7.3.1 Klasse List (gb.data)

Die Klasse *List* implementiert eine zirkuläre, doppelt-verkettete Liste von Elementen. Die Klassifikation als "doppelt-verkettet" bedeutet, dass jedes Element der Liste mit einem vorhergehenden und einem nächsten Element der Liste verbunden ist. Bei einer "zirkulären" Liste ist der Nachfolger des letzten wieder das erste Element und der Vorgänger des ersten Elements ist das letzte Element.

Man kann diese Art von Listen also ringförmig sowohl vorwärts als auch rückwärts durchlaufen. Die Elemente in einer verketteten Liste sind vom Typ Variant. Das ermöglicht es Ihnen in einer verketteten Liste alles zu speichern – von Integer über Float und String bis hin zu Objekten.

7.3.1.1 Eigenschaften

Die Eigenschaften einer verketteten Liste sind von der Anzahl her überschaubar und werden in der folgenden Tabelle angegeben:

Eigenschaft	Datentyp	Beschreibung
.Backwards	.List.Backwards (ReadOnly)	Virtuelles Objekt zur Enumeration rückwärts
.Count	Integer (ReadOnly)	Anzahl der Elemente der Liste
.Current	Variant	Wert des aktuellen Listenelements

Tabelle 7.3.1.1.1: Eigenschaften der Klasse List

- Über eine Liste kann zum Beispiel mit "For Each" iteriert werden. Dabei werden alle Werte vom ersten bis zum letzten durchlaufen. Der Wert von *.Current* verändert sich dabei nicht.
- Mit "For Each vElement In hList" zum Beispiel durchläuft die Variant-Variable "vElement" alle Elemente der Liste "hList" nacheinander vom ersten bis zum letzten.
- Die Eigenschaft *List.Backwards* kann benutzt werden, um die Operation von "For Each" umzukehren: "For Each vElement In hList.Backwards" resultiert in der umgekehrten Reihenfolge von Werten, die "vElement" annimmt.

7.3.1.2 Methoden eines List-Objektes

Eine verkettete Liste besitzt immer ein virtuelles root-Element, das keinen Wert hat. Bei der Instantiierung eines List-Objektes ist nur das root-Element vorhanden. Es selbst ist sein eigener Vorgänger und Nachfolger. In diesem Fall wird die Liste auch als "leer" bezeichnet.

Das root-Element liegt logisch immer zwischen dem ersten und dem letzten Element. Sie haben jedoch keinen Zugriff auf dieses root-Element. Es wird bei allen Operationen übergangen, da es keinen Wert trägt, aber das root-Element hilft bei der Erklärung einiger Methoden.

Methode	Beschreibung
Clear()	Entfernen aller Elemente
Append(aElement)	aElement an den Anfang anfügen; hinter das root-Element
Prepend(pElement)	pElement an das Ende anfügen; vor das root-Element
Take(Optional iIndex)	Zurückgeben und Entfernen des aktuellen Elements. Es kann optional ein Index übergeben werden. Ist dies der Fall, wird das Element mit dem angegebenen Index relativ zum root-Element zurückgegeben und entfernt. Negative Indizes sind erlaubt. Sie laufen rückwärts vom root-Element.
MoveFirst()	Das aktuelle Element auf das erste der Liste zeigen lassen
MoveLast()	Das aktuelle Element auf das letzte der Liste zeigen lassen
MoveNext()	Das aktuelle Element auf seinen Nachfolger zeigen lassen
MovePrev()	Synonym für MovePrevious()
MovePrevious()	Das aktuelle Element auf seinen Vorgänger zeigen lassen
FindFirst(vValue)	Das aktuelle Element auf den ersten Fund eines bestimmten Wertes in der Liste zeigen lassen

FindLast(vValue)	Das aktuelle Element auf den letzten Fund eines Wertes zeigen lassen
FindNext(vValue)	Beginnend beim Nachfolger des aktuellen Elements einen Wert vorwärts suchen. Diese Methode kann über das Ende der Liste wieder zum Anfang umschlagen und dort weitersuchen. Nachdem jedes Element einmal untersucht wurde, bricht die Methode ab.
FindPrev()	Synonym für FindPrevious(vValue)
FindPrevious(vValue)	Beginnend beim Vorgänger des aktuellen Elements einen Wert rückwärts suchen. Diese Methode kann über den Anfang der Liste wieder zum Ende umschlagen.

Tabelle 7.3.1.2.1: Methoden der Klasse List

- Wie viele Klassen in Gambas besitzt auch List ein aktuelles Element – Current. Die Eigenschaft List.Current gibt seinen Wert zurück oder setzt ihn. Mithilfe der Move*()-Methoden kann man ein anderes Element zum aktuellen Element bestimmen.
- Bei der Instanziierung eines List-Objektes ist das aktuelle Element in einem undefinierten Zustand, denn es existiert noch kein (echtes) Element in der Liste. Es wird als "invalid" bezeichnet. Sollte der Wert eines invaliden Elements abgefragt werden, wird Null zurückgegeben. Ein Fehler wird ausgelöst, wenn ein invalides Element entfernt werden soll.
- Das aktuelle Element wird *invalid*, wenn eine der Find*()-Methoden den zu suchenden Wert nicht findet.

7.3.1.3 Der Unterschied zwischen List und Array (Variant[])

Man beachte, dass die Find*()-Methoden nicht gut mit den CPU-Datencaches zusammenarbeiten. Die Listenelemente werden separat, jedes für sich, im Speicher des Rechners alloziert und dann lose miteinander verbunden. Moderne Rechner profitieren sehr stark von ihren schnellen Cache-Zwischenspeichern, in denen u.a. nach der Heuristik, der *Spatialität* von Zugriffen, Werte gespeichert werden, um auf diese später besonders schnell zugreifen zu können. Der Rechner nimmt also an, wenn ein Programm auf einen Speicherblock zugreift, dass es kurze Zeit später auf einen weiteren Speicherblock ganz in der Nähe zugreifen wird. Bei Listen ist das aber nicht der Fall, da ihre Elemente beliebig im gesamten Hauptspeicher verteilt sein können. Die Iteration über viele Listenelemente (wie beim Suchen von Werten in der Liste) macht damit sämtliche Daten in den CPU-Datencaches unbrauchbar und das wiederum verlangsamt die Ausführung des gesamten Programms.

Es besteht in diesem Zusammenhang auch die Möglichkeit, auf das i-te Element in einer Liste mit "hList[i]" zuzugreifen. Das ist aber eine sehr zeitintensive Operation, da hierfür alle Elemente vom ersten bis zum i-ten Element durchlaufen werden müssen. Ähnlich wie bei List.Take() kann der Index "i" negativ sein, um Elemente vom Ende der Liste rückwärts abzuzählen.

Eine Liste ist nicht für beliebigen Zugriff auf die Elemente gedacht, wie man es von einem Array gewohnt ist, sondern für sequenzielle Zugriffsmuster (MoveNext() und MovePrev()). Eine Liste kann sehr viel effizienter als ein Array sein, wenn man sie nicht zu oft zum Finden von Werten oder zum Durchwandern aller Elemente gebraucht. Die separate Speicherung der Elemente der Liste hat nämlich den Vorteil, dass ein sehr geringer und vor allem konstanter Aufwand betrieben werden muss, wenn ein Listenelement hinzugefügt oder entfernt werden soll. Wird bei einem Array hingegen ein Element hinzugefügt oder entfernt, kann es sein, dass das gesamte Array an einen anderen Ort kopiert werden muss, was bei großen Arrays sehr aufwändig ist.

Ob eine Liste oder ein Array benutzt wird, hängt vom beabsichtigten Zugriffsmuster ab und muss im Vorfeld sorgfältig überlegt werden. Im Zweifelsfall sollte man immer zu einem Array greifen. Bei allen durchgeführten Stress-Tests war die Variant[]-Klasse der List-Klasse stets überlegen.