den Hauptspeicher geladen.

Die Aufteilung der Objekte auf Objektblöcke ist prinzipiell völlig unabhängig von der Verbindung der Objekte im generic Tree oder im visual Tree. Auf modernen Rechnern ist der geringe Performanceverlust, der auftritt, wenn man Objekte "Kreuz und Quer" über viele Objektblocks verlinkt, zu vernachlässigen. Trotzdem ist es schon aus Gründen der Übersichtlichkeit ratsam, zusammengehörende Objekte, z.B. alle Objekte einer Dialogbox, im UI-Code zusammenhängend zu deklarieren. R-BASIC platziert sie dann automatisch im gleichen Objektblock.

GetObjBlockHandle

Syntax: **<handleVar>** = **GetObjBlockHandle**(**<obj>**)
<obj>: Variable oder Ausdruck vom Typ OBJECT
<handleVar>: Variable vom Typ HANDLE

Um unter R-BASIC mit Objektblöcken arbeiten zu können, benötigen Sie ein Handle auf den Objektblock. Die Funktion **GetObjBlockHandle()** liefert das Handle des Objektblocks, in dem sich das übergebene Objekt befindet.

GetObjBlockSize

Syntax: **<numVar>** = **GetObjBlockSize**(**<han>**)
<han>: Variable oder Ausdruck vom Typ HANDLE
Es muss das Handle eines Objektblocks sein.

Die Funktion **GetObjBlockSize()** liefert die aktuelle Größe des Objektblocks, dessen Handle der Funktion übergeben wurde.

Der GEOS Speichermanager ist auf Blockgrößen von 6 kByte bis 8 kByte optimiert. Natürlich kann er auch mit Blöcken von wenigen Bytes umgehen, z.B. mit Objektblöcken, die nur ein einziges Objekt enthalten - das ist jedoch nicht effizient und kostet unnötig Systemhandles (jeder Block benötigt sein eigenes Handle). Werden die Blöcke hingegen zu groß kann es länger dauern bis der Speichermanager einen Platz im Hauptspeicher für diesen Block gefunden hat. Unter Umständen muss er dazu andere Blöcke, die gerade nicht benutzt werden auf die Festplatte auslagern. Das kann dauern.
Blockgrößen von 10 oder 16 Kilobytes sind dabei noch kein echtes Problem, bei Blockgrößen von z.B. 40 kByte oder mehr kann es jedoch zu den gefürchteten "Hauptspeicher voll" Meldungen kommen.

Um diesbezügliche Probleme zu vermeiden geht R-BASIC beim Compilieren des UI-Codes folgendermaßen vor:
Vor dem Anlegen eines neuen Objekts prüft es die Größe des aktuell verwendeten Objektblocks. Sollte dieser bereits mehr als 6 kByte groß sein, so wird vor dem Anlegen des neuen Objekts ein weiterer Objektblock angelegt. Das neue und alle folgenden Objekte werden dann in dem neuen Objektblock gespeichert. Dabei berücksichtigt R-BASIC, dass einige Objekte zur Laufzeit weiteren Speicher benötigen oder benötigen könnten. Beispielsweise speichern drei der vier Text-Objekte (Memo, InputLine und VisText) ihren Text in ihrem eigenen Objektblock. Die Instance-Variable maxLen gibt an, wie viele Zeichen der Text maximal enthalten kann. R-BASIC berücksichtigt das bei der Berechnung der Objektblockgröße. Deswegen sind Objektblocks, die Text-Objekte enthalten, anfangs häufig kleiner als 6 kByte. Ähnliches gilt für DynamicList-Objekte. Diese erzeugen zur Laufzeit ihre eigenen Children. R-BASIC berücksichtigt das pauschal mit 1,5 kByte, was in den meisten Fällen völlig ausreicht.

CreateObjBlock

Syntax: **<handleVar>** = **CreateObjBlock ( )**

Die Funktion **CreateObjBlock( )** legt einen neuen, leeren Objektblock an. Sie liefert das Handle des Objektblocks zurück. Dieses benötigen Sie, wenn Sie neue Objekte in diesem Block anlegen wollen (CreateObject(), siehe Kapitel 2.1.5) oder den Objektblock später wieder vernichten wollen.

DestroyObjBlock

Syntax: **DestroyObjBlock <han>**
<han>: Variable oder Ausdruck vom Typ HANDLE
Es muss das Handle eines Objektblocks sein.

Die Anweisung **DestroyObjBlock** vernichtet einen Objektblock und gibt den damit verbundenen Speicher wieder frei.

Wichtig:
- Der Objektblock darf keine Objekte mehr enthalten. Verwenden Sie dazu DestroyObject() (siehe Kapitel 2.1.5).
- Es ist dringend davon abzuraten Objektblöcke zu vernichten, die vom UI-Compiler erzeugt wurden. Vernichten Sie nur Objektblöcke, die mit CreateObjBlock() angelegt wurden.

2.1.4 Beeinflussung der Objektblöcke im UI-Code

Diese Kapitel richtet sich an fortgeschrittene Programmierer.

R-BASIC kümmert sich um Objektblöcke und die damit zusammenhängenden Dinge weitgehend selbständig. Die meisten Programmierer werden daher niemals selbst mit der Verwaltung von Objektblöcken zu tun haben.

Es gibt jedoch einige wenige Situationen die das direkte Eingreifen des Programmierers erfordern. Das sind konkret:
- DynamicList Objekte, die grafische Elemente anzeigen.
- Zuweisungen von grafischen Captions zur Laufzeit
- Textobjekte mit sehr großen Texten

Diese sowie die dafür nötigen Hintergrundinformationen sind hier beschrieben.

UI-Anweisungen

Syntax im UI-Code	Aufgabe
ForceNewObjBlock	Einen neuen Objektblock anfordern

Nach dem Compilieren erhalten Sie eine Tabelle, die wie folgt aussehen könnte. Die Spalte "Ab Zeile" enthält die Zeile im UI-Code, in der ein neuer Objektblock angelegt wurde. Daraus können Sie ermitteln welche Objekte in welchem Objektblock gespeichert sind. "Größe" enthält die anfängliche Größe des Objektblocks in Byte. "Objekte" enthält die Anzahl der im Objektblock gespeicherten Objekte. Beachten Sie, dass das Application-Objekt in einen eigenen Objektblock compiliert wird.

```
Objektblöcke compiliert: 3
Ab Zeile     Größe      Objekte
   36         4280         33
  196         1592          5
Interner Objektblock: Application
```

Wie bereits oben erwähnt brauchen Sie hier im Normalfall nicht einzugreifen, insbesondere dann nicht, wenn Ihnen die Blockgrößen sehr klein erscheinen. R-BASIC wählt einen guten Kompromiss auf anfänglicher Blockgröße und dem möglichen Blockwachstum zur Laufzeit. Ein nachträgliches Verschieben eines Objekts in einen anderen Block ist leider nicht möglich.

Es gibt jedoch einige Situationen, die R-BASIC nicht vorhersehen kann. Diese sind im Folgenden erklärt. Sie beziehen sich alle auf Veränderungen der Objekte zur Laufzeit. Als Richtwert kann gelten, dass Objektblöcke zur Laufzeit nicht größer als 14 bis 16 kByte werden sollten. Optimal sind weniger al 10 kByte. Wie Sie das herausbekommen ist weiter unten erklärt. Bei mehr als 20 kByte Blockgröße ist es dringend zu empfehlen gegenzusteuern.

Fall 1: DynamicList Objekte

Ein häufiger Fall, in dem die Blockgröße kritisch anwachsen kann, sind dynamische Listen, die grafische Elemente anzeigen. Dynamische Listen erzeugen Ihre Children zur Laufzeit selbst. Dafür wird Platz im Objektblock benötigt. Es zählen dabei nur die gleichzeitig sichtbaren Listeneinträge. R-BASIC unterstellt der Liste bei der Berechnung der Objektblockgröße einen Platzbedarf von 2 kByte, das entspricht ca. 25 gleichzeitig angezeigten Listeneinträgen mit je 20 Zeichen Text-Caption. Aber auch ein realer Bedarf von 4 oder 6 kByte sind kein Problem - falls man nicht mehrere solcher Listen im gleichen Objektblock hat.
Ein echtes Problem können aber viele Listeneinträge mit einer Grafik sein (Anweisung **ItemGString**). In diesem Fall ist es eine gute Idee das DynamicList-Objekt ein einem eigenen Objektblock, nur für dieses Objekt, unterzubringen. Dazu verwenden Sie die unten beschriebene Anweisung ForceNewObjBlock.

Fall 2: Grafische Captions

Auch die "Objekt-Beschriftung" (Caption$ oder grafische Captions) wird im gleichen Objektblock gespeichert wie das Objekt selbst. Text-Captions (Caption$) stellen dabei niemals ein Problem dar, da sie nur wenige Bytes umfassen. Wenn sie einen Text durch einen anderen ersetzen ändert sich die Objektblockgröße nur um wenige Bytes, ggf. wird sie sogar kleiner.
Haben Sie jedoch im UI-Code eine Text-Caption (oder gar keine) angegeben und weisen einem Objekt **zur Laufzeit** eine Grafik als Caption zu (Anweisungen **CaptionIcon**, **CaptionPicture**, **CaptionImage** und **CaptionGString**), so wird auch diese im Objektblock des Objekts gespeichert. Wenn Sie dies bei mehreren Objekten tun kann das den Objektblock zu stark vergrößern. Beispielsweise nimmt ein Icon der Bitmap-Größe 48x30 Pixel im ungünstigsten Fall (TrueColor, unkomprimiert) ca. 4 kByte ein, bei 8 Bit unkomprimiert sind es immer noch 1,4

kByte. Weitere Informationen dazu finden Sie im Kapitel 3.1 (Caption: Die Objekt-Beschriftung).
Auch hier gilt: Weisen Sie die Grafik bereits im UI-Code zu, erkennt R-BASIC die Größe und verteilt die fraglichen Objekte auf verschiede Objektblöcke.

Fall 3: Textobjekte

Der hier beschriebene Fall ist kommt eher selten vor, aber man sollte ihn kennen. Wie bereits oben erwähnt speichern Textobjekte (außer LargeText) ihren Text in ihrem eigenen Objektblock. Wie groß dieser Text werden kann hängt von der Instance-Variablen maxLen des Textobjektes ab. Der Standardwert für maxLen ist 1024.
Wenn Sie **zur Laufzeit** den Wert für maxLen deutlich vergrößern (gegenüber dem Wert, der beim Compilieren festgelegt wurde) und das auch ausnutzen (d.h. so viel Text dort speichern) wird der Objektblock größer als vom Compiler angenommen. Machen Sie das für viele Textobjekte aus dem gleichen Block kann es zu einem Problem werden.
Es macht daher Sinn bei Textobjekten den Wert für maxLen im UI-Code so klein wie möglich, aber auch so groß wie nötig zu wählen. Dann löst R-BASIC das Problem für Sie, indem es die Objekte auf mehrere Blöcke verteilt.
Eine Verkleinerung des Wertes für maxLen zur Laufzeit stellt dagegen niemals ein Problem dar.

Wie erkennt man, ob es ein Problem gibt?

Um Informationen über einen Objektblock zu erhalten muss man zuerst das "Handle" des Blocks ermitteln. Die Funktion **GetObjBlockHandle**() liefert das Handle des Objektblocks, in dem sich das übergebene Objekt befindet. Danach kann man mit der Funktion **GetObjBlockSize**() die aktuell gültige Größe des Objektblocks ermitteln.
Wenn Sie den Verdacht haben, dass ein Objektblock zu groß geworden ist, rufen Sie diese beiden Routinen für ein Objekt aus diesem Block, wie im folgenden Beispiel gezeigt:

```
DIM han AS HANDLE
DIM size
  han = GetObjBlockHandle(MyDanamicList)
  size = GetObjBlockSize ( han)
  MsgBox Str$(size)
```

Dabei ist es normal, wenn die Größe eines Objektblocks zur Laufzeit etwas größer ist, als in der Tabelle vom Compiler angegeben. GEOS arbeitet mit den Objektblocks. Z.B. fügt es bei einigen Objekten je nach Bedarf eigene (interne) Instance-Werte hinzu oder löscht diese wieder. Beim Löschen wird der Speicher zwar als "frei" markiert, der Block aber nicht unbedingt sofort verkleinert. Daher ist es auch normal, wenn die Größe vom Mal zu unterschiedlich ist, auch wenn Sie "gar nichts gemacht" haben. Sie können jedoch gut erkennen, ob der Objektblock eine kritische Größe (mehr als 16 bis 20 kByte) erreicht hat oder nicht.

Was kann man tun?

Der einzige Weg, Einfluss auf die Verteilung der Objekte auf die Objektblöcke zu nehmen, ist, dem UI-Compiler anzuweisen, einen neuen Objektblock anzulegen.

ForceNewObjBlock

Syntax im UI-Code: **ForceNewObjBlock**

Platzieren Sie den UI-Befehl **ForceNewObjBlock** (Erzwinge neuen Objektblock) je nach Situation vor oder nach dem kritischen Objekt. Es ist auch möglich die fraglichen Objekte (oder das einzelne Objekt) in zwei ForceNewObjBlock-Anweisungen einzuschachteln.

```
ForceNewObjBlock

DynamicList MyBigList
  ....
  END Object

ForceNewObjBlock
```

Findet der UI-Compiler eine ForceNewObjBlock-Anweisung schließt er den aktuell verwendeten Objektblock und legt für die folgenden Objekt einen neuen an.
Tipp: Legen Sie die fraglichen Objekte ans Ende aller UI-Anweisungen, dann werden die davor befindlichen Objektblöcke nicht unnötig eingekürzt.

2.1.5 Anlegen und Vernichten von Objekten zur Laufzeit

Diese Kapitel richtet sich an fortgeschrittene Programmierer.

Üblicher Weise legt R-BASIC die im UI-Code deklarierten Objekte an, wenn das Programm compiliert wird. R-BASIC bietet Ihnen aber auch die Möglichkeit, weitere Objekte zur Laufzeit anzulegen und wieder zu vernichten. Für GenericClass Objekte (z.B. Button oder Menu) wird diese Möglichkeit eher selten genutzt. Einige komplexe Programme, wie z.B. der Grafikbetrachter Gonzo, nutzen aber die gebotenen Möglichkeiten für VisualClass Objekte sehr intensiv. Gonzo legt für jede Grafikdatei, die es findet, ein eigenes Objekt an. Dieses Kapitel beschreibt, wie man Objekte zur Laufzeit anlegt, damit arbeitet, wieder vernichtet und was es dabei zu beachten gilt.
Beachten Sie, dass die VisContent Objekte über eingebaute Methoden verfügen, die das Anlegen und Vernichten von VisObj-Objekten stark vereinfachen.

BASIC-Anweisungen

Syntax im BASIC-Code	Aufgabe
<objVar> = CreateObject (**<han>**, **<objClass>**)	Neues Objekt erzeugen
DestroyObject **<obj>**	Objekt vernichten

Wie im Kapitel 2.1.3 beschrieben verwaltet GEOS alle Objekte in Objektblöcken. Um ein Objekt zur Laufzeit anzulegen müssen Sie außer der Klasse des Objekts auch den Objektblock spezifizieren, in dem das Objekt angelegt werden soll. Das typische Vorgehen beim Arbeiten mit selbst angelegten Objekten ist im Folgenden beschrieben. Es unterscheidet sich für GenericClass-Objekte und VisualClass-Objekte nicht.

1. Erzeugen Sie einen neuen Objektblock mit **CreateObjBlock()**.
2. Legen Sie die neuen Objekte mit **CreateObject()** an. Initialisieren Sie die Instancevariablen und binden Sie die Objekte in den Tree ein.
3. Arbeiten Sie mit den Objekten.
4. Nachdem Sie die Objekte nicht mehr brauchen, sollen Sie sie mit **DestroyObject()** vernichten. Dadurch wird der Speicher im zugehörigen Objektblock freigegeben.
5. Am Schluss, nachdem alle Objekte im Objektblock vernichtet wurden, sollten Sie den Objektblock mit **DestroyObjBlock()** vernichten.

Die Befehle **CreateObjBlock()** und **DestroyObjBlock** sind im Kapitel 2.1.3 beschrieben.

CreateObject

Syntax: **<objVar>** = **CreateObject** (**<han>**, **<objClass>**)
<han>: Handle eines Objektblocks
<objClass>: Bezeichnung einer Objektklasse

Die Funktion **CreateObject()** legt ein neues Objekt in dem Objektblock an, dessen Handle der Funktion übergeben wurde. Als Objektklasse (dem "Typ" des Objekts) sind alle R-BASIC Klassen, außer Application, zulässig.

Beispiele:

```
DIM  o, p, q AS OBJECT

o = CreateObject (han, Button)
p = CreateObject (han, Dialog)
q = CreateObject (han, Menu)
```

Um das Objekt nutzen zu können müssen Sie noch seine Instance-Variablen initialisieren und das Objekt in den Tree einbinden.

Ausführliches Beispiel: siehe unten

Hinweise:
- Je nach Klasse benötigt jedes Objekt einige 10 bis einige 100 Byte. Objektblöcke sollten nicht zu groß werden. Die Idealgröße liegt zwischen 6 kByte und 8 kByte. Verwenden Sie im Zweifelsfall die Funktion **GetObjBlockSize()** um die aktuelle Größe des Objektblocks zu ermitteln. Details dazu sind im Kapitel 2.1.3 beschrieben.
- Auch wenn es möglich ist: Sie sollten keine Objekte in den Objektblocks anlegen, die vom Compiler erzeugt wurden.

DestroyObject

Syntax: **DestroyObject** **<obj>**
<obj>: Variable oder Ausdruck vom Typ OBJECT
Das Objekt wird vernichtet.

Die Anweisung **DestroyObject** vernichtet ein Objekt. Damit ein Objekt vernichtet werden kann darf es nicht mehr im Tree eingebunden oder auf andere Weise mit anderen Objekten verbunden sein. R-BASIC erzeugt einen Laufzeitfehler, wenn diese Bedingungen verletzt sind, um einen Absturz des Systems zu verhindern.

Hinweis:
- Auch wenn es möglich ist: Sie sollten keine Objekte vernichten, die vom Compiler erzeugt wurden.

Beispiel: Anlegen eines Objekt-Trees

```
DIM h as HANDLE
DIM d, t, b as OBJECT

h = CreateObjBlock()
d = CreateObject(h, Dialog)
d.Caption$ = "Neuer Dialog"
d.Parent = DemoPrimary,1

t = CreateObject (h, Memo)
t.Caption$ = "Text eingeben"
t.justifyCaption = J_TOP
t.Parent = d, 0

b = CreateObject(h, button)
b.Caption$ = "Fertig"
b.ActionHandler = TestAction
b.Parent = d,1
```

Beispiel: Vernichten eines Objekt-Trees

```
BUTTONACTION TestAction
  Print "Button pressed!"
  Print t.text$

  b.Parent = NullObj(), 0
  DestroyObject b

  t.Parent = NullObj(), 0
  DestroyObject t

  d.Parent = NullObj(), 0
  DestroyObject d

  DestroyObjBlock h

  END ACTION
```

2.2 Ausgabe von Grafik

Grafische Ausgaben auf den Schirm gehören zu den Kernaufgaben einer Programmiersprache. R-BASIC bietet diesbezüglich eine Fülle von Möglichkeiten. In diesem Kapitel finden Sie einen Überblick über die Möglichkeiten und die dazu verwendeten Objekte. Unter R-BASIC erfolgt die Ausgabe von Grafik - das schließt Textausgaben mit der Anweisung PRINT ein - immer auf das aktuelle "Screen" Objekt. Mehr zu Screenobjekt finden sie im Kapitel 2.3 unten.

2.2.1 Objekte zur Grafikausgabe

Im Folgenden werden die zur Grafikausgabe verwendbaren Objektklassen aufgezählt und ihre Vor- und Nachteile unter dem Aspekt der Grafik- und Textausgabe gegenübergestellt. Eine ausführliche Beschreibung der einzelnen Objektklassen finden sie in den folgenden Kapiteln des Objekthandbuchs.

BitmapContent

Ein BitmapContent (Container für eine Bitmap) stellt eine Bitmap bereit, in die man Grafik und Text zeichnen kann. Üblicherweise ist das BitmapContent Objekt als "Content" eines View Objekts gesetzt. Das ist notwendig, um die Bitmap auf den Schirm zu zeichnen, aber es ist ausdrücklich erlaubt auch in eine Bitmap, die nicht mit einem View Objekt verbunden ist, zu zeichnen. Die Veränderungen werden sichtbar, wenn das BitmapContent Objekt das nächste Mal mit einem View verbunden (als Content gesetzt) wird.

Der große Vorteil von BitmapContent Objekten ist, dass alle Grafikausgaben automatisch gespeichert werden. Muss sich das Objekt neu darstellen erfolgt das ohne das Zutun von BASIC Code einfach durch Neuzeichnen der Bitmap.

Nachteilig ist der relativ große Speicherbedarf. Außerdem kann es Farbabweichungen geben, wenn die Farbtiefe der Bitmap nicht mit der der verwendeten Zeichenbefehle übereinstimmt. Zeichnet man z.B. True-Color Grafiken in eine 8-Bit-Bitmap werden die Farben heruntergerechnet. Da Grafikausgaben aus Performancegründen immer parallel auf den Schirm und auf die Bitmap erfolgen ist dieser Effekt meist erst nach einer Neudarstellung des Objekts zu sehen, was besonders störend wirken kann.

Canvas

Ein Canvas Objekt zeichnet eine Grafik, indem es seinen **OnDraw** Handler aufruft. Dieser Handler zeichnet dann die eigentliche Grafik auf den Schirm. Das Canvas-Objekt eignet sich sehr gut für kleine, einfache Grafiken, die zur Laufzeit gezeichnet werden können und sich nicht oder nur selten ändern. Das können z.B. einfache Grafikbefehle oder ein Bild aus der Picture-List sein.

Von Vorteil ist die einfache Handhabung des Objekts. Außer um das Schreiben des OnDraw Handlers muss man sich um nichts kümmern. Auch der geringe Speicherbedarf des Objekts ist unter GEOS ein Vorteil.

Im normalen (ungepufferten Modus) ist nachteilig, dass der OnDraw Handler jedes Mal gerufen wird, wenn sich das Objekt neu darstellen muss, z.B. weil ein Teil des Objekts durch ein Menü verdeckt war. Das ist langsam und belastet den BASIC-Thread. Für einfache Anwendungen ist das jedoch ausreichend.
Im gepufferten Modus merkt sich das Canvas-Objekt die Grafikbefehle intern als GString und spielt ihn mit hoher Geschwindigkeit wieder ab, wenn sich das Objekt neu darstellen muss.

Image

Imageobjekte sind dafür ausgelegt Bilder aus eine externen Bilddatei z.B. BMP, ICO, PCX, JPG ...) darzustellen. Sie müssen dem Objekt nur Name und Speicherort der Datei mitteilen, um den Rest kümmert es sich allein.
Da Imageobjekte das Bild aus der Datei intern in einen GEOS Bitmap kopieren benötigen sie ähnlich viele Speicher wie BitmapContent Objekte.

Imageobjekte sind dafür ausgelegt das Bild darzustellen. Eine Bearbeitung (wie mit einem BitmapContent Objekt) ist nicht möglich.

VisContent

Das VisContent Objekt wird als "content" eines View Objekts in den Tree eingebunden. Es hat einen OnDraw-Handler, so wie ein Canvas-Objekt. Deshalb kann das VisContent-Objekt direkt Grafik und Text ausgeben. Aber üblicher Weise erledigen die Grafikausgabe die Children des VisContent-Objekts. Das sind Objekte der Klasse VisObj.

VisObj

Objekte der Klasse VisObj sind die Children eines **VisContent** Objekts. Deswegen werden Sie immer innerhalb eines **View** Objekts dargestellt. Wie das Canvas-Objekt besitzen sie einen OnDraw Handler und einen gepufferten Modus. Durch die Darstellung innerhalb eines View ergeben sich Möglichkeiten, die mit einem Canvas Objekt nicht möglich sind, z.B. Scrolling und Zoom. Außerdem kann man VisObj Objekte mit der Maus positionieren.

Generic Class: Grafische Captions

Alle von der GenericClass abstammenden Objekt können kleine (!) Grafiken als Caption darstellen (Instancevariablen CaptionImage, CaptionPicture, Caption-GString oder CaptionIcon). Gedacht ist dieses Feature für grafische Button-Beschriftungen und grafische Listenelemente, es eignet sich aber auch für Logos und andere kleine Grafiken, die zur Laufzeit nicht oder nur selten geändert werden. Wenn Sie grafische Captions zur Laufzeit verändern sollten Sie die Größe der Grafiken im Blick haben. Details dazu finden Sie im Kapitel 3.1 (Caption: Die Objekt-Beschriftung).

2.2.2 Konzepte zur Grafikausgabe

Dieses Kapitel enthält einen Überblick über die in R-BASIC verfügbaren Konzepte zur Grafikausgabe. Im Einzelnen sind das

- Einfache grafische Kommandos
- PRINT und BlockGrafik
- Die Picture-List
- Graphic Strings
- Bitmaps

Einfache grafische Kommandos

Dem R-BASIC Programmierer stehen eine große Auswahl von einfachen grafischen Kommandos wie Line, Circle, FillEllipse usw. zur Verfügung. Diese, sowie die das dazugehörige Koordinatensystem werden ausführlich im Kapitel 2.8 (Grafik) des R-BASIC Programmierhandbuchs besprochen. Besonders hingewiesen werden soll hier auf die Systemvariable **graphic**, über die die Eigenschaften aller Grafikbefehle wie Linienfarbe, Linienbreite und Flächenattribute (z.B. Füllmuster) bis hin zum MixMode eingestellt werden können. Die daraus resultierenden Möglichkeiten gehen weit über die klassischen BASIC Befehle wie INK und COLOR hinaus.

PRINT und BlockGrafik

Der Befehl PRINT ist unter BASIC für alles, was mit Textausgabe auf den Grafikschirm zusammenhängt, zuständig. Er wird ausführlich im Kapitel 2.9 (Textausgabe) des R-BASIC Programmierhandbuchs besprochen. Welche Schriftart, Schriftgröße usw. verwendet wird, wird mit der Systemvariablen **printFont** kontrolliert. Wie man darauf zugreift ist im Kapitel 2 (Verwendung von Schriften) des Handbuchs "Spezielle Themen" beschrieben. Für die Formatierung von Zahlen bei der Ausgabe mit Print oder dem BASIC Befehl Str$ ist die Systemvariable **numberFormat** zuständig, deren Beschreibung Sie im Kapitel 1 (Formatierung von Zahlen) des Handbuchs "Spezielle Themen" finden.

Eine spezielle, dabei sehr einfache und gleichzeitig leistungsfähige Art, Grafiken zu zeichnen, sind die sogenannten **Blockgrafiken**. Dabei wird der PRINT-Befehl verwendet, statt der Buchstaben werden aber kleine Grafiken (z.B. 32x32 Pixel) auf den Schirm gezeichnet. Jedem Buchstaben kann dabei eine eigene Grafik zugeordnet werden, so dass sich sehr komplexe Bilder aus wenigen Grafikelementen zeichnen lassen. Blockgrafikelemente lassen sich am einfachsten mit dem Blockgrafik Editor, der über das Tools-Menü von R-BASIC erreichbar ist, erstellen.
Wie man den Blockgrafik Modus einsetzt ist im Kapitel 3 (Verwendung des Block-Grafik-Modus) des Handbuchs "Spezielle Themen" beschrieben.

Die Picture-List

Die Picture-List ist eine komfortable Möglichkeit Grafiken im der Codedatei selbst unterzubringen um sie zur Laufzeit zu verwenden. Sie können diese Bilder auf den Screen zeichnen oder als grafische Aufschrift für Objekte verwenden. Die Picture-List wird über das Menü "Extras" - "Picture-List" verwaltet.
Eine ausführliche Beschreibung der Arbeit mit der Picture-List finden Sie im R-BASIC Programmierhandbuch, Kapitel 2.8.6.2 (Verwendung der "Picture-List").

Graphic Strings

Ein Graphic String (im Folgenden kurz GString) ist eine Folge von Grafikbefehlen oder Textausgaben, die gemeinsam gespeichert werden. Dieser GString kann später beliebig oft "abgespielt" werden. Dabei werden die enthaltenen grafischen Kommandos mit hoher Geschwindigkeit ausgeführt, viel schneller als dies als Folge von BASIC-Anweisungen möglich ist.

GStrings sind ein tief im GEOS System verwurzeltes und sehr leistungsfähiges Konzept. Beispielsweise erfolgt der Austausch von Grafiken zwischen verschiedenen Programmen über die Zwischenablage immer als GStrings. Der gepufferte Modus verschiedener BASIC Objekte (z.B. Canvas oder VisObj) ist ebenfalls mit GStrings realisiert.

Unter R-BASIC können Sie GStrings für verschiedene Zwecke verwenden
- Aufzeichnung von Grafikanweisungen zur späteren Verwendung
- Grafische Captions für Objekte
- Grafische Einträge in Listen
- Arbeit mit der Zwischenablage

Der Zugriff auf GStrings erfolgt unter R-BASIC über Handles. Sie können GStrings aufzeichnen, wiedergeben (auf den Screen zeichnen) oder wieder freigeben. In einem GString können grundsätzlich alle Grafikausgaben gespeichert werden. Das schließt explizit andere GStrings, Texte (PRINT-Anweisung) und Bitmaps (Bilder oder Blockgrafik) ein. Andere GStrings oder Bitmaps werden dabei in den GString kopiert. Die R-BASIC Library "VMFiles" bietet außerdem Funktionen an, einen GString in eine Datei zu schreiben bzw. ihn von dort zu laden.

Eine ausführliche Beschreibung der Arbeit mit GStrings finden Sie im R-BASIC Programmierhandbuch, Kapitel 2.8.5 (Arbeit mit Graphic Strings).

Bitmaps

Bitmaps sind digitalisierte Bilder. Sie bestehen aus einer rechteckigen Anordnung von einzelnen Bildpunkten (Picture Element: Pixel). Jedem Pixel kann eine eigene Farbe zugeordnet werden.
Innerhalb von GEOS werden Bitmaps **immer** im GEOS-internen Bitmapformat gespeichert, **unabhängig** davon, aus welcher Quelle (z.B. welchem Dateiformat) die Bilder stammen. Dieses interne GEOS Format erlaubt die Existenz einer Transparenz-Maske als auch einer Palette für jede Bitmap.

Die **Transparenz-Maske** muss man sich als zusätzliche Farbebene vorstellen, die neben den eigentlichen Bilddaten gespeichert wird. Dabei gibt es für jedes Pixel genau ein Transparenzbit. Ist es gesetzt (=1) wird das entsprechende Pixel des Bildes dargestellt. Ist es nicht gesetzt (=0) ist das Bild an dieser Stelle durchsichtig.

Viele Bilder enthalten weniger als 3 Byte (24 Bit) pro Pixel. Ein üblicher Wert ist z.B. 8 Bit pro Pixel. Das spart Speicher. Aber damit können sie nur 256 der 16 Mio. möglichen Farben darstellen. Welche das sind, kann mit einer **Palette** festgelegt werden. Jeder Paletteneintrag besteht aus genau 3 Byte. Damit wird jedem möglichen Farbwert des Bildes (0 ... 255 bei 8 Bit pro Pixel) eine True-Color Farbe zugeordnet. Durch eine geschickte Wahl der Palette kann man sehr realistische Bilder erzeugen.
Enthält ein Bild mit 8 Bit pro Pixel keine Palette wird die GEOS Standardpalette verwendet. Diese Bilder werden etwas schneller gezeichnet, aber viele Probleme mit der Farbdarstellung entstehen deswegen, weil die Windows-Standardpalette nicht mit der GEOS-Palette identisch ist.
Es ist deshalb oft eine gute Idee 256-Farb-Bilder in die GEOS Palette umzurechnen. Dazu eignet sich z.B. das GEOS Tool Sigma.

In R-BASIC erfolgt die Verwaltung einer Bitmap üblicher Weise mit einem **BitmapContent** Objekt. Dieses Objekt kümmert sich selbständig um den benötigten Speicherplatz sowie um die Darstellung der Bitmap auf dem Schirm. Es ermöglicht es in die Bitmap zu zeichnen oder Farbwerte auszulesen und bietet außerdem Funktionen zum Bearbeiten der Transparenz-Maske und der Palette. Eine Beschreibung des BitmapContent Objekts finden Sie weiter hinten im Objekthandbuch. Dort sind auch die Strukturen von Bitmap, Transparenz-Maske und Palette beschrieben.

Zusätzlich bietet R-BASIC die Möglichkeit Bitmaps analog zu den GStrings über **Handles** anzusprechen. Über diesen Weg können Bitmap zwischen Teilen des BASIC Programms ausgetauscht und in andere Bitmaps oder in GStrings gezeichnet werden. Die R-BASIC Library "VMFiles" bietet außerdem Funktionen an, Bitmaps in eine Datei zu schreiben bzw. ihn von dort zu laden.
Der Bitmapzugriff über Handles ist ausführlich im Anschnitt 2.8.6.4 (Bitmaps und Bitmap Handles) des R-BASIC Programmierhandbuchs beschrieben.

2.3 Arbeit mit dem Screen

Sowohl alle Grafik-Befehle als auch der Print-Befehl müssen wissen, wohin die Ausgaben erfolgen sollen. Dieses Ausgabe-Ziel ist ein Objekt, das als Screen-Objekt bezeichnet wird. Als Screen-Objekte können folgende Objekte dienen:

- BitmapContent
- VisContent
- VisObj
- Canvas
- Image
- PrintControl (nur während des Druckens)

Das BitmapContent Objekt ist das einzige, das global (permanent) als Screen arbeiten kann. Damit können Sie von verschiedenen Teilen des Programms, insbesondere von verschiedenen Action Handlern aus, in die Bitmap zeichnen. Die anderen Objekte werden automatisch zum Screen, wenn ihr OnDraw Handler (für PrintControl Objekte: ihr OnPrint Handler) gerufen wird. Sollte beim Aufruf des OnDraw oder OnPrint Handlers ein globaler Screen gesetzt sein, wird dieser deaktiviert, so lange der Handler läuft und anschießend wieder reaktiviert. Daher können Sie vom OnDraw bzw. OnPrint Handler aus direkt Grafik oder Text ausgeben.
Zusätzlich können Sie die genannten Objekte jederzeit temporär zum Screen machen, indem Sie die Screen-Variable mit dem Objekt belegen. Das ist z.B. bei der Arbeit mit der Maus sinnvoll. Sie müssen sich in diesem Fall aber selbst darum kümmern, dass der zuvor gesetzte Screen wieder hergestellt wird, sonst kann GEOS crashen. Beispiele dazu finden Sie in den Kapiteln zur Arbeit mit der Maus (Spezielle Themen, Kapitel 17) und bei der Beschreibung der Objektklassen VisContent und VisObj (Kapitel 5 im Objekthandbuch).

Sie können in R-BASIC zwischen verschiedenen globalen Screen-Objekten wechseln. Außerdem können Sie Einstellungen am Koordinatensystem wie Skalierung und Verschiebung vornehmen. Erfahrene Programmierer haben außerdem die Möglichkeit komplexe Operationen mit dem Koordinatensystem durchzuführen.

2.3.1 Die Screen-Variable

Variable	Syntax im UI-Code	Im BASIC-Code
Screen	—	lesen, schreiben
DefaultScreen	DefaultScreen	—

Die globale Variable **Screen** enthält das aktuelle Screen-Objekt. Sie kann im BASIC-Code gelesen, geschrieben oder in Vergleichen verwendet werden. Auch der direkte Zugriff auf Instance-Variablen (z.B. n = Screen.bitmapFormat(0)) ist zulässig.

Syntax	Lesen:	**<objVar>** = **Screen**
	Schreiben:	**Screen** = **<obj>**

Am Programmstart wird die Variable **Screen** mit demjenigen Objekt belegt, dass die Anweisung **DefaultScreen** im UI-Code hat. Die Anweisung DefaultScreen ist nur für BitmapContent Objekte zulässig.

Syntax UI-Code: **DefaultScreen**

Wird ein Objekt erstmalig zum Screen, so werden alle Grafik- und Font-Einstellungen auf den Standardwert zurückgesetzt, das Ausgabe-Window wird auf Maximum gesetzt und der Cursor wird links oben platziert. Die Farben werden auf die durch den **defaultColor**-Wert des neuen Screen-Objekts bestimmten Werte gesetzt. Ist kein **defaultColor**-Wert spezifiziert wird Schwarz auf Weiß eingestellt. Verliert ein Objekt den Status "Screen" zu sein, so speichert es die genannten Werte intern und stellt sie wieder her, wenn es erneut zum Screen wird. Sollte das nicht möglich sein, weil Sie z.B. zwischenzeitlich die Bitmapgröße geändert haben, werden wieder die Standardwerte verwendet.

Beispiel:
Umschalten zwischen den beiden Objekten MyContent1 und MyContent2.

```
IF Screen = MyContent1 THEN
    Screen = MyContent2
  ELSE
    Screen = MyContent1
  END IF
```

Wird die Variable Screen mit einem "leeren" Objekt belegt (Funktion **NullObj()**), so gehen alle Grafik- und Textausgaben ins Leere.

Beispiel 1:

```
Screen = NullObj()
```

Beispiel 2:

```
IF Screen = NullObj() THEN Screen = MyContent1
```

2.3.2 Clipping

Unter Clipping versteht man, dass Grafiken und Texte nicht über die vorgegeben Geometriegrenzen hinaus geschrieben werden. Beim Zeichnen in eine Bitmap wird nicht über deren Rand hinaus gezeichnet, Objekte die in einem View-Objekt sind zeichnen nicht über die Grenzen des View hinaus. GEOS und damit R-BASIC handelt das automatisch, so dass Sie sich im Allgemeinen nicht darum kümmern müssen. Sie haben jedoch die Möglichkeit den Bereich, in den Grafik- und Textausgaben gehen sollen, zusätzlich einzuschränken.

ScreenSetClipRect

Die Routine ScreenSetClipRect schränkt die Ausgabe von Grafik und Text auf den angegeben Koordinatenbereich ein.

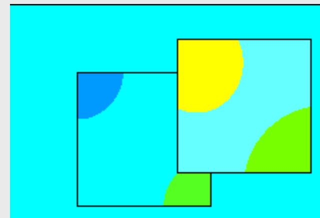
Syntax: **ScreenSetClipRect x0, y0, x1, y1**
x0, y0: linke obere Ecke des Clip-Rechtecks
x1, y1: rechte untere Ecke des Clip-Rechtecks

Beispiel:

```
Paper LIGHT_BLUE
CLS

ScreenSetClipRect 50, 50, 150, 150
FillEllipse 15, 15, 85, 85, BLUE
FillEllipse 115, 115, 185, 185, GREEN
Rectangle 50, 50, 150, 150, black

ScreenSetClipRect 125, 25, 225, 125
Paper LIGHT_CYAN
CLS
FillEllipse 105, 5, 175, 80, YELLOW
FillEllipse 175, 75, 295, 195, LIGHT_GREEN
Rectangle 125, 25, 225, 125, BLACK
```



Hinweise:
- Das aktuell eingestellte Clipping-Rechteck lässt sich nicht ermitteln.
- ScreenSaveState / ScreenRestoreState (siehe nächster Abschnitt) speichern das Clipping-Rechteck nicht. Ein nach einem ScreenSaveState ausgeführtes ScreenSetClipRect hat auch nach dem ScreenRestoreState Gültigkeit.
- Bei einigen Objekten, z.B. einem Canvas, kann ein zu großes Clipping-Rechteck auch dazu führen, dass das Objekt über seine Grenzen hinaus zeichnen kann.
- In den meisten Fällen stellt ScreenSetClipRect 0, 0, MaxX, MaxY das originale Clipping-Rechteck wieder her.

2.3.3 Speichern und Wiederherstellen des Screen-Status

Gelegentlich gibt es die Situation, dass man Änderungen in den Einstellungen für Grafik und/oder Text (Farben, Font, Linienbreite ...) vornehmen will, die aber anschließend wieder zurückgenommen werden sollen. Oft passiert das innerhalb einer Routine oder in einem Action-Handler, der Grafiken ausgibt aber sicherstellen will, dass die aktuell Screen-Einstellungen nicht verändert werden. Die Anweisung **ScreenSaveState** speichert alle den Screen betreffenden Einstellungen - mit Ausnahme der BlockFonts und des Clipping-Rechtecks. **ScreenRestoreState** stellt den gesicherten Zustand wieder her. Alle zwischenzeitlich vorgenommenen Änderungen für Text und Grafik gehen verloren.

ScreenSaveState kann mehrfach hintereinander aufgerufen werden, die Einstellungen werden von **ScreenSaveState** jeweils in einen eigenen Speicherbereich kopiert und von **ScreenRestoreState** in der umgekehrten Reihenfolge wieder restauriert. Dieses Vorgehen stellt sicher, dass eine Routine die Kombination ScreenSaveState / ScreenRestoreState verwenden kann, unabhängig davon, ob andere Routinen dies auch tun.

Beispiel: Die Routine zeichnet einen blauen Kreis ohne die aktuell eingestellten Linieneigenschaften zu verändern.

```
SUB  BlueCircle ( x, y as integer )
  ScreenSaveState
  graphic.lineColor = BLUE
  graphic.lineWidth = 4
  Circle x, y, 50
  ScreenRestoreState
END SUB
```

ScreenSaveState

Speichert die aktuellen Einstellungen für Grafik und Text.

Syntax: **ScreenSaveState**
Parameter: keine

ScreenRestoreState

Stellt die Einstellungen für Grafik und Text wieder her. Jede Routine **muss** genau so viele ScreenRestoreState wie ScreenSaveState aufrufen.

Syntax: **ScreenRestoreState**
Parameter: keine

Hinweise:

- Wird die Systemvariable **Screen** neu belegt, so gibt R-BASIC den eventuell noch von ScreenSaveState angeforderten Speicher frei, d.h. es werden genau so viele ScreenRestoreState ausgeführt, wie es offene ScreenSaveState gibt. Erst dann wird der Screen umgeschaltet.
- In folgenden Fällen kann ScreenRestoreState nach einem ScreenSaveState die Daten nicht wieder herstellen und es kommt zu einem Laufzeitfehler:
  1. Die Größe des Screenobjekts (z.B. der Bitmap) wurde zwischenzeitlich geändert.
  2. PC/GEOS wurde zwischenzeitlich heruntergefahren. Um diesen Fall zu vermeiden muss jede Routine genauso viele ScreenRestoreState wie ScreenSaveState aufrufen, weil PC/GEOS nicht herunterfahren kann, während noch ein Handler oder eine Routine läuft.
- Die BlockFont Zeichensätze (siehe Block-Grafik-Modus, Handbuch "Spezielle Themen", Kapitel 3) sind global. Sie werden weder vom Umschalten des Screens noch von **ScreenSaveState** / **ScreenRestoreState** verändert.
- Das Clipping-Rechteck wird von **ScreenSaveState** nicht gespeichert. Ein mit **ScreenSetClipRect** eingestellter Clipping-Bereich hat auch nach **Screen-RestoreState** weiter Gültigkeit.

2.3.4 Anpassen des Koordinatensystems

In diesem Abschnitt werden Routinen beschrieben, die einfache Operationen mit dem Koordinatensystem durchführen. Dabei handelt es sich um Verschiebung, Skalierung und Rotation. Diese Operationen werden als Koordinatentransformationen bezeichnet. Sie wirken immer auf die **nachfolgenden** Grafik- oder Text-Ausgaben. Bereits vorhandene Grafiken oder Texte werden nicht beeinflusst.

Werden mehrere Transformationen nacheinander ausgeführt, so berechnet GEOS intern eine "resultierende" Transformation, so dass es für die Performance keinen Unterschied macht wie viele Transformationen angewendet wurden.

Beachten Sie, dass die Reihenfolge der Transformationen wichtig ist, es ist etwas anderes ob man erst verschiebt und dann rotiert oder umgekehrt (siehe Beispiel unten).

Normalerweise begrenzt die PRINT-Anweisung die Ausgabe von Texten auf die ursprünglichen Koordinaten des Screens. Wenn Sie das Koordinatensystem geändert, z.B. nach rechts unten verschoben haben, kann Print z.B. den Zugriff auf Bereiche links und oberhalb des neuen Nullpunkts (negative Cursor-Koordinaten) verweigern. Sie müssen dann in den LAYOUT-Modus wechseln, in dem alle Cursor-Restriktionen, allerdings auch der automatische Zeilenumbruch, deaktiviert sind. Verwenden Sie dazu die Anweisung:

```
Print Chr$(19)                'oder gleichwertig Print "\19"
```

Print Chr$(17) ruft den PAGE-Modus auf, Print Chr$(18) den Scroll-Modus.

ScreenSetTranslation

Verschiebt den Ursprung des Koordinatensystems (Lage des Koordinatenursprungs) um die angegebenen Werte.

Syntax: **ScreenSetTranslation** **xMove**, **yMove**
xMove: Verschiebung in x-Richtung
yMove: Verschiebung in y-Richtung

ScreenSetScale

Skaliert die Achsen des Koordinatensystems um die angegebenen Werte. Negative Werte sind zulässig, sie bewirken eine Spiegelung in der entsprechenden Richtung. Bezugspunkt ist immer der aktuelle Koordinatenursprung.

Syntax: **ScreenSetScale** **xScale**, **yScale**
xScale: Streckung in x-Richtung
yScale: Streckung in y-Richtung

ScreenSetRotation

Rotiert das Koordinatensystem um den angegebenen Winkel. Die Drehung erfolgt um den aktuellen Koordinatenursprung

Syntax: **ScreenSetRotation alpha**
alpha: Drehwinkel im Gradmaß
Drehung erfolgt gegen den Uhrzeigersinn

Beispiel: Drehung um 45° nach rechts

```
ScreenSetRotation  -45
```

ScreenResetTransformation

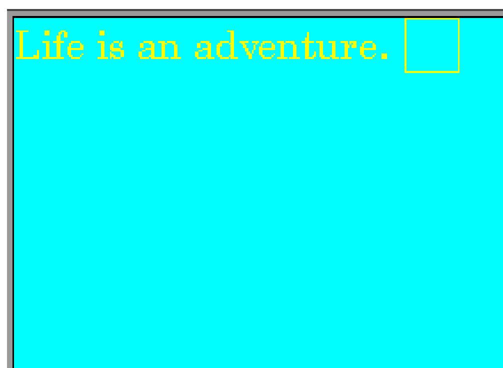
Nimmt alle Koordinatentransformationen zurück. Der Koordinatenursprung wird nach links oben gesetzt, Drehung und Skalierung werden zurückgesetzt.

Syntax: **ScreenResetTransformation**

Beispiel 1: Unterschiedliche Transformationen.

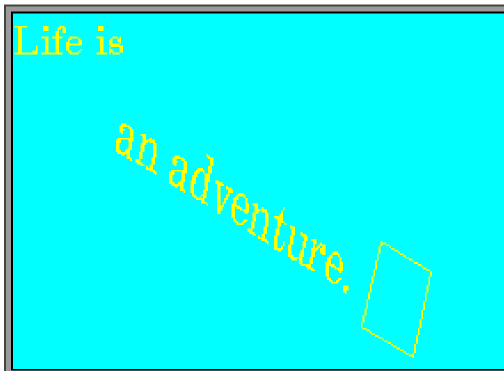
Textausgabe ohne Transformation:

```
FontSetGeos( FID_CRANBROOK, 20 )
Print "Life is ";
Print "an adventure.";
Rectangle 220, 0, 250, 30
```



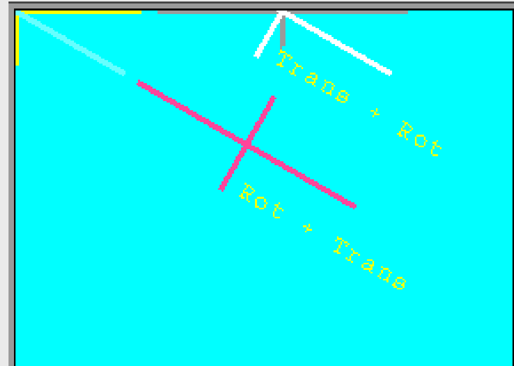
Gedrehte und skalierte Text-Ausgabe, die Reihenfolge spielt eine Rolle!

```
FontSetGeos( FID_CRANBROOK, 20 )
Print "Life is ";
ScreenSetScale 1, 1.7     ' y: 170%
ScreenSetRotation -20     ' 20° im Uhrzeigersinn
Print "an adventure.";
Rectangle 220, 0, 250, 30
```



Linkes Bild: Erst skaliert, dann rotiert (wie im Code gezeigt)
Rechtes Bild: Anweisungen ScreenSetScale und ScreenSetRotation vertauscht
(erst rotiert, dann skaliert)

Beispiel 2: Translation und Rotation.
Das Kreuz kennzeichnet jeweils den neuen Koordinatenursprung.




```
graphic.linewidth = 3
Line -70, 0, 70, 0, YELLOW
Line 0, -30, 0, 30, YELLOW

ScreenSetRotation -30
Line -70, 0, 70, 0, LIGHT_CYAN
Line 0, -30, 0, 30, LIGHT_CYAN
ScreenSetTranslation 150, 0
Line -70, 0, 70, 0, LIGHT_RED
Line 0, -30, 0, 30, LIGHT_RED
Print at 1,1;"Rot + Trans"

ScreenResetTransformation     ' wieder Ausgangssituation
                              ' herstellen
ScreenSetTranslation 150, 0
Line -70, 0, 70, 0, LIGHT_GRAY
Line 0, -30, 0, 30, LIGHT_GRAY
ScreenSetRotation -30
Line -70, 0, 70, 0, WHITE
Line 0, -30, 0, 30, WHITE
Print at 1,1;"Trans + Rot"
```

2.3.5 Komplexe Manipulation des Koordinatensystems

Hinweis: Dieses Kapitel ist etwas für erfahrene und mathematisch versierte Programmierer. Die meisten Programmierer werden niemals mit hier dargestellten Zusammenhängen arbeiten.

Wie bereits oben erwähnt berechnet GEOS bei Anwendung mehrerer Koordinatentransformationen eine "resultierende" Transformation. Dazu wird intern die Matrizenrechnung verwendet, die dazugehörige Struktur heißt Transformationsmatrix. Sie können die Transformationsmatrix lesen, verändern, eine eigene erstellen und sie wieder setzen. Dieses Kapitel beschreibt die dazugehörigen Routinen **ScreenGetTransMatrix**, **ScreenSetTransMatrix**, die Struktur **TransMatrix** und die mathematischen Grundlagen.

Jedes Mal, wenn GEOS einen einzelnen Punkt mit den Koordinaten (x; y) auf dem Bildschirm darstellt, wendet es die aktuell gültige Transformationsmatrix auf diesen Punkt an, um die Bildschirmkoordinaten (x'; y') zu erhalten. Dieses Verfahren wird für jeden einzelnen Punkt einer Linie, eines Buchstabens usw. angewendet. Diese Berechnungen sind für eine schnelle Grafikausgabe optimiert und in GEOS extrem effizient implementiert. In Matrizenschreibweise sieht das so aus:

$$(x' ; y' ; 1) = \begin{bmatrix} a1 & a2 & 0 \\ b1 & b2 & 0 \\ c1 & c2 & 1 \end{bmatrix} \cdot (x ; y ; 1)$$

Im Einzelnen berechnen sich die neuen Koordinaten somit folgendermaßen:

$$x' = a1 * x + b1 * y + c1$$
$$y' = b1 * x + b2 * y + c2$$

Beispiele für Transformation-Matrizen:

Standard Transformation:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Skalierung:
$$\begin{bmatrix} scaleX & 0 & 0 \\ 0 & scaleY & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ transX & transY & 1 \end{bmatrix}$$

Wird eine neue Koordinatentransformation angewendet (z.B. mit ScreenSetTranslation), so berechnet GEOS das Kreuzprodukt der alten Transformationsmatrix mit der neuen, wobei sich eine resultierende Transformationsmatrix ergibt, die alle bereits vorhandenen und die neue Koordinatentransformation enthält. Am Rechenaufwand bei der Darstellung auf dem Bildschirm, d.h. an der Ausgabegeschwindigkeit ändert sich dadurch nichts.

Erfahrene oder ambitionierte Programmierer können ihre eigene, komplexe Transformationsmatrix erstellen und diese mit **ScreenSetTransMatrix** anwenden. Oder man sichert die aktuelle Transformationsmatrix mit **ScreenGetTransMatrix** um sie später wieder zu verwenden. Dabei ist es erlaubt die Transformations-

matrix von einem Screen-Objekt zu lesen und einem anderen Screen-Objekt zuzuweisen.

Die Struktur **TransMatrix** enthält die 6 variablen Elemente der Transformations-Matrix und ist in R-BASIC folgendermaßen definiert:

```
STRUCT  TransMatrix
  a1, b1, c1      AS REAL
  a2, b2, c2      AS REAL
  END STRUCT
```

ScreenGetTransMatrix

Liest die aktuell gültige Transformationsmatrix des aktuellen Screen-Objekts aus. Die Klammern sind erforderlich, da es sich um eine Funktion handelt, d.h. sie liefert einen Wert zurück. Ist kein Screen-Objekt gesetzt ist das Ergebnis unbestimmt und sollte nicht verwendet werden.

Syntax: **<tm>** = **ScreenGetTransMatrix ( )**
<tm>: Variable vom Typ **TransMatrix**

ScreenSetTransMatrix

Wendet eine Transformationsmatrix auf den aktuellen Screen an. Ist kein Screen-Objekt gesetzt wird die Operation ignoriert.

Syntax: **ScreenSetTransMatrix** (**<tm>**)
<tm>: Variable oder Ausdruck vom Typ **TransMatrix**

2.4 Objekte individualisieren

Der folgende Abschnitt setzt voraus, dass Sie sich bereits etwas im GEOS Objektsystem auskennen und auch mit der Verwendung von Strukturen vertraut sind.

Es gibt Situationen, in denen es nötig ist, dass ein Objekt neben den vom System vorgegebenen Daten weitere Informationen speichern muss. Ein einfaches Beispiel ist ein Canvas-Objekt, dass entweder einen Kreis oder ein Quadrat zeichnen soll. Sie können die Information, ob ein Kreis oder ein Quadrat gezeichnet werden soll, natürlich für jedes Objekt in einer eigenen globalen Variablen speichern. Das ist aber nicht nur schlechter Stil, sondern wird bei mehreren Objekten auch schnell sehr unübersichtlich und damit fehlernfällig.
Die bessere Lösung ist, die Information im Objekt selber zu speichern. R-BASIC bietet Ihnen für diese Situation die Instancevariable **privData**. PrivData nimmt eine Strukurvariable beliebigen Typs auf und speichert sie im Objekt selbst. Hier können Sie z.B. ablegen, ob ein Kreis oder ein Quadrat gezeichnet werden soll. Außerdem können Sie - wenn Sie wollen - die Größe, die Farbe und beliebige weitere Informationen speichern.

VisObj-Objekte haben zusätzlich die Instancevariable **visDataValue**. Sie enthält einen numerischen Wert (LongInt-Bereich), mit dem Sie bei Bedarf verschiedene VisObj-Objekte auseinanderhalten können, ohne auf die relativ umständliche Verwendung der Instancevariablen privData zurückzugreifen. Außerdem können Sie visDataValue bereits im UI-Code zuweisen.

Fortgeschrittene Programmierer können in seltenen Situationen den Bedarf haben, dass sie eine Routine erst dann aufrufen wollen, wenn der aktuelle Actionhandler vollständig abgearbeitet ist. Typische Beispiele sind hier der OnPrint Handler (bei dem man das Screen-Objekt nicht ändern darf) oder ein OnMouse~ bzw. der OnKeyPressed Handler (die meist zeitkritisch sind). R-BASIC löst dieses Problem, indem man für Objekte eigene, private ("custom") Handler definieren kann. Actionhandler unterbrechen sich niemals, sondern werden immer nacheinander abgearbeitet. Der Aufruf eines solchen Handlers führt also dazu, dass die aktuelle Routine (genauer: der komplette aktuell laufende Handler) zuerst vollständig abgearbeitet wird bevor der neue Handler ausgeführt wird. Um einen Custom Handler für ein Objekt festzulegen verwenden Sie die Instancevariable **customHandler**. Custom Handler müssen als **CustomAction** deklariert sein. Um einen Custom Handler aufzurufen verwenden Sie die Methode **CustomApply**.

PrivData

PrivData nimmt eine einzelne Strukturvariable (also maximal 3500 Bytes) auf. Diese Instancevariable ist für **alle** Klassen definiert. PrivData ist zuweisungs-kompatibel mit jeder Art von Struktur, es wird weder eine Typ- noch eine

Größenprüfung ausgeführt. Es ist daher vernünftig beim Schreiben und beim Lesen der Daten den gleichen Struktur-Datentyp zu verwenden.

Schreiben: Sie müssen die Größe der zu schreibenden Daten angeben.

Lesen: Es werden so viele Bytes gelesen, wie die Variable auf der linken Seite der Zuweisung aufnehmen kann. Enthält privData weniger Bytes, so wird der Rest mit Nullen aufgefüllt.

Es ist zulässig, mehrfach hintereinander Strukturen verschiedenen Typs und verschiedener Größe in die Instancevariable zu schreiben. R-BASIC optimiert jedes Mal den verwendeten Speicher, so dass kein Platz verschwendet wird.

Syntax Schreiben: **<obj>.privData = <struct>, size**
<struct>: Strukturausdruck beliebigen Typs
size Größe der Struktur
Lesen: **<sturctVar> = <obj>.privData**
<structVar>: Strukturvariable des Typs, der beim Schreiben verwendet wurde.

Beispiel:
Ein Canvas-Objekt soll einen Kreis oder ein Quadrat in einer vorgegebenen Farbe zeichnen. Wir benötigen:
- einen Strukturtyp, der die Informationen enthält,
- eine Routine, die die Werte setzt,
- ein Canvas-Objekt,
- einen OnDraw Handler für das Canvas Objekt

Der Strukturtyp sei folgendermaßen defininiert:

```
STRUCT ImgData
  isCircle   as Integer
  color      as Integer
End Struct
```

Zum Belegen der Instancewerte dient die folgende Routine. Die zweite Routine (SetCanvasToRect) ist hier nicht aufgeführt.

```
SUB SetCanvasToCircle( col as Integer)
DIM pd AS ImgData
  pd.isCircle = TRUE
  pd.color = col
  MyCanvas.privData = pd, SIZEOF(pd)
  MyCanvas.Dirty                          ' Neudarstellung auslösen
End Sub
```

Das Canvas-Objekt sei wie folgt definiert. Beachten Sie, dass wir privData nicht definieren brauchen, es ist für alle Objekte automatisch verfügbar.

```
CANVAS  MyCanvas
  OnDraw = DrawFigure
  fixedSize = 70, 70
End Object
```

Schließlich benötigen wir noch den OnDraw-Handler, der die privData-Werte ausliest und verwendet.

```
DRAWACTION DrawFigure
DIM priv as ImgData

  priv = sender.privData
  INK priv.col
  IF priv.isCircle THEN
    FillEllipse 10, 10, 60, 60
  ELSE
    FillRect 10, 10, 60, 60
  END IF
End Action
```

CustomHandler

CustomHandler enthält den Namen des Actionhandlers, der mit der Methode CustomApply aufgerufen werden soll.

Syntax	UI- Code:	**CustomHandler** = <Handler>
	Schreiben:	<obj>.**CustomHandler** = <Handler>

Ein Custom Handler muss als CustomAction deklariert sein:

Handler-Typ	Parameter
CustomAction	(sender as object, actionData as integer)

CustomApply

Die Methode CustomApply ruft den CustomHandler eines Objekts auf. Ihr wird ein Integer-Wert übergeben, der an den Handler weitergereicht wird.

Syntax:	<obj>.**CustomApply** actionData
	actionData: Integerwert, der an CustomHandler übergeben wird.

Beispiel (einfach, deswegen nicht sehr sinnvoll):
Ein Button mit einem ActionHandler und Primary mit einem CustomHandler.

```
BUTTON MyButton
  Caption$ "Drück mich!"
  ActionHandler = PressHandler
End Object

Primary MyPrimary
  CustomHandler = CHandler
End Object
```

Der Actionhandler:

```
ButtonAction PressHandler
  Print "Text 1"
  MyPrimary.CustomApply   1
  Print "Text 2"
  MyPrimary.CustomApply   -7
  Print "Text 3"
End Action
```

Der CustomHandler wird erst ausgeführt, wenn der ActionHandler fertig ist

```
CustomAction CHandler
  Print "DATA = ";actionData
End Action
```

Wenn der Nutzer den Button drückt erscheint folgendes:

```
  Text 1
  Text 2
  Text 3
  DATA = 1
  DATA =-7
```

(Leerseite)