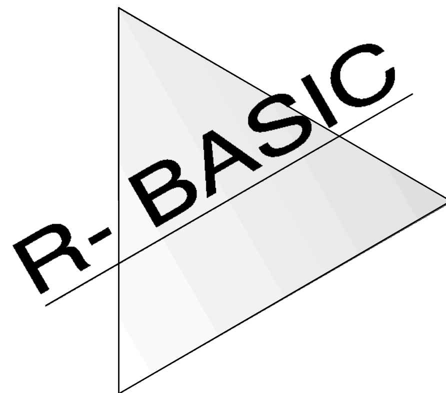
R-BASIC

Einfach unter PC/GEOS programmieren



Handbuch

Teil 2 - Objekte
Volume 4: Generic Objekt Klassen (2)

Version 0.8 Beta

Inhaltsverzeichnis - Volume 4

4.8 Listen-Objekte 4
- 4.8.1 Überblick 4
- 4.8.2 Option und OptionGroup 5
- 4.8.3 RadioButton und RadioButtonGroup 9
- 4.8.4 DynamicList 14
- 4.8.5 Listen-Objekte im Delayed Mode 17

4.9 View und Content 18
- 4.9.1 Überblick 18
- 4.9.2 Das View 18
  - 4.9.2.1 View Geometrie 19
  - 4.9.2.2 Verbindung zum Content 21
  - 4.9.2.3 Children eines View-Objekts 23
- 4.9.3 Das Content 23
- 4.9.4 Das BitmapContent 24

4.10 Text-Objekte 28
- 4.10.1 Überblick 28
- 4.10.2 Der Text 29
- 4.10.3 Text-Objekt Actions 30
- 4.10.4 Text-Formatierung 33
- 4.10.5 Text-Objekte im Delayed Mode 33

Noch ohne Zuordnung
- Maushandling
- Kbd Handling
- Drucken
- Timer

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 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 den Teil 2 des Handbuchs.

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 darauf folgenden Kapiteln vorausgesetzt.

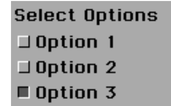
Grundlegenden Befehle und Konzepte, die die R-BASIC Programmiersprache ausmachen finden Sie im Teil 1. Dort wird auch erklärt, wie man das R-BASIC Oberfläche benutzt und Sie erfahren vieles über Variablen, Schleifen, Verzweig-ungen, Unterprogramme und andere grundlegende Dinge.

Der dritte Teil des Handbuchs widmet sich speziellen Themen, wie der Arbeit mit Dateien oder die Verwendung von Schriften.

4.8 Listen-Objekte

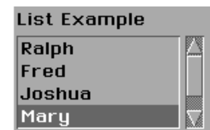
4.8.1 Überblick

In R-BASIC gibt es 3 Typen von Listen-Objekten:



- Die **OptionGroup** verwaltet eine Liste von Option-Objekten, das jedes für sich und abhängig voneinander den Zustand "ein" oder "aus" haben kann.

- Die **RadioButtonGroup** verwaltet eine Liste von RadioButton-Objekten, die einzeln oder in Gruppen ausgewählt (selektiert) werden können. Die RadioButtonGroup ist ein sehr vielseitiges Objekt das insbesondere dann zum Einsatz kommt, wenn die Anzahl der Listeneinträge von vorneherein bekannt und unveränderlich ist.



- Die **DynamicList** stammt von der RadioButtonGroup ab und ist daher genauso vielseitig wie diese. Sie wird eingesetzt, wenn die Anzahl der Listeneinträge nicht von vorneherein bekannt ist und / oder sich während des Programmablaufs verändert.

Alle Listen können einen ActionHandler aufrufen, wenn vom Nutzer ein Eintrag selektiert bzw. geändert wird. Über die Instance-Variable **look** kann das Aussehen der Listen weitgehend verändert werden. So können alle Listen-Objekte - nicht nur die DynamicList - als scrollbare Listen auftreten. Die Größe eines als scrollbare Liste erscheinenden List-Objekts wird häufig über den Geometrie-Hint **fixedSize** eingestellt. Beispiele dazu finden Sie bei der Beschreibung der Instance-Variablen **look** der OptionGroup.

Die folgenden Ausführungen gehen zunächst grundsätzlich davon aus, dass die Listen-Objekte im normalen Modus (nicht im sogenannten "Delayed Mode") arbeiten. Das ist der Normalfall wenn man nicht spezielle Hints setzt, um in den Delayed Mode zu kommen. Dieser "Delayed Mode" ist ausführlich im Kapitel 3.4.2 (Delayed Mode und Status-Message) dieses Handbuchs beschrieben.

Action-Handler-Typen:

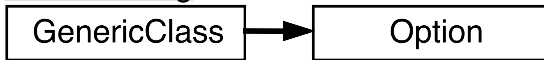
Handler-Typ	Parameter
ListAction	(sender as object, selection as word, modified as word, numSelections as word)

Alle ActionHandler der List-Objekte müssen als **ListAction** deklariert sein. Die Bedeutung der Parameter "selection", "modified" und "numSelections" variiert je nach Listen-Objekt und ActionHandler. Gelegentlich sind einige der Parameter auch bedeutungslos für den speziellen Fall.

4.8.2 Option und OptionGroup

Option

Abstammung



Spezielle Instance-Variablen

Instance-Variable	Syntax im UI-Code	Im BASIC-Code
identifier	identifier = numWert	lesen, schreiben

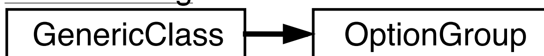
identifier

Die Instance-Variable **identifier** identifiziert das einzelne Option-Objekt. Sie ist vom Typ WORD. Der Wert muss eine 2er-Potenz sein (nur genau 1 Bit gesetzt, d.h. einer der Werte 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 81912, 16384, 32768). Option-Objekte müssen Children einer OptionGroup sein. Innerhalb einer OptionGroup darf jeder Identifier-Wert nur genau einmal vorkommen.

Syntax	UI- Code:	**identifier** = numWert
	Lesen:	<numVar> = <obj> . **identifier**
	Schreiben:	<obj>.**identifier** = numWert

OptionGroup

Abstammung

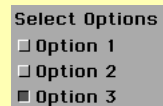


Eine OptionGroup managed eine Liste von Option-Objekten (bis zu 16). OptionGroup-Objekte können nur Option-Objekte als Children haben.

Beispiel UI-Code:

```
OptionGroup ListOfOptions
    Caption$= "Select Options"
    justifyCaption = J_TOP
    Children = bool1, bool2, bool3
    OrientChildren = ORIENT_VERTICALLY
    ApplyHandler = BoolApply
    selection = 4
    END Object

Option bool1: caption$="Option 1":identifier = 1: end object
Option bool2: caption$="Option 2":identifier = 2: end object
Option bool3
    Caption$="Option 3" : identifier = 4:
    end object
```



Zur Demonstration wurde in an einigen Stellen im Code oben die Syntax mit mehreren, durch Doppelpunkt getrennten Anweisungen verwendet, die auch im UI-Code zulässig ist.

Spezielle Instance-Variablen

Instance-Variable	Syntax im UI-Code	Im BASIC-Code
selection	identifier = numWert	lesen, schreiben
look	look = numWert	lesen, schreiben
modified	modified = numWert	lesen, schreiben
ApplyHandler	ApplyHandler = <Handler>	nur schreiben
StatusHandler	StatusHandler = <Handler>	nur schreiben

Methoden:

Methode	Aufgabe
SendStatus	Status-Handler aufrufen

selection

Option's können den Zustand "ein" oder "aus" haben. Die Instance-Variable **selection** der OptionGroup enthält die Summe der identifier derjenigen Option's, die auf "ein" sind. Genauer gesagt ist es die logische Oder-Verknüpfung der **identifier**. Wenn man sich an die Regel hält, dass Option-Identifier nur 2er-Potenzen sein dürfen (was man unbedingt sollte), sind beide Aussagen gleichwertig.

Syntax	UI- Code:	**selection** = numWert
	Lesen:	<numVar> = <obj> . **selection**
	Schreiben:	<obj>.**selection** = numWert

look

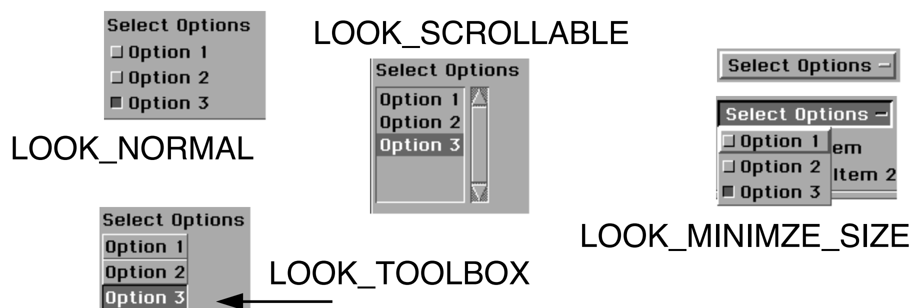
Instance-Variable **look** bestimmt das Aussehen der OptionGroup und ihrer Option-Objekte (engl. look : Aussehen, äußere Erscheinung). Funktionell sind alle Looks identisch.

Syntax	UI- Code:	**look** = numWert
	Lesen:	<numVar> = <obj> . **look**
	Schreiben:	<obj>.**look** = numWert

Für alle Listen-Objekte stehen folgende Looks zur Verfügung:

Konstante	Wert	Aussehen
LOOK_NORMAL	0	Klassisches Aussehen
LOOK_SCROLLABLE	1	Scrollbare Liste
LOOK_MINIMIZE_SIZE	2	Minimaler Platzverbrauch
LOOK_TOOLBOX	4	ToolBox-Style

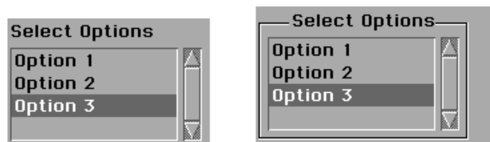
Die Look-Werte können kombiniert werden (mit +), was insbesondere bei LOOK_MINIMIZE_SIZE + LOOK_TOOLBOX gelegentlich Sinn macht. Ungültige bzw. widersprüchliche Kombinationen können jedoch zu seltsamen Effekten führen.



Das Bild zeigt die Liste mit dem UI-Code von oben, jedoch jeweils mit verschiedenen Werten für **look** gesetzt.

Insbesondere bei scrollbaren Listen besteht häufig der Bedarf die Größe und die Anzahl der gleichzeitig sichtbaren Listeneinträge festzulegen. Dazu eignet sich der Geometrie-Hint **fixedSize**. Die in den folgenden Bildern dargestellten Listen haben die Instance-Variable **look** auf LOOK_SCROLLABLE und folgenden fixedSize-Hint gesetzt:

```
fixedSize = 15 + ST_AVG_CHAR_WIDTH, 4 + ST_TEXT_LINES, 4
```



Die Listen-Objekte im Bild sind 15 Zeichen breit und das Listen-Fenster ist 4 Zeilen hoch (4 + ST_TEXT_LINES). Der letzte Parameter (4) bestimmt, dass 4 Einträge im Fenster gleichzeitig dargestellt werden sollen. Sinnvollerweise ist er identisch mit der Höhe, gemessen in Textzeilen. Die Liste rechts im Bild hat - zur Demonstration - zusätzlich den Hint **DrawInBox** gesetzt.

modified

Die Instance-Variable **modified** der OptionGroup enthält die logische OR-Verknüpfung der Identifier derjenigen Option's, die seit der letzten Apply-Aktion modifiziert wurden.

Beachten Sie, dass ein Verändern des Objekts vom BASIC-Code aus (z.B. Belegen der Instance-Variable **selection**), das Objekt nicht als "modified" markiert, d.h. der Wert der Instance-Variablen **modified** wird nicht verändert. Sie können dies bei Bedarf selbst machen, indem Sie die Anweisung "<obj>.modified = numWert" verwenden, wobei "numWert" ein einzelner Identifier oder die OR-Verknüfung mehrerer Identifier der Option Objekte aus der OptionGroup sein soll.

Syntax		
	UI- Code:	**modified** = numWert
	Lesen:	<numVar> = <obj> . **modified**
	Schreiben:	<obj>.**modified** = numWert

Wenn Sie die Instance Variable **modified** lesen, werden Sie feststellen, dass sie Null enthält, es sei denn, Sie haben sie explizit auf auf einen anderen Wert gesetzt. Ändert der Nutzer nämlich des Zustand eines Option-Objekts, so passiert intern folgendes:

- Die Instance-Variable **modified** wird mit dem Identifier des betroffenen Option-Objkets belegt.
- Es wird geprüft ob ein ApplyHandler vorhanden ist und dieser wird ggf. aufgerufen. Der Wert von **modified** wird dem Handler übergeben.
- Die Instance-Variable **modified** wird zurückgesetzt (mit Null belegt).

Hinweis: Im sogenannten Delayed Mode (siehe entsprechendes Kapitel weiter unten) werden die letzten beiden Schritte nicht ausgeführt, so dass die Instance-Variable **modified** eine eigene Bedeutung erhält.

ApplyHandler

Der **ApplyHandler** der OptionGroup wird aufgerufen, wenn eines der Option-Objekte in der Group geändert wird. Er muss als **ListAction** deklariert sein.

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

Beispiel, passend zum UI-Code oben:
```
ListAction BoolApply
    Print selection; modified
    IF modified AND 1 THEN Print "Option 1 geklickt"
    IF modified AND 4 THEN Print "Option 3 geklickt"
    END Action
```

Der Parameter **selection** enthält die selektierten Options (OR-Verknüpfung, d.h. die Summe der Identifier).
Der Parameter **modified** enthält den Identifier, der geändert wurde und so den Apply-Handler auslöste.
Die Abfrage erfolgt mit dem logischen Operator AND, siehe Beispiel

Achtung! Der Parameter numSelections ist hier bedeutungslos.

Hinweis: Es ist möglich den ApplyHandler der OptionGroup manuell (vom BASIC-Code aus) zu aktivieren. Dazu wird die von der GenericClass geerbte Methode **Apply** verwendet. Da ApplyHandler nur ausgelöst werden, wenn das Objekt "modified" ist, muss es vorher als "modified" markiert werden. Alternativ könnte man dem Objekt auch den Hint *ApplyEvenIfNotModifie*d geben.

Beispiel:

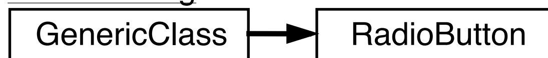
```
ListOfOptions.modified = TRUE
ListOfOptions.Apply
```

Eine ausführliche Beschreibung dazu finden Sie im Kapitel 3.4 (Die "Apply"-Message) dieses Handbuchs.

4.8.3 RadioButton und RadioButtonGroup

RadioButton

Abstammung



Spezielle Instance-Variablen

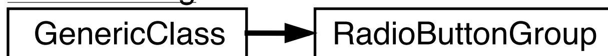
Instance-Variable	Syntax im UI-Code	Im BASIC-Code
identifier	identifier = numWert	lesen, schreiben

identifier

RadioButton’s haben einen **identifier**, der sie identifiziert. Es ist vom Typ WORD und muss innerhalb einer RadioButtonGroup eindeutig sein. 65535 (alle Bits im WORD gesetzt) ist nicht zulässig, auch wenn es keine sofortige Fehlermeldung gibt. Dieser Wert wird verwendet wenn kein RadioButton-Objekt selektiert ist / werden soll. Für ihn gibt es die Konstante NONE_SELECTED.
RadioButton-Objekte müssen Children einer RadioButtonGroup sein.

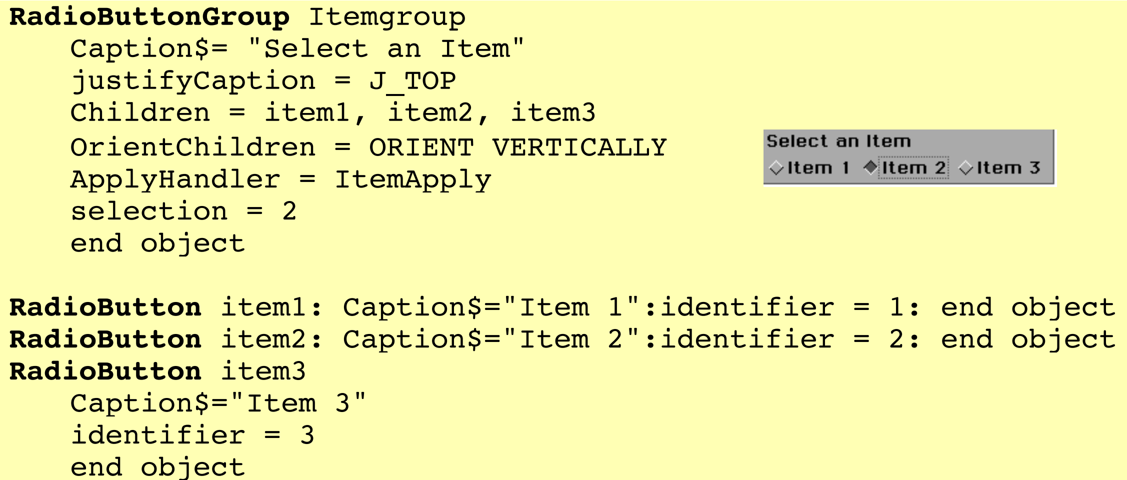
RadioButtonGroup

Abstammung



Eine RadioButtonGroup managed eine Liste von RadioButton-Objekten (theoretisch bis über 65000). RadioButtonGroup-Objekte können nur RadioButton-Objekte als Children haben.

Beispiel:




```
RadioButtonGroup Itemgroup
    Caption$= "Select an Item"
    justifyCaption = J_TOP
    Children = item1, item2, item3
    OrientChildren = ORIENT VERTICALLY
    ApplyHandler = ItemApply
    selection = 2
    end object

RadioButton item1: Caption$="Item 1":identifier = 1: end object
RadioButton item2: Caption$="Item 2":identifier = 2: end object
RadioButton item3
    Caption$="Item 3"
    identifier = 3
    end object
```

Die UI-Anweisungen für die Objekte item1 und item2 wurden zur Demonstration in einer Zeile untergebracht. Dazu wird - wie im BASIC-Code auch - ein Doppelpunkt zur Trennung verwendet.

Spezielle Instance-Variablen

Instance-Variable	Syntax im UI-Code	Im BASIC-Code
behavior	behavior = numWert	lesen, schreiben
look	look = numWert	lesen, schreiben
selection	selection = numWert	lesen, schreiben
numSelections	—	nur lesen
DisplayCurrentSelection	DisplayCurrentSelection	—
modified	modified = numWert	lesen, schreiben
ApplyHandler	ApplyHandler = <Handler>	nur schreiben
StatusHandler	StatusHandler = <Handler>	nur schreiben
DoublePressHandler	DoublePressHandler = <Handler>	nur schreiben

Methoden:

Methode	Aufgabe
SendStatus	Status-Handler aufrufen

behavior

Die Instance-Variable **behavior** bestimmt, wie sich die Group bezüglich Selektionsmöglichkeiten der Einträge verhält.

Konstante	Wert	Bedeutung
LB_EXCLUSIVE	0	Das ist der Default-Wert. Nur genau ein Element kann selektiert sein
LB_EXCLUSIVE_NONE	1	Ein oder kein Element kann selektiert sein. Ist kein Element selektiert, wird als "selection" 65535 (NONE_SELECTED) geliefert.
LB_EXTENDED_SELECTION	2	Wie LB_EXCLUSIVE_NONE, aber der Nutzer kann die Selektion durch Ziehen mit der Maus oder durch Shift-Klick oder Ctrl-Klick "erweitern". Es können also mehrere Elemente selektiert sein.
LB_NON_EXCLUSIVE	3	Jedes Element kann unabhängig von den anderen einzeln selektiert werden. Wie bei einer OptionGroup. Die Liste sieht dann auch so aus wie eine OptionGroup. XXX noch keine Abfrage der einzelnen Items implementiert - daher akt nicht nutzbar.

look

Instance-Variable **look** bestimmt das Aussehen der RadioButtonGroup und ihrer RadioButton-Objekte (engl. look : Aussehen, äußere Erscheinung). Funktionell sind alle Looks identisch.

Syntax		
	UI- Code:	**look** = numWert
	Lesen:	<numVar> = <obj> . **look**
	Schreiben:	<obj>.**look** = numWert

Für alle Listen-Objekte stehen die bei der OptionGroup beschriebenen Looks (LOOK_NORMAL, LOOK_SCROLLABLE, LOOK_MINIMIZE_SIZE und LOOK_TOOLBOX) zur Verfügung. Dort finden Sie auch Bilder und weitergehende Informationen dazu.

selection

Die Instance-Variable **selection** enthält den Identifier des aktuell selektierten RadioButton-Objekts. Ist kein Objekt selektiert, enthält sie den Wert 65535 (NONE_SELECTED). Falls mehrere Objekte selektiert sind, enthält **selection** den Identifier eines der selektierten Objekte. XXX Einzelabfrage noch nicht implementiert

Syntax	UI- Code:	**selection** = numWert
	Lesen:	<numVar> = <obj> . **selection**
	Schreiben:	<obj>.**selection** = numWert

Konstante	Wert	Bedeutung
NONE_SELECTED	65535	Spezialwert für die Instance-Variable "selection" wenn kein Eintrag selektiert ist oder kein Eintrag selektiert werden soll. **Behavior** sollte den Wert LB_EXTENDED_SELECTION oder LB_EXCLUSIVE_NONE haben.

numSelections

Die Instance-Variable **numSelections** enthält die Anzahl der aktuell selektierten Listeneinträge.

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

DisplayCurrentSelection

Der Hint **DisplayCurrentSelection** beeinflusst RadioButtonsGroups mit **look** = LOOK_MINIMIZE_SIZE und bewirkt, dass im minimierten Zustand statt des Caption-Textes der RadioButtonGroup der Text des aktuell selektierten RadioButtons angezeigt wird.

Syntax	UI- Code:	**DisplayCurrentSelection**

```
RadioButtonGroup Itemgroup
    Caption$= "Select an Item"
    Children = item1, item2, item3
    look = LOOK MINIMIZE SIZE
    DisplayCurrentSelection
    fixedSize = 15 + ST AVG CHAR WIDTH, 1 + ST TEXT LINES
    ApplyHandler = ItemApply
    selection = 2
    end object
```

Das Bild zeigt die RadioButtonGroup entsprechend dem obigen Code, links ohne und rechts mit dem Hint **DisplayCurrentSelection**.

modified

Die Instance-Variable **modified** der RadioButtonGroup enthält die Information, ob die Selektion der RadioButtonGroup seit der letzen Apply-Aktion geändert wurde.

Beachten Sie, dass ein Verändern des Objekts vom BASIC-Code aus (z.B. Belegen der Instance-Variable **selection**), das Objekt nicht als "modified" markiert, d.h. der Wert der Instance-Variablen **modified** wird nicht verändert. Sie können dies bei Bedarf selbst machen, indem Sie die Anweisung "<obj>.modified = TRUE" verwenden.

Syntax	UI- Code:	**modified** = TRUE I FALSE
	Lesen:	<numVar> = <obj> . **modified**
	Schreiben:	<obj>.**modified** = TRUE I FALSE

Wenn Sie die Instance Variable **modified** lesen, werden Sie feststellen, dass sie Null enthält, es sei denn, Sie haben sie explizit auf auf einen anderen Wert gesetzt. Ändert der Nutzer nämlich Auswahl innerhalb der Liste, so passiert intern folgendes:

- Die Instance-Variable **modified** wird mit TRUE belegt.
- Es wird geprüft ob ein ApplyHandler vorhanden ist und dieser wird ggf. aufgerufen. Der Wert von **modified** wird dem Handler übergeben.
- Die Instance-Variable **modified** wird zurückgesetzt (mit FALSE belegt).

Hinweis: Im sogenannten Delayed Mode (siehe entsprechendes Kapitel weiter unten) werden die letzten beiden Schritte nicht ausgeführt, so dass die Instance-Variable **modified** eine eigene Bedeutung erhält.

ApplyHandler

Der **ApplyHandler** der RadioButtonGroup wird aufgerufen, wenn der User ein Element selektiert / die Selektion ändert. Er muss als **ListAction** deklariert sein.

Beispiel, passend zum UI-Code oben:
```
ListAction   ItemApply
    Print selection; numSelections
    END Action
```

Der Parameter **selection** enthält den Identifier, der geändert wurde und so den Apply-Handler auslöste.
Der Parameter **numSelections** enthält die Anzahl der der selektierten Elemente.
Der Parameter **modified** ist immer 65535, d.h. alle Bits des WORD sind gesetzt. Das ist das Analogon zur Integer-Konstanten TRUE (= –1).

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

Hinweis: Es ist möglich den ApplyHandler der RadioButtonGroup manuell (vom BASIC-Code aus) zu aktivieren. Dazu wird die von der GenericClass geerbte Methode **Apply** verwendet. Da ApplyHandler nur ausgelöst werden, wenn das Objekt "modified" ist, muss es vorher als "modified" markiert werden. Alternativ könnte man dem Objekt auch den Hint *ApplyEvenIfNotModified* geben.

Beispiel:

```
    Itemgroup.modified = TRUE
    Itemgroup.Apply
```

Eine ausführliche Beschreibung dazu finden Sie im Kapitel 3.4 (Die "Apply"-Message) dieses Handbuchs.

DoublePressHandler

Der **DoublePressHandler** wird aufgerufen, wenn der User ein Element mit der Maus doppelklickt. War der Eintrag bis dahin noch nicht selektiert wird vorher der **ApplyHandler** aufgerufen. Die Parameter entsprechen denen des ApplyHandlers.

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

Hinweis: GEOS unterstützt einen DoublePressHandler nur, wenn die Instance-Variable **behavior** auf LB_EXCLUSIVE oder LB_EXTENDED_SELECTION gesetzt ist. R-BASIC kann nichts dafür - sorry.

4.8.4 DynamicList

Abstammung

Eine DynamicList ist eine erweiterte RadioButtonGroup. Daher erbt sie alle Instance-Variablen und Fähigkeiten dieser Klasse (behavior, look, selection, modified, ApplyHandler, DoublePressHandler, arbeit im Delayed Mode usw.). Zusätzlich hat sie folgende Besonderheiten:

- Die Instance-Variable **look** steht per default auf LOOK_SCROLLABLE. Sie können das natürlich im UI-Code ändern.
- Eine DynamicList hat im UI-Code **keine Children**. Sie erzeugt und verwaltet ihren Children (Listeneinträge) selbst.
- Sie **müssen** der Instance-Variable **numItems** auf eine Wert ungleich Null setzen und einen **QueryHandler** für eine DynamicList schreiben, sonst werden keine Listeneinträge angezeigt.

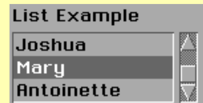
Spezielle Instance-Variablen

Instance-Variable	Syntax im UI-Code	Im BASIC-Code
numItems	numItems = numWert	lesen, schreiben
ItemText$	—	nur schreiben
QueryHandler	QueryHandler = <Handler>	nur schreiben

So arbeitet eine DynamicList

Nehmen wir an, wie haben eine DynamicList mit 5 Einträgen, die hier als Items bezeichnet werden, so wie im Code-Beispiel dargestellt.

```
DynamicList DynList
    Caption$ = "List Example"
    justifyCaption = J_TOP
    numItems = 5
    fixedSize = 15 + ST_AVG_CHAR_WIDTH, 3 + ST_TEXT_LINES, 3
    selection = 3
    ApplyHandler = MyApplyHandler
    QueryHandler = MyListQuery
    END Object
```



Die Instance-Variable **numItems** bestimmt, wie viele Items eine DynamicList hat. Die Items sind selbst Objekte, sie werden aber im UI-Code nicht aufgeführt, sondern die DynamicList erzeugt sie bei Bedarf selbst. Die von der RadioButtonGroup-Klasse geerbte Instance-Variable **selection** legt fest, welcher Eintrag am Anfang selektiert ist. Ebenfalls von der RadioButtonGroup Klasse geerbt ist die Instance-Variable ApplyHandler. **FixedSize** hingegen ist von der GenericClass geerbt und legt im Beispiel fest, dass die Liste 15 Zeichen breit und 3 Zeilen hoch ist, wobei 3 Items gleichzeitig angezeigt werden sollen.

Will eine DynamicList eines ihrer Items darstellen, so benötigt sie eine Information, welchen Text das Item darzustellen hat. Dazu ruft sie den **QueryHandler** auf (engl. to query: anfordern). Diesem wird die Nummer des Items, das dargestellt werden soll, übergeben. Die Zählung beginnt dabei immer mit Null. Der Handler muss den anzuzeigenden Text ermitteln und ihn an die DynamicList übergeben, wie im Beispielcode dargestellt.

```
LISTACTION MyListQuery
DIM name$   AS String

  ON selection SWITCH
    CASE 0: name$ = "Ralph"   : END CASE
    CASE 1: name$ = "Fred"    : END CASE
    CASE 2: name$ = "Joshua"  : END CASE
    CASE 3: name$ = "Mary"    : END CASE
    CASE 4: name$ = "Antoinette": END CASE
    DEFAULT:  name$ = "no name"             'Nur zur Sicherheit!
  END SWITCH

  sender.ItemText$(selection) = name$

  END Action
```

numItems

Die Instance-Variable **numItems** bestimmt, wie viele Listeneinträge die DynamicList hat. Sie kann im sowohl UI-Code als auch im BASIC-Code gesetzt werden.

Syntax	UI- Code:	**numItems** = numWert
	Lesen:	<numVar> = <obj> . **numItems**
	Schreiben:	<obj>.**numItems** = numWert

Es ist explizit zulässig, **numItems** im UI-Code nicht oder mit Null zu belegen, was die gleiche Wirkung hat. Die Liste bleibt dann zunächst leer. Der Ändern Sie den Wert von **numItems** im BASIC-Code, so stellt sich die Liste neu dar. Hat **numItems** den Wert Null, so sendet die DynamicList keine Query-Message aus. Es ist daher eine gute Idee, **numItems** am Programmende auf Null zu setzen.

ItemText$

Die Instance-Variable **ItemText$**(index) kann nur im BASIC-Code geschrieben werden. Das passiert üblicherweise im QueryHandler der DynamicList. Über **ItemText$**(index) wird der Liste mitgeteilt, welchen Text das entsprechende Item darzustellen hat. "Index" bestimmt, welchem Item der Text zugeordnet wird.

Syntax	Schreiben:	<obj>.**ItemText$(**index**)** = "Text"

QueryHandler

Der **QueryHandler** wird automatisch aufgerufen, wenn die DynamicList eines seiner Items darstellen will. QueryHandler müssen als **ListAction** deklariert sein. Der Parameter **selection** enthält die Nummer des Items, für dass ein Text angefordert wird. Die anderen Parameter sind bedeutungslos.
Die korrekte Reaktion des QueryHandlers ist, wie im Code-Beispiel oben, die Instance-Variable **ItemText$**(selection) der DynamicList zu belegen.

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

Das Zusammenspiel zwischen DynamicList und QueryHandler funktioniert automatisch, so dass Sie sich nicht weiter darum kümmern müssen. Sie müssen nur sicherstellen, dass der QueryHandler zu jedem selection-Wert, der von der Liste kommen kann (Null ... numItems–1), einen passenden Text bereitstellt. Dabei ist es, wie im Beispiel-Code oben, sinnvoll auch "unerwartete" Fälle zu berücksichtigen, falls sie später etwas ändern oder ein Programmierfehler auftritt.

4.8.5 Listen-Objekte im Delayed Mode

Alle Listen-Objekte können im "Delayed Mode" (engl.: verzögerter Modus) arbeiten. Dazu muss man dem Objekt selbst bzw. einem seiner Parents im UI-Code den Hint **MakeDelayedApply** geben oder man bindet das Objekt als Child in einem Dialog ein, dessen **dialogType** Instance Variable auf DT_DELAYED_APPLY gesetzt ist. Dieser "Delayed Mode" ist ausführlich im Kapitel 3.4.2 (Delayed Mode und Status-Message) dieses Handbuchs beschrieben, eine Beschreibung des Dialog-Objekts im Delayed Mode finden Sie im Kapitel 4.6.6.5.

Instance Variable	Syntax im UI-Code	Im BASIC-Code
StatusHandler	StatusHandler = <Handler>	nur schreiben

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

Der **StatusHandler** wird im Delayed Mode statt des ApplyHandlers gerufen wenn der Nutzer die Auswahl innerhalb der Liste ändert. Der ApplyHandler hingegen wird erst auf Anforderung gerufen (siehe Kapitel 3.4.2).

Die Instance-Variable **modified** kann einen Wert ungleich Null enthalten, nämlich dann wenn das Objekt vom User modifiziert wurde, der ApplyHandler aber noch nicht gerufen wurde. Der Aufruf des ApplyHandlers setzt auch im Delayed Mode den modified-Status zurück. Falls kein ApplyHandler gesetzt ist wird der modified-Status wird immer dann zurückgesetzt, wenn der ApplyHandler gerufen werden müsste.

Bei einer OptionGroup gilt außerdem:
- Ändert der Nutzer den Zustand eines Option-Objkets der OptionGroup, so wird die Instance-Variable **modified** mit dessen identifier logisch OR vernüpft.
- Der Parameter "modified" enthält folglich sowohl beim StatusHandler als auch beim ApplyHandler die logische OR-Verknüfung der Identifier der seit dem letzten Aufruf des ApplyHandler veränderten Option-Objekte.

Methode	Aufgabe
SendStatus	Status-Handler aufrufen

Syntax BASIC-Code: <obj>.**SendStatus**

Die Methode **SendStatus** fordert das Objekt auf, seinen StatusHandler aufzurufen (d.h. seine Status-Message zu senden).

4.9 View und Content

4.9.1 Überblick

Die View Objektklasse stellt ein "Fenster" bereit, in dem beliebige grafische Daten - einschließlich Texte - dargestellt werden können. Die Inhalte dieses Fensters, d.h. die grafischen Daten, werden vom "Content"-Objekt (content: engl. Inhalt) bereitgestellt. Dieses Objekt muss nur die Daten darstellen können.

Alles andere, wie Scrolling, Zoom oder Clipping (engl. to clip: beschneiden) macht das View. Das View fordert bei Bedarf das Content-Objekt auf, sich selbst darzustellen, aber welcher Teil der Darstellung auf dem Bildschirm erscheint (Clipping), ob er vergrößert oder verkleinert ist usw., darum kümmert sich das View.

Während das View ein GenericClass Objekt ist, ist das Content ein VisualClass Objekt. Da beide jedoch zusammengehören werden sie hier auch gemeinsam besprochen.
Es gibt zwei Objektklassen, die als Content für ein View dienen können: Die Klasse "Content" selbst, die beliebige Daten darstellen kann und die Klasse "BitmapContent", die eine editierbare Bitmap bereitstellt.

XXX ViewControl : probabely in next release XXXX

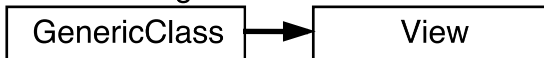
Die Kombination der View/Content ist sehr universell und für praktisch alle Anwendungsgebiete geeignet. Das Ausnutzen dieser Möglichkeiten erfordert daher eine gewisse Einarbeitung in die Eigenschaften der beiden Objektklassen und ihrem Zusammenspiel, wobei man um gelegentliches Experimentieren nicht herumkommt. Die Verwendung der Klasse BitmapContent stellt bereits einen Spezialfall dar, der in der Grundkonfiguration sehr einfach zu handhaben ist.
Es gibt in R-BASIC weitere, spezialisierte und damit noch einfacher zu handhabende Möglichkeiten, Grafiken darzustellen. Einen Überblick über diese Möglichkeiten, die ohne die Verwendung eines View-Objekts auskommen, und Verweise zu den entsprechenden Abschnitten in diesem Handbuch finden Sie im Kapitel 2.2.

4.9.2 Das View

Das View-Objekt ist die Schnittstelle zwischen ihrer Programm-UI und dem darzustellenden grafischen Inhalt. Es arbeitet in vielen Situationen automatisch mit seinem Content-Objekt zusammen und sorgt bei Bedarf für das Scrolling, Clipping und den Zoom.

XXX In der aktuellen Version: nur wenige Grundfunktionen implementiert XXX

Abstammung:



Spezielle Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
hControl	hControl = numWert	lesen, schreiben
vControl	vControl = numWert	lesen, schreiben
Content	Content = <obj>	lesen, schreiben
contentSize	contentSize = sizeX, sizeY	lesen, schreiben

Spezielle Action-Handler: keine

4.9.2.1 View Geometrie

hControl, vControl

Die Felder **hControl** und **vControl** legen fest, wie sich das View in horizontraler (hControl) oder vertikaler (vControl) Richtung darstellt. Zulässige Werte sind Kombinationen der HVC_-Konstanten (HVC: horzontal-vertical-control). Die Werte sind so gewählt, dass jede Konstant genau ein Bit gesetzt hat. Mehrere Konstanen können mit + oder OR verknüpft werden, die Abfrage, ob ein bestimmter Wert gesetzt ist kann mit der logischen AND Funktion erfolgen.

Syntax UI-Code: **hControl** = numWert
vControl = numWert
numWert ist eine Kombination der HVC_-Konstanten
Lesen: <numVar> = <obj>.**hControl**
<numVar> = <obj>.**vControl**
Schreiben: <obj>.**hControl** = numWert
<obj>.**vControl** = numWert
numWert ist eine Kombination der HVC_-Konstanten

Folgende Konstanten stehen zur Verfügung. Hier nicht aufgeführte Werte (z.B. 2) sollten auch nicht verwendet werden, da ihre Wirkung unbestimmt ist.
Auch die Kombination widersprüchlicher Werte (z.B. HVC_SCROLLABLE + HVC_NO_SCROLLBAR) kann seltsame Folgen haben.

Konstante	Wert dezimal	Wert hex	gesetztes Bit
HVC_SCROLLABLE	128	&H80	7
HVC_TAIL_ORIENTED	32	&H20	5
HVC_NO_SCROLLBAR	16	&H10	4
HVC_NO_LARGER_THAN_CONTENT	8	&H08	3
HVC_NO_SMALLER_THAN_CONTENT	4	&H04	2
HVC_KEEP_ASPECT_RATIO	1	&H01	0

HVC_SCROLLABLE
: Das View soll in diese Dimension scrollbar sein. Die Scrolleisten werden auch gezeigt, wenn es eigentlich nicht erforderlich ist.

HVC_TAIL_ORIENTED
: Bestimmt, dass das View das untere/rechte Ende des Content-Bereichs weiterhin darstellen soll, wenn dieser Bereich bereits dargestellt wird und sich die Größe des Content-Objekts ändert.

HVC_NO_SCROLLBAR
: Das View soll keine Scrollleisten in der entsprechenden Dimension anzeigen, auch wenn es scrollbar ist.

HVC_NO_LARGER_THAN_CONTENT
: Das View soll sich in der gegeben Dimension nicht größer als das Content machen, wobei der Wert, der in der **contentSize** Instance-Variable steht, massgebend ist. Per Default gibt es keine Restriktionen bezüglich der View-Größe.

HVC_NO_SMALLER_THAN_CONTENT
: Das View soll sich in der gegeben Dimension nicht kleiner als das Content machen, wobei der Wert, der in der **contentSize** Instance-Variable steht, massgebend ist. Per Default gibt es keine Restriktionen bezüglich der View-Größe.

HVC_KEEP_ASPECT_RATIO
: Das View soll das Seitenverhältnis in der Darstellung beibehalten, indem es seine Größe in der gegebenen Dimension basierend auf der Größe der anderen Dimension berechnet.

Beispiel UI-Code:

```
View  MyView
    hControl = HVC_NO_LARGER_THAN_CONTENT +
           HVC_NO_SMALLER_THAN_CONTENT
    vControl = HVC_SCROLLABLE
    Content = MyBitmapContent
END Object
```

Beispiel: Abfrage mit AND, ob das View scrollbar ist

```
IF   MyView.vControl  AND  HVC_SCROLLABLE  THEN ...
```

Da ein View ein generisches Objekt ist können alle Geomtrie-Hints aus dem Kapitel 3.3, insbesondere aus dem Kaptiel 3.3.4 (Objektgröße) mit einem View verwednet werden. Sie sind der Einfachheit halber hier noch einmal aufgeführt. Beachten Sie, dass die ~Size-Hints mit den Werten für hControl bzw. vControl in Konflikt geraten können.

GenericClass Hint	UI-Code Syntax	Im BASIC Code
ExpandWidth	ExpandWidth	—
ExpandHeight	ExpandHeight	—
initialSize	initialSize = x, y [, count]	lesen, schreiben
minimumSize	minimumSize = x, y [, count]	lesen, schreiben
maximumSize	maximumSize = x, y [, count]	lesen, schreiben
fixedSize	fixedSize = x, y [, count]	lesen, schreiben
xSize	—	nur lesen
ySize	—	nur lesen

4.9.2.2 Verbindung zum Content

Content

DIe Instance-Variable **Content** (engl.: Inhalt) enthält das aktuelle Content-Objekt des View's. Es kann im BASIC-Code gelesen und geschrieben werden. Wird dem View ein neues Content-Objekt zugeordnet stellt sich das View automatisch neu dar, so dass der neue Content sichtbar wird. Dabei kommuniziert das View mit dem neuen Content und kann, je nach dem, welche Werte in hControl bzw. vControl (XXX später auch viewAttrs XXX), gegebenfalls seine Größe neu bestimmen.

Ein eventuell vorher mit dem View verbundenes Content-Objekt wird dabei automatisch abgekoppelt. Die Zuweisung eines "leeren" Content-Objekts mit der Funktion **NullObj()** ist zulässig.

Syntax UI-Code:	**Content** = <obj>
	<obj> muss namentlich aufgeführt werden.
	Variablen sind im UI-Code nicht zulässig.
Lesen:	<objVar> = <obj>.**Content**
Schreiben:	<obj>.**Content** = <obj2>

Beachten Sie, dass die Zuweisung eines neuen Content-Objekts den aktuelle "Screen" nicht beeinflusst (siehe auch Kapitel 2.3.1). Wenn das alte Content-

Objekt der "Screen" war, bleibt er es auch. Alle Grafik- und Text-Ausgaben gehen weiterhin an dieses Objekt. Möglicherweise müssen Sie also zusätzlich das neue Content-Objekt auch der Screen-Variablen zuweisen.

Beispiel UI-Code:

```
View  MyView
    hControl = HVC_SCROLLABLE
    vControl = HVC_SCROLLABLE
    Content = MyBitmapContent
END Object
```

Beispiele BASIC-Code:

```
DIM ob as OBJECT

ob = MyView.Content
MyView.Content = MyOtherContent
....
IF  ob = MyBitmapContent   THEN
    MyView.Content = NullObj()
    END IF
```

contentSize

Die Instance-Variable **contentSize** speichert die x- und y-Ausdehung der vom Content-Objekt des View darzustellenden grafischen Daten. Gemeinsam mit seiner eigenen Größe und einem eventuell eingestellten Zoom-Faktor kann das View dann z.B. entscheiden, ob es Scrollbalken verwenden muss und wie groß deren "innerer Balken" zu sein hat.

In vielen Fällen verwaltet das View die **contentSize** automatisch, indem es mit dem Content-Objekt kommuniziert. Bei Bedarf kann **contentSize** aber sowohl im UI-Code als auch im BASIC-Code geschrieben werden.

XXX Das kann nötig sein, wenn Sie die Größe des Content-Objekts "hinter dem Rücken" des View ändern oder (??) -- ggf Wizzard anpassen!
nächste Release: Attribut Manuelles verwalten gesetzt ??XXX

Syntax			
	UI-Code:	**contentSize** = xSize, ySize	
	Lesen:	<numVar> = <obj>.**contentSize** (0)	! x-Size
		<numVar> = <obj>.**contentSize** (1)	! y-Size
	Schreiben:	<obj>.**contentSize** = xSize, ySize	

Beispiel UI-Code:

```
View  MyView
    < .. andere Instance-Variablen hier .. >
    contentSize = 100, 200
    END Objekt
```

Beispiele BASIC-Code:

```
DIM breite  AS  Real

breite = MyView.contentSize(0)
Print "Alte Größe = "; breite ; " x "; MyView.contentSize(1);
    "Pixel"

MyView.contentSize = 64, 48
```

4.9.2.3 Children eines View-Objekts

Im Normalfall besitzt ein View keine Children. Möglich ist das jedoch, da ein View von der GenericClass abstammt. Um die Children innerhalb des View zu platzieren, können Sie **den Childen** die - ebenfalls von der GenericClass geerbten Hint **placeObject** geben. Dieser bestimmt, in welchem Bereich des View das Child platiert werden soll. Dabei stehen die folgenden Werte zur Verfügung:

Wert	Objekt wird plaziert ...
16	X-Scroller-Bereich
32	Y-Scroller Bereich
64	Linke Seite des View
128	Obere Seite des View
256	Rechte Seite des View
512	Untere Seite des View

Beispiel: Platziere eine Button auf de linkes Seite des View

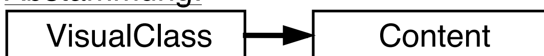
```
View      MyView
          Children = MyButton
          <... >
          END

Button    MyButton
          Caption$ = "Neu zeichnen"
          placeObject = 64
          < .. >
          END Object
```

4.9.3 Das Content

XXX aktuell noch kein Content-Objekt implementiert / immer drawen! XXX

Abstammung:

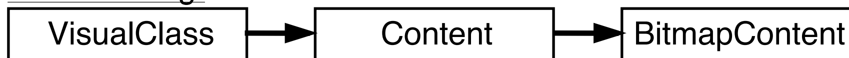


Da Content Objekte von der VisualClass abstammen, kommen Sie nicht in den GenericTree der Programms. Statt dessen werden die über die Instance-Variable "Content" eines View's mit dem View verbunden. Das View muss aber in den GenericTree des Programm eingebunden werden.

4.9.4 Das BitmapContent

Objekte der Klasse **BitmapContent** verwalten eine editierbare Bitmap. Bitmaps sind digitalisierte Bilder. Sie bestehen aus einer rechteckigen Anordnung von einzelnen Bildpunkten (Picture Element: Pixel). Jedem Pixel kann eine eigene Farbe zugeordnet werden. In die Bitmaps der Klasse **BitmapContent** kann Text oder Grafik geschrieben werden. Das BitmapContent-Objekt legt die zugehörige Bitmap automatisch selbst an, so dass sie sofort benutzt werden kann.
Einen kompletten Überblick über die Möglichkeiten von R-BASIC Grafik auszugeben, finden Sie im Kapitel 2.2.

Abstammung:



Da BitmapContent Objekte von der VisualClass abstammen, kommen Sie nicht in den GenericTree der Programms. Statt dessen werden die über die Instance-Variable "Content" eines View's mit dem View verbunden. Das View muss aber in den GenericTree des Programm eingebunden werden.

Beispiel UI-Code:
Das View "MyView" stellt eine 320x256 Pixel große True-Color Bitmap dar. Es kommuniziert automatisch mit dem BitmapContent "MyBitmap" um seine Größe auf 320x256 Pixel zu setzen, so das die ganze Bitmap sichtbar ist.
"DefaultScreen" stellt das BitmapContent als "Standard-Ausgabe-Objekt" für Grafik- und Textausgaben ein.

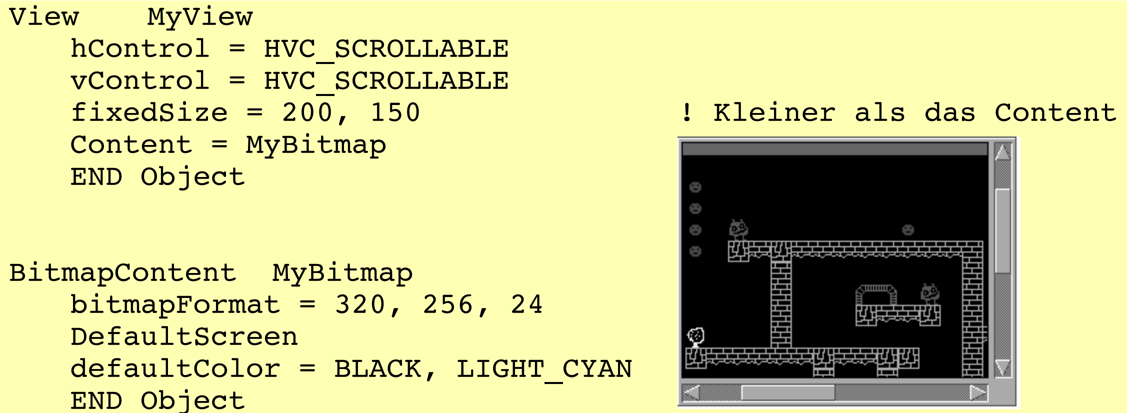
```
View    MyView
    vControl = HVC_NO_LARGER_THAN_CONTENT
                                                            +
    HVC_NO_SMALLER_THAN_CONTENT
    hControl = HVC_NO_LARGER_THAN_CONTENT
                                                            +
    HVC_NO_SMALLER_THAN_CONTENT
    Content = MyBitmap
    END Object

BitmapContent                               MyBitmap
    bitmapFormat = 320, 256, 24
    DefaultScreen
    defaultColor = BLACK, LIGHT_CYAN
    END Object
```

In vielen Fällen wird der im Code oben verwendete Fall (kein Scrolling der Bitmap, kein Zoom) aureichend sein. Ein BitmapContent ist jedoch ein vollwertiges Content-Objekt und kann daher z.B. auch in einem scrollbaren View dargestellt werden:

```
View     MyView
    hControl = HVC_SCROLLABLE
    vControl = HVC_SCROLLABLE
    fixedSize = 200, 150          ! Kleiner als das Content
    Content = MyBitmap
    END Object


BitmapContent  MyBitmap
    bitmapFormat = 320, 256, 24
    DefaultScreen
    defaultColor = BLACK, LIGHT_CYAN
    END Object
```



BitmapContent-Objekte sind auch dann voll nutzbar, wenn sie nicht mit einem View verbunden sind, d.h. sie können zum "Screen" gemacht (vgl. Kapitel XXXX) oder als "DefaultScreen" gesetzt werden. Natürlich werden sie dann nicht auf dem Bildschirm erscheinen. Grafik- und Textausgaben gehen dann "im Hintergrund" in die Bitmap und werden sichtbar, sobald das Objekt an ein View gekoppelt wird (z.B. mit MyView.Content = MyBitmapContent). Insbesondere ist es möglich zwischen zwei BitmapContent Objekten hin- und herzuschalten. Sie können die eine Bitmap im Hintergrund ändern, während die andere sichtbar ist - und dann die Veränderungen mit MyView.Content = .. auf "einen Schlag" sichbar machen.

Spezielle Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
bitmapFormat	bitmapFormat = x, y, n [, flags]	lesen, schreiben
defaultColor	defaultColor = fg, bg	lesen, schreiben
DefaultScreen	DefaultScreen	—

bitmapFormat

Die Instance-Variable **bitmapFormat** speichert die Größe, die Farbtiefe und weitere Eigenschaften der Bitmap. R-BASIC unterstützt die Farbtiefen 1 (schwarz/weiss), 8 (256 Farben) und 24 (True Color, 16 Mio Farben). Die Farbtiefe 4 (16 Farben) wird von R-BASIC nicht unterstützt. Verwenden Sie statt dessen 8 Bit Farbtiefe.

XXX Die Unterstüzung von Masken (transparenz) und Paletten wird erst in der nächsten Release implementiert XXXX

Syntax UI-Code: **bitmapFormat** = x, y, n [, flags]
x: Breite der Bitmap
y: Höhe der Bitmap
n: Farbtiefe (zulässige Werte: 1, 8, 24)
flags: XX noch nicht unterstützt. Nicht verwenden

Lesen: <numVar> = <obj>.**bitmapFormat** (0) ’ Breite
<numVar> = <obj>.**bitmapFormat** (1) ’ Höhe

	<numVar> = <obj>.**bitmapFormat** (2)	' Farbtiefe
	<numVar> = <obj>.**bitmapFormat** (3)	' flags
Schreiben:	<obj>.**bitmapFormat** = x, y, n [, flags]	

Beispiel UI-Code: siehe oben

Wenn Sie im BASIC-Code die Variable **bitmapFormat** belegen (schreiben), so wird die Bitmap neu angelegt. Alle vorhandenen Informationen (Grafik, Text..) gehen vorloren. Die Bitmap darf dabei weiterhin als Content einen Views gesetzt sein, muss es aber nicht.

Beispiele BASIC-Code:
Lesen der Werte:

```
DIM  b, h, f  as WORD
    b = MyBitmap.bitmapFomat (0)        ' Breite
    h = MyBitmap.bitmapFomat (1)        ' Höhe
    f = MyBitmap.bitmapFomat (2)        ' Farbtiefe

    Print "Bitmapgröße:" b; "x"; h; "Pixel, "; f; "Bit pro Pixel"
         ! z.B.    320 x 256 Pixel,  24 Bit pro Pixel
```

Neu anlegen der Bitmap: 800 x 600 Pixel, 256 Farben

```
    MyBitmap.bitmapFormat = 800, 600, 8
```

defaultColor

Die Instance-Variable **defaultColor** enthält die Farben, die beim Initialisieren der Bitmap (erstmaliges bzw. Neuanlegen der Bitmap) verwendet werden. Außerdem werden sie verwendet, wenn das Objekt zum "**Screen**" wird. Das tritt auf, wenn das Objekt die Anweisung DefaultScreen im UI-Code enthält oder wenn es der Systemvariablen **Screen** direkt zugewiesen wird (vergleiche Kapitel 2.3.1 "Die Screen-Variable").

BitmapContent-Objekte ohne die Anweisung **defaultColor** verwenden die Farben "schwarz auf weiß".

Syntax UI-Code:	**defaultColor** = fg, bg fg: Vordergrund (foreground) bg: Hintergrund (background) fg und bg müssen Indexfarben sein. RGB-Farben sind nicht zulässig.	
Lesen:	<numVar> = <obj>.**defaultColor** (0)	' fg
	<numVar> = <obj>.**defaultColor** (1)	' bg
Schreiben:	<obj>.**defaultColor** = fg, bg	

Beim Anlegen der Bitmap löscht R-BASIC die Bitmap in der Hintergrundfarbe **bg**. Wird das zugehörige BitmapContent-Objekt zum Screen setzt R-BASIC die

Farben folgendermaßen:
  Hintergrundfarbe: bg
  Text-, Linien- und Flächenfarbe: fg
Das ist prinzipiell so, als würde automatisch die Anweisung "COLOR fg, bg" ausgeführt.

DefaultScreen

Diese Anweisung im UI-Code bewirkt, dass das entsprechende BitmapContent als "Standard-Ausgabe-Objekt" festgelegt wird. Es wird dazu automatisch in der Systemvariablen **Screen** gespeichert (vergleiche Kapitel 2.3.1 "Die Screen-Variable"). Alle Grafik- oder Textausgaben (XXX die nicht in einem Draw-Handler erfolgen XXX) gehen damit automatisch auf dieses Objekt.

Syntax UI-Code: **DefaultScreen**

Beispiel UI-Code:

```
BitmapContent  MyBitmap
    bitmapFormat = 320, 256, 24
    DefaultScreen
    END Object
```

Hinweis für Profis: Es ist zwar meist sinnvoll, aber nicht zwingend erforderlich, dass das Content-Objekt, dass als "DefaultScreen" gesetzt ist, mit einen View verbunden ist. Grafik- und Textausgaben gehen dann zwar in die Bitmap werden aber erst sichtbar, sobald das Objekt an ein View gekoppelt wird (z.B. mit MyView.Content = MyBitmapContent). Damit kann man z.B. komplexe Ausgabe-Operationen "im Hintergrund" abwickeln.

4.10 Text-Objekte

4.10.1 Überblick

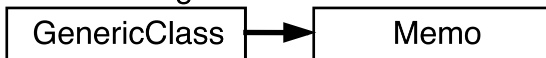
Die R-BASIC Text-Objekte erlauben das einfache Eingeben von Text, ohne dass Sie als Programmierer sich um irgendwelche Details kümmern müssen. Die Text-Objekte behandeln Tastatur- und Mausereignisse selbständig und erzeugen bei Bedarf eine vertikale Scroll-Leiste. Sie registrieren, ob der Text vom Nutzer verändert wurde und können bei Bedarf Messages aussenden, um den Rest des Programms über bestimmten Ereignissen zu informieren.

In R-BASIC stehen zwei Text-Objekt Klassen zur Verfügung. Die Klasse **Memo** stellt einen einfachen Text-Editor bereit. Sie unterstützt einen automatischen Zeilenumbruch und die Enter-Taste beginnt einen neuen Absatz. Die Klasse **InputLine** dagegen ist für die Eingabe einzeiliger Texte, z.B. von Dateinamen, gedacht. Die Enter-Taste löst bei InputLine-Objekten den Apply-Handler des Objekts aus. Ansonsten unterscheiden sich die Instance-Variablen, Action-Handler usw. der beide Text-Objekt-Klassen nicht.

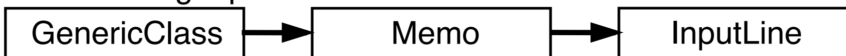
Weder das Memo noch das InputLine-Objekt unterstützt die Formatierung einzelner Buchstaben, Worte oder Absätze. Alle Format-Informationen (z.B. Font, Textgröße, Textstil, Ausrichtung usw.) gelten immer für den gesamten Text des Objekts.
XXX In der aktuelle Version sind nur die Grundfunktionen der Text-Objekte implementiert. Text-Formatierungen und weitere Funktionen werden in der nächsten Release folgen.

Die folgenden Ausführungen gehen zunächst grundsätzlich davon aus, dass die Text-Objekte im normalen Modus (nicht im sogenannten "Delayed Mode") arbeiten. Das ist der Normalfall wenn man nicht spezielle Hints setzt, um in den Delayed Mode zu kommen. Dieser "Delayed Mode" ist ausführlich im Kapitel 3.4.2 (Delayed Mode und Status-Message) dieses Handbuchs beschrieben.

Abstammung Memo:



Abstammung InputLine:



Spezielle Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
text$	text$ = "text"	lesen, schreiben
maxLen	maxLen = numWert	lesen, schreiben
textLen	—	nur lesen
modified	modified = numWert	lesen, schreiben
ApplyHandler	ApplyHandler = <Handler>	nur schreiben

StatusHandler	StatusHandler = <Handler>	nur schreiben
OnModified	OnModified = <Handler>	nur schreiben

Methoden:

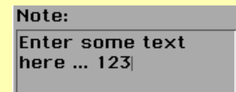
Methode	Aufgabe
SendStatus	Status-Handler aufrufen

Action-Handler-Typen:

Handler-Typ	Parameter
TextAction	(sender as object, isModified as integer, textLen as word)

Beispiel: Ein typisches Memo-Text-Objekt

```
Memo memo1
    Caption$ = "Note:"
    justifyCaption = J_TOP
    text$ ="Enter some text here ..."
    maxlen = 100
    fixedSize = 30 + ST_AVG_CHAR_WIDTH, 5 + ST_TEXT_LINES
    END Object
```



Beispiel: Ein typisches InputLine-Text-Objekt

```
InputLine NameText
    Caption$ = "Name:"
    text$ ="Setag, Llib"
    maxLen = 100
    ExpandWidth
    ApplyHandler = ApplyNameText
    END Object
```

Da Text-Objekte von der GenericClass abstammen erben sie alle Eigenschaften , Hints und Fähigkeiten dieser Klasse. Für Text-Objekte sind besonders die Fähigkeiten zum Geometrie-Management (Kapitel 3.3) von Bedeutung.

4.10.2 Der Text

text$	text$ = "text"	lesen, schreiben
maxLen	maxLen = numWert	lesen, schreiben
textLen	—	nur lesen

text$

Die Instance-Variable **text$** enthält den eigentlichen Text des Objekts. Sie kann

gelesen und geschrieben werden. Das Text-Objekt stellt den neuen Text automatisch dar, wenn sie der Instance-Variable **text$** einen Wert zuweisen.

Syntax	UI-Code:	**text$** = "text"
	Lesen:	<stringVar> = <obj>.**text$**
	Schreiben:	<obj>.**text$** = "text"

maxLen

Die Instance-Variable **maxLen** enthält die maximale Länge des Textes, den das Objekt verwalten kann. Der Default-Wert liegt bei 1024, das ist die maximale Größe, die eine String-Variable in R-BASIC speichern kann. Erlaubt sind Werte von 1 bis 4096.

Syntax	UI-Code:	**maxLen** = numWert
	Lesen:	<numVar> = <obj>.**maxLen**
	Schreiben:	<obj>.**maxLen** = numWert

- Setzen Sie **maxLen** auf eine Wert, der kleiner als die aktuelle Textlänge ist, so wird der Text abgeschnitten.
- Der Nutzer kann nicht mehr Text eingeben, als durch **maxLen** festgelegt ist.
- Es ist immer eine gute Idee **maxLen** so klein wie nur möglich zu wählen. Beispielsweise ist zur Eingabe von GEOS-Dateinamen ein Wert von 32 für **maxLen** vernünftig, da GEOS-Dateinamen nicht länger als 32 Zeichen werden können. Der Text wird immer gemeinsam mit dem Text-Objekt in den Speicher geladen wird. Bei mehreren Text-Objekten oder großen Texten kann es sonst passieren, dass die Meldung "Hauptspeicher voll" kommt und GEOS crasht. Der Speichermanager von GEOS ist für Speicherblöcke um 8 kByte optimiert.
  XXX Aufteilung in Speicherblöcke (Ressourcen) noch nicht implementiert.
- XXX Aktuell noch keine Funktionen zum Lesen von Text-Abschnitten implementiert. Sie sollten daher **maxLen** nicht größer als 1024 wählen (obwohl 4096 zulässig sind), sonst kann R-BASIC den Text möglicherweise nicht mehr lesen (Laufzeitfehler "String zu lang").

textLen

Die Instance-Variable **textLen** enthält die Länge des aktuellen Textes.

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

4.10.3 Text-Objekt Actions

modified	modified = numWert	lesen, schreiben
ApplyHandler	ApplyHandler = <Handler>	nur schreiben

OnModified	OnModified = <Handler>	nur schreiben

TextAction	(sender as object, isModified as integer, textLen as word)

Anmerkungen zu TextAction

Der Parameter **isModified** ist eine Kopie der modified-Instance-Variable des Objekts. In den meisten Fällen ist er TRUE, weil die Handler üblicherweise nur aufgerufen werden, wenn das Objekt durch den Nutzer verändert wurde.
Der Parameter **textLen** enthält einfach die aktuelle Textlänge.

modified

Die Instance-Variable **modified** enthält die Information, ob der Text seit dem letzten Aussenden der Apply-Message (Aufruf des Apply-Handlers) vom Nutzer modifiziert wurde (**modified**=TRUE) oder nicht (**modified**=FALSE). Text-Objekte eines neu gestarteten Programms sind zunächst ebenfalls "nicht modified".

Syntax	UI-Code:	**modified** = TRUE I FALSE
	Lesen:	<numVar> = <obj>.**modified**
	Schreiben:	<obj>.**modified** = TRUE I FALSE

Beachten Sie, dass ein Verändern des Textes vom BASIC-Code aus (z.B. Belegen der Instance-Variable **text$**), den Text nicht als "modified" markiert, d.h. der Wert der Instance-Variablen **modified** wird nicht verändert. Sie können dies bei Bedarf selbst machen, indem Sie die Anweisung "<obj>.modified = TRUE" verwenden.
Das Aussenden der Apply-Message setzt den Modified-Status zurück (**modified** = FALSE).

ApplyHandler

Die Instance-Variable **ApplyHandler** enthält den Namen des Action-Handlers, der aufgerufen wird, wenn der Text sein Änderungen anwenden will (engl. to apply: Anwenden). **Apply-Handler** müssen als **TextAction** deklariert sein.
Beachten Sie: Der Apply-Handler wird nur aufgerufen, wenn der Text "modified", d.h. vom Nutzer verändert ist. Dies passiert automatisch, wenn der Nutzer den Text ändert, Sie können es aber auch vom BASIC Code aus machen, indem Sie die Instance-Variable modified mit TRUE belegen.

- **InputLine**-Objekte haben im Allgemeinen einen Apply-Handler. Er wird aufgerufen, wenn der Nutzer nach Eingabe eines Textes im Eingabefeld auf die Enter-Taste drückt.

- **Memo**-Objekte haben häufig keinen Apply-Handler.

- Sie können ein Text-Objekt (Memo und InputLine) veranlassen, seinen Apply-Handler aufzurufen, indem Sie die von der GenericClass geerbte Methode **Apply** verwenden. Das Objekt muss aber als "modified" markiert (siehe oben). Alternativ könnte man dem Objekt auch den Hint *ApplyEvenIfNotModifie*d geben. Eine ausführliche Beschreibung dazu finden Sie im Kapitel 3.4 (Die "Apply"-Message) dieses Handbuchs.

```
SUB ForceApply ( obj as OBJECT )
    obj.modified = TRUE                  ' zu Sicheheit!
    obj.Apply
    END SUB
```

OnModified

Gelegentlich benötigt man eine Information, wenn ein Text-Objekt durch eine Nutzereingabe vom "nicht modified" in den "modified" Zustand übergeht. Die Instance-Variable **OnModified** enthält den Namen des Action-Handlers, der aufgerufen wird, wenn das Text-Objekt erstmalig nach Aussenden der letzten Apply-Message vom Nutzer verändert wird. **OnModified**-Handler müssen als **TextAction** deklariert sein.
Das Aussenden der Apply-Message setzt den "modified" Zustand des Text-Objekt zurück. Gibt der Nutzer nun Text ein, so wird der **OnModified**-Handler aufgerufen. Das heißt im Umkehrschluss, dass der **OnModified**-Handler in folgenden Fällen nicht gerufen wird:
- Setzen der **modified** Instance-Variable vom BASIC Code aus.
- Verändern des Textes vom BASIC-Code aus (z.B. Belegen der Instance-Variable **text$**).
- Die Instance-Variable modified steht bereits auf TRUE, z.B. weil sie vom BASIC-Code aus gesetzt wurde, bevor der Nutzer etwas eingegeben hat.

Beispiel: Text-Objekt mit Apply- und OnModified-Handler. Der Nutzer soll einen Dateinamen eingeben und ihn durch drücken der Enter-Taste im Text-Objekt "anwenden" können. Oder er soll einen extra Button dazu verwenden. Dieser soll aber erst aktiv sein, nachdem der Nutzer etwas eingegeben hat. Die SUB "DoSaveFile" muss natürlich irgendwo definiert sein und wird hier nicht mit aufgeführt.
UI-Code

```
Button  SaveFileButton
    Caption$ = "Save File"
    ActionHandler = ButtonSaveFile
    enabled = FALSE                      ' Zunächst inaktiv
    END Object

InputLine FileNameText
    maxLen = 32                          ' max. 32 Zeichen sinnvoll
    ApplyHandler = TextSaveFile
    OnModified = TextIsModified
    END Object
```

BASIC-Code

```
ButtonAction ButtonSaveFile
    DoSaveFile                              ' macht die Arbeit
    END Action

TextAction TextSaveFile
    DoSaveFile                              ' macht die Arbeit
    END Action

TextAction TextIsModified
    SaveFileButton.enabled = TRUE           ' Button freischalten
    END Action
```

4.10.4 Text-Formatierung

XXX noch nicht implementiert

XXX Doku-Konzept insgesamt druchdenken

4.10.5 Text-Objekte im Delayed Mode

Alle Text-Objekte können im "Delayed Mode" (engl.: verzögerter Modus) arbeiten. Dazu muss man dem Objekt selbst bzw. einem seiner Parents im UI-Code den Hint **MakeDelayedApply** geben oder man bindet das Objekt als Child in einem Dialog ein, dessen **dialogType** Instance Variable auf DT_DELAYED_APPLY gesetzt ist. Dieser "Delayed Mode" ist ausführlich im Kapitel 3.4.2 (Delayed Mode und Status-Message) dieses Handbuchs beschrieben, eine Beschreibung des Dialog-Objekts im Delayed Mode finden Sie im Kapitel 4.6.6.5.

Instance Variable	Syntax im UI-Code	Im BASIC-Code
StatusHandler	StatusHandler = <Handler>	nur schreiben

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

Der **StatusHandler** wird im Delayed Mode statt des ApplyHandlers gerufen wenn der Nutzer z.B. nach Eingabe eines Textes in einem InputLine-Objekt auf die Enter-Taste drückt. Der ApplyHandler hingegen wird erst auf Anforderung gerufen (siehe Kapitel 3.4.2).

Die Instance-Variable **modified** kann einen Wert ungleich Null enthalten, nämlich dann wenn der Text vom User modifiziert wurde, der ApplyHandler aber noch nicht gerufen wurde. Der Aufruf des ApplyHandlers setzt auch im Delayed Mode

den modified-Status zurück.

Methode	Aufgabe
SendStatus	Status-Handler aufrufen

Syntax BASIC-Code: <obj>.**SendStatus**

Die Methode **SendStatus** fordert das Objekt auf, seinen StatusHandler aufzurufen (d.h. seine Status-Message zu senden).