Das Basic Buch

Handbuch zum Entwickeln von Kbasic Anwendungen

Erste Ausgabe

Diese Ausgabe bezieht sich auf KBasic Version 1.6. Stellen Sie sicher, dass Sie die aktuelle Version von KBasic benutzen (Januar 2008).

Über dieses Buch

In diesem Buch finden Sie alle notwendigen Informationen, um erfolgreich in KBasic zu programmieren. Wenn Sie das Buch durchgearbeitet haben, werden Sie in der Lage sein, schnell einfache oder komplexe KBasic Programme zu schreiben, ob mit GUI-Features oder einfach nur Standard-BASIC, ob mit oder ohne moderner grafischer Benutzeroberfläche (z. B. mit Formularen). Für viele Probleme des Programmieralltags stellt es Lösungshilfen bereit. Wenn Sie gerade beginnen, sich mit KBasic zu beschäftigen, werden Sie sich vielleicht mit diesem Buch an einen ruhigen Ort zurückziehen und durch das Lesen der ersten Kapitel mehr über die Sprache erfahren wollen. Später dann, in einem freien Moment, werden sie wahrscheinlich die Beispiele dieses Buches ausprobieren.

Der Hauptzweck dieses Buches ist es jedoch, aufgeschlagen neben der Tastatur zu liegen, um Ihnen als schnelles und praktisches Nachschlagewerk während des Programmierens zu dienen. Der weniger erfahrene KBasic-Programmierer erhält eine umfangreiche Einführung in die Programmiersprache KBasic. Durch zahlreiche Beispiele für die praktische Verwendung der wichtigsten Sprachelemente wird ihnen die ereignis- und objektorientierte Arbeitsweise von KBasic nähergebracht.

Einen Referenzteil finden sie in der KBasic-Entwicklungsumgebung selbst. Deshalb wurde auf eine ausführliche Referenz in diesem Buch verzichtet. In der KBasic-Entwicklungsumgebung finden Sie eine komplette, detaillierte Auflistung aller Objekte, ihrer Ereignisprozeduren, Methoden und Eigenschaften sowie eine vollständige Referenz aller Operatoren und Befehle. Neben präzisen Syntaxdefinitionen finden Sie dort jeweils eine ausführliche Erläuterung der verfügbaren Parameter.

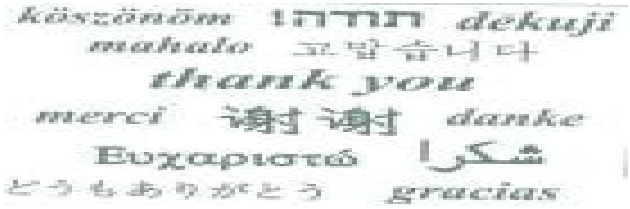
Warum Sie dieses Buch lesen sollten

Dieses Buch ist interessant für Leute mit einigem BASIC-Hintergrundwissen, aber auch absolute Programmier-Anfänger werden sich leicht zurechtfinden. Sollten Sie erfahrener Visual Basic-Entwickler sein, werden Sie sich sehr schnell heimisch fühlen. Dieses Buch ist für Leute gedacht, die

- einfach mal in die Programmierwelt hineinschnuppern wollen
- eine einfache und mächtige Programmiersprache für Linux, Mac und Windows lernen möchten
- erfahrene C/C++, Java oder Visual Basic-Entwickler, die einen Umstieg oder die Erweiterung ihrer Fähigkeiten anstreben
- gehört haben, dass es eine neue coole Programmiersprache für Linux/Mac und Windows gibt und die sich das nicht entgehen lassen wollen

Außerdem: Dieses Buch

- ist eines der wichtigsten Bücher, wenn Sie ernsthaft mit KBasic programmieren möchten
- gibt Ihnen ein solides Hintergrundwissen über die KBasic-Programmiersprache
- zeigt Ihnen Schritt für Schritt die wichtigsten Merkmale von KBasic



Danksagung

Ich bin allen Leuten, die an diesem Buch mitgearbeitet haben, sehr dankbar. Besonderer Dank geht an Nadja, meine Freundin und an jeden, der KBasic Professional gekauft oder mich sonst wie unterstützt hat und ein großer Dank geht an meine Eltern. Besonderen Dank geht auch an Herrn Lengfeld und Brigitte Hepp, die mir bei der Fehlersuche außerordentlich hilfreich waren. Verbleibende Fehler gehen natürlich auf eigenes Verschulden zurück.

Über den Autor

Bernd Noetscher ist Software-Entwickler und Hauptentwickler der KBasic-Programmiersprache. In seiner Freizeit geht er gerne Tanzen, liest viele Bücher und spielt Klavier, interessiert sich für Theater und Programmkino.

Seine private Internetseite ist www.berndnoetscher.de

Mithelfer

Ich danke allen Leuten, die dieses Buch Korrektur gelesen haben. Besonders erwähnen möchte ich hier an dieser Stelle Herrn Dr. Kaiser und Herrn Lengfeld. Sollten noch Fehler vorhanden sein, so bitte ich dies zu entschuldigen.

Geben Sie uns Feedback

Wir bitten Sie, uns dabei zu helfen, dieses Buch zu verbessern. Als Leser sind Sie der wichtigste Kommentator und Kritiker meines Buches. Wir schätzen Ihre Meinung und möchten wissen, was Ihnen gefällt oder was wir besser machen könnten. Wenn Sie einen Fehler in einem Beispielprogramm oder im Text gefunden haben, dann lassen Sie es uns wissen. Schreiben Sie eine Email an info@kbasic.com. Danke!

Wer aufgehört hat besser zu sein, hat aufgehört gut zu sein!

Nicht aufzuhören uns zu verbessern, bedeutet für uns, die Bedürfnisse unserer Kunden zu befriedigen, denn unser Hauptziel ist die Zufriedenheit unserer Kunden.

Um so ein hohes Ziel zu erreichen, brauchen wir Ihre Hilfe. Wenn wir irgendeinen Fehler gemacht haben, teilen Sie ihn uns bitte mit. Schreiben Sie eine Email an info@kbasic.com.

Neue Reihe von Handbüchern

KBasic wurde von KBasic Software entwickelt und hat im Internet und in der Geschäftswelt ein unglaubliches Interesse unter den Entwicklern geweckt. Wir haben uns verpflichtet, diese aufregende neue Technologie mit einer Reihe von Handbüchern zu dokumentieren. Diese Serie wird einleitende Bände, Sprach- und API-Referenzwerke und spezialisierte Texte über fortgeschrittene Themen der Programmierung in KBasic, wie Datenbanken und Netzwerke, umfassen.

Konventionen / Schreibweise in diesem Buch

Normaler Text erscheint in Arial Schrift. Hier ein Beispiel:
This is normal text

Syntax und Source code erscheint in Courier New und grauem Hintergrund: Hier das Beispiel:

```
Dim i As Integer
```

Wichtige Stichwörter werden kursiv hinterlegt:
Arguments

Quellen für weitere Informationen

Die Internetseite des Herstellers von KBasic: www.KBasic.com

Information rund um BASIC: Visual Basic 6 Internetarchive unter
www.Google.com

Inhaltsverzeichnis

Einführung

Sie haben vermutlich schon viel über KBasic gehört. Sie würden dieses Buch nicht lesen, wenn sie KBasic nicht für eine wichtige neue Technologie hielten. Sie haben sich jetzt schon vermutlich die Testversion von KBasic heruntergeladen und ein wenig mit den Beispielprogrammen herumgespielt. Nun reizt es Sie, Ihre eigenen KBasic-Programme zu schreiben...

Wollten Sie schon immer wissen, wie Computerprogramme funktionieren? Wollten Sie schon immer an Ihrer Tastatur sitzen und Bilder oder Spiele auf dem Computerbildschirm zaubern? Wenn ja, dann sind Sie vielleicht ein zukünftiger Softwareentwickler, der nur darauf wartet loszulegen.

Möglicherweise denken Sie, Programmieren ist eine komplizierte Angelegenheit. Jedesmal, wenn Sie ein einfaches Skript schreiben, bekommen Sie graue Haare. Wenn Sie so denken, wird Ihnen dieses Buch zeigen, dass das Programmieren Ihres Computers Spaß macht, aber vor allem nicht schwierig ist. Bevor Sie mit dem Programmieren loslegen, ist es sinnvoll, dass Sie ein grundlegendes Verständnis über alles haben. Sie haben wahrscheinlich schon eine Vorstellung, was ein Programm ist und wie es funktioniert. Klar, sonst würden Sie nicht dieses Buch anfangen zu lesen.

Nachdem Sie die Einführung durchgelesen haben, werden Sie entweder feststellen, dass Ihre Programmkenntnisse sehr gut sind, oder Sie werden erkennen, dass Sie soviel über Softwareentwickeln wissen, wie über Flugzeugbau. In beiden Fällen lohnt es sich mit KBasic zu beginnen...

Das große Geheimnis

Die Welt der Computerprogrammierung hat ein wohlgehütetes Geheimnis. Softwareentwickler reden nicht oft darüber. Wenn Sie seit langem schon Computer benutzt haben, werden Sie es kaum glauben können. Nichtsdestotrotz ist es so wahr wie der Himmel blau ist. Sie sind dabei, ein interessantes Geheimnis zu erfahren. Computer sind dumm. Genauer gesagt, ein Computer kann überhaupt nichts von alleine. Ohne Programmierer wären Computer nutzlos wie Autos ohne Benzin. Computer können nur das, was Ihnen zuvor befohlen wurde. Wenn Sie einen Augenblick darüber nachdenken, werden Sie herausfinden, dass das bedeutet, Computer können nur Aufgaben erledigen, die von Menschen lösbar und ausgedacht sind.

Warum sind also Computer trotzdem nützlich? Das Gute an Computern ist, dass Sie nicht intelligent sind, sondern viele Rechenoperationen in einen Bruchteil von Sekunden durchführen können. Software-Entwickler sind die Personen, die den Computern befehlen, was zu tun ist. Das gilt aber nicht für Leute, die vor dem Computer sitzen und ihn einfach benutzen. Wenn Sie z. B. einen Text schreiben, geben Sie dem Computer keine Befehle. Sie benutzen lediglich die Befehle, die dieses Computerprogramm kennt. Es ist das Computerprogramm, das dem Computer sagt, was er tun soll. Sie benutzen vielleicht Ihren Computer für Anwendungen, wie z. B. Computerspiele, Tabellenkalkulation oder Zeichnen. In allen Fällen arbeiten Sie mit einem Computerprogramm, welches die Befehle für den Computer bereitstellt.

Als Computernutzer sind Sie an der Spitze der Befehlskette eines Computers. Wenn Sie Ihrem Computer direkt Befehle geben möchten, müssen Sie programmieren lernen.

Warum sollten Sie Programmieren erlernen?

Es gibt viele Gründe Programmieren zu lernen. Nur Sie können wissen, warum Sie programmieren lernen möchten; einige Gründe könnten sein:

- Sie lernen Programmieren für die Arbeit oder für die Schule
- Sie wollen in der Lage sein Programme zu schreiben, die Sie wirklich brauchen
- Sie möchten verstehen wie Computerprogramme funktionieren
- Sie möchten Ihre Freunde oder Bekannte beeindrucken
- Sie sind auf der Suche nach einem neuen Hobby

Das alles sind gute Gründe. Egal was der Grund ist, Sie werden bestimmt feststellen, wie interessant, faszinierend und süchtigmachend Programmieren sein kann.

Was ist ein Computerprogramm?

Ein Computerprogramm ist eine Liste von Anweisungen, die dem Computer sagt, was er tun soll. Der Computer folgt diesen Anweisungen oder Befehlen Schritt für Schritt, bis er das Ende des Programms erreicht hat. Jede Zeile eines Computerprogramms ist für gewöhnlich ein Befehl, den der Computer ausführt. Jeder Befehl erledigt nur eine kleine Aufgabe, wie z. B. Zahlen addieren oder einen Text auf dem Bildschirm ausgeben. Aber wenn man hunderte von Befehlen, oder sogar tausende miteinander kombiniert, kann man tolle Sachen machen: mathematische Berechnungen durchführen, ein Dokument drucken, Bilder malen oder einfach 'SimCity' spielen.

Als nächstes werden Sie sehen, dass Computerprogramme mit verschiedenen Sprachen programmiert werden können.

Programmiersprachen

Computer verstehen kein Deutsch oder Englisch; sie verstehen auch keine andere menschliche Sprache. Sie können noch nicht einmal BASIC verstehen, die Computersprache, mit der KBasic verwandt ist. Computer verstehen nur eine Sprache: Maschinensprache. Diese besteht nur aus zwei Zahlen, 1 und 0. Aber Programmiersprachen wie BASIC erlauben Programmierern, Programme mit einer Englisch-ähnlichen Sprache zu entwickeln. Dort gibt es einen Übersetzer, der diese Englisch-ähnliche Sprache in Maschinensprache übersetzt, sodass der Computer es verstehen kann. Die KBasic Programmiersprache ist ein Dialekt der BASIC Programmiersprache, die entwickelt wurde, Entwicklern zu helfen, außerhalb der Maschinensprachenwelt zu programmieren. KBasic ersetzt viele der Maschinenbefehle aus 0 und 1 mit Wörtern und Symbolen, die wir Menschen besser lesen und benutzen können. Desweiteren ermöglicht KBasic Programmierern, visuell die Oberfläche eines Programms zu erstellen. So ist es viel einfacher für einen Softwareentwickler mit dem Computer zu kommunizieren. Anstatt wie eine Maschine zu denken, kann man jetzt wie ein Mensch denken.

Namensgeschichte

Manchmal sehen Sie den Namen BASIC mit Kleinbuchstaben geschrieben (also Basic). Sogar KBasic benutzt diese Schreibweise. Jedoch BASIC wurde ursprünglich als Abkürzung mit Großbuchstaben gelesen. BASIC steht für folgende englische Abkürzung: **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.

Ein BASIC Programm benutzt Symbole und Wörter (und ein paar Zahlen), die Menschen verstehen können. Wie ist es möglich, dass Computer BASIC-Programme verstehen können? Die Lösung ist, wenn Sie KBasic starten, starten Sie damit automatisch auch einen Übersetzer (Interpreter oder Compiler), ein spezielles Programm, das die Wörter und Symbole eines KBasic Programms in Maschinensprache übersetzt. Der Computer hätte keine Idee, was er mit einem BASIC Programm machen müsste, ohne die Übersetzung des Compilers.

Alles in allem existieren viele Arten von Computerprogrammiersprachen u. a. Java, C++ und BASIC... Aber alle Computersprachen haben eines gemeinsam: Sie können von Menschen gelesen werden, und müssen deswegen in Maschinensprache übersetzt werden, bevor der Computer sie verstehen kann.

Alle Arten von BASIC

Es gibt viele Arten von BASIC Programmiersprachen, KBasic ist nur eine davon. Neuere Versionen von DOS hatten QBasic installiert. Sie können zu Ihrem lokalen Softwarehändler gehen und Visual Basic kaufen. Alle diese Software erlaubt Ihnen Computersoftware mit BASIC zu erstellen, aber alle implementieren die BASIC Programmiersprache leicht unterschiedlich. Im Gegensatz zu fast allen BASIC Programmiersprachen erlaubt KBasic Ihnen Anwendungen auch für Mac OS X und Linux und nicht nur für Windows zu erstellen.

Compiler und Interpreter

Einige Computersprachen, wie einige BASIC Dialekte, übersetzen die Programmbefehle Zeile für Zeile während der Ausführung des Programms in Maschinensprache, sogenannte Interpreter. Andere Sprachen, wie z. B. Java oder KBasic, haben einen Compiler, der das gesamte Programm komplett auf einmal übersetzt, bevor es ausgeführt wird. In allen Fällen muss das Programm erst in Maschinensprache übersetzt werden.

Alle Compiler übersetzen ein Programm in eine ausführbare Datei, die ohne die Hilfe des Compilers ausgeführt werden kann. Eine ausführbare Datei ist, einfach gesagt, ein Maschinensprachenprogramm, das vom Computer direkt ausgeführt werden kann. Mit ein paar Ausnahmen haben fast alle Programmiersprachen Compiler. Aber es ist auch heutzutage schwierig, zwischen den Arten von Programmiersprachen - ob Compiler oder Interpreter - zu unterscheiden, weil es viele Mischformen gibt, wie z. B. Java oder KBasic.

Der Entwicklungsprozess

Nun, da Sie einiges über Computerprogramme erfahren haben, wie wäre es, wenn Sie einfach eins schreiben würden. Ein Computerprogramm zu schreiben ist nicht schwer, jedoch kann es ein langer Prozess sein. Ein Computerprogramm zu schreiben führt dazu, dass man Entwicklungsschritte hat, wie beim Schreiben eines Berichts oder einer Geschichte. Die folgenden Zeilen sollen diese Schritte verdeutlichen:

1. Entwurf des Programmkonzepts und der Bildschirmmasken, wie sie auf dem Bildschirm aussehen sollen
2. Erstellung des Programms mittels dem KBasic Quelltexteditor und Formulardesigner
3. Abspeichern des Programms
4. Ausführen des Programms, um es zu testen
5. Fehler korrigieren
6. Zurück zu Schritt 3.

Die meisten Schritte im Entwicklungsprozess wiederholen sich immer wieder, da Fehler entdeckt und korrigiert werden. Sogar erfahrene Programmierer können in der Regel keine fehlerfreien Programme schreiben, oder aber das Programm ist extrem kurz. Programmierer verbringen oft Stunden damit, ihre Anwendung zu optimieren und zu korrigieren, länger als das eigentliche Programmieren dauert. Das Optimieren und Korrigieren ist wichtig, da wir Menschen Fehler machen. Weiterhin ist unser Gehirn nicht in der Lage, jedes Detail zu wissen, dass für ein fehlerfreies Programm notwendig wäre. Viele von uns sind glücklich, wenn Sie sich schon den Namen des aktuellen Bundespräsidenten merken können. Aber wenn ein Programm unerwartet abstürzt oder etwas Seltsames macht, kann man nur hoffen, den Fehler zu finden. Computerexperten sagen: „Fehlerfreie Programme? Die gibt es nicht!“ Nachdem Sie angefangen haben große Programme zu schreiben, werden Sie sehen wie wahr das ist.

Angriff der Insekten

„Bug“ ist ein englisches Wort, das in der Computersprache einen Fehler in einem Computerprogramm bezeichnet. Der Name entstand während der Zeit der ersten Computer, als diese noch groß wie Räume waren und gewöhnliche Insekten, Käfer etc. (“Bugs“) manchmal verantwortlich dafür waren, dass Computer nicht einwandfrei funktionierten. Bevor man sein Computerprogramm veröffentlicht, sollte man versuchen, so viele Fehler wie möglich aufzuspüren.

Ist Programmieren schwierig?

Nachdem Sie nun erfahren haben, was es bedeutet, ein Computerprogramm zu erstellen, wird Ihnen vielleicht ein wenig unwohl. Jedenfalls haben Sie jetzt dieses Buch, das Ihnen versprochen hat, Sie das Programmieren mit KBasic zu lehren. Niemand hat Sie vor Maschinensprache, Interpreter, Compiler und Programmierfehler gewarnt. So ist jetzt Programmieren einfach oder nicht? Wahrscheinlich - ja und nein.
Es ist einfach, kleine Programme in KBasic zu schreiben. Die KBasic-Programmiersprache ist logisch, Englisch-ähnlich und einfach zu verstehen. Mit nur ein wenig Übung können Sie einfache und nützliche Programme schreiben. Alles was Sie brauchen ist die Zeit dieses Buch zu lesen und den Wunsch, ein paar eigene Programme zu schreiben

Was Sie in diesem Buch erfahren, ist genug für einen Programmierer, der nicht plant, ein Profi-Programmierer zu werden.
Wenn Sie jedoch eine Karriere als Softwareentwickler anstreben, haben Sie noch viel zu lernen, was nicht in diesem Einführungsbuch über Programmieren gezeigt wird. Zum Beispiel: Betrachten wir eine Textverarbeitung wie z. B. OpenOffice, die in vielen Jahren von vielen Softwareentwicklern erstellt wurde. Um so eine komplexe Software zu schreiben, braucht man gute Kenntnisse über die Funktionsweise des Computers. Zusätzlich muss man viele Jahre Erfahrung beim Programmieren professioneller Software gesammelt haben. Dennoch können Sie jetzt schon eine Menge mit KBasic erreichen, egal ob Sie daran interessiert sind, kleine Anwendungen zu schreiben oder sogar Computerspiele. Und Schritt für Schritt werden Sie entdecken, dass Programmieren nicht so kompliziert ist wie Sie vielleicht dachten.

Wieviel von KBasic muss ich können um es benutzen zu können?

Es ist möglich, KBasic Programme zu erstellen ohne eine Zeile Quelltext eingeben zu müssen. Es gibt viele Möglichkeiten bereits geschriebene KBasic Programme zu bekommen, wie z. B. Bücher, das Internet oder sogar Freunde und Kollegen. Sie können diese Programme einfach in KBasic übernehmen und haben dann eine fertigerstellte Anwendung. Manchmal jedoch brauchen Sie einfache oder komplexere Anpassungen, und darin liegt die wahre Herausforderung an Ihre Programmierkenntnisse.

Was Sie noch wissen müssen

Bevor Sie mit dem Programmieren loslegen, ist es wichtig zu wissen, was Computerprogramme sind und wie sie funktionieren. Dieses Hintergrundwissen wird Ihnen helfen zu verstehen, warum KBasic manche Sachen macht, die es macht; desweiteren wird es helfen, Fehler in Ihrem Programm zu finden. Sie sollten folgende Punkte wissen:

Ein Computer macht genau das, was ihm zuvor von einem Menschen instruiert wurde. Ein Computerprogramm ist eine Liste von Befehlen, die der Computer von Anfang bis Ende ausführt.

Ein BASIC Programm muss, bevor der Computer es verstehen kann, in Maschinensprache übersetzt werden. KBasic's Compiler macht dies automatisch. Ein Computerprogramm zu schreiben ähnelt sehr dem Schreiben eines Textes. Sie müssen viele Prototypen schreiben, bis es vollständig und fertig ist. Programmieren mit KBasic ist so schwierig oder einfach, wie Sie es möchten.

Was es einfach macht, KBasic zu verstehen, ist folgendes:
Sie sollten ein gutes Hintergrundwissen über objektorientiertes Programmieren haben, z. B. durch C++. Viele Konzepte und Prinzipien von KBasic sind ähnlich denen in C++ oder Java. Aber es gibt auch ein paar große Unterschiede zwischen KBasic und C++ oder Java.

Wenn Sie Datenbanken benutzen, sollten Sie ein fundiertes Wissen über SQL (structured query language) und über das Client-Server-Modell haben.

Einführung in die KBasic Programmiersprache

Hallo und herzlich willkommen zur KBasic Programmiersprache. Dieses Kapitel befasst sich mit

- was genau KBasic ist
- warum Sie KBasic lernen sollten (die Merkmale und die Vorteile gegenüber anderen Programmiersprachen)
- erste Schritte mit KBasic, welches Hintergrundwissen gebraucht wird und einige Grundlagenkenntnisse
- wie Sie Ihr erstes KBasic Programm schreiben

Was ist KBasic?

KBasic ist eine leistungsfähige Programmiersprache, die einfach, intuitiv und schnell erlernbar ist. Und da sie vor allem schon vertraut und bekannt ist, stellt KBasic für Linux, Mac OS X und Windows eine weitere Brücke zwischen diesen Systemen dar. KBasic ist eine neue Programmiersprache, ein weiterer BASIC-Dialekt, verwandt mit Visual Basic 6 und Java. Genauer gesagt ist KBasic eine objektorientierte und ereignisgesteuerte Programmiersprache, entwickelt von KBasic Software (www.kbasic.com). Es wurde speziell für Windows, Mac OS X und Linux entwickelt und an die Bedürfnisse von GUI-Entwicklern angepasst und besitzt somit eine Stärke in der GUI-Anwendungsentwicklung.

Unter C/C++ Entwicklern hat BASIC im allgemeinen einen schlechten Ruf als Kinderprogrammiersprache. Dass dem nicht so ist, wird mit diesem Buch deutlich. Mit KBasic können Sie fast alle Anwendungen schreiben, die Sie mit C/C++ vielleicht geschrieben hätten, gäbe es KBasic nicht.

Kurz und bündig in einem Satz

KBasic ist eine einfach benutzbare, objektorientierte, interpretierte, robuste, plattformunabhängige, performante und moderne Sprache.

Warnung

Um Kompatibilität zu zukünftigen Version von KBasic sicherzustellen, sollten sie beim Programmieren in KBasic vermeiden, bestimmte alte Sprachelemente, die aus historischen Gründen noch unterstützt werden, zu verwenden. Verwenden sie stattdessen die modernen Sprachelemente. Mehr Details dazu finden sie in den jeweiligen Hilfeeinträgen zu den einzelnen Menüpunkten innerhalb der KBasic-Entwicklungsumgebung. Beachten Sie bitte auch die Hinweise zur Migration von Visual Basic 6 am Ende dieses Buches. Außerdem wurden in diesem Buch nur die neuen, aktuellen und modernen Sprachfeatures erläutert, da viele Sprachkonstrukte nur noch aus historischen Gründen von KBasic unterstützt werden, z. B. sind 'GoSub', 'DefInt', 'On Error GoTo' veraltete Sprachelemente und werden in diesem Buch nicht erläutert.

Beispiele

Alle Beispiele sind der einfachthalber für den Autor in Englisch gehalten.

KBasics Vergangenheit, Gegenwart und Zukunft

KBasic wurde ursprünglich als Open Source Software entwickelt. Das Projekt wurde von Bernd Noetscher im Sommer 2000 ins Leben gerufen. Nach etlichen Enttäuschungen durch mangelnde Hilfsbereitschaft und Unterstützung durch Freiwillige traf er die Entscheidung, die Programmiersprache als kommerzielle Entwicklung für Linux, Mac OS X und Windows bereitzustellen, um so sicherzustellen, dass die hohen Produktansprüche erfüllt sind. Gegenwärtig wird an KBasic Version 2.0 gearbeit mit noch mehr Benutzerfreundlichkeit, Leistungsfähigkeit der Sprache, weiteren Anbindungen an Programmierbibliotheken und neuen objektorientierten Features.

Was muss man noch über KBasic wissen?

KBasic ist objektorientiert

Für sie als Programmierer bedeutet es, dass sie sich auf die Daten in Ihrer Applikation und auf die Methoden, mit denen sie die Daten manipulieren, konzentrieren können anstatt nur in Prozeduren zu denken. Wenn sie mit prozeduraler Programmierung in BASIC vertraut sind, werden sie vermutlich zu der Überzeugung gelangen, dass Sie die Art, wie sie ihre Programme entwerfen, ändern müssen, wenn sie KBasic benutzen. Aber das müssen sie nicht. Sie können weiterhin prozedural programieren. Wenn sie sehen, wie mächtig dieses neue Paradigma „objektorientiert“ ist, werden Sie sich aber schnell daran gewöhnen, denn mittels Objektorientierung ist man viel leichter in der Lage, wiederverwendbare, komplexe Anwendungsprogramme übersichtlich und modular zu entwickeln. KBasic ist objektorientiert hinsichtlich der Anbindung an C++ Programmierbibliotheken, oder selbstdefinierte KBasic-Klassen, also mit echter Vererbung. Im Gegensatz zu anderen Programmiersprachen wurde KBasic von vornherein objektorientiert entworfen. Die meisten Dinge in KBasic sind Objekte; die einfachen Datentypen, wie numerisch, Zeichen und boolesche Werte, sind die einzigen Ausnahmen. Obwohl KBasic so entworfen wurde, dass es ähnlich aussieht wie VB6, werden sie merken, dass KBasic viele Schwierigkeiten von VB6 bereinigt.

KBasic ist einfach zu lernen

KBasic ist eine einfache Sprache. Ein Designziel von KBasic ist die einfache Benutzung sowie schnelle Entwicklung, leichtere Fehlersuche und einfache Erlernung. KBasic ist nach Standard BASIC modelliert, ist abgesehen von den Visual Basic Komponenten syntax-kompatibel zu Visual Basic 6 und kommt damit der Mehrzahl der Programmierer bekannt vor; dadurch ist die Migration einfach und schnell. KBasic enthält objektorientierte Ideen von Java und C++, wobei verwirrende Elemente weggelassen wurden. KBasic verwendet wie in Java Referenzen statt Pointer auf Objekte, darüber hinaus führt der „garbage collector“ zu einer für den Programmierer einfachen Speicherverwaltung, ganz automatisch. Ihnen bleiben herumirrende Zeiger, ungültige Zeigerreferenzen und Speicherlöcher erspart und Sie können Ihre Zeit stattdessen darauf verwenden, die Funktionalität Ihres Programms zu entwickeln. KBasic ist tatsächlich eine vollständige und elegante Sprache.

Interpretiert

KBasic erzeugt Pseudo-Code statt direkten Maschinencode. Um ein KBasic Program tatsächlich ablaufen zu lassen, benutzen Sie die KBasic Entwicklungsumgebung (IDE) oder den Menüpunkt „Binärdatei erstellen“ (nur Professional Edition). KBasic ist somit eine interpretierte Sprache. KBasic Pseudo-Codes stellen ein architekturunabhängiges Format zur Verfügung: der Code ist dazu entworfen worden, Programme effizient auf verschiedene Plattformen zu transportieren.

Robust

KBasic wurde entworfen, um sehr zuverlässige robuste Software zu schreiben. KBasic macht selbstverständlich die Qualitätskontrolle nicht unnötig. Es ist immer noch möglich, unverlässliche Software zu schreiben. Jedoch eliminiert KBasic bestimmte Arten von Programmierfehlern, so dass es deutlich einfacher wird, zuverlässige Software zu schreiben. KBasic ist eine typisierte Sprache, was es ermöglicht, während der Übersetzung sehr ausführliche Prüfungen potentieller Typinkonsistenzen durchzuführen. KBasic ist strenger typisiert als Standard BASIC. Eine der wichtigsten Eigenschaften hinsichtlich der Zuverlässigkeit ist das Speichermodell. KBasic unterstützt keine Zeiger, was die Gefahr ausschließt, Speicher zu überschreiben und Daten unbrauchbar zu machen. Ähnlich wirkt auch der KBasic „garbage collector“, der auch hilft, Speicherlöcher und andere bösartige Fehler, die mit dynamischer Speicherallozierung und -deallozierung in Zusammenhang stehen, zu verhindern. Außerdem prüft der KBasic-Interpreter auch während der Laufzeit verschiedene Dinge, so zum Beispiel, ob Felder und Zeichenketten sich noch innerhalb ihrer Grenzen befinden. Die Ausnahmebehandlung ist eine andere Möglichkeit, die KBasic bietet, um robustere Programme zu schreiben. Eine Ausnahme ist ein Signal für einen außergewöhnlichen Zustand, der wie ein Fehler aufgetreten ist. Durch Benutzung der „try/catch/finally“ Befehle können Sie alle ihre Fehlerbehandlungsroutinen an einer Stelle gruppieren , was es erleichtert, Fehler zu behandeln und zu beheben.

Performanz

KBasic ist eine interpretierende Sprache. Deswegen wird sie niemals so schnell wie eine übersetzende Sprache wie etwa C sein. Tatsächlich ist KBasic etwa 20 mal langsamer als C. Bevor sich jetzt aber entsetzt abwenden, seinen sich der Tatasche bewusst, dass diese Geschwindigkeit mehr als ausreicht für interaktive, grafische Anwendungen, die in den meisten Fällen nur darauf warten, dass der Benutzer etwas tut. Wenn Sie die Leistung betrachten, ist es wichtig, sich zu überlegen, wo KBasic im Spektrum der verfügbaren Programmiersprachen steht. An einem Ende des Spektrums liegen hohe, voll-interpretierende Sprachen wie PHP und UNIX-Shells. Diese Sprachen sind gut zum Entwurf geeignet und sehr portabel, aber auch sehr langsam. Am anderen Ende sind die unteren, übersetzenden Sprachen, wie C oder C++. Diese Sprachen bieten hohe Leistung, aber sie sind nicht so portabel und verlässlich.

KBasic liegt in der Mitte dieses Spektrums. Die Leistung von KBasics interpretiertem Pseudo-Code ist viel besser als die der höheren Script-Sprachen, aber es bietet immer noch die Einfachheit und Portabilität dieser Sprachen. Obwohl KBasic nicht so schnell ist wie eine compilierende Sprache, so bietet es doch die architekturunabhängige Umgebung, in der es möglich ist, zuverlässige Programme für Windows, Mac OS X und Linux zu schreiben.

Warum KBasic erfolgreich ist

Traditionell litten Programmiersprachen unter der Idee, dass man alles über Bord werfen und ganz von vorne anfangen müsse; mit neuen Konzepten und einer neuen Syntax. Dies wurde damit begründet, dass es besser auf lange Sicht ist, Altlasten loszuwerden. Dies mag wahr sein - auf lange Sicht. Aber eigentlich war viel von diesen Altlasten gut verwertbar. Am verwertbarsten waren vielleicht nicht die existierenden Quelltexte, sondern die existierenden Programmierfähigkeiten mit genau dieser Sprache. Wenn Sie ein erfolgreicher VB6 Entwickler sind und Sie alles über Bord werfen müssten, was Sie über VB6 wüssten, würden Sie viel weniger produktiv für viele Monate werden, bis Sie sich dem neuen Programmierparadigma angepasst hätten. Wohingegen, wenn Sie die VB6 Kenntnisse mitnehmen und erweitern könnten, würden Sie weiterhin produktiv sein, während Sie sich dem neuen Paradigma anpassen wie z. B. objektorientierte Programmierung. Da jeder sein eigenes Programmierverständnis hat, würde der Schritt zu einer neuen Programmiersprache an sich schon anstrengend genug sein. Das Problem mit dem Lernen einer neuen Programmiersprache ist die Produktivität. Keine Firma kann es sich leisten, plötzlich einen produktiven Softwareentwickler zu verlieren, bloß weil er gerade eine neue Programmiersprache lernt. KBasic ist sehr ähnlich zu VB6, die Syntax ist die gleiche, nur gibt es noch Erweiterungen und das Programmiermodell ist dasselbe. Es erlaubt Ihnen weiterhin schnell brauchbaren Quelltext zu programmieren und allmählich auf die neuen Möglichkeiten umzusteigen, genau in dem Tempo, in dem Sie diese verstehen und lernen. Dies mag einer der wichtigsten Gründe für den Erfolg von KBasic sein. Und außerdem ist viel von Ihrem bestehenden VB6 Quelltexten in KBasic weiterhin nutzbar.

Ein einfaches Beispiel

Genug davon! Jetzt sollten sie einen guten Eindruck davon haben, worum es bei KBasic eigentlich geht. Lassen sie uns aufhören, über abstrakte Schlagworte zu reden und stattdessen einen konkreten BASIC Quelltext anschauen. Jetzt wollen wir uns einem allgegenwärtigen Favoriten zuwenden: „Hallo Welt“

```
' program beginning

CLS
Print "Hello World!"
Print
Print
Print "                / `._      .       .       _.' \"
Print "              '.@ = `.      \     /     .' = @.'"
Print "               \ @`.@ `.     \   /    .' @.'@ / "
Print "                \;`@`.@ `.    \ /    .' @.'@`;/  "
Print "                 \`.@ `.@ `'.(*).'` @.' @.'/    "
Print "                  \ '=._`. @ :=: @ .'_.=' /     "
Print "                   \ @   '.'..'='..'.'   @ /     "
Print "                    \_@_.==.: = :.==._@_/       "
Print "                    /  @ @_.: = :._@ @  \       "
Print "                   /@  _.-'  : = :  '-._ @\     "
Print "                  /`'@ @ .-': = :'-.@ @`'`\     "
Print "                  \.@_.=` .-: = :-. `=._@./     "
Print "                    \._.-'   '.'   '-._./       "
Print
Print "... you just did your first kbasic program!"
```

Sie können dieses Programm direkt in ein leeres Quelltextfenster in KBasic eintippen und starten lassen. Auf dem Bildschirm wird dann ein schöner Schmetterling erscheinen.

Gehen wir nun zum nächsten Kapitel.

Rapid Application Development (RAD)

Sie können KBasic für die schnelle Entwicklung von grafischen Anwendungen verwenden. Der Formulardesigner ermöglicht Ihnen per „drag & drop“ und Mausklick:

- Entwurf der Benutzeroberfläche Ihres Programms
- Bestimmung des Verhaltens der Steuerelemente
- Festlegung der Beziehung zwischen der Benutzerschnittstelle und dem Rest des Programms

Zusätzlich zu den grafischen Möglichkeiten können Sie mit KBasic noch folgende Arbeiten schnell und einfach erledigen:

- Neue Programmelemente erstellen. In KBasic ist ein Programmelement:
- Ein Projekt: es enthält u. a. Klassen, Module und Formulare
- Ein Formular: die Oberflächen Ihres Programms
- Klassen/Module: die Programmteile. Klassen können Methoden und andere Anweisungen enthalten.
- Sub/Function/Methode: können ebenfalls Anweisungen enthalten
- Importieren und Exportieren von Quelltexten

KBasic gibt Ihnen die Möglichkeiten professionelle Software zu erstellen. Besonderheiten sind:

- Debuggen Ihres Quelltextes während Ihr Programm abläuft
- Übersicht der Quelltexte bezogen auf Projekt oder Klasse (Klassenbrowser)

Wie bekomme ich KBasic?

KBasic Professional Version

KBasic Professional kann man käuflich erwerben. Es wird vom Hersteller unter www.kbasic.com zum Verkauf angeboten und kostet etwa 24,95 Euro (inklusive Versandkosten / Stand Januar 2008). Der Käufer erhält drei Programmversionen: eine für Windows, eine für Mac OS X und eine für Linux; und eine Lizenz, die gültig ist für Windows, Mac OS X und Linux und das Recht beliebig viele KBasic Programme kommerziell ohne weitere Lizenzgebühren zu entwickeln.

Bei Fragen wenden sie sich bitte an: sales@kbasic.com

Installation von KBasic / Installationsbeschreibung / Systemvoraussetzungen

KBasic ist einfach zu installieren, kommt mit einem Installationsprogramm und braucht mit vollständiger Dokumentation etwa 200 MB Speicherplatz.

Offiziell unterstützt werden:

- Windows Vista/XP/2000, aber auch ältere Windowsversionen
- Linux mit KDE >= 3.3.3 (oder Qt >= 4.2)
- Mac OS X (>= 10.3)

An Arbeitsspeicher reichen unter Windows mind. 64 MB RAM und der Prozessor sollte mind. 400 Mhz schnell sein. Bildschirmauflösung 1024*768, 32bit, Echtfarben

KBasic starten

Sie können KBasic starten indem Sie folgende Aktion durchführen:

- unter Linux auf das KBasic Icon - auf dem Desktop klicken
- unter Windows auf das KBasic Icon - auf dem Desktop klicken
- unter Mac OS X auf das KBasic Icon - auf dem Desktop klicken

Nach dem Start von KBasic sehen Sie das Hauptfenster von KBasic. Dieses Fenster beinhaltet weitere Fenster, wie z. B. Fenster zum Editieren von Quelltexten oder das Projektfenster, das das aktuelle Projekt und seine Elemente anzeigt.

Häufig gestellte Fragen

Was ist der Unterschied zwischen KBasic und Visual Basic 6?

KBasic ist VB6 sehr ähnlich, was es VB6 Programmierern einfach macht, KBasic zu lernen. Aber es gibt eine Menge wichtiger Unterschiede zwischen KBasic und VB6, was zum Beispiel die Preprozessoren und den Mechanismus zur Ausnahmebehandlung betrifft. Eines der Hauptgebiete, auf dem sich KBasic und VB6 unterscheiden, ist natürlich, dass KBasic eine echte objektorientierte Sprache ist und Mechanismen besitzt, um Klassen zu definieren, diese zu vererben und Objekte daraus zu erzeugen, die Instanzen dieser Klassen sind. KBasic enthält alle syntaktischen Sprachelemente von Visual Basic 6 und unterstützt auch Objekt-Komponenten ähnlich denen von Visual Basic 6. Die Visual Basic 6 Syntax wird also vollständig unterstützt. Darüber hinaus bietet KBasic weitere objektorientierte Features und Erweiterungen.

Ein Programm in KBasic besteht aus einer oder mehreren Klassendefinitionen, Modulen, Formularen oder einfach nur Funktionen oder Prozeduren. Genauso wie in VB6 ist es möglich im globalen Namensraum, globale Funktionen und Variablen und Konstanten zu deklarieren. KBasic unterstützt zur Zeit nicht das bedingte Übersetzen, wird es aber in Zukunft tun (z. B. '#ifdef' und '#if'). Theoretisch gesehen ist bedingte Übersetzung in KBasic auch nicht nötig, da es eine platformunabhängige Sprache ist und es deswegen keine Plattformabhängigkeiten gibt, die diese Technik benötigen. Desweiteren sind die einfachen Datentypen gleichbedeutend, jedoch unterscheidet sich die Größe mancher Datentypen: 'Boolean' ist 1 Byte, 'Integer' ist 32 bit und 'Long' ist 64 bit in KBasic.

Kann ich bestehende Visual Basic 6 Programme automatisch importieren lassen?

KBasic enthält einen Import-Assistent für VB6 Programme. Zur Zeit müssen Sie aber manchmal manuell Änderungen vornehmen, falls einige VB6 Komponenten nicht erkannt wurden.

Auf welchen Plattformen ist KBasic verfügbar?

Zur Zeit gibt es eine Version für Linux, Mac OS X und für Windows.

Wo finde ich weitere Information zu KBasic. Beispiele, Dokumentation und Neuigkeiten?

Zuerst lesen Sie dieses Buch. Dann gibt es noch www.kbasic.com, wo Informationen und Neuigkeiten verbreitet werden. Beispiele finden sich genug im Internet. Es gibt etliche Visual Basic 6 oder BASIC Programmierarchive, die meistens ohne Änderungen oder mit kleineren Anpassungen direkt in KBasic übernommen werden können.

KBasic-Syntax

Die Syntax in einem KBasic Hilfethema für eine Methode, Anweisung oder Funktion zeigt alle Elemente, die für die korrekte Verwendung der Methode, Anweisung oder Funktion benötigt werden. In den Beispielen dieses Abschnitts wird erklärt, wie die am häufigsten verwendeten Syntaxelemente interpretiert werden müssen.

Beispiel: Syntax der MsgBox-Funktion

```
MsgBox(prompt[, buttons] [, title])
```

Argumente, die in eckigen Klammern eingeschlossen sind, sind optional. (Geben Sie diese Klammern in Ihrem KBasic-Code nicht an.) Das einzige Argument, das Sie für die MsgBox-Funktion bereitstellen müssen, ist der Text für die Anzeige (prompt).

Argumente für Funktionen und Methoden können im Code entweder mit Hilfe ihrer Position oder ihres Namens angegeben werden. Um die Argumente mit Hilfe ihrer Position anzugeben, sollten Sie der in der Syntax angegebenen Reihenfolge folgen, und trennen Sie jedes Argument durch ein Komma.

Beispiel:

```
MsgBox("The answer is right!", 0, "window with answer")
```

Wenn Sie ein Argument mit Hilfe des Namens angeben möchten, verwenden Sie den Argumentnamen, gefolgt von einem Doppelpunkt und einem Gleichheitszeichen (:=) sowie dem Argumentwert. Sie können diese benannten Argumente in jeder Reihenfolge angeben.

Beispiel:

```
MsgBox(title:="window with answer", prompt:="The answer is right!")
```

In der Syntax von Funktionen und von einigen Methoden werden die Argumente in geschweiften Klammern eingeschlossen.

```
Option Compare {Binary | Text}
```

In der Syntax der 'Option Compare'-Anweisung kennzeichnen die geschweiften Klammern und die vertikalen Linien eine vorgeschriebene Wahl zwischen zwei Elementen. (Geben Sie die Klammern nicht in der KBasic-Anweisung an). Die folgende Anweisung gibt z. B. an, dass innerhalb des Moduls Zeichenfolgen in einer Sortierreihenfolge verglichen werden, die nicht auf die Groß-/Kleinschreibung achtet.

```
Option Compare Text
```

Syntax der Dim-Anweisung

```
Dim VarName[([Indexes])] [As Type] [, VarName[([Indexes])] [As
Type]] ...
```

In der Syntax der 'Dim'-Anweisung entspricht das Wort 'Dim' einem erforderlichen Schlüsselwort. Das einzige erforderliche Element ist VarName (der Variablenname). Die folgende Anweisung z. B. erstellt drei Variablen: myVar, nextVar und thirdVar. Diese werden automatisch als 'Variant'-Variablen deklariert (oder 'Double' in 'Option VeryOldBasic').

```
Dim myVar, nextVar, thirdVar
```

Das folgende Beispiel deklariert eine Variable als 'String'. Das Angeben eines Datentyps spart Speicherplatz und kann dazu beitragen, Fehler im Code zu entdecken.

```
Dim myAns As String
```

Geben Sie den Datentyp für jede Variable an, um mehrere Variablen in einer Anweisung zu deklarieren. Variablen, die ohne einen Datentyp deklariert werden, werden automatisch als 'Variant' deklariert (oder 'Double' in 'Option VeryOldBasic').

```
Dim x As Integer, y As Integer, z As Integer
```

In der folgenden Anweisung wird x und y der Datentyp 'Variant' zugewiesen. Nur z wird der Datentyp 'Integer' zugewiesen.

```
Dim x, y, z As Integer
```

Sie müssen Klammern hinzufügen, wenn Sie eine Datenfeldvariable deklarieren. Die Indizes sind optional. Die folgende Anweisung dimensioniert ein dynamisches Datenfeld, myArray.

```
Dim myArray(100) ' old style, not recommended
Dim myArrayModernStyle[200]
```

KBasic ist nicht nur eine Programmiersprache, sondern drei

Wenn Sie einen der folgenden Befehle verwenden, können Sie den Modus von KBasic umschalten:

- Wenn Sie die neuesten KBasic Sprachelemente benutzen möchten (Standard)

```
Option KBasic
```

- Wenn Sie alten VB6 Code verwenden möchten (nicht empfohlen)

```
Option OldBasic
```

- Wenn Sie sehr altes BASIC verwenden möchten, wie z. B. QBasic (nicht empfohlen)

```
Option VeryOldBasic
```

Es ist möglich alle drei Modi in einem Programm zu benutzen, z. B. benutzt ein Modul 'Option OldBasic', weil es alten VB6 Code enthält und die restlichen Module 'Option KBasic', weil diese neuen KBasic Code enthalten. Dazu müssen Sie nur eine der oben genannten Zeilen am Anfang des Moduls plazieren. Standard ist 'Option KBasic'.

Softwareentwicklung mit KBasic

Eine typische KBasic-Anwendung besteht aus Formularen, Modulen und Klassen und anderen Objekten, die Sie zu einer Einheit verbinden. Formulare und deren Steuerelemente sowie das Ändern von Daten in einem Feld oder das Anklicken eines Befehlssymbols durch den Benutzer entsprechen Ereignissen. Sie können die Reaktion auf ein Ereignis steuern, indem Sie diesem Ereignis KBasic-Code zuordnen.

Ereignisgesteuerte gegenüber traditionelle Programmierung

Bei einem traditionellen „prozeduralen“ Programm steuert die Anwendung, welche Teile des Codes ausgeführt werden, unabhängig von Ereignissen. Die Ausführung startet mit der ersten Codezeile und folgt einem vorgegeben Weg durch die Anwendung; bei Bedarf werden einzelne Prozeduren aufgerufen.

In ereignisgesteuerten Anwendungen löst eine Aktion des Benutzers oder ein Systemereignis eine Ereignisprozedur aus. Die Reihenfolge, in der der Code ausgeführt wird, hängt also davon ab, in welcher Reihenfolge die Ereignisse geschehen; dies wiederum hängt von den Aktionen des Benutzers ab. Dies ist das Prinzip graphischer Benutzeroberflächen und ereignisgesteuerter Programmierung: Der Benutzer agiert und das Programm reagiert entsprechend.

Funktionsweise ereignisgesteuerter Anwendungen

Ein Ereignis ist eine Aktion, die von Formularen oder Steuerelementen erkannt wird. Objekte in KBasic erkennen eine vordefinierte Gruppe von Ereignissen automatisch und reagieren auf ein bestimmtes Ereignis. Damit ein Steuerelement in einer bestimmten Weise auf ein Ereignis reagiert, können Sie für dieses Ereignis eine Ereignisprozedur in KBasic schreiben.

So läuft eine typische ereignisgesteuerte Anwendung ab:

1. Ein Anwender startet die Anwendungen
2. Das Formular oder ein Steuerelement in dem Formular empfängt ein Ereignis. Das Ereignis kann durch den Benutzer ausgelöst werden (z. B . einen Tastenanschlag oder einen Mausklick) oder durch Ihren Code (z. B. das Ereignis „Öffnen“, wenn Ihr Code ein Formular öffnet).
3. Wenn für das Ereignis eine Ereignisprozedur vorliegt, wird diese ausgeführt.
4. Die Anwendung wartet auf das nächste Ereignis.

Einige Ereignisse lösen automatisch weitere Ereignisse aus. Näheres finden sie dazu in der Hilfeumgebung von KBasic.

Die drei Entwicklungsschritte

Eine einfache KBasic Anwendung zu erstellen ist fast einfacher als Kuchenbacken. Sie brauchen bloß drei Schritte zu durchlaufen, und nach denen werden Sie ein Programm haben, das außerhalb der KBasic Umgebung lauffähig ist wie jede andere Anwendung. Diese drei Schritte sind folgende:

1. Erstellen der Programmoberfläche
2. Schreiben des Programmquelltextes, was das Programm veranlasst, dass zu tun, was es tun soll
3. Übersetzen des Programms, sodass das Programm ohne KBasic lauffähig ist (nur mit der Professional Edition von KBasic). Das Übersetzen wird mit Hilfe des KBasics Compiler durchgeführt.

Natürlich sind viele Details in diesen Schritten versteckt, besonders, wenn Sie ein größeres Programm entwickeln. Während Sie daran arbeiten, werden Sie diese Schritte in der Reihenfolge durcharbeiten, in der sie oben erwähnt sind. Aber Sie werden auch zwischen den Schritten hin und her springen, z. B. von Schritt 2 zu Schritt 1, wenn Sie die Oberfläche verfeinern wollen, nachdem Sie den Initialprogrammcode erstellt haben.

Objektorientierte Programmierung mit KBasic

Objektorientierte Programmierung ist eine der wichtigsten Konzepte der letzten Jahre und hat sich auf breiter Linie durchgesetzt. Dort tauchen Begriffe auf wie

- was sind Klassen und Objekte; und wie stehen sie zueinander in Beziehung
- das Verhalten und Attribute der Klassen und Objekte
- Vererbung von Klassen und wie dadurch das Programmdesign beeinflusst wird

Objekte und Klassen

KBasic ist eine objektorientierte Sprache. Objektorientiert ist ein Begriff, der so häufig gebraucht wird, dass er praktisch gar keine konkrete Bedeutung mehr hat. Objektorientiert im Sinne KBasics deckt folgende Gebiete ab:

- Klassen und Objekte in KBasic
- Erzeugen von Objekten
- „garbage collection“ zur Freigabe unbenutzter Objekte
- den Unterschied zwischen Klassen-Variablen und Instanzvariablen und den Unterschied zwischen Klassenmethoden und Instanzmethoden
- die Erweiterung einer Klasse, um eine Unterklasse zu erzeugen
- das Überschreiben von Klassenmethoden und das dynamische Suchen nach Methoden
- Abstrakte Klassen

Wenn Sie ein Java Programmierer sind oder mit anderen objektorientierten Sprachen Erfahrungen gesammelt haben, sollten Ihnen viele dieser Konzepte bekannt vorkommen. Wenn sie diese Erfahrung nicht besitzen, brauchen sie keine Angst zu haben, denn im Prinzip sind diese Konzept nicht besonders schwer zu erlernen.

Objektorientierte Programmierung verfolgt die Idee, das jedes Objekt Bestandteil eines anderen Objekts ist. Genau wie in der richtigen Welt ein Auto aus vielen Objekten besteht: Karosserie, Reifen, Felgen, Lenkrad usw.

Klassen sind die Baupläne für Reifen, Felgen usw. Objekte werden nach diesen Bauplänen erstellt. Somit gibt es eine Klasse Reifen, aber viele Objekte Reifen, alle nach dem Bauplan der Klasse erstellt. Man sagt auch eine Instanz einer Klasse, wobei man statt Instanz auch Objekt sagen kann. Eine Instanz ist somit ein dynamisches, während der Laufzeit erzeugtes Objekt.

Jede Klasse besteht aus folgenden Elementen:

- Geerbte Klasse
- Attributen
- Methoden

Geerbte Klasse bedeutet alle Attribute und Methoden der vererbenden Klasse, also die Vorlage einer Klasse, (und deren geerbten Klassen) befinden sich auch in dieser Klasse zusätzlich zu den speziell in dieser Klasse angegeben.

Attribute sind Eigenschaften, Konstanten und Variablen, die zu dieser Klasse gehören.

Methoden sind die Prozeduren und Funktionen dieser Klasse. Methoden sind unstreitbar der wichtigste Teil einer objektorientierten Programmiersprache. Klassen und Objekte bilden also nur das Rahmenwerk, die Klassen- und Instanzvariablen speichern die Daten und somit die Objekteigenschaften. Die Methoden dagegen definieren das Verhalten der Objekte und damit auch das Verhalten der Objekte unter- und miteinander. Variablen und Methoden können sowohl nur klassenbezogen (1mal vorhanden im Speicher) oder instanzbezogen deklariert sein (so oft wie Instanzen im Speicher vorhanden sind) im Deklarationsbereich der Klasse (außerhalb einer Methode):

- Instanzvariable
  ```
  Dim instanceVar
  ```
- Instanzmethode
  ```
  Sub instanceSub()
  ```
- Klassenvariable
  ```
  Static staticVar
  ```
- Klassenmethode
  ```
  Static Sub staticSub()
  ```

Klassenbezogen bedeutet: man kann diese Variable oder Methode unabhängig von einem Objekt verwenden. Instanzbezogene werden immer mit einem Objekt verwendet. Instanzvariablen ähneln den lokalen Variablen: lokale Variablen existieren nur beim Aufruf der Prozedur oder Funktion, Instanzvariablen existieren nur zur Lebenszeit eines Objektes. Klassenvariablen sind global und in der Objektinstanz verfügbar. Klassenvariablen sind gut für die Kommunikation zwischen verschiedenen Objekten der gleichen Klasse, oder gut zum Speichern von globalen Daten.

Auch Klassen können Konstanten definieren, das geschieht wie bei einer Variablendeklaration in einer Klasse im Deklarationsbereich der Klasse.

Klassen können all ihre Elemente an die Nachkommen weitergeben, d.h. vererben.

Vererbung von Klassen

Vererbung ist das Wesentliche der objektorientierten Programmierung.

KBasic unterstützt einfache Vererbung, d.h. eine Klasse erbt nur von einer Klasse und nicht von mehreren. Es verhält sich hier genauso wie bei Java. In Zukunft wird das Konzept der Interfaces auch unterstützt werden, was der Mehrfachvererbung recht nahe kommt. Wie in Java müssen in KBasic alle Objekte dynamisch mit 'New' erzeugt werden.

Eine Klasse ist also eine Sammlung von Daten und Methoden, die auf der Basis dieser Daten arbeiten. Die Daten und Methoden dienen zusammengenommen dazu, den Inhalt und die Fähigkeiten eines Objektes zu definieren.

Objekte sind Instanzen einer Klasse: Wir können mit einer Klasse selbst fast nichts anfangen, wir benötigen eine Instanz der Klasse, ein einzelnes Objekt, dass aus den Variablen- und Methodendefinitionen der Klasse besteht.

Das ist genau, was objektorientierte Programmierung genannt wird; die Objekte stehen hier im Mittelpunkt, nicht der Funktionsaufruf.

KBasic-Syntax

Wenn Sie Programme schnell schreiben und lesen möchten, müssen Sie sich mit der Syntax von KBasic auskennen. Die Syntax legt fest, wie Sie Ihre Programme schreiben. Sie definiert, wie jedes einzelne Programmelement geschrieben werden muss, wie diese miteinander in Beziehung stehen und wie diese benutzt werden können. Auf den folgenden Seiten werden Sie mit der Syntax vertraut gemacht.

Anweisungen und Ausdrücke

Eine Anweisung in KBasic gibt eine vollständige Aktion an. Sie kann Schlüsselwörter, Operatoren, Variablen, Konstanten und Ausdrücke enthalten. Jede Anweisung gehört zu einer der folgenden drei Kategorien:

- Deklarationsanweisungen bzw. Definitionsanweisungen, die eine Klasse, Module, Variable, Konstante oder Prozedur, Funktion, Methode, Eigenschaft benennen und auch gegebenenfalls einen Datentyp angeben.
- Zuweisungsanweisungen, die einen Wert oder Ausdruck einer Variablen oder Konstanten oder Eigenschaft zuweisen.
- Ausführbare Anweisungen, die Aktionen bewirken. Diese Anweisungen können eine Methode oder Funktion, Prozedur und eine Schleife oder eine Verzweigung zu Code-Blöcken ausführen. Ausführbare Anweisungen enthalten oftmals mathematische oder bedingte Operatoren.

Fortsetzen einer Anweisung über mehrere Zeilen

Eine Anweisung passt normalerweise in eine Zeile. Sie können aber auch eine Anweisung in der nächsten Zeile fortsetzen, indem Sie ein Zeilenfortsetzungszeichen (_) verwenden. Im folgenden Beispiel wird die ausführbare Anweisung 'MsgBox' über zwei Zeilen fortgesetzt:

```
Sub field()
      Dim myVar As String
      myVar = "Bernd"
      MsgBox Prompt:="Hello " & myVar, _
            Title:="Hello"
End Sub
```

Sie können auch mehrere Anweisungen in einer einzigen Zeile schreiben, wenn Sie sie jeweils durch einen Doppelpunkt (:) trennen. Zum Beispiel:

```
Print „Hi“: Print „Ho“
```

Variablen und Datentypen

Häufig ist es erforderlich, beim Ausführen von Berechnungen mit KBasic Werte vorübergehend zwischenzuspeichern. Sie möchten z. B. verschiedene Werte berechnen, diese vergleichen und je nach Ergebnis des Vergleichs unterschiedliche Operationen mit den Werten ausführen. Sie müssen die Werte speichern, um sie zu vergleichen, weil Sie diese Werte jedoch nur solange bereithalten müssen, wie ihr Code abläuft, ist es nicht sinnvoll sie in einer Datei zu speichern.

KBasic verwendet Variablen, um Werte zu speichern, genauer gesagt, zwischenzuspeichern, denn nach dem Beenden des Programms sind alle Variablen wieder gelöscht. Eine Variable kann ständig ihren Wert ändern, indem ihr eine andere Variable, eine Konstante oder ein Ausdruck zugeordnet wird. Eine Variable ist ein Speicherplatz, der jedoch nicht in einer Datei, sondern innerhalb von KBasic vorliegt. Genau wie eine Datei, besitzt auch eine Variable einen Namen (das Wort, das sie verwenden, um auf den in der Variablen enthalten Wert zu verweisen) und außerdem einen Datentyp (der bestimmt, welche Art von Daten die Variable enthalten kann).

KBasic kennt Instanzvariablen, Klassenvariablen, globale und lokale Variablen. Instanzvariablen gehören zu einem Objekt, Klassenvariablen zu einer Klasse. Globale Variablen sind überall verwendbar und lokale Variablen gehören zu einer Prozedur, Funktion oder Methode.

Deklaration von Variablen / Explizite Deklaration

Bevor Sie Variable benutzen können, müssen Sie diese deklarieren. Das bedeutet: Sie müssen den Namen und den Datentyp der Variablen angeben. Die 'Dim'-Anweisung deklariert eine Variable. Diese 'Dim'-Anweisung gehört zu einer Gruppe von Anweisungen, die Variablen deklarieren. Weitere Anweisungen zum Deklarieren sind: 'Redim', 'Static', 'Public', 'Private', 'Protected' und 'Const'.

```
Dim myName As String
```

Variable können an jeder Stelle innerhalb einer Prozedur oder Funktion deklariert werden und nicht nur am Anfang einer Prozedur oder Funktion. Sinnvollerweise werden aber Variablen am Anfang einer Prozedur oder Funktion deklariert.

```
Sub doEvent()
  Dim myName As Integer
End Sub
```

Jede Zeile Ihres Quelltextes kann verschiedene Deklarationen für mehrere Variablen beinhalten. Welchen Datentyp eine Variable erhält, hängt davon ab, wie Sie Ihre Variable deklarieren. Wenn Sie für jede Variable explizit einen Datentyp angeben, wird nicht der Standarddatentyp für die Variable herangezogen. Dieser Standarddatentyp hängt davon ab, in welchem Sprachmodus sich Ihr Quelltext gerade befindet. Im 'KBasic Mode' und 'OldBasic Mode' ist es 'Variant', in 'VeryOldBasic' ist es 'Double'.

```
Dim firstname, surname, lastname As String
```

name ist Typ 'Variant'
surname ist Typ 'Variant'
lastname ist Typ 'String'

Oder explizit mit verschiedenen Typen:

```
Dim myName As String, age As Integer
```

Wenn Sie einen bestimmten Datentyp verwenden möchten, müssen Sie Ihn für jede Variable angeben, die Sie deklarieren, ansonsten wird der Standarddatentyp verwendet. In der folgenden Anweisung werden alle Variablen als 'Integer' deklariert.

```
Dim intX As Integer, intY As Integer, intZ As Integer
```

In der folgenden Anweisung ist intX und intY als 'Variant' und intZ als 'Integer' deklariert.

```
Dim intX, intY, intZ As Integer
```

Variablen können einen Startwert (Initialwert) haben. Dafür gibt es zwei Möglichkeiten:

Entweder

```
Dim name = „sunny sun“ As String ' style not recommended
```

oder

```
Dim name As String = „sunny sun“ ' recommended
```

Wenn viele Variablen in einer Zeile deklariert sind, wie im folgenden Beispiel, hat nur lastname einen Startwert zugewiesen bekommen („sunny sun“).

```
Dim name, surname, lastname As String = „sunny sun“
```

Wenn kein Wert bei der Deklaration zugewiesen wird, wird der Standardwert benutzt, der normalerweise 0 ist, wenn es eine numerische Variable ist. Wenn die Variable vom Typ 'String' ist, hat der 'String' die Länge 0 (““). 'Object'-Variablen bekommen den Wert 'Null'; 'Variant' ist 'Empty'.

Variablennamen dürfen nicht länger als 128 Zeichen lang sein und dürfen bestimmte Zeichen wie folgt beschrieben nicht beinhalten: Punkt, Komma und Sonderzeichen. Eine Ausnahme ist der Unterstrich (_). Dieser ist erlaubt. Wichtig! Normalerweise unterscheidet KBasic zwischen groß- und kleingeschriebenen Variablennamen, aber um kompatibel zu zukünftigen Versionen zu sein, sollten Sie die Variablennamen immer gleich schreiben. Name, name, naME bezeichnen also normalerweise die gleiche Variable.Aber es existieren auch Ausnahmen. Wenn Sie die Qt-Anbindung benutzen, müssen Sie immer den Variablen- oder Methodennamen so schreiben wie dieser in der Dokumentation angeben ist.

Beachten Sie die folgenden Regeln, wenn Sie Prozeduren, Funktionen, Argumente, Konstanten und Variablen in KBasic benennen:

- Benutzen Sie einen Buchstaben (A-Z, a- z) oder Unterstrich (_) als erstes Zeichen
- Benutzen Sie nicht Leerzeichen ( ), Punkt (.), Ausrufezeichen (!), oder eines der folgenden Zeichen bzw. Sonderzeichen wie z. B. @, &, $, #, „ in Ihren Variablennamen.
- Der Name darf Ziffern beinhalten. Das erste Zeichen darf aber keine Ziffer sein.
- Im allgemeinen: Benutzen Sie keine Namen, die bereits in KBasic vordefiniert sind wie z. B. Schlüsselwörter ('If' etc.), Builtins ('Print' etc.), Klassennamen etc., weil dies sonst zu einer Namenskollision führen würde, wenn das Element der KBasic-Sprache statt der Variable erwartet werden würde.
- Sie können keine Variable mit demselben Namen im gleichen Sichtbarkeitsbereich verwenden, z. B. können Sie keine Variable zweimal in der gleichen Prozedur deklarieren, aber Sie können eine Variable mit demselben Namen z. B. in einem Modul und in einer Prozedur deklarieren.
- Sie dürfen keine reservierten Wörter wie z. B. Schlüsselwörter oder Builtins ('Print' etc.) verwenden.

Deklaration von Variablen in verschiedenen Sichtbarkeitsbereichen

Eine Deklarationsanweisung kann innerhalb einer Prozedur/Funktion/Methode zur Erstellung einer Variablen auf Prozedurebene oder zu Beginn eines Moduls oder Klasse im Deklarationsabschnitt zur Erstellung einer Variablen auf Modul-/Klassenebene plaziert werden und damit der Gültigkeitsbereich der Variablen bestimmt werden.

Der Gültigkeitsbereich einer Variablen kann während der Ausführung des Codes nicht verändert werden. Sie können jedoch Variablen mit denselben Namen und unterschiedlichen Gültigkeitsbereichen deklarieren. Deklarieren Sie z. B. eine globale Variable mit dem Namen 'Bus' und anschließend innerhalb einer Prozedur eine lokale Variable mit demselben Namen 'Bus'. Referenzen innerhalb der Prozedur auf den Namen 'Bus' greifen auf die lokale Variable zu; Referenzen außerhalb der Prozedur verweisen dann auf die globale Variable.

In dem folgenden Beispiel wird die Variable sName erstellt und der Datentyp 'String' angegeben.

```
Dim sName As String
```

Ist diese Anweisung Teil einer Prozedur, so kann die Variable sName nur in dieser Prozedur verwendet werden. Befindet sich diese Anweisung im Deklarationsabschnitt eines Moduls, so ist die Variable sName für alle Prozeduren innerhalb dieses Moduls, nicht aber für Prozeduren in anderen Modulen des Projekts verfügbar. Damit diese Variable für alle Prozeduren des Projekts verfügbar ist, stellen Sie ihr die 'Public'-Anweisung voran. Beispiel:

```
Public sName As String
```

Verwenden der 'Public'-Anweisung

Sie können die 'Public'-Anweisung zur Deklaration öffentlicher Variablen auf Modul-/Klassenebene verwenden.

```
Public sName As String
```

Verwenden der 'Private'-Anweisung

Sie können die 'Private'-Anweisung zur Deklaration privater Variablen auf Modul-/Klassenebene verwenden, was bedeutet, dass diese nur im gleichen Sichtbarkeitsbereich verfügbar sind.

```
Private myName As String
```

Private Variablen können nur von Prozeduren/Funktionen/Methoden des gleichen Moduls oder der gleichen Klasse verwendet werden.

Wird die 'Dim'-Anweisung auf Modul-/Klassenebene verwendet, gleicht sie der 'Private'-Anweisung. Möglicherweise möchten Sie die 'Private'-Anweisung verwenden, um die Lesbarkeit des Codes zu erhöhen und die Interpretation zu vereinfachen.

Verwenden der 'Protected'-Anweisung

Sie können die 'Protected'-Anweisung zur Deklaration von 'Protected'-Variablen auf Klassenebene verwenden. Diese Variablen können nur von Methoden der gleichen Klasse und Unterklasse verwendet werden und dienen dazu, die Vererbungshierarchie in den Klassen zu unterstützen. Normalerweise, verwendet man diese nur in sehr großen Projekten.

```
Protected myName As String
```

Verwenden der 'Static'-Anweisung

'Static' hat drei Bedeutungen, die abhängig vom Context sind.

- *'Static' innerhalb einer Klasse aber ausserhalb einer Methode*
  Wenn Sie die 'Static'-Anweisung anstelle einer 'Dim'-Anweisung verwenden, so wird die Variable als Klassenvariable deklariert und kann somit ohne Instanzen dieser Klasse verwendet werden und existiert damit auch nur einmal im Speicher während der Programmausführung.
- *'Static' ausserhalb einer Klasse in einer Prozedur/Funktion/Methode*
  Wenn Sie die 'Static'-Anweisung anstelle einer 'Dim'-Anweisung verwenden, so verliert diese Variable zwischen den Prozedur-/Funktionsaufrufen nicht ihren Wert. Diese ist bei Rekursivaufrufen einer Prozedur nur einmal vorhanden.
- *'Static' ausserhalb einer Klasse vor einer Prozedur/Funktion/Methode*
  Wenn sie alle lokalen Variablen einer Prozedur zu statischen Variablen machen möchten, geben sie an den Anfang der Prozedurdeklaration das reservierte Wort 'Static' an. Dies deklariert alle lokalen Variablen der Prozedur als statische Variablen, unabhängig davon, ob sie mit der Anweisung 'Dim' oder dem reservierten Wort 'Static' deklariert werden. Geben sie das reservierte Wort 'Static' vor einer beliebigen Funktions- oder Prozedurdeklaration an, auch wenn diese mit 'Private', 'Protected' oder 'Public' deklariert ist.

Automatische Deklaration von Variablen / Implizite Deklaration

Aus historischen Gründen wird auch ein Modus innerhalb KBasic unterstützt, der es erlaubt, Variablen nicht deklarieren zu müssen. In diesem Modus wird automatisch eine Variable angelegt, wenn sie in ihrem Code einen Namen verwenden, der keinem Namen einer existierenden Variablen entspricht. Es ist jedoch dringend zu empfehlen, alle Variablen explizit zu deklarieren und gleichzeitig deren Datentyp festzulegen. Wenn Sie ihre Variablen deklarieren, wird die Fehlersuche in ihrem Code wesentlich vereinfacht und das Problem von Schreibfehlern in Variablennamen vermieden.

Zum Abschalten der Deklarationspflicht von Variablen verwenden Sie folgende Codezeilen:

```
Option OldBasic
Option Explicit Off
```

Verwenden der Option Explicit-Anweisung
(nur gültig in 'Option OldBasic' und 'Option VeryOldBasic')

Sie können somit implizit eine Variable in KBasic deklarieren, indem Sie diese in einer Zuweisungsanweisung verwenden. Alle implizit deklarierten Variablen sind vom Typ 'Variant' ('Option OldBasic') oder 'Double' ('Option VeryOldBasic'). Variable des Typs 'Variant' erfordern mehr Speicherressourcen als die meisten anderen Variablen. Ihre Anwendung ist effizienter, wenn Sie Variablen explizit und mit einem bestimmten Datentyp deklarieren. Das explizite Deklarieren aller Variablen verringert das Auftreten von Fehlern aufgrund von Namenskonflikten oder Schreibfehlern.

Wenn Sie in KBasic keine impliziten Deklarationen verwenden möchten, setzen Sie die 'Option Explicit'-Anweisung in einem/r Modul/Klasse vor alle anderen Prozeduren/Funktionen/Methoden. Diese Anweisung erzwingt die explizite Deklaration aller Variablen innerhalb des/r Moduls/Klasse. Enthält ein Modul/Klasse die 'Option Explicit'-Anweisung, so tritt ein Fehler zur Kompilierungszeit auf, wenn KBasic auf einen Variablennamen trifft, der noch nicht deklariert oder falsch geschrieben wurde.

Wenn sie im 'Option KBasic' arbeiten, sind alle Variablen zu deklarieren. Es ist also 'Explicit On' standardmäßig eingestellt, da 'Option KBasic' standardmäßig eingestellt ist.

Hinweis: Sie müssen feste und dynamische Datenfelder immer explizit deklarieren. Sie können 'Option OldBasic' und 'Option KBasic' mischen, indem sie eine/ein Klasse/Modul als 'Option OldBasic' und die andere als 'Option KBasic' deklarieren und im Bereich 'Option OldBasic' 'Explicit Off' eingestellt haben. Es wird aber ausdrücklich empfohlen immer die Variablen explizit zu deklarieren.

Lokale Variablen

Der Wert einer lokalen Variablen in einer Prozedur ist nur lokal für diese Prozedur vorhanden, sie können nicht aus einer Prozedur auf eine Variable einer anderen Prozedur zugreifen. Sie können daher dieselben Variablennamen in verschiedenen Prozeduren verwenden, ohne dass es zu Konflikten oder versehentlichen Änderungen kommt.

Beispiel:

```
Sub test1()
  Dim i As Integer

  i = 1
End Sub

Sub test2()
  Dim i As Integer

  i = 19
End Sub
```

Zuweisungsanweisungen

Zuweisungsanweisungen weisen einen Wert oder einen Ausdruck einer Variablen oder Konstanten/Eigenschaften zu. Zuweisungsanweisungen enthalten immer ein Gleichheitszeichen (=). Das folgende Beispiel weist den Rückgabewert der 'InputBox'-Funktion der Variablen yourName zu.

```
Dim yourName As String
yourName = InputBox("What is your name?")
MsgBox "Your name is " & yourName
```

Die 'Let'-Anweisung ist optional und wird normalerweise nicht verwendet. 'Let' existiert aus historischen Gründen. Die oben aufgeführte Zuweisungsanweisung kann z. B. folgendermaßen geschrieben werden.

```
Let yourName = InputBox("What is your name?")
```

Die 'Set'-Anweisung wird in VB6 verwendet, um ein Objekt einer Variablen zuzuweisen, die als Objekt deklariert wurde. In KBasic ist 'Set' überflüssig. Das Schlüsselwort 'Set' ist nicht erforderlich und nicht erlaubt.

Wurde also eine Variable deklariert, kann man ihr einen Wert zuweisen mittels des Zuweisungsoperators (=).

```
Dim Cool As Boolean
Dim myName As String

Cool = True
Name = „Julie“
```

Cool enthält dann 'True' und Name dann 'Julie'.

Lebensdauer von Variablen

Die Zeit, in der eine Variable ihren Wert behält oder existiert, wird Lebensdauer genannt. Der Wert einer Variablen kann sich während ihrer Lebensdauer ändern, sie behält aber irgendeinen Wert. Verlässt eine Variable ihren Gültigkeitsbereich, so verfügt sie über keinen Wert mehr.
Wenn eine Prozedur ausgeführt wird, werden alle Variablen dieser Prozedur bei der entsprechenden 'Dim'-Anweisung initialisiert. Jedes Element einer Variablen eines benutzerdefinierten Typs wird so initialisiert, als wäre es eine separate Variable. Diese lokalen Variablen existieren nur für die Ausführungsdauer der Prozedur. Wenn die Prozedur beendet ist, wird der Wert der Variablen wieder gelöscht. Beim nächsten Ausführen der Prozedur werden alle lokalen Variablen der Prozedur neu initialisiert. Damit eine Variable ihren Wert auch nach Beendigung der Prozedur beibehält, deklarieren sie diese mit Hilfe der Anweisung 'Static'.

Wenn Sie eine Objektvariable deklarieren, wird Platz im Speicher reserviert, der Wert der Objektvariablen wird aber auf 'Null' ('Nothing') gesetzt, bis Sie einen Objektverweis zuweisen.
Wenn der Wert der Variablen während der Ausführung Ihres Codes nicht verändert wird, behält sie ihren initialisierten Wert, bis sie ihren Gültigkeitsbereich verläßt.
Eine auf Prozedurebene mit der 'Dim'-Anweisung deklarierte Variable behält also einen Wert, bis die Ausführung der Prozedur beendet wird. Ruft die Prozedur andere Prozeduren auf, behält die Variable während der Ausführung dieser Prozeduren ihren Wert.

Bei 'Static'-deklarierten Prozedurvariablen in Modulen existiert diese Variable genau einmal, unabhängig vom rekursiven Aufruf dieser Prozedur. Wird eine Variable also auf Prozedurebene mit dem Schlüsselwort 'Static' deklariert, behält die Variable ihren Wert so lange, wie Code in einem beliebigen Modul ausgeführt wird. Wenn die Ausführung des gesamten Codes beendet wird, verliert die Variable ihren Gültigkeitsbereich und ihren Wert. Ihre Lebensdauer entspricht der Lebensdauer einer Variablen auf Modulebene.
Eine Variable auf Modulebene unterscheidet sich von einer statischen Variablen. In einem Standardmodul oder einem Klassenmodul behält sie ihren Wert, bis die Ausführung des Codes angehalten wird. In einem Klassenmodul als Instanzvariable behält sie ihren Wert, solange eine Instanz der Klasse existiert. Klassenvariablen, die mit 'Static' in der Klasse deklariert wurden, behalten ihren Wert bis zum Ende des Programms. Variablen auf Klassen-/Modulebene benötigen Speicherressourcen, bis Sie deren Werte zurücksetzen; darum sollten Sie diese nur dann verwenden, wenn es unbedingt notwendig ist.

Wenn Sie das Schlüsselwort 'Static' vor einer Prozedur- oder Funktion-Anweisung verwenden, werden die Werte aller Variablen auf Prozedurebene in der Prozedur zwischen den Aufrufen beibehalten.

Deklarationsort

Sie sollten in KBasic jede Variable und jede Methode in einer Klasse oder zumindest in einem Modul definieren. Damit kann jede KBasic-Variable und KBasic-Methode durch ihren vollqualifizierten Namen, der aus Klassenname und Feldname besteht (sowie der Variablen oder Methodename) - alle durch Punkte getrennt – referenziert werden.

Syntax:

```
Dim Name[([Index])] [As Type] [, Name[([Index])] [As Type]] ...

Dim Name [= Expression] [As Type]

Dim Name [As Type] [= Expression]

[Public | Protected | Private | Dim | Static] Name [= Expression]
[As Type]
```

Variablenbezeichnungen

Unter Programmierern hat sich eine mehr oder weniger einheitliche Schreibweise für Programmcode durchgesetzt. Wenn Sie sich an diese Konventionen halten, erleichtern Sie es anderen, Ihren Code zu lesen und weiterzuverwenden. Zudem vereinfachen Sie damit aber auch sich selbst die Arbeit. Variablennamen beginnen typischerweise mit einem Kleinbuchstaben, der zweite Buchstabe ist groß. Dabei steht der Anfangsbuchstabe als Kürzel für den Datentyp:

Ein „o“ bezeichnet eine Objektvariable, zum Beispiel „oDokument1“.
Ein „i“ oder „n“ steht bei einer numerischen Variablen für Ganzzahlen ('Integer').
Ein „f“ oder „d“ kennzeichnet eine numerische Variable für Gleitkommazahlen ('Double').
Ein „b“ wird bei Wahrheitsvariablen wie beispielsweise „bZustand1“ verwendet.
Ein „s“ bezeichnet eine Stringvariable.
Ein „t“ kennzeichnet eine Datumsvariable.
Ein „v“ wird bei Variablen ohne speziellen Datentyp verwendet ('Variant')

Datentypen

Datentypen beschreiben die Art der gespeicherten Daten. Neben dem Namen muss eine Variable auch einen Typ haben. KBasic unterstützt viele VB6-Datentypen und darüber hinaus noch viele mehr. Mögliche Typen sind:

- einfacher Datentyp (z. B. 'Integer')
- Name einer Klasse (ob selbstdefinierte Klasse, Qt-Anbindung oder KBasic-Klasse)
- benutzerdefinierter Datentyp
- benutzerdefinierte Aufzählung

Einfache Datentypen

Dabei handelt es sich um Zahlen und Texte. Sie werden Einfache genannt, weil sie in KBasic fest eingebaut sind und keine Klassen darstellen. Jeder der verschiedenen Datentypen hat einen verschiedenen Zahlenbereich. Wenn ein Wert zu groß für eine Variable wird, wird er einfach abgeschnitten.

Die folgende Tabelle enthält die von KBasic unterstützten Datentypen sowie deren Speicherbedarf und Wertebereiche.

Datentyp / Speicherbedarf in Bytes / Wertebereich

- *Boolean* / 1 / 'True' oder 'False'
  Variablen vom Datentyp 'Boolean' werden als 8-Bit-Zahlen (1 Byte) gespeichert, die nur die Werte 'True' oder 'False' annehmen können. Mit den Schlüsselwörtern 'True' und 'False' weisen Sie einer Variablen vom Typ 'Boolean' einen von zwei Zuständen zu.
- *Byte* / 1 / 0..255
  Variablen vom Datentyp 'Byte' werden als einzelne 8-Bit-Zahlen (1 Byte) ohne Vorzeichen gespeichert und haben einen Wert im Bereich von 0 bis 255.
- *Short* / 2 / -2^15 bis 2^15-1
  Variablen vom Datentyp 'Short' werden als 16-Bit-Zahlen (2 Bytes) in einem Bereich von -32.768 bis 32.767 gespeichert.
- *Integer* / 4 / -2^31 bis 2^31 –1
  Variablen vom Datentyp 'Integer' werden als 32-Bit-Zahlen (4 Bytes) in einem Bereich von -2147483648 bis 2147483647 gespeichert. Das Typkennzeichen für 'Integer' ist das Zeichen (%). Mit Variablen vom Datentyp 'Integer' können Sie auch Aufzählungswerte darstellen.
- *Long* / 8 / -2^63 bis 2^63 –1
  Variablen vom Datentyp 'Long' (lange Ganzzahl) werden als 64-Bit-Zahlen (8 Bytes) mit Vorzeichen gespeichert. Das Typkennzeichen für 'Long' ist das Zeichen (&)
- *String / je nach Länge / unendlich*
  Es gibt zwei Arten von Zeichenfolgen: Zeichenfolgen variabler Länge und Zeichenfolgen fester Länge. Zeichenfolgen variabler Länge können bis zu 2 Milliarden (oder 2^31) Zeichen enthalten. Standardmäßig sind Zeichenfolgenvariablen oder -argumente Zeichenfolgen variabler Länge; die 'String'-Variable vergrößert oder verkleinert sich, wenn sie neue Werte zuweisen. Sie können außerdem Zeichenfolgen mit fester Länge deklarieren. Das Typkennzeichen für 'String' ist das Zeichen ($). Die Codes für Zeichen vom Datentyp 'String' liegen im Bereich von 0 bis 255 (einschließlich). Die ersten 128 Zeichen (0 bis 127) entsprechen den Buchstaben und Symbolen auf einer US-amerikanischen Standardtastatur. Diese ersten 128 Zeichen stimmen mit den im ASCII-Zeichensatz definierten Zeichen überein. Die zweiten 128 Zeichen (128 bis 255) sind Sonderzeichen, z. B. Buchstaben aus internationalen Alphabeten, Akzentzeichen, Währungssymbole und Symbole für mathematische Brüche. Alle Zeichen dürfen innerhalb eines Strings stehen, z. B.
  ```
  myStringVar = „1234 Names symbols .!/& keywords IF ? - “
  ```

- *String fester Länge / Länge*
  Zeichenfolgen fester Länge können bis zu 2 Milliarden Zeichen (oder 2^31) Zeichen enthalten. Diese finden nur in einer 'Type'...'End Type'-Struktur Verwendung.
- *Character* / 2 /
  ist reserviert, wird in Zukunft unterstützt werden
- *Single* / 4 /
  Variablen vom Datentyp Single (Gleitkommazahl mit einfacher Genauigkeit) werden als 32-Bit-Gleitkommazahlen (4 Bytes) nach IEEE im Bereich von -3,402823E38 bis -1,401298E-45 für negative Werte und von 1,401298E-45 bis 3,402823E38 für positive Werte gespeichert. Das Typkennzeichen für Single ist das Ausrufezeichen (!).
- *Double* / 8 /
  Variablen vom Datentyp Double (Gleitkommazahl mit doppelter Genauigkeit) werden als 64-Bit-Gleitkommazahlen (8 Bytes) nach IEEE im Bereich von -1,79769313486232E308 bis -4,94065645841247E-324 für negative Werte und von 4,94065645841247E-324 bis 1,79769313486232E308 für positive Werte gespeichert. Das Typkennzeichen für Double ist das Zeichen (#).
- *Object / 4 / + Größe des Objects*
  Variablen vom Datentyp 'Object' werden als 32-Bit-Adressen (4 Bytes) gespeichert, die auf Objekte in einer Anwendung verweisen. Einer Variablen, die als 'Object' deklariert wurde, kann anschließend mit der '='-Anweisung ein Verweis auf jedes von der Anwendung erzeugte Objekt zugewiesen werden.
- *Date* / 8
  Wird als Zahl gespeichert (8 Bytes). Ein Datumsliteral muss durch das Zeichen (#) eingeschlossen sein, zum Beispiel: #1993-12-31# .
- *Currency* / 8
  Variablen vom Datentyp Currency werden als 64-Bit-Zahlen (8 Bytes) in einem ganzzahligen Format gespeichert und durch 10.000 dividiert, was eine Festkommazahl mit 15 Vorkomma- und 4 Nachkommastellen ergibt. Diese Darstellung ergibt einen Wertebereich von -922.337.203.685.477,5808 bis 922.337.203.685.477,5807. Das Typkennzeichen für Currency ist das Zeichen (@). Der Datentyp 'Currency' ist für Berechnungen mit Geldbeträgen und für Festkommaberechnungen, die eine hohe Genauigkeit erfordern, gut geeignet.

- *Variant* / 40
  Ein Wert vom Typ 'Variant', der ein Datenfeld enthält, benötigt 40 Bytes zusätzlich zu dem Speicher, der für das Datenfeld alleine benötigt wird.
- *Klassen* /
  Siehe 'Object'
- benutzerdefinierter Datentyp / siehe weiter im Text

Benutzerdefinierte Typen

Diese sind ein Relikt aus der Vor-Objektorientierten Zeit und Verbund von Variablen verschiedenen Typs ähnlich einer Klasse, aber ohne Methoden. Dabei können benutzerdefinierte Typen selbst benutzerdefinierte Typen beinhalten, neben den einfachen Datentypen auch Objektenreferenzen.

Syntax:

```
Type Name
 Name [(Index)] As Type
 Name As Type
  ...
End Type
```

Klassentypen/Objekte:

Variable können Objekte speichern, genauer ausgedrückt, Verweise auf Objekte speichern. Verweise werden auch Referenzen genannt. Variable vom Datentyp 'Object' werden als 32-Bit-Adressen (4 Bytes) gespeichert, die auf Objekte in einer Anwendung verweisen. Bei einer Variablen, die als 'Object' deklariert wurde, muss nicht (anders als in VB6) anschließend mit der 'Set'-Anweisung ein Verweis auf das Objekt zugewiesen werden. In KBasic genügt die einfache '='-Anweisung.

Obwohl eine Variable, die mit dem Datentyp 'Object' deklariert wurde, damit einen Verweis auf jedes beliebige Objekt enthalten kann, erfolgt die Verbindung mit dem Objekt, auf das verwiesen wurde, fast immer zur Laufzeit (Binden zur Laufzeit). Sie können eine frühere Bindung (Binden zur Kompilierungszeit) erzwingen, indem Sie den Objektverweis einer Variablen zuweisen, die durch den Namen einer Klasse deklariert wurde. Das gleiche können Sie mit einer Typumwandlungsanweisung erreichen.

Typ Variant

Der Datentyp 'Variant' wird automatisch festgelegt, wenn Sie bei der Deklaration einer Konstanten, einer Variablen oder eines Arguments keinen Datentyp angegeben haben. Einzige Ausnahme: im Modus 'VeryOldBasic' wird standardmäßig 'Double' statt 'Variant' festgelegt. Variablen vom Datentyp 'Variant' können Zeichenfolgen, Datums-, Zeit-, boolesche oder numerische Werte enthalten und diese Werte automatisch umwandeln. Der 'Variant'-Datentyp speichert somit alle möglichen Datentypen. Eine Variant-Variable benötigen mehr Speicherplatz (was nur für große Prozeduren oder komplexe Klassen/Module von Bedeutung ist), und der Zugriff auf diese Werte erfolgt langsamer, als wenn explizit eine Variable eines anderen Datentyps angegeben worden wären. Deshalb sollten Sie die Verwendung des Variantdatentyps vermeiden.

Das folgende Beispiel erstellt drei Variablen vom Datentyp 'Variant':

```
hisVar = 3

Dim myVar
```

```
Dim yourVar As Variant
```

Die erste Anweisung impliziert eine Variantvariable automatisch (nur innerhalb von 'Option OldBasic' und 'Option Explicit Off').

Zusätzlich kann 'Variant' die folgenden Werte speichern:

- 'Null' ('Nothing) – 'Nothing' wird aus Kompatibilitätsgründen unterstützt
- 'Empty'

Variablensuffix

Variablennamen können auch über den Datentyp der Variablen etwas aussagen. Früher wurden im Variablennamen gleich der Datentyp mittels bestimmter Symbole am Ende des Variablennames mit angeben. KBasic unterstüzt diese alte Schreibweise aus Kompatibilitätsgründen. Statt der Suffixe sollten Sie besser am Anfang des Variablennamens einen Buchstaben zur Kennzeichnung des Variablentyps verwenden (moderner Stil). Folgende Typkennzeichen werden unterstützt:

Beispiele:

```
Dim i% ' Integer variable
Dim l& ' Long variable
Dim s$ ' String variable
```

Die folgenden Symbole werden unterstützt.

- *Integer*
  Das Typkennzeichen für Integer ist das Zeichen (%).
- Long
  Das Typkennzeichen für Long ist das Zeichen (&).
- String
  Das Typkennzeichen für String ist das Dollarzeichen ($).
- Single
  Das Typkennzeichen für Single ist das Ausrufezeichen (!).
- Double
  Das Typkennzeichen für Double ist das Zeichen (#).
- Currency
  Das Typkennzeichen für Currency ist das Zeichen (@).

Kommentare

An vielen Stellen in den Beispielen in diesem Buch wird das Kommantersymbol (') verwendet. Kommentare können eine Prozedur/Funktion oder eine bestimmte Anweisung erklären. KBasic ignoriert Kommentare beim Ausführen der Prozeduren. Kommentare können überall eingefügt werden. Einen Kommentar können Sie z. B. folgendermaßen einfügen: Geben Sie einen Apostroph gefolgt von einem Kommentar am Ende der Anweisungzeile ein, um diesen Kommentar auf die gleiche Zeile zu setzen wie die Anweisungen. Standardmäßig werden Kommentare als grüner Text angezeigt.

KBasic kennt vier Arten von Kommentaren. Das Symbol (‘) und 'REM' z. B.

```
REM this is a comment

‘ this is a comment, too
```

und wie in Java

```
/* comment begin
and comment end */

/** docu comment
begin and docu
comment end */
```

Kommentare werden von KBasic überlesen und dienen dem Software-Entwickler zu Dokumentationszwecken, machen somit die Arbeit leichter und helfen Personen, die sich den Code zu einem späteren Zeitpunkt anschauen, das Verständins zu erleichtern. Kommentare beschreiben normalerweise, wie der Code arbeitet.

Benennungskonventionen allgemein

Wenn Sie Code in KBasic schreiben, deklarieren und benennen Sie viele Elemente (Sub-Prozeduren und Funktionen, Variablen und Konstanten und so weiter). Für die Namen der Prozeduren, Variablen und Konstanten etc., die Sie in Ihrem KBasic-Code deklarieren, gilt:

- sie müssen mit einem Buchstaben oder dem Unterstreichungszeichen (_) beginnen
- sie dürfen nur aus Buchstaben, Ziffern und dem Unterstreichungszeichen (_) bestehen; Satz- und Leerzeichen sind nicht erlaubt
- sie dürfen eine bestimmte Länge nicht überschreiten
- es dürfen keine reservierten Worte verwendet werden

Ein reserviertes Wort ist ein Wort, das KBasic als Teil seiner Sprache verwendet. Hierzu zählen vordefinerte Anweisungen (z. B. 'If' und 'Then'), Funktionen (z. B. 'Mid') und Operatoren (z. B. 'Mod'). Eine vollständige Liste dieser Worte finden Sie am Ende des Buches.

Literale

Neben Schlüsselwörtern, Symbolen und Namen besteht ein KBasicprogramm auch aus Literalen. Ein Literal ist eine Zeichenfolge, die einen Wert repräsentiert. Literale sind z. B. 12, „Hi“ oder 'True'.

Es gibt mehrere verschiedene Arten von Literalen:

- *Byte, Short, Integer, Long*
  `1, 2, -44, 4453`
- *Hex*
  `&HAA43`
- *Binär*
  `&B11110001`
- *Octal*
  `&O1234`
- *Single (Dezimal)*
  Hier wird die englische Schreibweise verwendet: Trennzeichen ist der Punkt.
  `21.32, 0.344, -435.235421.21`
- Double *(Dezimal)*
  Hier wird die englische Schreibweise verwendet: Trennzeichen ist der Punkt.
  `212.23`
- Currency
  Hier wird die englische Schreibweise verwendet.
  `45.3@`
- *Date / Time*
  Hier wird die englische Schreibweise verwendet:
  Datumsliteral #yyyy-mm-dd# / Zeitliteral #hh:mm:ss#
- *String*
  Ein beliebige Zeichenfolge, wobei das erste und letzte Zeichen ein (“) ist, somit Anfang und Ende bestimmt. Ein doppeltes (“) also (““) innerhalb einer Zeichenfolge wird als (“) interpretiert. Es gibt keine Fluchtzeichen wie in C++. Diese werden mittels Stringfunktionen in KBasic abgebildet. z. B. “hello“.
  Für ein (“) innerhalb eines Strings müssen Sie zweimal (“) angeben:
  `"ein ""lustiges"""` Stück wird zu `"ein "lustiges" Stück`

- *Boolean*
  Entweder '`True`' oder '`False`'
  Beim Umwandeln anderer numerischer Datentypen in Werte des Typs 'Boolean' wird 0 zu 'False', und alle anderen Werte werden zu 'True'. Beim Umwandeln von Werten des Datentyps Boolean in andere Datentypen wird 'False' zu 0 und 'True' zu -1 oder 1 (je nach Kontext).

Ausdrücke

Ein Ausdruck symbolisiert einen Wert in KBasic. Er kann Schlüsselwörter, Operatoren, Variablen, Konstanten und selbst Ausdrücke enthalten. Beispiele für Ausdrücke sind:

- 1 + 9 (numerische Zahl Operator numerische Zahl)
- myVar (Variable)
- myVar TypeOf Object (Variable Schlüsselwort Klassenname)

Ausdrücke können einen Wert zurückgeben oder nicht: Wenn ein Ausdruck rechts der '='-Anweisung steht oder berechnet wird, muss/gibt der Ausdruck einen Wert zurückgeben.

```
myVar = 1 + 9
```

1 + 9 ergibt 10 und dieser Ausdruckswert wird in myVar abgespeichert.

Es ist genauso, wenn der Ausdruck für einen Parameter im Prozeduraufruf steht.

```
MyProcedure(1 + 9)
```

Ausdrücke, wenn sie berechnet werden, ergeben somit einen Rückgabewert. Das Ergebnis kann einer Variablen zugewiesen werden oder wie im zweiten Beispiel als Prozedurparameter verwendet werden. Ausdrücke werden in der Regel zur Laufzeit Ihres Programms berechnet.

Konstanten

Konstanten haben, im Gegensatz zu Variablen, einen konstanten, d. h. nicht veränderlichen Wert. Durch die Deklaration einer Konstanten können Sie einem Wert einen aussagefähigen Namen zuweisen. Verwenden Sie die 'Const'-Anweisung um eine Konstante zu deklarieren und deren Wert festzulegen. Nachdem eine Konstante deklariert wurde, kann sie nicht mehr geändert werden, und es kann ihr kein neuer Wert zugewiesen werden. Beispiel:

```
Const border As Integer = 377
```

Die 'Const'-Anweisung deklariert die Konstante border und gibt den Datentyp 'Integer' und einen Wert von 337 an.

Es gelten dieselben Regeln wie für das Erstellen von Variablennamen. Eine 'Const'-Anweisung besitzt genau wie eine Variablendeklaration, einen Gültigkeitsbereich. Um eine Konstante zu erstellen, die nur innerhalb einer Prozedur existiert, deklarieren Sie sie in dieser Prozedur. Um eine Konstante zu erstellen, die allen Prozeduren innerhalb eines Moduls zur Verfügung stehen soll, deklarieren Sie diese im Deklarationsbereich des Moduls. Sie können somit eine Konstante innerhalb einer Prozedur oder zu Beginn einer/s Klasse/Moduls im Deklarationsabschnitt deklarieren. Konstanten auf Klassen-/Modulebene sind standardmäßig privat. Stellen Sie der 'Const'-Anweisung das Schlüsselwort 'Public' voran, um eine öffentliche Konstante auf Klassen-/Modulebene zu deklarieren. Sie deklarieren eine private Konstante explizit, indem Sie der 'Const'-Anweisung das Schlüsselwort 'Private' voranstellen, womit Ihr Code einfacher zu lesen und zu interpretieren ist. Weitere Informationen finden Sie unter dem Thema "Gültigkeitsbereich".

In dem folgenden Beispiel wird die 'Public'-Konstante conAge als 'Integer' deklariert und ihr der Wert 34 zugewiesen.

```
Public Const conAge As Integer = 34
```

Konstanten können als einfache Datentypen deklariert werden als eine der folgenden Datentypen: Boolean, Byte, Short, Integer, Long, Currency, Single, Double, Date, String oder Variant. Da Sie den Wert einer Konstanten bereits kennen, können Sie den Datentyp in einer 'Const'-Anweisung angeben.

Sie können mehrere Konstanten in einer Anweisungszeile deklarieren:

```
Const conAge As Integer = 39, conSalary As Currency = 3500
```

Syntax:

```
Const Name = Expression

Const Name [As Type] = Expression [, Name [As Type] = Expression]
...

[Public | Protected | Private] Const Name [As Type] = Expression
```

Operatoren und Operanden

KBasic unterstüzt alle VB6-Operatoren und darüberhinaus noch viele mehr. Operatoren sind z. B. '+' oder '-' , kurzum, ein Operator führt eine Operation aus. Operanden sind die Zahlen, Strings oder Variablen, mit denen etwas gemacht wird. KBasic unterstützt Operatorenüberladen in den Qt-Anbindungen. In Zukunft wird es auch möglich sein, innerhalb von KBasic-Klassen Operatorüberladungen vorzunehmen, genauso wie in C++.

Es gibt verschiedene Arten von Operatoren.

Berechnungsoperatoren

Diese dienen zur Durchführung arithmetischer Berechnungen mit zwei Operanden. Jeder dieser Operatoren braucht zwei Operanden. Einer links, einer rechts vom Operator. Ausnahme: Der Subtraktionsoperator (-) kann zum Negieren verwendet werden, z. B. -9 bedeutet negative 9; +9 bedeutet positive 9; 9 bedeutet positive 9

Berechnungsoperatoren sind:

- *+ Addition*
  Dient zur Addition zweier Zahlen. Auch Zeichenketten können addiert werden. Wenn Sie aber dafür den Operator '+' verwenden, können Sie nicht immer bestimmen, ob eine Addition oder eine Zeichenverkettung erfolgt. Für die Verkettung zweier Zeichenfolgen sollten Sie deshalb den Operator '&' verwenden, um Mehrdeutigkeiten auszuschließen und Code zu erstellen, der sich selbst dokumentiert.
  z. B. `3 + 4  ( = 7)`
- *- Subtraktion*
  Dient zum Bilden der Differenz von zwei Zahlen oder zum Bilden des negativen Werts eines numerischen Ausdrucks.
  z. B. `5 - 7  ( = -2)`
- ** Multiplikation*
  Dient zur Multiplikation zweier Zahlen.
  z. B. `33 * 7  ( = 231)`
- */ Division*
  Dient zur Division zweier Zahlen und gibt ein Fließkomma-Ergebnis oder eine Ganzzahl zurück, wenn keiner der Operanden ein Fließkommaausdruck ist.
  z. B. 2 / 5 (= 2)
- *\ Integer-Division*
  Dient zur Division zweier Zahlen und gibt ein ganzzahliges Ergebnis zurück. Das Ergebnis ist vom Typ 'Integer'/'Long' ist.
  z. B. `29 / 5  ( = 5)`
- *Mod Modulo, auch Divisionsrest*
  Gibt den Rest einer ganzzahligen Division zweier Zahlen zurück.
  z. B. `45 Mod 10  ( = 5)`

Eine Integerdivision ergibt einen Integerwert. Nachkommastellen werden bei der Integerdivision abgeschnitten. Modulo gibt den Divisionsrest zurück. Z. B. 21 'Mod' 10 ergibt 1. Wenn einer der Datentypen 'Integer' ist, ist das Ergebnis vom Typ 'Integer'. Ist einer 'Long' dann ist der Ergebnistyp auch 'Long'. Kommen Fließkommazahlen mit ins Spiel so ist das Ergebnis eine Fließkommazahl, entweder 'Double' oder 'Single'. Generell bestimmt der größere Datentyp den Ergebnistyp der Operation.

Achtung! Sollten Sie eine Fließkommaergebnis erwarten, aber nur Ganzzahlenausrdrücke verwenden, wird das Ergebnis der Operation vom Typ Ganzzahl sein.

Das Problem "Fließkommazahlen": Hier gibt es in KBasic wie in jeder Sprache eine Unschärfe bei der Berechnung. Das liegt an der Darstellung der Fließkommazahl. Je nachdem wieviel Stellen vorgesehen sind insgesamt und wie diese aufgeteilt sind zwischen Mantisse und Exponent, kommt es früher oder später zu solchen Unschärfen. Wer mit Gleitkommazahlen rechnet, verlässt sich auf das Ergebnis, und das kann man nicht. Daher ist es manchmal besser, eine Fließkommazahl nach Integer zu wandeln, die Berechnung im Integer zu machen und dann die entsprechende Darstellung (Kommastellen) auszuwählen. Das ist aber Sache des Programmierers.

Beispiele:

x ist 10, y ist 4

```
Print x + y ' is 14
Print x - y ' is 6
Print x * y ' is 40

Print x / y ' is 2.5 --> 2 (because, both operands are Integers)
Print 10.0 / 4.0 ' is 2.5

Print x \ y ' is 2
Print x Mod y ' is 5
```

Zuweisungsoperatoren

Die folgenden von C++ bekannten Operatoren werden zur Zeit nur für die Qt-Anbindung unterstützt. Neben den gewöhnlichen Zuweisungsoperator (=) auch die von C++ bekannten, erweiterten Zuweisungsoperatoren im Zusammenhang mit den Anbindungen in den Klassen vollständig.
+=' Addieren-Zuweisen (bsp. var += 1 entspricht var = vara + 1)
-= Subtrahieren-Zuweisen (bsp. var -= 1 entspricht var = vara - 1)
/= Dividieren-Zuweisen (bsp. var /= 1 entspricht var = vara / 1)
*= Multiplizieren-Zuweisen (bsp. var *= 1 entspricht var = vara * 1)

Für die Qt-Anbindung werden noch weitere C/C++ Zuweisungssoperatoren unterstützt, wie z. B. |= Oder-Zuweisen. Weitere Informationen finden Sie in den Dokumentationen zu den Qt-Anbindungen.

Inkrement und Dekrement

Die von C++ bekannten Operatoren '++' und '–' werden zur Zeit nur für Qt-Anbindung unterstützt. Man kann aber stattdessen var = var + 1 bzw. var = var – 1 schreiben. Oder INC(var) (inkrement, also +1) oder DEC(var) (dekrement, also -1) in naher Zukunft benutzen.

Vergleiche

Diese führen zwischen zwei Operanden einen Vergleich durch und liefern ein Ergebnis, das wahr oder falsch ist.
KBasic hat mehrere Operatoren, um Ausdrücke zu vergleichen:

- = *Gleich*
  z. B. x = 3
- <> *Ungleich*
  z. B. x <> y
- < *Kleiner*
  z. B. x < 3
- > *Größer*
  z. B. x > 4
- <= *Kleiner gleich*
  z. B. x <= 4
- >= *Größer gleich*
  z. B. x >= 3

Für die Qt-Anbindung werden noch die entsprechenden C/C++ Vergleichsoperatoren unterstützt, wie z. B. '==' C++-Vergleich.

Logische Operatoren (Boolesche Operatoren)

KBasic unterstützt echte logische Operatoren für Bedingungen wie C/C++ anders als Visual Basic 6.

- *AndAlso*
  beide Bedingungen müssen zutreffen
- *OrElse*
  eine der Bedingungen muss nur zutreffen

Man kann aber auch auf die Bitoperatoren ausweichen.

Bitoperatoren

Diese führen Verknüpfungen zweier Operanden auf Bit-Ebenen durch. Dadurch entsteht entweder das Ergebnis wahr oder falsch.

- *And (Und)*
  e.g. 4 And 6
  Dient zum Durchführen einer logischen Konjunktion zwischen zwei Ausdrücken.Das Argument Ergebnis ist nur dann 'True', wenn beide Ausdrücke 'True' sind.

```
a  | b  || a AND b
---------------
0  | 0  ||    0
0  | 1  ||    0
1  | 0  ||    0
1  | 1  ||    1
```

  Das Ergebnis ist also nur dann 1 / TRUE / wahr, wenn BEIDE Operanden wahr sind.

Beispiel:

```
            dez.   bin.
            4      0100
            6      0110
            ----------
      AND   4      0100
```

und das ist das, was hier passiert (Bitoperation).

Man kann solche Operationen gut zum Maskieren einsetzen oder auch zum Auswählen von einzelnen Bits. Natürlich kann man damit auch „rechnen“, wie obiges Beispiel zeigt.

Dies gilt für die übrigen Bitoperationen in entsprechender Weise. Die Formulierung bei 'Eqv“ kommt als Kurzbeschreibung den Bitoperationen sehr nahe.

- *Or (Oder)*
  e.g. 33 Or 8
  Dient zum Durchführen einer logischen Disjunktion von zwei Ausdrücken. Sobald einer der beiden Ausdrücke 'True' ist, ist Ergebnis ebenfalls 'True'.

- *Xor (exklusives Oder)*
  e.g. 77 Xor 3
  Wenn genau ein Ausdruck 'True' ist, so ist Ergebnis ebenfalls 'True'.
- *Not (Negation)*
  e.g. Not 5
  Führt eine logische Negation eines Ausdrucks durch. Darüber hinaus invertiert der Operator 'Not' die Bit-Werte einer beliebigen Variablen.
- *'BitAnd', 'BitOr', 'BitXor', 'BitNot'* entsprechen 'And', 'Or', 'Xor' und 'Not' und sind zur Vollständigkeit unterstützt.

Ein Ergebnis lautet jeweils -1 ('True'), wenn sämtliche Bits gesetzt sind (binär 1111 1111) und 0 wenn die Bits nicht gesetzt sind (binär 0000 0000).

Die von C++ bekannten Operatoren << und >> werden zur Zeit nur für die Qt-Anbindung unterstützt. << und >> als Shiftoperatoren in KBasic selbst, werden noch nicht unterstützt. Man kann aber stattdessen SHL(stellen) (nach links) oder SHR(stellen) (nach rechts) benutzen.

Andere Operatoren

- *^ Potenz*
  Dient zum Potenzieren einer Zahl mit einem Exponenten. Die Zahl nach '^' darf nur negativ sein, wenn Exponent ein ganzzahliger Wert ist. Werden in einem einzigen Ausdruck mehrere Potenzierungen ausgeführt, so wird der Operator '^' jeweils in der Reihenfolge seines Auftretens von links nach rechts ausgewertet.
- *Like (Mustervergleich)*
  Dient zum Vergleichen zweier Zeichenfolgen.
  Wenn Zeichenfolge und Muster übereinstimmen, ist Ergebnis 'True'. Bei fehlender Übereinstimmung ist Ergebnis 'False'. Dem Vergleich liegt dabei der ASCII-Code zugrunde, nach dem jedem Textzeichen intern ein numerischer Wert zugeordnet ist.
- *New* erstellt ein neues Objekt anhand einer Klasse
- *( ) oder [ ]*
  Indexzugriff auf Felder
- *. Punkt*
  wird bei Methodenaufrufen, Enumerationen, oder Feldzugriff verwendet
- *& (siehe Stringberechnung ff.)*
  Dient zur Verkettung von Zeichenfolgen.
- *Eqv*
  bedeutet x Eqv y = Not (x Xor y)
  Der Operator Eqv führt einen bitweisen Vergleich gleich positionierter Bits in zwei numerischen Ausdrücken durch.
- *Imp*
  bedeutet x Imp y = Not (x And Not y)
  Der Operator Imp führt einen bitweisen Vergleich gleich positionierter Bits in zwei numerischen Ausdrücken durch.
- *Is*
  Der Operator Is dient zum Vergleichen von Verweisen auf Objekte. Er vergleicht nicht die Objekte oder deren Werte, sondern überprüft lediglich, ob sich zwei Objektverweise auf dieselbe Klasse beziehen.

Für die Qt-Anbindung unterstüzte Operatoren, eigentlich C/C++ Operatoren sind:

- ++ Inkrement
- -- Dekrement
- [ ] Indexzugriff
- |= Oder-Zuweisung
- &= Ond-Zuweisung
- | Oder
- ! Nicht
- == Ist gleich
- != Ungleich
- ^= Potenzzuweisung
- << Shift links
- >> Shift rechts

Stringberechnung

Auch Strings können addiert werden. Das Ergebnis ist dann wiederum vom Typ 'String'. Das nennt man String-Konkatination. Dafür existieren zwei Operatoren, es gibt keinen Unterschied zwischen den beiden Operaoren: beide ergeben einen String. Wenn aber ein Operand kein String ist, kann es zu unterschiedlichen Ergebnissen mit dem '+'-Operator kommen. Deshalb sollten Sie vorzugsweise den '&'-Operator verwenden.

- '+'
- '&'

Beispiel: Name + „is a „ + color + „bird“

Ist der erste Operand ein 'String' ist, werden die restlichen zu Strings umgewandelt. Wenn einer der Operanden kein String ist, dann bewirkt '&' immer eine Stringberechnung, wohingegen '+' in bestimmten Fällen eine numerische Berechnung ergeben würde.

Der '&' - Operator existiert auch aus historischen Gründen.

Operatorauswertungsreihenfolge

KBasic untestüzt auch die VB6-Operatorenreihenfolge. Wenn ein Ausdruck mehrere Operationen enthält, werden die einzelnen Teilausdrücke in einer bestimmten Rangfolge ausgewertet und aufgelöst, die als Operatorvorrang bezeichnet wird. Wenn Ausdrücke Operatoren aus mehreren Kategorien enthalten, werden zunächst die arithmetischen Operatoren, dann die Vergleichsoperatoren und zuletzt die logischen Operatoren ausgewertet. Die Vergleichsoperatoren haben alle dieselbe Priorität und werden daher von links nach rechts in der Reihenfolge ihres Auftretens ausgewertet. Allgemein wird ein Ausdruck von links nach rechts ausgewertet.

Z. B.
X = 10 + 10 / 2 ergibt 15 und nicht 10, da zuerst '/' vor '+' berechnet wird. Das ist die sogenannte Punkt vor Strich Regelung.

Hier die ausführliche Übersicht mit absteigender Priorität. Erwähnt werden hier der Vollständigkeitshalber auch die C/C++ Operatoren (in Klammern), die in der Qt-Anbindung Verwendung finden. Für die arithmetischen und logischen Operatoren gilt die folgende Rangfolge:

- . ( ) [ ]
- Not, BitNot, !, ++, --, (unary +), (unary -)
- New
- ^
- * / \ Mod
- + - &
- Imp Eqv
- < > <= >= = Is Like == !=
- And BitAnd (& C/C++ And)
- Or Xor BitOr BitXor (| C/C++ Or) (^ C/C++ Xor)
- AndAlso
- OrElse

Die Multiplikation und Division innerhalb eines Ausdrucks sind gleichrangig und werden von links nach rechts in der Reihenfolge ihres Auftretens ausgewertet. Dasselbe gilt für Additionen und Subtraktionen, die zusammen in einem Ausdruck auftreten.

Mit runden Klammern kann diese Rangfolge außer Kraft gesetzt werden, damit werden bestimmte Teilausdrücke vor anderen Teilausdrücken ausgewertet. In Klammern gesetzte Operationen haben grundsätzlich Vorrang. Innerhalb der Klammern gilt jedoch wieder die normale Rangfolge der Operatoren.

Der Zeichenverkettungsoperator (&) ist zwar kein arithmetischer Operator, liegt aber in der Rangfolge zwischen den arithmetischen Operatoren und den Vergleichsoperatoren. Der Operator 'Like' ist eigentlich ein Operator zum Mustervergleich, wird aber in der Rangfolge den anderen Vergleichsoperatoren gleichgestellt.

Vermeiden von Namenskonflikten

Ein Namenskonflikt tritt auf, wenn Sie einen Bezeichner erstellen oder verwenden möchten, der von KBasic als bereits definiert erkannt wurde. In einigen Fällen können Namenskonflikte zu Fehlermeldungen wie "Mehrdeutiger Name" oder "Mehrfachdeklaration im aktuellen Gültigkeitsbereich" führen. Namenskonflikte können aber auch unbemerkt bleiben und zu Fehlern im Code sowie zu falschen Programmergebnissen führen, insbesondere, wenn Sie nicht alle Variablen vor der ersten Verwendung explizit deklarieren.

Die meisten Namenskonflikte können beseitigt werden, wenn Sie das Konzept der Gültigkeitsbereiche von Bezeichnern für Daten, Objekte und Prozeduren verstanden haben. In KBasic gibt es mehrere Ebenen für Gültigkeitsbereiche: Prozedurebene, private Klassen-/Modulebene und öffentliche Klassen-/Modulebene und protected Klassenebene. Namenskonflikte können in den folgenden Situationen auftreten:

- Wenn auf einen Bezeichner aus mehreren Gültigkeitsbereichen zugegriffen werden kann.
- Wenn ein Bezeichner zwei verschiedene Bedeutungen in demselben Gültigkeitsbereich hat.

Prozeduren können z. B. in verschiedenen Modulen denselben Namen haben. Eine Prozedur mit dem Namen Sub1 kann daher in den Modulen Mod1 und Mod2 definiert werden. Wenn die Prozeduren aus den jeweils zugehörigen Modulen aufgerufen werden, treten keine Konflikte auf. Es kann jedoch zu einem Fehler kommen, wenn Sub1 aus einem dritten Modul aufgerufen wird und eine explizite Kennzeichnung zur Unterscheidung der beiden Sub1-Prozeduren fehlt.

Die meisten Namenskonflikte können behoben werden, indem Sie dem Namen eine genaue Kennzeichnung voranstellen, die den Modulnamen enthält. Beispiel:

```
aModule.aSub(Module1.Var1)
```

In diesem Code wird die Sub-Prozedur aSub aufgerufen, wobei Var1 als Argument übergeben wird. Sie können die Kennzeichner beliebig kombinieren, solange die Unterscheidung gleicher Bezeichner gewährleistet ist.

KBasic ordnet jedem Verweis auf einen Bezeichner die "nächstliegende" Deklaration eines übereinstimmenden Bezeichners zu. Wenn myID z. B. in zwei Modulen eines Projekts (Mod1 und Mod2) als Public deklariert ist, können Sie auf die in Mod2 deklarierte Variable mit dem Namen myID ohne explizite Kennzeichnung aus Mod2 zugreifen, müssen die Variable aber explizit (als Mod2.myID) kennzeichnen, um darauf aus Mod1 zuzugreifen. Wenn Sie auf myID aus einem dritten, direkt referenzierten Modul zugreifen, wird die erste Deklaration verwendet, die bei der Suche in folgenden Bereichen gefunden wird: Es gibt keine festgelegte Reihenfolge der Klassen/Module innerhalb des Projekts.

Typische Fehler, die aufgrund von Namenskonflikten auftreten, sind mehrdeutige Namen, doppelte Deklarationen, nichtdeklarierte Bezeichner und nichtgefundene Prozeduren. Sie können Namenskonflikte und Fehler im Zusammenhang mit Bezeichnern vermeiden, indem Sie jede/s Klasse/Modul mit der Anweisung 'Option Explicit' in 'OldBasic Mode' beginnen. Damit erzwingen Sie die explizite Deklaration von Variablen, bevor sie verwendet werden. Im Modus 'Option KBasic', der standardmäßig aktiviert ist, werden automatisch explizite Deklarationen erwartet.

Bearbeiten des Quelltextes

Die Bearbeitung von Code unterscheidet sich kaum von der Textbearbeitung in einem beliebigen Texteditor. Der blinkende vertikale Strich, der Cursor, kennzeichnet die Stelle auf dem Bildschirm, an der geschriebener oder eingefügter Text erscheint. Das Quelltextfenster stellt ihnen Befehle, wie z. B. Suchen und Ersetzen als Hilfe bei der Quelltextbearbeitung zur Verfügung.

Mit Objekten Arbeiten

Weil KBasic eine objektorientierte Programmiersprache ist, werden Sie wahrscheinlich diese objektorientierten Möglichkeiten nutzen wollen und mit Objekten arbeiten wollen.

Folgende Fragen tauchen auf:

- Wie erstelle ich eine Klasse?
- Wie erstelle ich ein Objekt (eine Instanz) einer Klasse?
- Wie rufe ich Klassenvariablen, Klassenmethoden, Instanzvariablen, Instanzmethoden auf?
- Wie konvertiere ich ein Objekt in ein anderes?

Neue Objekte erstellen

Entweder Sie erstellen ein Objekt von einer bereits vorhandenen Klasse, die von KBasic angeboten wird, oder Sie erstellen ein Objekt von einer ihrer eigenen Klasse in einer Klassendatei.

Objekte erzeugen / Benutzen Sie 'New'

Die Deklaration einer Variablen, die ein Objekt beinhalten soll, erzeugt noch nicht das Objekt selbst. Diese Variable beinhalten nur eine Referenz auf dieses Objekt. Um tatsächlich ein Objekt zu erzeugen, müssen sie das Schlüsselwort 'New' benutzen. Der Speicher für neu angelegte Objekte wird dynamisch bereitgestellt. Das Erzeugen eines Objektes mit 'New' ist in KBasic fast das gleiche, wie wenn Sie in C malloc() aufrufen, um Speicher für eine Instanz einer struct bereitzustellen. Es entspricht noch mehr der Benutzung des 'New'-Operators in Java. Nach dem Schlüsselwort 'New' müssen Sie den zu benutzenden Klassennamen angeben. 'New' ist somit ein spezieller Operator, der von einer Klasse ein Objekt erstellt.

Beispiel:

```
s = New myClass()
```

Die Klammern sind nicht unbedingt notwendig, es sei denn, es werden Argumente beim Erstellen eines neuen Objekts übergeben. So kann der passende 'Constructor' der Klasse Argumente erwarten, die angegeben werden müssen. Ob Argumente benötigt werden, legt die jeweilige Klasse in der Deklaration des Konstructors fest. Ein 'Constructor' ist eine spezielle Prozedur, die durch 'New' und durch die passenden Argumente aufgerufen wird und die ein Objekt von der Klasse zurückgibt, in der sich der 'Constructor' befindet. Anders als bei gewöhnlichen Methoden/Prozeduren können Sie den 'Constructor' nicht direkt aufrufen. Stattdessen wird dieser automatisch beim Benutzen von 'New' aufgerufen.

Weiteres Beispiel:

```
s = New myTimer(begin, end)
```

Was macht 'New'?

Bei Benutzung von 'New' wird eine neue Instanz einer Klasse erstellt. Der passende 'Constructor' der Klasse wird aufgerufen.
Es gibt zwei wichtige Bemerkungen über das Benennen und Deklarieren von Konstruktoren:

- Der Name des Konstruktors muss immer gleich dem Klassennamen sein.
- Der Rückgabewert ist implizit eine Instanz der Klasse. Es wird weder eine Rückgabetyp in der Konstruktor-Deklaration angegeben, noch wird das Schlüsselwort 'Function/Sub' benutzt. Es wird implizit 'Me' zurückgegeben; ein Konstruktor darf keine 'Return'-Anweisung benutzen, um einen Wert zurückzugeben.

Syntax:

```
objectVariable = New ClassName[(Arguments)]

objectVariable = New ClassName()

objectVariable = New ClassName
```

Mehrere Konstruktoren

Manchmal möchten Sie die Möglichkeit haben, ein Objekt auf verschiedene Arten zu initialisieren, je nachdem, was in der Situation gerade am besten geeignet ist. Kein Problem: eine Klasse kann mehrere Konstruktoren haben. Diese unterscheiden sich anhand der Argumente.

Beispiel:

```
Class test1

  Constructor test1()
  End Constructor

  Constructor test1(i As Integer)
  End Constructor

  Constructor test1(i As Integer, m As String)
  End Constructor

End Class
```

'Null' (oder 'Nothing')

Der Standardwert für alle Objektvariablen ist 'Null'. 'Null' (oder 'Nothing') ist ein reservierter Wert, der die Abwesenheit einer Referenz anzeigt, z. B. wenn eine Variable nicht auf ein Objekt weißt. 'Null' kann nur Objektvariablen zugewiesen werden.

Automatische Freigabe nicht mehr benutzter Objekte

Es könnte der Eindruck entstehen, dass KBasic voller Speicherlöcher ist, weil wir niemals die Objekte freigeben. Das ist aber nicht der Fall. KBasic benutzt eine Technik, die „garbage collection“ genannt wird. Diese gibt automatisch Objekte, die nicht mehr benötigt werden, frei. Ein Objekt wird dann nicht mehr benötigt, wenn niemand es mehr referenziert. Das bedeutet, dass Sie sich keine Gedanken über das Freigeben von Speicherplatz oder das Zerstören von Objekten machen – der „garbage collector“ kümmert sich darum. Das nötige Referenzieren und Dereferenzieren von Objekten wird automatisch von KBasic übernommen und Sie können sich auf das konzentrieren, was wichtig ist: z. B. den Algorithmus, an dem Sie arbeiten.

Manuelles Löschen der Objekte

Sie können aber auch explizit die Objekte freigeben, indem sie alle Referenzvariablen auf 'Null' setzen.

```
myVar = Null
```

Objekte, die nicht mehr vom Programm benutzt werden und damit nicht mehr referenziert sind, werden also automatisch von KBasic aus dem Speicher entfernt. Das tritt genau dann ein, wenn keine Objektvariable mehr auf dieses Objekt zeigt, da diese entweder automatisch nach dem Verlassen der Prozedur gelöscht wurden oder den Wert 'Null' (oder 'Nothing') angenommen haben. Möchten Sie explizit ein Objekt freigeben, müssen Sie alle Objektvariablen, die auf auf ein und dasselbe Objekt verweisen, auf 'Null' (oder 'Nothing') zeigen lassen.

Bsp.:

```
objectVar = Null
objectVar = Nothing
```

Wenn Sie C/C++ Programmierer sind, dann wird es vielleicht einige Gewöhnungszeit benötigen, bis Sie bereitgestellte Objekte sich selbst überlassen, ohne sich über den Speicher Gedanken zu machen. Wenn Sie sich daran gewöhnt haben, werden Sie diese Eigenschaft als sehr nützlich zu schätzen wissen.

Das bedeutet also für Ihre Programme, dass Sie, wenn Sie mit einem Objekt fertig sind, dieses Objekt vergessen können – der „garbage collector“ findet es und kümmert sich darum.

Eine Klasse erstellen

Eine Klasse zu definieren ist einfach. Normalerweise erbt jede Klasse von der Klasse 'Object'. Mittels 'Inherits' kann eine beliebige Klasse stattdessen beerbt werden. Die Deklaration einer Klasse besteht aus dem Klassennamen und dem Namen der Elternklasse, von der die Klasse erbt, den Konstruktoren und den Destruktor-Methoden, den Klassenvariablen und Klassenmethoden, den Instanzvariablen und Instanzmethoden und Klasseneigenschaften und natürlich Klassenkonstanten etc.

```
Class oak Inherits tree
```

Bestandteile sind:

- Variablen/ Konstanten / Eigenschaften / benutzerdefinierte Typen / Aufzählungen
- Konstruktoren
- Destruktor
- Funktionen
- Prozeduren/Methoden

```
End Class
```

Eine Klasse kann nur in einer Module-Datei oder Class-Datei erstellt werden. In einem Formularmodul ist es nicht möglich. Auch können mehrere Klassen in einer Datei definiert werden.

Klassen werden im Gegensatz zu Methoden nicht ausgeführt

Ein neue Klasse enthält lediglich den Deklarationsbereich und noch keine Methoden. Die einzelnen Methoden müssen Sie selbst erstellen. Eine Klasse enthält kein Hauptprogramm, sondern nur den Deklarationsbereich und Methoden. Sie führen nicht eine Klasse, sondern die darin enthaltenen Methoden als Reaktion auf Ereignisse aus, oder Sie rufen diese Methoden aus Ausdrücken oder anderen Methoden auf.

Zugriff auf Objekte

Wie Sie vielleicht schon in den verschiedenen Beispielcodefragmenten gesehen haben, wird auf die Elemente eines Objektes mit dem Punkt (.) zugegriffen:

Beispiel:

```
classeName.variable
```

Diese Syntax erinnert uns an den Zugriff auf die Felder eines benutzerdefinierten Typs in QBasic.

Zugriff auf Klassen- und Instanzvariablen

Mit dem Punkt (.) wird auf Variablen einer Klasse zugegriffen:

Beispiel:

```
myObject.variable
```

Bei Klassenvariablen wird der Name der Klasse angegeben. Bei Instanzvariablen der Name des Objekts:

```
className.classVariable
objectName.instanceVariable
```

Wichtig!

Klassenvariablen existieren nur ein einziges mal, nämlich in der Klasse und können jederzeit verwendet werden. Man muss auch vorher kein Objekt erstellt haben. Die Klasse ist während des ganzen Programmablaufs präsent und entspricht in etwa einer globalen Variable (globale Variablen sind in KBasic zwar erlaubt, sollten aber vermieden werden). Instanzvariablen dagegen sind Teil eines Objekts und existieren nur solange das Objekt existiert. Bei der Definition der Variablen wird mittels 'Static' angezeigt, dass die Variable keine Instanzvariable sondern eine Klassenvariable ist.

Beispiel:

```
Static myClassVariable As Integer
Private myInstanceVariable As String
```

Weiteres Beispiel:

```
myClass.classVariable = 23

Dim o As myClass
```

Zugriff auf eine Methodenvariable:

```
o.instanceVariable = 99
```

Klassen- und Instanzmethoden

Genauso wie bei den Variablen gibt es auch Methoden, die immer präsent sind: die Klassenmethoden. Sie sind verwendbar, ohne dass vorher mit 'New' ein Objekt erstellt wurde, und entsprechen in etwa globalen Funktionen. Innerhalb von Klassenmethoden ist es nicht erlaubt, 'Me' oder 'Parent' zu verwenden, da diese Klassenmethoden ja zu keiner Instanz einer Klasse gehören.

Die Instanzmethoden dagegen beziehen sich auf ein Objekt und treten immer nur mit diesem auf. Bei der Definition einer Methode wird mittels 'Static' angezeigt, dass die Methode keine Instanzmethode sondern eine Klassenmethode ist.

Beispiel:

```
Static Sub myClassMethod()
...
End Sub

Sub myInstanceMethod
...
End Sub
```

Methoden aufrufen

Auch hier wird wieder der Punkt (.) verwendet.

```
myObject.myMethod()
```

References to objects

Mittels der '='-Anweisung kann eine Variable auf das gleiche Objekt verweisen wie eine andere Variable.

Beispiel:

```
Dim i As tree
Dim s As tree

i = New tree
s = i
```

Beide Variablen 's' und 'i' verweisen damit auf das gleiche Objekt 'tree'.
Hinweis: 'Set', bekannt aus Visual Basic 6, wird bei Objektzuweisung in KBasic nicht verwendet, da es in KBasic überflüssig ist und durch das Weglassen die Lesbarkeit vereinfacht.

Objekte kopieren

Da Objekte nicht mit Wert übergeben werden, kopiert die Zuweisung eines Objekts in ein anderes in KBasic nicht den Wert des Objektes. Stattdessen müssen Sie eine geeignete Methode in ihrer Klasse definieren, die das Kopieren übernimmt. In Zukunft wird KBasic ein spezielle Methode unterstützten, die in der Klasse 'Object' bereitgestellt wird, die 'Clone'-Methode.

Überprüfen von Objekten auf Gleichheit

Eine weitere Folge der Tasache, dass Objekte durch Referenz übergeben werden, ist, dass der '='-Operator testet, ob zwei Variable auf das gleiche Objekt zeigen und nicht, ob die Inhalte der Objekte gleich sind. Um tatsächlich zu überprüfen, ob zwei Objekte gleich sind, müssen sie eine geeignete Methode in ihrer Klasse definieren, die das Vergleichen übernimmt. In Zukunft wird KBasic eine spezielle Methode unterstützten, die in der Klasse 'Object' bereitgestellt wird, die 'Equals'-Methode.

Wiederholung: Objekte, Eigenschaften, Methoden und Ereignisse

Ein **Objekt** stellt ein Element einer Anwendung dar, beispielsweise eine Tabelle, ein Formular oder einen Bericht. In KBasic-Quelltexten müssen Sie ein Objekt bestimmen, bevor Sie eine der Methoden des Objekts anwenden oder den Wert einer der Eigenschaften ändern können.

Eine **Auflistung** ist ein Objekt, das verschiedene andere Objekte enthält. Normalerweise, aber nicht immer, vom gleichen Typ. KBasic enthält in Zukunft die Forms-Auflistung aller Form-Objekte einer Anwendung.

Eine **Methode** ist eine Aktion, die ein Objekt ausführen kann. Add ist z. B. eine Methode des ComboBox-Objekts, da es einem Kombinationsfeld einen neuen Eintrag hinzufügt.

Die folgende Prozedur verwendet die Add-Methode, um einem Kombinationsfeld ein neues Element hinzuzufügen.

```
Sub addNewEntry(newEntry As String)
     Combo1.Append(newEntry)
End Sub
```

Eine **Property** (Eigenschaft) ist ein Attribut eines Objekts, das eines der Merkmale des Objekts definiert, beispielsweise die Größe, Farbe, die Position auf dem Bildschirm oder das Verhalten des Objekts (z. B. ob es aktiviert oder sichtbar ist). Sie ändern die Werte der Eigenschaften eines Objekts, um die Merkmale dieses Objekts zu verändern.

Geben Sie nach dem Verweis auf ein Objekt einen Punkt, den Eigenschaftsnamen, ein Gleichheitszeichen (=) und den neuen Eigenschaftswert an, um den Wert einer Eigenschaft festzulegen. Die folgende Prozedur beispielsweise ändert den Titel eines KBasic-Formulars durch Festlegen der Caption-Eigenschaft.

```
Sub changeName(newTitle)
     myForm.Caption = newTitle
End Sub
```

Einige Eigenschaften können Sie nicht festlegen. Das Hilfethema zu einer Property enthält Informationen darüber, ob Sie diese festlegen (lesen-schreiben), nur lesen (schreibgeschützt), oder nur schreiben können.

Sie können Informationen über ein Objekt erhalten, indem Sie den Wert einer seiner Eigenschaften zurückgeben. Die folgende Prozedur verwendet ein Meldungsfeld zur Anzeige des Titels, der im oberen Bereich des derzeit aktiven Formulars angezeigt wird.

```
MsgBox Me.Caption
```

Ein **Ereignis** ist eine Aktion, die von einem Objekt erkannt wird, beispielsweise das Klicken mit der Maus oder das Drücken einer Taste. Dafür müssen Sie Code schreiben, so dass das Objekt auf das Ereignis reagieren kann. Ereignisse können durch eine Benutzeraktion oder durch Programm-Code hervorgerufen werden oder durch das System ausgelöst werden.

```
Sub Form_OnOpen()
    Print "Hi"
End Sub
```

Erstellen von Objektvariablen

Sie können eine Objektvariable genauso behandeln wie das Objekt, auf das sie verweist. Sie können die Eigenschaften des Objekts festlegen oder zurückgeben oder eine seiner Methoden aufrufen.

So erstellen Sie eine Objektvariable:

1. Deklarieren Sie die Objektvariable
2. Weisen Sie der Objektvariable ein Objekt zu

Deklarieren einer Objektvariablen

Deklarieren Sie eine Objektvariable mit der 'Dim'-Anweisung oder mit einer der anderen Deklarationsanweisungen ('Public', 'Private', 'Protected' oder 'Static'). Eine Variable, die auf ein Objekt verweist, muß den Typ 'Variant', 'Object' oder einen spezifischen Objekttyp (z. B. 'Form') haben. Die folgenden Deklarationen sind z. B. zulässig:

```
' Object1 as Variant
Dim Object1

' Object1 as general Object
Dim Object1 As Object

' Object1 as Form-Object
Dim Object1 As Form
```

Wenn Sie eine Objektvariable verwenden, ohne sie vorher zu deklarieren, erhält sie standardmäßig den Datentyp 'Variant'.

Die Deklaration einer Objektvariablen mit dem Datentyp 'Object' bietet sich in Situationen an, in denen der spezifische Objekttyp vor dem Ausführen der Prozedur nicht bekannt ist. Mit dem Datentyp 'Object' können Sie einen allgemeinen Verweis auf ein beliebiges Objekt erstellen.

Ist Ihnen der spezifische Objekttyp bekannt, auf den eine Objektvariable verweisen soll, dann sollten Sie die Objektvariable mit diesem Typ deklarieren. Wenn die von Ihnen verwendete Anwendung z. B. den Objekttyp "Sample" verwendet, können Sie eine Objektvariable für dieses Objekt mit einer der folgenden Anweisungen deklarieren:

```
Dim Objekt1 As Object        ' Object

Dim Objekt1 As Sample        ' Sample
```

Die Deklaration spezifischer Objekttypen ermöglicht automatische Typüberprüfungen, der Code kann schneller ausgeführt werden und ist verständlicher.

Zuweisen einer Objektvariablen zu einem Objekt

Mit der '='-Anweisung weisen Sie einer Objektvariablen ein Objekt zu. Einer Objektvariablen kann ein Objektausdruck oder 'Null' (oder 'Nothing') zugewiesen werden. Die folgenden Zuweisungen zu Objektvariablen sind z. B. zulässig:

```
Object1 = Object2     ' object reference
Object1 = Null        ' set no object reference
```

Sie können die Deklaration einer Objektvariablen mit der Zuweisung eines Objekts kombinieren, indem Sie das Schlüsselwort 'New' mit der '='-Anweisung verwenden.

Beispiel:

```
Dim Object1 As Object = New Object    ' creation and assignment
```

Durch Festlegen einer Objektvariablen auf den Wert 'Null' (oder 'Nothing') wird die Zuordnung eines bestimmten Objekts zu einer Objektvariablen beendet. Dadurch vermeiden Sie ein versehentliches Ändern des Objekts bei einer Veränderung der Variablen.

Beispiel:

```
If Not Object1 Is Null Then
  ' object variable references an object
  ...
End If
```

Verweisen auf die aktuelle Instanz eines Objekts / 'Me' / 'Parent'

Mit dem Schlüsselwort 'Me' verweisen Sie auf die aktuelle Instanz des Objekts, dessen Code momentan ausgeführt wird. Alle mit dem aktuellen Objekt verbundenen Prozeduren haben Zugriff auf das Objekt, auf das mit 'Me' verwiesen wird. Die Verwendung von 'Me' eignet sich besonders gut, um Informationen über die aktuelle Instanz eines Objekts an eine Prozedur in einem anderen Modul zu übergeben. Das Schlüsselwort 'Me' / 'Parent' verhält sich wie eine implizit deklarierte Variable.

Es steht allen Prozeduren in einem Klassenmodul automatisch zur Verfügung. Wenn eine Klasse mehrere Instanzen hat, stellt 'Me' somit eine Möglichkeit dar, um sich auf diejenige Instanz der Klasse zu beziehen, in der Code gerade ausgeführt wird.

Beispiel:

```
Sub changeObjectColor(Object1 As Control)
     Object1.BackColor = RGB(Rnd * 256, Rnd * 256, Rnd * 256)
End Sub

changeObjectColor(Me) ' statement inside the object
```

Weitere Verwendung von 'Parent'

Es gibt eine spezielle Benutzung des Schlüsselwortes 'Parent', die Bedeutung erlangt, wenn eine Klasse mehrere Konstruktoren besitzt. Es kann von einem Konstruktor benutzt werden, um einen anderen Konstruktor der Klasse aufzurufen. Es gibt eine sehr wichtige Einschränkung für diese 'Parent'-Syntax: sie darf nur als erste Anweisung in einem Konstruktor erscheinen. Ihr können natürlich noch weitere Anweisungen zur Initialisierung folgen, die eine bestimmte Version des Konstruktors noch vornehmen muss. Der Grund dieser Einschränkung hat mit dem automatischen Aufruf der Konstruktormethoden der Elternklasse zu tun.

Beispiel:

```
Class movies

  Protected sMovieName As String

  Sub printName
    print sMovieName
  End Sub

  Constructor movies(s As String)
    sMovieName = s
  End Constructor

End Class


Class movies2 Inherits movies

  Constructor movies2(ByRef s As String)
    Parent.movies(s + "2")
  End Constructor

End Class


Dim k As Integer = 9

Dim m As New movies2("final fantasy")

m.printName()
```

Umwandlung von Objekten und einfachen Datentypen

Für einfache Datentypen existieren Funktionen zum Umwandeln, wie z. B. 'CInt'
Bei Objekten gibt es auch die Möglichkeit, ein Objekt umzuwandeln, d. h. ein Objekt in seinen Vorgänger in der Objekthierarchie oder Nachfolger in der Objekthierarchie umzuwandeln. Hierbei wird das Objekt erweitert oder reduziert. Man muss sich dabei an der Vererbungshierarchie orientieren. Oben steht die Elternklasse aller Klassen, nämlich 'Object', weiter unten folgen dann alle davon abgeleiteten Klassen.

Implizites Upcasting und Downcasting

Downcasting: das Objekt o enthält mehr Methoden als Objekt m und ist somit eine Erweiterung:

```
Dim m As Control
Dim o As CommandButton ' control is parent class
o = m
```

Upcasting: das Objekt enthält weniger Methoden und ist somit eine Reduzierung:

```
Dim h As Control
Dim k As CommandButton ' control is parent class
h = k
```

Objekte vergleichen

Wenn Sie (=) mit Objektverweisen benutzen, ergibt es 'True', wenn beide Objektverweise auf das gleiche Objekt zeigen. Um inhaltlich sinnvolle Vergleiche zu ermöglichen müssen Sie in den Klassen geeignete Methoden für den Vergleich implementieren, die sie dann zum Vergleichen der Objekte benutzen.

Typinformationen zur Laufzeit

Typbestimmung zur Laufzeit:

```
If TypeOf myObject Is myClass Then
```

Gibt 'True' zurück, wenn das Objekt vom Typ der gefragten Klasse ist.

Syntax:

```
TypeOf objectVariable Is ClassName
```

Kinderklassen und Vererbung

Es gibt die Möglichkeit bestehende Klassen ob eigen oder KBasic-Klassen elegant um Funktionalität zu erweitern. Der Vorgang wird Vererbung genannt. Dabei wird eine Elternklasse angenommen, die als Vorlage für die neue Klasse dienen soll. Alle Eigenschaften der Elternklasse befinden sich in der neuen Klassen, also alle Methoden, Variablen und Eigenschaften. Man kann aber diese Methoden überschreiben und somit die neue Klasse den Bedürfnissen anpassen. Der gesamte Vorgang des Übernehmens der Elternklasse wird Vererbung genannt.

Die Vererbung ist ein sehr mächtiges Werkzeug der objektorientierten Programmierung.

Mittels 'Inherits' wird die Elternklasse bei der Definition der neuen Klasse bestimmt:

```
Class movies

  Protected sMovieName As String

  Sub printName
    print sMovieName
  End Sub

  Constructor movies(s As String)
    sMovieName = s
  End Constructor

End Class


Class movies2 Inherits movies
...
End Class
```

Weil jede Klasse eine Elternklasse besitzt, formen Klassen in KBasic eine Klassenhierarchie, die als Baum mit 'Object' als Wurzel und vielen Ästen (den Kinderklassen) dargestellt werden kann.

Konstruktoren verketten

Wenn sie eine Klasse definieren, garantiert KBasic, dass die Konstruktormethode der Klasse immer aufgerufen werden kann, wenn eine Instanz der Klasse erzeugt wird. KBasic garantiert auch, dass der Konstruktor aufgerufen wird, wenn eine Instanz einer Kinderklasse erzeugt wird. Damit dies auch wirklich geschieht, muss KBasic sicherstellen, dass jede Konstruktormethode die Konstruktormethode der Elternklasse aufruft. Wenn die erste Anweisung in einem Konstruktor nicht ein expliziter Aufruf des Konstruktors der Elternklasse mittels 'Parent' ist, dann fügt KBasic implizit den passenden Parent()-Aufruf ein – das heißt, es ruft den Konstruktor der Elternklasse ohne Argument oder mit passenden Argumenten auf.

Es gibt eine Ausnahme zu der Regel, dass KBasic 'Parent()' implizit aufruft, wenn Sie dies nicht explizit tun. Wenn die erste Zeile des Konstruktors der Aufruf eines anderen Konstruktors in derselben Klasse mittels der 'Me()'-Syntax ist, wird KBasic den Elternklassenkonstruktor nicht implizit ausführen. Aber beachten Sie folgendes: wenn der Konstruktor, der mit der 'Me()'-Syntax aufgerufen wird, selbst nicht explizit 'Parent()' aufruft (oder wieder 'Me()' aufruft), dann ruft KBasic 'Parent()' implizit auf. Wenn also Konstrutormethoden sich innerhalb einer Klasse gegenseitig aufrufen, so muss doch eine (explizit oder implizit) die Konstruktormethode der Elternklasse aufrufen.

Das bedeutete, dass die Aufrufe der Konstruktoren verkettet sind. Jedesmal, wenn ein Objekt erzeugt wird, wird eine Reihe von Konstruktoren aufgerufen, von einer Kinderklasse zur Elternklasse bis hin zu 'Object' an der der Spitze der Klassenhierarchie. Weil der Elternklassekonstruktor immer zuerst aufgerufen wird, wird immer als erstes der Konstruktor von 'Object' aufgerufen, gefolgt von der Kinderklasse und die Klassenhierarchie hinunter zu der Klasse, die instanziert wurde.

Der Standardkonstruktor

Wenn ein Konstruktor keinen Elternklassekonstruktor aufruft, tut KBasic dies implizit. Sollte eine Klasse ohne Konstruktor deklariert sein, so fügt KBasic automatisch einen Standardkonstruktor hinzu, der für das implizite Aufrufen des Elternklassekonstruktors zuständig ist.

Es kann verwirrend sein, wenn KBasic implizit einen Konstruktor aufruft oder einen Konstruktor in die Klassendefinition einfügt – es passiert etwas, dass nicht in ihrem Code erscheint. Daher ist es besser, immer einen passenden Konstruktor zu definieren.

Verborgene Variablen

Eine bestehende Variable in der Elternklasse kann bei der Vererbung durch eine neue Variable des gleichen Typs überlagert werden. Obwohl beide den gleichen Namen besitzen, ist es möglich, beide Variablen mit dem selben Namen anzusprechen. Man muss bei der bestehenden Variablen aus der Elternklasse 'Parent' und einen Punkt (.) vor den Variablennamen schreiben:

```
Parent.myVariable  ' access parent variable
myVariable         ' access me variable
```

Verborgene Methoden

So wie eine Variable in einer Klasse eine Variable gleichen Namens in der Elternklasse verbergen kann, kann auch eine Methode in einer Klasse eine Methode mit dem gleichen Namen in der Elternklasse verbergen. In einer gewissen Weise geschieht dies auch. Verborgene Methoden werden überschriebene Methoden genannt. Eine bestehende Methode in der Elternklasse kann bei der Vererbung durch eine neue Methode des gleichen Typs und Argumenten überlagert/überschrieben werden, und obwohl beide den gleichen Namen besitzen, ist es möglich, beide Methoden mit dem selben Namen anzusprechen. Man muss wie bei der bestehenden Methode aus der Elternklasse 'Parent' und einen Punkt (.) vor den Methodenamen schreiben:

```
Parent.myMethod()      ' access parent method
myMethod()             ' access me method
```

Methoden überschreiben

Wenn eine Klasse eine Methode definiert, die den gleichen Namen, Rückgabetyp und die gleichen Argumente wie eine Methode der Elternklasse besitzt, dann überschreibt diese Methode die der Elternklasse. Wenn die Methode für ein Objekt dieser Klasse aufgerufen wird, dann wird die neue Definition der Methode benutzt, nicht aber die alte Definition der Elternklasse (Elternklasse und Kindklasse nutzen beide die gleiche neue Definition!).

Das Überschreiben von Methoden ist eine wichtige und nützliche Technik in der objektorientierten Programmierung. Sie dürfen das Überschreiben von Methoden aber nicht mit dem Überladen von Methoden verwechseln. Überladen bedeutet, mehrere Methoden (in der gleichen Klasse) mit dem gleichen Namen, aber unterschiedlichen Argumenten zu definieren. Dies hat mit dem Überschreiben von Methoden nichts zu tun, und es ist sehr wichtig, dies nicht zu verwechseln.

Überschreiben ist nicht verbergen

Obwohl KBasic die Variablen und Methoden einer Klasse auf viele Art analog behandelt, ist dies beim Überschreiben von Methoden und dem Verbergen von Variablen überhaupt nicht der Fall: Sie können verborgene Variable einfach erreichen, indem Sie ein Objekt in den entsprechenden Typ umwandeln. Sie können überschriebene Methoden jedoch nicht auf diese Art aufrufen. Dafür müssen Sie explizit 'Parent' und den Punkt (.) angeben. In jedem Fall ist KBasic gezwungen, die Vererbung zu berücksichtigen und darauf zu achten, dass nach einer Überschreibung einer Methode diese neue Methodendefiniton entsprechend aufgerufen wird und auch von den Elternklassen benutzt wird. Das geschieht mittels dynamischem Auswählen während des Programmablaufes und nicht während der Codeerzeugung.

Dynamisches Suchen von Methoden

KBasic weiß beim Aufruf einer Methode nicht immer, welche die richtige während des Kompilierens ist, kann dies aber während der Programmausführung herausfinden, da es ja weiß, welche Objekte aus welchen Klassen stammen. Das dynamische Auswählen von Mehoden ist schnell, aber natürlich nicht so schnell wie das direkte Aufrufen der passenen Methode. Glücklicherweise gibt es eine Reihe von Fällen, in denen KBasic diese Technik nicht anwenden muss: Klassenmethoden können nicht von Kinderklassen beerbt werden und können deshalb auch nicht von den Kinderklassen überschrieben werden. Das bedeutet, KBasic kann diese aufrufen, ohne die dynamische Methode zu benutzen.

Aufruf einer überschriebenen Methode

Wir haben jetzt die wichtigen Unterschiede zwischen dem Überschreiben von Methoden und dem Verbergen von Variablen gesehen. Trotzdem ist die Syntax, um in KBasic eine überschriebene Methode aufzurufen, der Syntax, auf eine verborgene Variable zuzugreifen, sehr ähnlich, denn beide benutzen das Schlüsselwort 'Parent'.

Beispiel:

```
Class A
  Dim i As Integer

  Function f()
    Return i
  End Functon

End Class

Class B Inherits A
  Dim i As Integer     ' this variable hides i in A

  Functon f()         ' this method overrides f() in A
    i = Parent.i + 1        ' accessing A.i
    Return Parent.f() + i   ' accessing A.F()
  End Functon
End Class
```

Erinnern Sie sich daran, dass es gleichbedeutend ist, 'Parent' zu benutzen, um auf eine verborgene Variable zuzugreifen. Auf der anderen Seite ist es nicht das gleiche, 'Parent' zu benutzen, um eine überschriebene Methode aufzurufen oder sie mittels 'Me' umzuwandeln. In diesem Fall hat 'Parent' den speziellen Zweck, das dynamische Auswählen der Methode zu umgehen und die spezifische Methode aufzurufen, die die Elternklasse definiert oder erbt.

Beachten Sie, dass diese Form von 'Parent' nicht als erste Anweisung in einer Methode erscheinen darf, wie dies der Fall ist, wenn Sie Konstruktormethoden der Elternklasse aufrufen. Es kann überall in der Methode stehen.

Ein weiteres Beispiel:

```
' class example
Class being

  Constructor being()
    Print "being.Constructor!!!!"
  End Constructor

  Sub cry()
    Print "being.cry"
  End Sub

End Class


Class body Inherits being

  Constructor body()
    Print "body.Constructor!!!!"
  End Constructor

  Sub cry()
    Print "body.cry"
  End Sub
```

```
End Class


Class face Inherits being

  Constructor face()
    Print "face.Constructor!!!!"
  End Constructor

  Sub cry()
    Print "face.cry"
  End Sub

End Class

Dim l[10] As being

l[3] = New being
l[4] = New face
l[5] = New body

' polymorphism
l[3].cry()
l[4].cry()
l[5].cry()
```

Daten verstecken und kapseln

Eine wichtige objektorientierte Technik ist das Verstecken von Daten (Variablen und Konstanten) in einer Klasse, wobei sie nur durch Methoden nach außen zur Verfügung stehen. Diese Technik wird oft auch als Kapselung bezeichnet, weil sich die Daten der Klasse in der Kapsel der Klasse abschirmen, wobei sie nur von vertrauenswürdigen Benutzern erreicht werden können, d. h. von den Methoden der Klasse. Warum sollten Sie das tun? Ein Grund ist, Ihre Klasse gegen versehentliche oder absichtliche Dummheit zu schützen. Eine Klasse enthält oft voneinander abhängige Daten, die in einem konsistenten Zustand bleiben müssen. Wenn Sie es einem Programmierer erlauben, die Variablen direkt zu manipulieren, so kann er eine Variable ändern, ohne auch die anderen entsprechend zu ändern und damit kann er einen inkonsistenten Zustand erzeugen. Wenn er stattdessen eine Methode aufrufen muss, um die Variable zu ändern, so kann diese Methode sicherstellen, dass alles Notwendige getan wird, um den Zustand konsistent zu halten. Außerdem: Wenn alle Variablen der Klasse versteckt sind, dann definieren die Methoden der Klasse die einzig möglichen Operationen, die auf den Objekten der Klasse ausgeführt werden können. Wenn Sie Ihre Methoden sorgfältig getestet und auf Fehler untersucht haben, können Sie sicher sein, dass Ihre Klasse so arbeitet, wie Sie es erwarten. Wenn Sie es erlauben, dass alle Variablen direkt manipuliert werden können, wird die Anzahl, die Sie zu überprüfen haben, unüberschaubar. Weitere Gründe können noch sein:

- interne Variablen, die auch extern sichtbar sind, bringen nur ihre API durcheinander. Die Anzahl der sichtbaren Variablen auf ein Minimum zu beschränken hält ihre Klassen schlank und elegant
- wenn eine Variable in ihrer Klasse sichtbar ist , müssen Sie sie dokumentieren. Sparen Sie sich die Zeit, indem Sie sie verstecken.

Sichtbarkeitsmodifikatoren

Um Variablen oder Methoden zu verstecken, müssen Sie sie als 'Private' deklarieren:

```
Class plane

  Private wings As Integer       ' only visible within this class

  Private Funcion countWings()   ' only visible within this class
    ...
  End Function

End Class
```

Eine 'private' - Variable einer Klasse ist nur für Methoden sichtbar, die in der Klasse definiert sind. Ähnlich kann eine 'private' - Methode nur von Methoden der gleichen Klasse aufgerufen werden. Private Methoden und Variablen sind in Kinderklassen nicht sichtbar und werden nicht von Kinderklassen geerbt, wie es mit den anderen Methoden und Variablen geschieht. Natürlich werden nicht-private Methoden, die intern private Methoden aufrufen, vererbt und sind in Kinderklassen sichtbar.

Außer 'Private' gibt es in KBasic noch 'Protected', 'Public' und die Standardstufe, die zutrifft, wenn keines der Schlüsselwörter 'Public', 'Private' oder 'Protected' benutzt wird. Die Standardstufe enspricht 'Private'. Eine private Variable/Methode ist nur innerhalb der eigenen Klasse sichtbar. Eine protected-Methode oder Variable ist natürlich in der Klasse sichtbar, in der sie definiert ist, und in allen Kinderklassen dieser Klasse. Eine public deklarierte Variable oder Methode ist überall sichtbar.

Ein paar Tipps zur Verwendung der Sichtbarkeitsmodifikatoren:
- Benutzen Sie 'Private' für Methoden und Variablen, die nur innerhalb einer Klasse benutzt werden sollen.
- Benutzen Sie 'Public' für Methoden, Konstanten und andere wichtige Variablen, die auch überall sichtbar sein sollen.

Abstrakte Klassen und Methoden

KBasic lässt uns eine Methode definieren, ohne sie implementieren zu müssen, indem diese Methode als abstrakt deklariert wird ('Abstract'). Eine abstrakte Methode hat keinen eigenen Programmteil (Befehle innerhalb der Methode). Eine abstrakte Klasse enthält abstrakte Methoden, entweder direkt oder indirekt durch das Erben von abstrakten Methoden in der Elternklasse. Sie hat nur die Signatur einer Definition. Abstrakte Klassen und Methoden dienen dazu, in der Klassenhierarchie bestimmte Klassen zu definieren, die als Vorlage für andere Klassen dienen können. Sie können aber selbst nicht instanziert werden, da wesentliche Elemente dort nicht implementiert werden können. Oftmals müssen Sie keine abstrakte Klassen oder Methoden definieren. Normalerweise müssen Sie keine abstrakten Klassen und Methoden verwenden bzw. erstellen.

Ein paar Regeln zu den abstrakten Methoden und den abstrakten Klassen:
- jede Klasse, die eine abstrakte Methode enthält, ist automatisch selbst abstrakt, eine abstrakte Klasse muss mindestens eine abstrakte Methode enthalten
- Eine abstrakte Klasse kann nicht instanziert werden
- eine Kinderklasse einer abstrakten Elternklasse kann instanziert werden, wenn sie alle abstrakten Methoden ihrer Elternklasse überschreibt und eine Implementierung für diese Methoden zur Verfügung stellt
- Wenn eine Kinderklasse einer abstrakten Klasse nicht alle abstrakten Methoden, die es erbt, implementiert, dann ist sie selber abstrakt

Datenfelder (Arrays)

Wenn Sie bereits mit anderen Programmiersprachen gearbeitet haben, sind Sie wahrscheinlich mit dem Konzept der Datenfelder vertraut. Sie können ein Datenfeld deklarieren, das mit einer Gruppe von Werten des gleichen Datentyps arbeitet. Ein Datenfeld ist eine einzelne Variable mit mehreren Speicherfeldern, in denen Werte gespeichert werden. Eine typische Variable verfügt dagegen nur über ein Speicherfeld, das auch nur einen Wert aufnehmen kann. Sie können auf das Datenfeld als Ganzes verweisen, wenn Sie auf alle darin befindlichen Werte verweisen möchten, oder auf die einzelnen Elemente. Datenfelder sind somit eine Aneinanderreihung von Variablen des gleichen Typs, auf die mittels einer Nummer (dem Index) zugegriffen wird.

Datenfelder werden genauso wie andere Variablen mit Hilfe der 'Dim'-, 'Static'-, 'Private'-, 'Protected'- oder 'Public'-Anweisungen deklariert. Der Unterschied zwischen skalaren Variablen (die keine Datenfelder sind) und Datenfeldvariablen besteht darin, dass Sie generell die Größe des Datenfelds angeben müssen. Ein Datenfeld, dessen Größe angegeben ist, ist ein Datenfeld fester Größe. Ein Datenfeld, dessen Größe bei Ausführung eines Programms geändert werden kann, ist ein dynamisches Datenfeld. Die maximale Größe eines Datenfelds hängt von dem verfügbaren Speicher ab.

Mittels Datenfeldern können Sie in vielen Fällen den kürzeren und einfacheren Code erstellen, weil Sie Schleifen benutzen können, die verschiedene Fälle effizient über eine Indexnummer abarbeiten. Datenfelder besitzen obere und untere Grenzen und die Elemente der Datenfelder liegen innerhalb dieser Grenzen zusammenhängend vor. Da KBasic für jede Indexnummer Speicherplatz reserviert, deklarieren Sie ein Datenfeld nur so groß , wie Sie es tatsächlich benötigen. Wenn ein Feld mit dem Datentyp 'Variant' deklariert wurde, können dessen einzelne Elemente verschiedene Arten von Daten enthalten (Zeichenfolgen, Zahlen oder Datumswerte).

Arten von Datenfeldern

Es gibt zwei Arten von Datenfeldern, die sich nur dadurch unterscheiden, dass sich die Größe dynamisch zur Laufzeit ändert.

- *Dynamische Datenfelder*
  ändern ihre Indexanzahl während des Programmsablaufs
- *Statische Datenfelder*
  feste Indexanzahl

Möglicherweise kennen Sie nicht die genaue erforderliche Länge für ein Datenfeld, oder Sie möchten die Länge des Datenfeldes zur Laufzeit ändern können. Die Länge des dynamischen Datenfeldes ist zu jeder Zeit veränderbar. Dynamische Datenfelder unterstützen eine effiziente Speicherverwaltung. Sie können z. B. für eine kurze Zeit ein großes Datenfeld verwenden und den Speicherplatz anschließend wieder für das System freigeben, wenn Sie das Datenfeld nicht mehr benötigen. Die Alternative besteht darin, ein Datenfeld mit der maximal erwarteten Länge zu deklarieren und anschließend alle nicht benötigten Datenfeldelemente zu ignorieren, was aber eine Speicherverschwendung darstellen könnte.

Deklaration

Deklaration eines Arrays mittels 'Dim' und der Größenangabe innerhalb von ( ) oder [ ]. Die Schreibweise mit () wird aus historischen Gründen noch unterstützt:

Syntax:

```
Dim variableName(Index) As Type    ' old VB6 style

Dim variableName[Index] As Type

Dim variableName[Index, Index, ...] As Type

Dim variableName[Index To Index] As Type

Dim variableName[Index To Index, Index To Index, ...] As Type
```

Beispiel:

```
Dim i[100] As Integer
```

Erstellt ein Feld mit 101 Integervariablen (Gezählt wird von 0 bis einschließlich 100).

Zugriff auf Arrayelemente

Sie erreichen ein Element eines Feldes, indem Sie einen ganzzahligen Ausdruck zwischen eckige oder runden Klammern nach dem Namen des Feldes plazieren: Mittels ( ) oder [ ]:

```
i[3] = 10
```

Zugriff mit Angabe der Position. Hier auf die Position 1 im Beispiel:

```
i[1] = 0
```

Es wird der Index daraufhin überprüft, ob er zu klein ist, oder ob er zu groß ist. Wenn der Index außerhalb der Feldgrenzen ist, so wird eine Fehlermeldung erzeugt. Dies ist ein weiterer Weg, wie KBasic versucht, Fehler zu verhindern.

Die Angabe des Index beim Feldzugriff kann dynamisch zur Laufzeit erfolgen. Möglich ist:

```
N = 100

Dim i[n] As Integer
```

Ober- und Untergrenze feststellen

'UBound' zum Ermittlen der Obergrenze:

```
UBound (arrayVar [,(Dimension])
```

'LBound' zum Ermittlen der Untergrenze:

```
LBound (arrayVar [,(Dimension])
```

Standardmäßig zählt der Index von „0“ an, das bedeutet, eine Dimensionierung „Dim Test(10) as Integer“ enthält 11 Elemente.

Explizite Bestimmung der Untergrenze

```
Dim i[50 To 100] As Integer
```

Erstellt ein Feld mit der Untergrenze 50.

Beispiel:

```
Dim curCosts [364] As Currency

Sub createArray ()
     Dim curCosts [364] As Currency
     Dim intI As Integer

     For intI = 0 To 364
          curCosts [intI] = 20
     Next
End Sub
```

Ändern der unteren Grenze

Sie können die 'Option Base'-Anweisung am Anfang eines Moduls dazu verwenden, den Standardindex des ersten Elements von 0 auf 1 zu ändern. Im folgenden Beispiel ändert die 'Option Base'-Anweisung den Index für das erste Element und die 'Dim'-Anweisung deklariert die Datenfeldvariable curCosts mit 365 Elementen. Davon wird aber ausdrücklich abgeraten.

```
Option Base 1

Dim curCosts[365] As Currency
```

Sie können auch explizit die untere Grenze eines Datenfelds über den 'To'-Abschnitt festlegen. Beispiel:

```
Dim curCosts[1 To 365] As Currency
Dim sWeekday[7 To 13] As String
```

Verwenden mehrdimensionaler Datenfelder

KBasic unterstüzt auch mehrdimensionale Felder. Die Anzahl der Dimensionen ist auf 32 beschränkt.
Beispiel: Erstellung eines dreidimensionalen Feldes (mit drei Dimensionen)

```
Dim i[100, 50, 400]
```

Die folgende Anweisung deklariert z. B. ein zweidimensionales, 5-zu-10-Datenfeld.

```
Dim sngMulti[1 To 5, 1 To 10] As Single
```

Wenn Sie sich das Datenfeld als Matrix vorstellen, so stellt das erste Argument die Zeilen und das zweite Argument die Spalten dar.

Verwenden Sie verschachtelte 'For-Next'-Anweisungen für die Verarbeitung mehrdimensionaler Datenfelder. Die folgende Prozedur füllt ein zweidimensionales Datenfeld mit Werten.

```
Sub changeArray ()
     Dim intI As Integer, intJ As Integer
     Dim sngMulti[1 To 5, 1 To 10] As Single

     ' set values of array
     For intI = 1 To 5
          For intJ = 1 To 10
               sngMulti[intI, intJ] = intI * intJ
               Print sngMulti[intI, intJ]
          Next
     Next
End Sub
```

Speichern von 'Variant'-Werten in Datenfeldern

Es gibt zwei Möglichkeiten, Datenfelder für Werte des Typs 'Variant' zu erstellen. Sie können ein Datenfeld als Datentyp 'Variant' deklarieren. Beispiel:

```
Dim varData[3] As Variant
varData[0] = "Christiane Roth"
varData[1] = "60529 Frankfurt"
varData[2] = 26
```

Sie können aber auch das von der 'Array'-Funktion zurückgegebene Datenfeld einer 'Variant'-Variablen zuweisen. varData im folgenden Beispiel ist nicht als Array deklariert, enthält aber nach der Zuweisung von Array(...) ein Array. Beispiel:

```
Dim varData As Variant
varData = Array("Christiane Roth", "60529 Frankfurt", 26)
```

Sie kennzeichnen die Elemente in einem Datenfeld von Variant-Werten über den Index, unabhängig von der bei der Erstellung des Datenfeldes verwendeten Methode. Beispiel:

```
Print "Data " & varDaten[0] & " has been stored."
```

Ändern der Größe eines dynamischen Datenfelds

Deklarieren Sie das Datenfeld mit der Anweisung 'Dim' und einer leeren Dimensionsliste (entweder [ ] oder ( )), z. B.

```
Dim a[] As Integer
```

Reservieren Sie im Programmverlauf die tatsächliche Anzahl von Elementen mit der Anweisung 'Redim'.

Die Anweisung 'Redim' kann nur in Prozeduren verwendet werden. Die Anweisung Redim unterstützt diesselbe Syntax wie 'Dim'. Jede 'Redim'-Anweisung kann die Anzahl der Elemente sowie die untere und obere Grenze für jede Dimension verändern. Die Anzahl der Dimensionen bleibt jedoch unverändert, wie in der ersten 'Redim'-Anweisung angegeben.

Jedesmal, wenn Sie die Anweisung 'Redim' ausführen, werden alle aktuell im Datenfeld gespeicherten Werte gelöscht, es sei denn, Sie verwenden das reservierte Wort 'Preserve'. KBasic setzt die Werte auf den Varianttyp 'Empty' (bei 'Variant'-Datenfeldern), auf den Wert 0 (bei numerischen Datenfeldern), auf eine leere Zeichenfolge (bei Stringdatenfeldern) oder auf ASCII-0 (bei Stringdatenfeldern mit fester Länge) zurück. Objekte erhalten den Wert 'Null'.
Beispiel:

```
Function calc() As Integer
  Dim Matrix1[] As Integer

  Redim Matrix1[22, 33]
  ...
```

Sie können statt fester Werte (Literale) auch Variable oder Ausdrücke verwenden:

```
Redim Matrix1 [a, b]
```

Syntax:

```
Redim variableName(Index)   ' old VB6 style

Redim variableName[Index]

Redim variableName[Index, Index, ...]

Redim variableName[Index To Index]

Redim variableName[Index To Index, Index To Index, ...]
```

Datenfelder löschen / zurücksetzen

Der Befehl 'Erase' löscht Datenfelder bzw. setzt sie zurück:

```
Erase array1[, array2]
```

Bei statischen Datenfeldern wird der Inhalt des Datenfeldes gelöscht, so dass sämtliche Elemente bei numerischen Datentypen den Wert 0 erhalten. Die Elemente in Stringdatenfeldern erhalten dagegen den Wert “”. Bei dynamischen Datenfeldern wird das gesamte Datenfeld gelöscht, der Speicherplatz wird freigegeben. Das Datenfeld kann dann mit 'Redim' neu dimensioniert werden. Bei Datenfeldern des Datentyps 'Variant' erhalten sämtliche Elemente den Typ 'Empty'. Bei Objektdatentypen erhält jedes Element den Wert 'Null'.

Steuern der Programmausführung

Die Anweisungen, die Entscheidungen und Schleifen in KBasic steuern, werden als Kontrollstrukturen bezeichnet. Die am häufigsten verwendeten Kontrollstrukturen in KBasic sind Entscheidungsstrukturen und Schleifenstrukturen.

Mit Hilfe bedingter Anweisungen und Schleifenanweisungen können Sie KBasic-Code erstellen, der Entscheidungen trifft und bestimmte Aktionen ausführt oder wiederholt. KBasic unterstützt alle VB6-Anweisungen. Entscheidungen ermöglichen das gezielte Ausführen von Anweisungen, Schleifen ermöglichen die wiederholte Ausführung einer Gruppe von Anweisungen.

Entscheidungen

Es bedarf der Verwendung von bedingten Anweisungen, um Entscheidungen zu treffen. Entscheidungsstrukturen enthalten bedingte Anweisungen, die überprüfen, ob eine Bedingung 'True/Wahr' oder 'False/Falsch' ist, und legen dann eine oder mehrere Anweisungen fest, die in Abhängigkeit vom Ergebnis der Überprüfung ausgeführt werden sollen. Normalerweise ist eine Bedingung ein Ausdruck, der einen Vergleichsoperator für den Vergleich eines Wertes oder einer Variablen mit einem anderen Wert oder einer anderen Variablen verwendet. KBasic untestüzt alle VB6-Anweisungen diesbezüglich und darüberhinaus noch weitere.

Einfache Entscheidung

Sie wird benutzt, wenn bestimmte Anweisungen aufgrund einer Bedingung ausgeführt werden sollen. Hat die Bedingung den Wert 'True', werden die Anweisungen nach 'Then' ausgeführt, die Anweisungen nach 'Else' werden übersprungen. Wenn aber die Bedingung den Wert 'False' hat, werden nur die Anweisungen nach 'Else' ausgeführt. Sollte die Anweisung nach 'Then' eine Zeilennummer sein, wird ein Sprung dorthin ausgeführt. Befinden sich Anweisungen erst in der nächsten Zeile nach 'Then', werden diese ausgeführt, bis ein 'End If' erreicht wird. Wenn sich das Programm im Modus 'Option KBasic' befindet (Standard), muss die Bedingung ein Booleanwert sein. In den anderen Modi werden auch numerische Werte ungleich „0“ als 'True' interpretiert.

Syntax:

```
If Expression Then Statement

If Expression Then Statement : Else Statement

If Expression Then LineNo

If Expression Then LabelName:

If Expression Then
  [Statements]
End If

If Expression Then
  [Statements]
```

```
Else
  [Statements]
End If

If Expression Then
  [Statements]
ElseIf Expression
  [Statements]
Else
  [Statements]
End If

If Expression Then
  [Statements]
ElseIf Expression
  [Statements]
ElseIf Expression
  [Statements]
Else
  [Statements]
End If

If Expression Then
  [Statements]
ElseIf Expression
  [Statements]
End If
```

'Else' definiert eine Standardbedingung in einer mehrzeiligen 'If'-Anweisung.

'ElseIf' definiert eine weitere Bedingung in einer mehrzeiligen 'If'-Anweisung.

'End If' beendet eine mehrzeilige 'If'-Anweisung.

Beispiel:

```
Dim i As Integer
Dim n As Integer

If i = 1 Then
  n = 11111
ElseIf i = 2 * 10 Then
  n = 22222
Else
  n = 33333
End If
```

'IIf' – Kurzes 'If'

Es gibt einen Wert zurück abhängig von einer Bedingung. Beispiel:

```
testing = IIf(Test1 > 1000, "big", "small")
```

Syntax:

```
IIf(Expression, ThenReturnExpression, ElseReturnExpression)
```

Mehrfachauswahl

Die 'Select Case'-Anweisung ist um einiges komplizierter als die 'If'-Anweisung, aber auch komfortabler.

In vielen Fällen möchten Sie den gleichen Wert oder Ausdruck mit verschiedenen Werten vergleichen, und abhängig von den Vergleichen bestimmten Code ausführen lassen. Genau für diesen Fall existiert die Mehrfachauswahl 'Select Case'.

Sie führt mehrere Vergleiche auf einmal durch. Der Bedingungsausdruck wird von 'Case'-Anweisungen zum Vergleich herangezogen. Die 'Select Case'-Anweisung wird mit 'End Select' abgeschlossen. Dazwischen befinden sich die gewünschten 'Case'-Anweisungen.

Jede 'Case'-Anweisung kann einen String-, numerischen oder booleschen Ausdruck annehmen. Sobald ein 'Case'-Ausdruck zu dem 'Select Case'-Ausdruck passt, werden die entsprechenden Anweisungen nach dieser 'Case'-Anweisung ausgeführt. Danach wird die 'Select Case'-Anweisung verlassen, die anderen 'Case'-Anweisungen werden übersprungen.

Sollten keine Treffer erzielt werden, werden nur die Anweisungen von 'Case Else' ausgeführt.

Syntax:

```
Select Case Expression
Case Expression
  [Statements]
Case Expression
  [Statements]
End Select

Select Case Expression
Case Expression
  [Statements]
Case Expression To Expression
  [Statements]
Case Is Expression
  [Statements]
Case Else
  [Statements]
End Select
```

Beispiel:

```
Dim i As Double
Dim n As Integer

i = 4

Select Case i
Case 0
```

```
  n = 0
Case 1, 2
  n = 1122
Case 4 To 10
  n = 441000
Case Is = 11
  n = 9999
Case Else
  n = 999999
End Select
```

Case Is = 11 ' Ausdruck muss gleich 9 sein, damit zugehörige Befehle ausgeführt werden
Case Is > 11 ' Ausdruck muss größer als 9 sein, damit zugehörige Befehle ausgeführt werden
Case Is < 11 ' Ausdruck muss kleiner als 9 sein, damit zugehörige Befehle ausgeführt werden

'Switch' – Kurzes 'Select Case'

Es gibt einen Wert zurück, abhängig von einer Liste von Bedingungen. Beispiel:

```
Dim s As String
Dim i As Integer
i = 1
s = Switch(i = 1, "Bible", i = 2, "Casanova")
Print s
```

Syntax:

```
Switch(Expression, ReturnExpression[, Expression, ReturnExpression,
... ])
```

'Choose' – Anderes kurzes 'Select Case'

Es gibt ein Auswahl-Ergebnis zurück, abhängig von einem Wert. Beispiel:

```
Dim s As String
s = Choose(1, "un", "deux", "troi")
Print s
```

Ergebnis: „un“

Syntax:

```
Choose(Expression, ReturnExpression[, ReturnExpression, ... ])
```

Expression muss größer „0“ sein!

Unbedingte Verzweigung

Vieles in diesem Kapitel hat sich auf bedingte Verzweigungen bezogen. Wie Sie sich sicherlich erinnern, kann man aber auch unbedingte Sprünge beim Programmieren durchführen. Diese Art von Sprüngen kann durch die 'GoTo'-Anweisung erreicht werden. Mittels 'GoTo' können Sie eine bestimmte Marke oder Zeile in Ihrem Programm anspringen.

'GoTo' führt eine unbedingte Verzweigung durch. Unbedingte Verzweigungen erfolgen in jedem Fall, d.h. ohne Erfüllung einer vorgegeben Bedingung:

Syntax:

```
GoTo {LineNo | Label:}

GoTo 125
GoTo nextStep:
```

With-Anweisung

Eine weitere nützliche Kontrollstruktur, die 'With'-Anweisung, führt eine Reihe von Anweisungen aus, ohne dass ein Objekt erneut angegeben werden muss. In KBasic müssen Sie normalerweise ein Objekt angeben, bevor Sie eine seiner Methoden ausführen oder eine seiner Eigenschaften ändern können. Sie können die 'With'-Anweisung verwenden, um ein Objekt für eine ganze Folge von Anweisungen anzugeben.
Mit der 'With'-Anweisung können Sie somit ein Objekt oder einen benutzerdefinierten Typ einmal für eine ganze Folge von Anweisungen festlegen. 'With'-Anweisungen reduzieren wiederholte Eingaben.

```
Sub formSection ()
     With myClass.
          .Value = 30
     End With
End Sub
```

Sie können 'With'-Anweisungen auch verschachteln:

```
Sub setValue ()
     With j
       .e.bkname = "Frankfurter Zoo"
       With .e
         .isbn ( 99 ) = 333
       End With
     End With
End Sub
```

Schleifenanweisungen

Die Anweisungen, die Entscheidungen und Schleifen in KBasic steuern, werden als Kontrollstrukturen bezeichnet. Die am häufigsten verwendeten Kontrollstrukturen in KBasic sind Entscheidungsstrukturen und Schleifenstrukturen.

Einige Schleifen wiederholen Anweisungen, bis eine Bedingung dem Wert 'False' entspricht, andere, bis eine Bedingung dem Wert 'True' entspricht. Es gibt weiterhin Schleifen, die Anweisungen für jedes Objekt in einer Auflistung oder mit einer festgelegten Anzahl an Wiederholungen ausführen.

Eine Schleife wird solange ausgeführt, bis die Abbruch-Bedingung der Schleife zutrifft. Einige der Schleifenarten werden nur aus Kompatibilitätsgründen unterstützt.

For Next

Diese Schleife wird verwendet, wenn genau bekannt ist, wie oft Anweisungen wiederholt werden sollen.

Syntax:

```
For counter = start To stop [Step stepExpression]
  [Statements]
Next [counter]
```

'counter' ist eine numerische Integer-Variable, die dazu benutzt wird, den Zählwert der Schleife zu speichern. 'counter' ist somit der Zähler. 'start' ist ein numerischer Ausdruck, der den Startwert des Zählwerts bestimmt. 'stop' ist ein numerischer Ausdruck, der den Endwert des Zählers bestimmt. Startwert und Endwert bestimmen zusammen die Anzahl der Schleifendurchläufe. Sie können die Veränderung des Zählers in jedem Schleifendurchlauf ändern, indem Sie die Erhöhung oder Erniedrigung mittels 'stepExpression' bestimmen. Standardmäßig wird der Zähler mit jedem Schleifendurchlauf um eins erhöht.

Wenn Sie 'Exit For' benutzen, wird die aktuelle 'For Next'-Schleife sofort verlassen und mit der nächsten gültigen Zeile in der Anweisungsfolge fortgesetzt. Für gewöhnlich wird 'Exit For' zusammen mit einer 'If'-Bedingung kombiniert (If irgendetwas Then Exit For).

Die 'Next'-Anweisung bestimmt das Ende der Schleife. Alle Anweisungen innerhalb 'For' und 'Next' werden mit jedem Schleifendurchlauf ausgeführt. Die Schleife wird solange wiederholt, bis der Startwert größer dem Endwert ist.

Hinweis:
Die Geschwindigkeit der 'For Next'-Schleifen ist abhängig vom Variablentyp des Zählers. Integervariablen werden am schnellsten verarbeitet.

```
' Example 1:
Dim ctr As Integer

For ctr = 1 To 5
  Print "Z";
Next ctr

' Output:
' ZZZZZ

' Example 2:
' Here we use STEP
' The resulting string is the same
' as in example 1
```

```
Dim ctr As Integer

As Integer ctr = 1 To 50 Step 10
  Print "Z";
Next ctr

' Output:
' ZZZZZ


' Example 3:
' These are nested loops.
' The "y" loop is the INNER one
' and thus will end first:

Dim x, y As Integer

For x = 1 To 5
  Print "Line" + Str$(x)
  For y = 1 To 5
    Print "Z";
  Next y
  Print "-"
Next x

' Output:
' Line 1
' ZZZZZ-
' Line 2
' ZZZZZ-
' Line 3
' ZZZZZ-
' Line 4
' ZZZZZ-
' Line 5
' ZZZZZ-
```

Optimieren von 'For...Next'-Schleifen

Der Datentyp 'Integer' benötigt nicht nur weniger Speicher als der Datentyp 'Variant', sondern Variablen dieses Typs lassen sich auch etwas schneller aktualisieren. Da Computer in der Regel mit einer sehr hohen Geschwindigkeit arbeiten, können Sie einen Unterschied normalerweise erst nach mehreren tausend Operationen feststellen.

Beispiel:

```
Dim rabbit As Integer ' counter of type Integer

For rabbit = 0 To 3276
Next rabbit


Dim hedgehog As Variant    ' counter of type Variant.
```

```
For hedgehog = 0 To 3276
Next hedgehog
```

Im ersten Fall wird etwas weniger Zeit für die Ausführung benötigt. Allgemein gilt: Je kleiner der Datentyp, desto schneller kann er aktualisiert werden. Werte vom Typ 'Variant' sind immer etwas langsamer als die direkt angegeben, entsprechenden Datentypen.

'For Each'

Verhält sich im Prinzip wie 'For Next'. Statt aber einen Zähler zu inkrementieren, wird eine Auflistung von Anfang bis zum Ende durchlaufen.

Syntax:

```
For Each element In collection
  [Statements]
Next
```

Andere Schleifen

Die folgenden Schleifen werden verwendet, wenn nicht genau bekannt ist, wie oft die Anweisungen in der Schleife ausgeführt werden müssen. Dafür existieren mehrere Schlüsselwörter: 'Do', 'While', 'Loop', 'Until' and 'Wend'.

Es gibt zwei Möglichkeiten zur Verwendung des Schlüsselworts 'While', um eine Bedingung in einer 'Do...Loop'-Anweisung zu überprüfen. Sie können die Bedingung überprüfen, bevor die Ausführung der Schleife begonnen wird oder nachdem die Schleife mindestens einmal ausgeführt wurde.
In der folgenden Prozedur SubBefore überprüfen Sie die Bedingung, bevor die Ausführung der Schleife begonnen wird. Wenn myNumber auf 9 anstatt auf 20 festgelegt wird, werden die Anweisungen innerhalb der Schleife niemals ausgeführt. In der Prozedur SubAfter werden die Anweisungen innerhalb der Schleife nur einmal ausgeführt, bevor die Bedingung dem Wert 'False' entspricht.

```
Sub SubBefore()
  Counter = 0
  myNumber = 20
  Do While myNumber > 10
    myNumber = myNumber - 1
    Counter = Counter + 1
  Loop
  MsgBox "Loop has been executed " & Counter & " time(s)."
End Sub

Sub SubAfter()
  Counter = 0
  myNumber = 9
  Do
    myNumber = myNumber - 1
```

```
    Counter = Counter + 1
  Loop While myNumber > 10
  MsgBox "Loop has been executed " & Counter & " time(s)."
End Sub
```

Es gibt zwei Möglichkeiten der Verwendung des Schlüsselworts 'Until', um eine Bedingung in einer 'Do...Loop'-Anweisung zu überprüfen. Sie können die Bedingung vor dem Eintreten in eine Schleife prüfen (dargestellt in der SubBefore-Prozedur), oder Sie prüfen die Bedingung, nachdem die Schleife mindestens einmal durchlaufen wurde (dargestellt in der SubAfter-Prozedur). Die Schleife wird fortgeführt, solange die Bedingung dem Wert 'False' entspricht.

```
Sub SubBefore()
  Counter = 0
  myNumber = 20
  Do Until myNumber = 10
    myNumber = myNumber - 1
    Counter = Counter + 1
  Loop
  MsgBox "Loop has been executed " & Counter & " time(s)."
End Sub

Sub SubAfter()
  Counter = 0
  myNumber = 1
  Do
    myNumber = myNumber + 1
    Counter = Counter + 1
  Loop Until myNumber = 10
  "Loop has been executed " & Counter & " time(s)."
End Sub
```

'Do While...Loop'

Eine weitere Möglichkeit, bestimmte Anweisungen zu wiederholen, besteht mit der 'Do While...Loop'-Schleife.

- 'Do'
Startet die Schleife und muss als erstes stehen

- 'Loop'
Beendet die Schleife und steht am Ende

- 'While'
Wiederholt die Schleife, solange die Bedingung 'True' ist

- Bedingung
Ein numerischer Ausdruck oder Zeichenkettenausdruck der 'True' oder 'False' ergibt

- 'Exit Do'
Verlässt die Schleife

- 'Iterate Do'
Springt direkt zur Schleifenbedingung

Syntax:

```
Do While Expression
  [Statements]
Loop
```

Beispiele:

Im ersten Beispiel wird die Schleife solange wiederholt, bis xyz einen anderen Wert als 5 annimmt:

```
Do While xyz = 5
(lines of code)
Loop
```

Bitte beachten Sie, dass die Anweisungen innerhalb der Schleife, die wiederholt werden sollen, niemals ausgeführt werden, wenn xyz nicht 5 ist.

Manchmal kann es sinnvoll sein, die Schleifenbedingung am Ende der Schleife zu platzieren, so dass die Anweisungen innerhalb der Schleife mindestens einmal ausgeführt werden:

```
Do
  (lines of code)
Loop While xyz = 5
```

Erst nach dem Ausführen der Anweisungen innerhalb der Schleife wird mittels der Schleifenbedingung überprüft, ob eine Wiederholung stattfinden soll.

Wenn Sie eine weitere Bedinung zum Abbruch der Schleife brauchen, können Sie jederzeit innerhalb der Schleife explizit die Schleife mittels der 'Exit'-Anweisung verlassen:

```
Do
  (lines of code)
  If abc = 0 Then Exit Loop
  (lines of code)
Loop While xyz <> 5
```

Dies bedeutet, wenn abc den Wert 0 hat, wird die Schleife sofort verlassen, ohne die restlichen Anweisungen der Schleife auszuführen.

Außerdem kann es manchmal sinnvoll sein, nur Teile der Schleifenanweisungen zu wiederholen. Mit Hilfe von 'Iterate' können Sie jederzeit zur Schleifenbedingung springen:

```
Do
  (lines of code)
  If abc = 0 Then Iterate Loop
  (lines of code)
Loop While xyz <> 5
```

Dies bedeutet, wenn abc den Wert 0 hat, wird die Schleifenbedingung überprüft. Wenn das Ergebnis 'True' ist, wird eine weitere Wiederholung der Schleifenanweisungen gestartet, ansonsten wird die Schleife sofort verlassen.

Sie können auch mehrere Bedingungen kombinieren, z. B. mit Hilfe von 'And' oder 'Or' :

```
Do While x < 10 And y < 10
  (lines of code)
Loop
```

Es wird nur solange wiederholt, wie x < 10 UND y < 10 sind.

Wichtiger Hinweis: Wenn Sie für verschachtelte Schleifen Bedingungen verwenden, die auf gleiche Variablen verweisen, müssen Sie besonders darauf achten, dass Sie nicht versehentlich eine Endlosschleife programmieren. Denn wird die Variable der einen Schleife immer zurückgesetzt, kann die andere Schleife niemals verlassen werden.

```
Do While counter < 10
  (lines of code)
  counter = 0
  Do
    counter = counter + 1
  Loop While counter <> 5
Loop
```

Deshalb ist es in der Regel sinnvoll, für verschiedene Schleifen unterschiedliche Variablen zu verwenden:

```
Do While counter < 10
  (lines of code)
  For counter = 1 To 5
    (lines of code)
  Next counter
Loop
```

Seien Sie auch vorsichtig, wenn Sie mehrere Kontrollstrukturen verschachteln. Beachten Sie die richtige Reihenfolge:

```
Do While counter < 10
  (lines of code)
  For i = 1 TO 5
    (lines of code)
  Loop
Next i
```

Das Beispiel wird zu einem Fehler führen, da 'Loop' unterhalb von 'Next' stehen muss.

Do...Loop Until

Wiederholt, bis die Bedingung 'True' ist. Beispiel:

```
x = 0
Do
  Print x
  x = x + 1
Loop Until x > 3
```

Syntax:

```
Do
  [Statements]
Loop Until Expression
```

Do...Loop While

Wiederholt solange die Bedingung 'True' ist. Beispiel:

```
x = 0
Do
  Print x
  x = x + 1
Loop While x < 3
```

Syntax:

```
Do
  [Statements]
```

```
Loop While Expression
```

Do Until...Loop

Wiederholt, solange die Bedingung 'False' ist. Beispiel:

x = 0

```
Do Until x > 3
  Print x
  x = x + 1
Loop
```

Syntax:

```
Do Until Expression
  [Statements]
Loop
```

While Wend

Wiederholt, solange die Bedingung 'True' ist. Beispiel:

```
While True ' Achtung! Endlosschleife hier
  Print 1234
Wend
```

Syntax:

```
While Expression
  [Statements]
Wend
```

While...End While

Wiederholt, solange die Bedingung 'True' ist. Enspricht 'While Wend'. Beispiel:

```
While True
  Print 1234
End While
```

Syntax:

```
While Expression
  [Statements]
End While
```

Vorzeitiges Verlassen einer Schleife

Normalerweise wird eine Schleife genau dann verlassen, wenn die Bedingung dies zulässt. Manchmal kann es aber sein, das schon früher eine Schleife verlassen werden muss, und dafür gibt es auch bestimmte Anweisungen. Sie können eine Schleife unter Verwendung der 'Exit Do'-Anweisung verlassen. Dabei verlässt 'Exit Do' die nächste zugeordnete Do-Schleife. Für 'For'-Schleifen verwenden Sie bitte 'Exit For':

- 'Exit For' – Verlassen einer 'For'-Schleife
- 'Exit Do' – Verlassen einer 'Do'-Schleife

Verwenden Sie z. B. die 'Exit Do'-Anweisung im Anweisungsblock einer 'If'-Anweisung oder einer 'Select Case'-Anweisung, wenn Sie eine Endlosschleife verlassen möchten. Entspricht die Bedingung dem Wert 'False', so wird die Schleife wie üblich ausgeführt.

Syntax:

```
Exit For
Exit Do
```

Vorzeitiges Überprüfen einer Schleifebedingung

Normalerweise wird eine Schleifenbedingung genau dann geprüft, wenn die Anweisungen abgearbeitet wurden. Manchmal kann es aber sein, das schon früher eine Schleifebedingung geprüft werden muss und dafür gibt es auch bestimmte Anweisungen. Sie können eine Schleifebedingung unter Verwendung der 'Iterate Do'-Anweisung frühzeitig überprüfen. Dabei springt 'Iterate Do' zur nächsten zugeordneten 'Do'-Schleife. Für 'For'-Schleifen verwenden sie bitte 'Iterate For':

- 'Iterate For' – vorzeitiges Prüfen einer 'For'-Schleifenbedingung
- 'Iterate Do' – vorzeitiges Prüfen einer 'Do'-Schleifebedingung

Syntax:

```
Iterate For
Iterate Do
```

Verschachtelte Kontrollstrukturen

Sie können Kontrollstrukturen in andere Kontrollstrukturen einbetten, z. B. einen 'If'-Block in eine 'For Next'-Schleife. Kontrollstrukturen, die in anderen Kontrollstrukturen eingebettet sind, werden als verschachtelt bezeichnet.

Kontrollstrukturen in KBasic können in beliebig viele Ebenen verschachtelt werden. Es ist sinnvoll, bei verschachtelten Entscheidungs- und Schleifenstrukturen deren Rumpf einzurücken, um den Code für sich und andere lesbarer zu gestalten.

Prozeduren

Möglicherweise waren Ihre KBasic-Programme bis jetzt recht kurz. Wenn Sie aber richtig große Anwendungen schreiben wollen, werden Sie feststellen, dass Sie sehr viele Seiten von Quelltexten erstellen werden. Wenn Programme größer werden, wird es auch schwieriger, sie zu überschauen und zu verwalten. Eine Möglichkeit dem entgegenzuwirken ist, die Quelltexte in verschiedenen Modulen und Prozeduren zu speichern und damit zu teilen. Eine Prozedur enthält dabei nur Anweisungen, die logisch zusammengehören.

Sie schreiben KBasic-Quelltexte in Einheiten, die als Prozeduren bezeichnet werden. Eine Prozedur ist eine Folge von KBasic-Anweisungen, die in den Schlüsselworten 'Sub'/'Function' und 'End Sub'/End Function' eingeschlossen sind. Man unterscheidet mehrere Arten von Prozeduren:

- *Sub (innerhalb eines Modules oder Klasse)*
  Sub-Prozeduren führen Aktionen aus, geben jedoch keinen Wert zurück und können nicht in Ausdrücken verwendet werden. Ereignisprozeduren sind immer Sub-Prozeduren. Einer Sub-Prozedur kann man Argumente übergeben. Eine Ereignisprozedur ist eine einem Formular zugeordnete Sub-Prozedur. Wenn KBasic erkennt, dass in einem Formular oder Steuerelement ein Ereignis aufgetreten ist, ruft es automatisch die für das Objekt und Ereignis festgelgete Ereignisprozedur auf.

- *Function (innerhalb eines Modules oder Klasse)*
  Funktionsprozeduren geben immer einen Wert, z. B. das Ergebnis einer Berechnung zurück und können zum Ausführen andere Aktionen verwendet werden. Da Funktionen Werte zurückgeben, können sie in Ausdrücken eingesetzt werden. Funktionen akzeptieren Argumente.

- *Sub (innerhalb einer Klasse)*
  besser genannt Methode ohne Rückgabewert, verhält sich genauso wie eine Sub-Prozedur.

- *Function (innerhalb einer Klasse)*
  besser genannt Methode mit Rückgabewert, verhält sich genauso wie eine Function-Prozedur.

Sub-Prozedur

Eine Sub-Prozedur kann Argumente, z. B. Konstanten, Variablen oder Ausdrücke verwenden, die beim Aufruf der Prozedur übergeben werden. Diese Werte, die Sie beim Aufrufen der Prozedur angeben, werden dann zur Berechnung oder Verarbeitung innerhalb der Prozedur verwendet. Vielen in KBasic eingebauten Funktionen können Argumente übergeben werden.

Syntax:

```
Sub Name([Arguments])
  [Statements]
```

```
End Sub

Sub Name([Arguments]) [Throws Name, ...]
  [Statements]
End Sub
```

Function-Prozedur

Eine Function-Prozedur kann Argumente, z. B. Konstanten, Variablen oder Ausdrücke verwenden, die beim Aufruf der Funktion übergeben werden. Diese Werte, die Sie beim Aufrufen der Funktion angeben, werden dann zur Berechnung oder Verarbeitung innerhalb der Funktion verwendet. Funktionen geben immer ein Ergebnis zurück. Vielen in KBasic eingebauten Funktionen können Argumente hinzugefügt werden.

Syntax:

```
Function Name([Arguments]) [As Type]
  [Statements]
End Function

Function Name([Arguments]) [As Type] [Throws Name, ...]
  [Statements]
End Function
```

Argumente

Sie können so viele Argumente angeben, wie Sie für Ihre Prozedur brauchen. Aber Sie müssen darauf achten, dass die Reihenfolge, in der Sie die Argumente erstellen, beim Aufrufen der Prozedur genau eingehalten wird.

Wenn eine Sub-Prozedur über keine Argumente verfügt, muss die Sub-Anweisung kein leeres Klammerpaar enthalten. Alle von den Sub- und End Sub-Anweisungen eingeschlossenen Anweisungen werden immer dann ausgeführt, wenn die Prozedur aufgerufen oder ausgeführt wird. Argumente steht für die Argumentnamen; wenn Sie mehrere Argumente definieren möchten, trennen Sie diese mittels Kommata. Jedes Argument sieht wie eine Variablendeklaration aus und funktioniert auch wie eine Variable in der Prozedur. Die Syntax für die einzelnen Argumente sieht folgendermaßen aus:

Syntax:

```
Name As Type

[ByVal | ByRef] Name As Type

[ByVal | ByRef] Name [As Type]

[ByVal | ByRef] Name [()][As Type]

[ByVal | ByRef] [Optional] Name [()][As Type] [= Expression]
```

```
[ByVal | ByRef] Name [[]][As Type]

[ByVal | ByRef] [Optional] Name [[]][As Type] [= Expression]
```

Typ kann einer der allgemeinen Datentypen, ein 'Variant'-Datentyp oder ein Objekttyp sein. Wenn Sie keinen Datentyp vorgeben, erhält das Argument den Standardatentyp, der normalerweise 'Variant' ist. Klammern nach Variablenmnamen zeigen an, dass es sich bei dem Argument um ein Datenfeld handelt.

Wenn Sie eine Sub-Prozedur aufrufen, geben Sie die Argumente an, die die Prozedur verwenden soll.

Mittels 'ByRef' und 'ByVal' bestimmen Sie, wie die Argumente an die Prozedur übergeben werden. 'ByVal' bedeutet, dass eine Kopie des Wertes an die aufgerufenen Procedure/Funktion übergeben wird. So wird sichergestellt, dass keine unbeabsichtigten Änderungen am Original vorgenommen werden. Es wird also nur mit dem Wert der Variablen gearbeitet und nicht mit ihr selbst. Dagegen wird mit 'ByRef' der Originalwert übergeben und dieser wird durch die Manipulation innerhalb der Prozedur/Funktion geändert.

Jedes einzelne Argument kann mit Hilfe von 'ByRef' oder 'ByVal' spezifiert werden.

Benannte und optionale Argumente

Mit dem Aufruf einer Sub- oder Function-Prozedur können Werte an aufgerufene Prozeduren unter Verwendung von benannten Argumenten übergeben werden. Sie können die benannten Argumente in beliebiger Reihenfolge auflisten. Beim Aufruf von Sub- oder Function-Prozeduren können Sie Argumente in der Reihenfolge, in der sie in der Prozedurdefinition auftreten, oder über den Namen und ungeachtet der Position bereitstellen.

Sie können diese Prozedur aufrufen, indem Sie die Argumente an der korrekten Position, jeweils getrennt durch ein Komma, angeben. Beispiel:

```
PassArg(Frank", 26)
```

Sie können diese Prozedur auch über die Angabe von benannten Argumenten, jeweils getrennt durch ein Komma, aufrufen.

```
PassArg(intAge = 26, strName := "Frank")
```

Beispiel:

```
Sub passArg(strName As String, intAge As Integer)
     Print strName, intAge
End Sub
```

Optionalen Argumenten geht das Schlüsselwort 'Optional' in der Prozedurdefinition voraus. Sie können ebenso einen Standardwert für das optionale Argument in der Prozedurdefinition angeben. Beispiel:

```
Sub optionaleArg(str As String, Optional strCountry As String)
...
End Sub
```

Wenn Sie eine Prozedur mit einem optionalen Argument aufrufen, können Sie wählen, ob Sie das optionale Argument angeben möchten oder nicht. Geben Sie das optionale Argument nicht an, so wird der Standardwert, falls vorhanden, verwendet. Ist kein Standardwert angegeben, entspricht das Argument einer Variablen des angegebenen Typs.

Mit Hife von 'IsMissing' können Sie optionale Argumente testen, ob diese übergeben wurden:

```
Sub optionaleArg(str As String, _
Optional strCountry As String)
     If IsMissing(strCountry) Then
          Print str
     Else
          Print str, strCountry
     End If
End Sub
```

```
optionaleArg(str := „test1“, strCountry := "Germany")
optionaleArg(str := „test1“)
```

Argumente nur mit einem Standardwert sind auch möglich. Sie sollten wenn möglich immer Standardwerte verwenden. Beispiel:

```
Sub optionaleArg(str As String, strCountry As String = "Germany")
...
End Sub
```

Schreiben einer Function-Prozedur

Eine Function-Prozedur ist eine Folge von KBasic-Anweisungen, die durch die Anweisungen 'Function' und 'End Function' eingeschlossen sind. Eine Function-Prozedur ähnelt einer Sub-Prozedur, kann aber auch einen Wert zurückgeben. Eine Function-Prozedur kann Argumente, wie z. B. Konstanten, Variablen oder Ausdrücke verwenden, die mit der aufrufenden Anweisung übergeben werden. Wenn eine Function-Prozedur über keine Argumente verfügt, muss deren Function-Anweisung kein leeres Klammernpaar enthalten. Eine Funktion gibt einen Wert zurück, indem ihrem Namen ein Wert in einer oder mehreren Anweisungen der Prozedur zugewiesen wird (alte Variante). Sie sollten aber für die Rückgabe eines Wertes das Schlüsselwort 'Return' verwenden (neue Variante).

Beispiel:

```
Sub Main()
     temp = InputBox(Prompt:= _
          "Fahrenheit?")
     MsgBox "Fahrenheit = " & Celsius(temp)
End Sub

Function Celsius(fGrad)
     Return (fGrad - 32) * 5 / 9
End Function

Main()
```

Aufrufen von Sub- und Function-Prozeduren

Damit Sie eine Sub-Prozedur aus einer anderen Prozedur aufrufen können, geben Sie den Namen der Prozedur sowie Werte für alle benötigten Argumente an. Die 'Call'-Anweisung wird nicht benötigt.

Sie können eine Sub-Prozedur verwenden, um andere Prozeduren anzuordnen, damit der Code verständlicher wird und einfacher zu überprüfen ist. Im folgenden Beispiel ruft die Sub-Prozedur Main die Sub-Prozedur MultiBeep auf und übergibt den Wert 56 als Argument. Nach Ausführung von MultiBeep wird die Steuerung wieder an Main zurückgegeben. Main ruft die Sub-Prozedur Meldung auf.

Bespiel:

```
Sub Main()
     MultiBeep(56)
     DoMessage
End Sub

Sub MultiBeep(no)
     Dim counter As Integer
     For counter = 1 To no
          Beep
     Next
End Sub

Sub DoMessage()
     Print "Tee time!"
End Sub

Main()
```

Hinzufügen von Kommentaren zu Ihrer Prozedur

Beim Erstellen einer neuen Prozedur oder Ändern von vorhandenem Code empfiehlt es sich, Kommentare hinzuzufügen. Diese Kommentare haben keinen Einfluß auf die Funktion des Codes. Sie verdeutlichen Ihnen und anderen Programierern lediglich, worum es bei diesem Code geht. Kommentare beginnen mit einem ('). Dieses Zeichen teilt KBasic mit, dass alle folgenden Wörter in dieser Zeile unberücksichtigt bleiben sollen. Es gibt noch weitere Möglichkeiten Kommentare zu kennzeichnen. Genaueres finden Sie in diesem Buch unter dem Punkt Kommentare.

Aufrufen von Prozeduren mit dem gleichen Namen

Sie können eine Prozedur in einem beliebigen Modul, das sich in demselben Projekt befindet wie das aktive Modul, genauso aufrufen wie eine Prozedur im aktiven Modul. Sie müssen lediglich den entsprechenden Modulnamen in der aufzurufenden Anweisung angeben, gefolgt vom obligatorischen Punkt (.). Beispiel:

```
Sub Main()
     Module1.myProcedure()
End Sub
```

Tips zum Aufrufen von Prozeduren

- Wenn Sie ein/e Klasse/Modul umbenennen, achten Sie darauf, dass der Name der/s Klasse/Moduls oder Projekts in allen aufzurufenden Anweisungen geändert wird. Andernfalls kann KBasic die aufgerufene Prozedur nicht finden. Sie können im Menü Bearbeiten den Befehl Ersetzen verwenden, um Text in einer/m Klasse/Modul zu suchen und zu ersetzen.
- Um Namenskonflikte in referenzierten Projekten zu vermeiden, geben Sie Ihren Prozeduren eindeutige Namen, sodass Sie eine Prozedur ohne Angabe eines Projekts oder Moduls aufrufen können.

Verlassen von Prozeduren

Sie können beliebig oft und an jeder Stelle innerhalb des Funktionsrumpfes die Prozedur verlassen. Dafür verwenden sie 'Exit Sub' oder bei Funktionen 'Exit Function'.

Syntax:

```
Exit Sub

Exit Function  ' inside a function-procedure
```

Aufruf von Sub-Prozeduren mit mehr als einem Argument

Das folgende Beispiel zeigt eine Sub-Prozedur, die mit mehr als einem Argument aufgerufen wird.

```
Sub Main()
     costs (99800, 43100)
     Call costs (380950, 49500) ' old style
End Sub

Sub costs (price As Single, income As Single)
     If 2.5 * income <= 1.8 * price Then
          Print "House to expensive."
     Else
          Print "House is cheap."
     End If
End Sub
```

Verwenden von Klammern beim Aufruf von Function-Prozeduren

Damit Sie den Rückgabewert einer Funktion verwenden können, weisen Sie die Funktion einer Variablen zu. Beispiel:

```
answer3 = MsgBox("Are you satisfied with your income?")
```

Wenn der Rückgabewert einer Funktion für Sie bedeutungslos ist, können Sie eine Funktion auf die gleiche Weise aufrufen wie eine Sub-Prozedur. Sie können die Klammern weglassen. Sie sollten aber immer Klammern beim Aufruf von Prozeduren verwenden. Beispiel:

```
MsgBox "Task done!"  ' old style
```

Schreiben von rekursiven Prozeduren

Prozeduren steht nur ein begrenzter Speicherplatz für Variablen zur Verfügung. Wenn eine Prozedur sich selbst aufruft, wird weiterer Speicherplatz verbraucht. Sich selbst aufrufende Prozeduren werden als rekursive Prozeduren bezeichnet. Eine rekursive Prozedur, die sich ständig selbst aufruft, löst schließlich einen Fehler aus. Beispiel:

```
Function doError(Maximum)
      doError = doError(Maximum)
End Function
```

Dieser Fehler kann auch versteckt in Erscheinung treten, wenn sich zwei Prozeduren permanent gegenseitig aufrufen oder wenn eine Abbruchbedingung für eine Rekursion niemals erfüllt ist. In manchen Situationen sind Rekursionen aber durchaus sinnvoll. Die folgende Prozedur ruft sich z. B. selbst auf, um Fakultäten zu berechnen:

```
Function Fakulty (N)
     If N <= 1 Then   ' end of recursive
       Fakulty = 1    ' (N = 0) no more calls
     Else ' Fakulty is called again
       ' wenn N > 0.
       Fakulty = Fakulty(N - 1) * N
     End If
End Function
```

Sie müssen eine rekursive Prozedur testen, um sicherzustellen, dass sie sich nicht zu oft aufruft und so einen Überlauffehler auslöst. Wenn es zu einem Fehler kommt, überprüfen Sie die Abbruchbedingung der Rekursion. Versuchen Sie dann, durch die folgenden Maßnahmen Speicher einzusparen:

- Entfernen Sie überflüssige Variablen.
- Vermeiden Sie, sofern möglich, die Verwendung des Datentyps 'Variant'.
- Überprüfen Sie die Logik der Prozedur erneut. Häufig können Sie anstelle einer Rekursion verschachtelte Schleifen verwenden.

Überladen von Prozeduren

In vielen Programmiersprachen ist der Name einer Prozedur ausreichend, um diese von anderen Prozeduren zu unterscheiden. Aber in KBasic werden Prozeduren nicht nur anhand des Prozedurnamens unterschieden, sondern auch anhand der Parameter. Sie können somit mehrere Prozeduren gleich benennen, solange nur die Parameter verschieden defininiert sind. KBasic ist dann in der Lage, eine Differenzierung zu übernehmen. Zwei Prozeduren sind nie gleich, außer sie haben den gleichen Namen und die gleiche Menge von Argumenten des gleichen Typs, die ihnen in der gleichen Reihenfolge übergeben werden. Wenn Sie eine Prozedur aufrufen und es mehr als eine Prozedur dieses Namens gibt, dann wählt KBasic automatisch die Prozedur aus, bei der die Datentypen zu den Argumenten in Ihrem Aufruf passen. Prozeduren mit dem gleichen Namen und unterschiedlichen Argumenten zu definieren, heißt Prozeduren überladen. Es kann sehr praktisch sein, Prozeduren nur dann den gleichen Namen zu geben, wenn sie ähnliche Funktionen mit nur leicht unterschiedlichen Rückgabewerten haben, auch wenn sie unterschiedliche Argumente haben. Verwechseln Sie das Überladen von Prozeduren nicht mit dem Überschreiben von Prozeduren.

Ereignisprozeduren

Wenn KBasic erkennt, dass in einem Formular oder Steuerelement ein Ereignis stattfindet, wird automatisch die für das Objekt und das Ereignis angegebene Ereignisprozedur aufgerufen.

Generelle Prozeduren

KBasic startet eine Ereignisprozedur als Reaktion auf ein bestimmtes Ereignis in einem Forlumar oder Steuerelement. Eine generelle Prozedur hingegen läuft nur dann ab, wenn sie explitzit aufgerufen wird. Eine Funktion ist ein Beispiel für generelle Prozeduren.

Wenn Sie mehrere verschiedene Ereignisprozeduren benötigen, um dieselbe Aktion auszuführen, können Sie an deren Stellen eine generelle Prozedur verwenden. Eine gute Programmierstrategie besteht darin, einen allgemeinen Code in eine separate generelle Prozedur zu schreiben und diesen Code über Ereignisprozeduren aufzurufen. Auf diese Weise entfällt die Notwendigkeit, den Code doppelt zu schreiben, und die Pflege der Anwendungen wird erleichtert.

Sie können generelle Prozeduren entweder in einem Formular oder in einem globalen Modul erstellen. Wenn Sie eine generelle Prozedur in einem Fomrularmodul erstellen, können Sie diese über jede Ereignisprozedur des Formulars aufrufen. Wenn die generelle Prozedur der gesamten Anwendung zur Verfügung stehen soll, schreiben Sie sie in ein globales Modul.

Funktionen

Eigentlich sind Funktion Prozeduren; diese haben aber einen Rückgabewert, was den einzigen Unterschied zu ihnen ausmacht:

Syntax:

```
Function Name([Arguments])[As Type]
  [Statements]
End Function

Function Name([Arguments])[As Type] [Throws Name, ...]
  [Statements]
End Function
```

Die Argumente einer Funktion werden genau wie die Argumente einer Sub-Prozedur verwendet. Die Definition einer Funktion erfolgt mit dem reservierten Wort 'Function'. Außerdem sollten Sie den Rückgabetyp festlegen. Für mehr Informationen siehe Kapitel Prozeduren.

Datentypen von Funktionen

Wie bei Variablen besitzt der von einer Funktion zurückgegeben Wert einen Datentyp. Wenn sie die Funktion definieren, können sie den Datentyp des zurückgegebenen Wertes deklarieren.

Wie bei Variabeln arbeitet KBasic auch mit Funktionen effizienter, wenn sie für die zurückgegebenen Werte explizit einen Datentyp deklarieren. Wenn Sie keinen Datentyp deklarieren, verwendet die Funktioen den Standarddatentyp, der normalerweise 'Variant' ist.

Funktionsrückgabe

Wenn Sie den Rückgabewert festlegen möchten, benutzen Sie 'Return', mit diesem Befehl verlassen Sie sofort die Funktion.

Syntax:

```
Function Name([Arguments])[As Type]
  [Statements]
  Return Expression
End Function
```

Eigenschaften

Viele Objekte und Steuerelemente haben Eigenschaften, die nicht anderes sind als Attribute. Zum Beispiel hat das Steuerelement 'CommandButton' die Eigenschaft (Property) 'Caption', die bestimmt, welcher Text auf dem Steuerelement erscheint. Neben solchen bereits definierten Eigenschaften können Sie in Ihren Klassen auch eigene Eigenschaften erstellen.

Eine Property-Prozedur ist eine Folge von KBasic-Anweisungen, die es einem Programmierer ermöglicht, benutzerdefinierte Eigenschaften zu erstellen und zu bearbeiten.

Syntax:

```
Property Set Name(Argument) ' old style
  [Statements]
End Property

Property Get Name As Type ' old style
  [Statements]
End Property


Property Name As Type ) ' modern style

  Get
    [Statements]
  End Get

  Set(Argument)
    [Statements]
  End Set

End Property
```

Datentypumwandlung

Bestehen eine Konstante, ein Ausdruck oder die zuzuordnende Variable aus unerschiedlichen numerischen Datentypen, so ist im Regelfall keine besondere Anpassung notwendig, da diese automatisch vollzogen wird. Dennoch gibt es Sonderfälle, in denen eine Typenkonvertierung erforderlich ist, sei es aus Gründen des Überlaufs (insbesondere bei Operationen mit Integerwerten) oder der Wiedergabe von Fließkommazahlen.

Folgende Konvertierungsfunktionen existieren, das Ergebnis wird gegebenfalls gerundet:

```
y% = CInt (expression)  ' returns Integer
y% = CLng (expression)  ' returns Long
y! = CSng (expression)  ' returns Single
y' = CDbl (expression)  ' returns Double
y@ = CCur (expression)  ' returns Currency
y = CBool (expression)  ' returns Boolean
y = CByte (expression)  ' returns Byte
y = CShort (expression) ' returns Short
y = CDate (expression)  ' returns Date
```

Sichtbarkeitsbereich

Mit Sichtbarkeitsbereich wird die Verfügbarkeit von Variablen, Konstanten oder Prozeduren zur Verwendung durch eine andere Prozedur bezeichnet, d. h. wo darf die Variable, Konstante oder Prozedur verwendet werden. Folgende Gültigkeitsbereiche gibt es:

- Innerhalb der Prozedur

- Global sichtbar

- Privat sichtbar - innerhalb eines Modules
- Global sichtbar - innerhalb eines Modules deklariert (Public)

- Privat sichtbar - innerhalb einer Klasse
- (Protected) sichtbar - innerhalb einer Klasse und Kinderklassen
- Global sichtbar - innerhalb einer Klasse deklariert (Public)

Sie bestimmen den Gültigkeitsbereich einer Variablen bei deren Deklaration. Durch die explizite Deklaration aller Variablen können Fehler aufgrund von Namenskonflikten zwischen Variablen mit unterschiedlichen Gültigkeitsbereichen vermieden werden.

Definieren von Gültigkeitsbereichen auf Prozedurebene

Sie können Variable und Konstanten auf Modulebene im Deklarationsabschnitt eines Moduls definieren. Variable auf Modulebene können entweder öffentlich oder privat sein. Öffentliche Variable sind für alle Prozeduren in allen Modulen eines Projekts verfügbar, private Variablen sind nur für die Prozeduren in diesem Modul verfügbar. Standardmäßig werden Variable mit der 'Dim'-Anweisung im Deklarationsabschnitt im privaten Gültigkeitsbereich deklariert. Durch Voranstellen des Schlüsselworts 'Private' jedoch wird der Gültigkeitsbereich der Variablen in Ihrem Code erkennbar.

```
Sub localVar()
  Dim str As String
  str = "This variable can only be used inside this procedure"
  Print str
End Sub

Sub outOfScope()
     Print str  ' causes an parser error
End Sub
```

Definieren von Gültigkeitsbereichen auf privater Modulebene

Sie können Variablen und Konstanten auf Modulebene im Deklarationsabschnitt eines Moduls definieren. Variablen auf Modulebene können entweder öffentlich oder privat sein. Öffentliche Variablen sind für alle Prozeduren in allen Modulen eines Projekts verfügbar, private Variablen sind nur für die Prozeduren in diesem Modul verfügbar. Standardmäßig werden Variablen mit der 'Dim'-Anweisung im Deklarationsabschnitt im privaten Gültigkeitsbereich deklariert. Durch Voranstellen des Schlüsselworts 'Private' jedoch wird der Gültigkeitsbereich der Variablen in Ihrem Code erkennbar.

Beispiel:

```
' add to the declaration section of your
' module the following lines
Private str As String

Sub setPrivateVariable()
  str = "This variable may be only used inside this module"
End Sub

Sub usePrivateVariable()
  Print str
End Sub
```

Definieren von Gültigkeitsbereichen auf öffentlicher Modulebene

Wenn Sie eine Variable auf Modulebene als öffentlich deklarieren, ist sie für alle Prozeduren in diesem Projekt verfügbar.

Beispiel:

```
' add to the declaration section of your
' module the following lines
Public str As String
```

Alle Prozeduren (ausgenommen Ereignisprodezuren) sind standardmäßig öffentlich. Wenn KBasic eine Ereignisprozedur erstellt, wird das Schlüsselwort 'Private' automatisch vor der Prozedurdeklaration eingefügt. Für alle anderen Prozeduren müssen Sie die Prozedur explizit mit dem Schlüsselwort 'Private' deklarieren, wenn die Prozedur nicht öffentlich sein soll.

Benutzerdefinierter Datentyp

Sind ein Relikt aus der vor-objektorienierten Zeit und ein Verbund von Variablen verschiedenen Typs ähnlich einer Klasse, aber ohne Methoden. Dabei können benutzerdefinierte Typen alle Arten von Datentypen enthalten, von einfachen Datentypen über Objekte bis hin zu benutzerdefinierten Typen.

Syntax:

```
Type Name
  Name [(Index)] As Type
  sStr As String
  oObj As Object
  ...
End Type
```

Aufzählungstyp

Ist eine Neuerung und hilft Konstanten zu gruppieren. Aufzählungstypen bestehen aus unterschiedlichen Integerwerten. Jeder Wert ist einem Namen zugeordnet.

Syntax:

```
Enum Name
  Name [= Expression]
  ...
End Enum
```

Klassen

Eine einfache Anwendung kann z. B. aus einem einzigen Formular bestehen, während sich der gesamte Code der Anwendung in einem eingebauten Formularmodul befindet. Wenn Ihre Awendungen jedoch größer und kompliziert werden, möchten Sie vielleicht denselben Code in verschiedenen Formularen verwenden. Dazu erstellen Sie eine separate Klasse, die Methoden enthält, die Sie aus jedem Formular aufrufen können. Mit der Zeit können Sie eine Bibliothek Ihrer Klassen aufbauen, die nützliche, häufig gebrauchte Methoden enthält.

Ihren Code speichern Sie somit in einer KBasic-Anwendung in Modulen oder Klassen. Sie können Ihren Programmcode mit Hilfe von Klassen ordnen. Jede Klasse besitzt einen einzelnen Deklarationsbereich und die von Ihnen hinzugefügten Methoden.

Ein Klasse kann enthalten:

- *Deklarationen*,
  für Variable und Konstanen, Eigenschaften, Aufzählungen

- *Methoden (auch Prozeduren genannt)*
  Dies sind Methoden, die nicht direkt einem bestimmten Objekt oder Ereignis zugeordnet sind. Sie können generelle Methoden in einer Klasse aufnehmen. Generelle Methoden können entweder Sub-Prozeduren (Prozeduren, die keinen Wert zurückgeben) oder Funktionen (Prozeduren, die einen Wert zurückgeben) sein.

- *Signal/Slots-Methoden*
  Dies sind Methoden, die in Verbindung mit der Qt-Anbindung vereinbart werden. Näheres dazu finden Sie in der Hilfe zur Qt-Anbindung der KBasic-Entwicklungsumgebung.

- *Eigenschaften*
  Sind Variablen einer Klasse, die über zwei spezielle Methoden bearbeitet werden können. Properties werden nicht mit Klammern aufgerufen, sondern direkt mit der '='-Anweisung benutzt.

Sie können in KBasic in einer Moduldatei oder Klassendatei mehrere Module oder Klassen unterbringen, davon wird aber dringend abgeraten.

Klassen werden im Gegensatz zu Prozeduren nicht ausgeführt

Ein neue Klasse enthält lediglich den Deklarationsbereich und noch keine Methoden. Die einzelnen Methoden müssen Sie selbst erstellen. Eine Klasse enthält kein Hauptprogramm, sondern nur den Deklarationsbereich, Methoden und Eigenschaften. Sie führen nicht eine Klasse, sondern die darin enthaltenen Methoden als Reaktion auf Ereignisse aus, oder sie rufen diese Methoden aus Ausdrücken oder anderen Methoden auf.

Bearbeiten einer Klasse

Die Bearbeitung von Code in einer Klasse unterscheidet sich kaum von der Textbearbeitung in einem beliebigen Texteditor. Der blinkende vertikale Strich, der Cursor, kennzeichnet die Stelle auf dem Bildschirm, an der geschriebener oder eingefügter Text erscheint. Das Quelltextfenster stellt Ihnen Befehle, wie z. B. Suchen und Ersetzen als Hilfe bei der Quelltextbearbeitung zur Verfügung.

Syntax:

```
[Abstract] Class Name Inherits ParentClassName

  [Static] Dim Name As Type
  [Static] Public Name As Type
  [Static] Protected Name As Type
  [Static] Private Name As Type
  Const Name As Type
  Public Const Name As Type
  Protected Const Name As Type
  Private Const Name As Type
  ...


  [Public | Protected | Private]
  Enum Name
    Name As Type
    ...
  End Enum
  ...


  [Public | Protected | Private]
  Type Name
    Name As Type
    ...
  End Type
  ...


  [Public | Protected | Private]
  Property Name As Type

    Get
      [Statements]
    End Get

    Set(Argument)
      [Statements]
    End Set

  End Property
  ...
```

```
  [Public | Protected | Private]
  Constructor Name([Arguments])
    [Statements]
  End Constructor
  ...


  [Public | Protected | Private]
  Destructor Name( )
    [Statements]
  End Destructor


  [Static] [Public | Protected | Private]
  Function Name([Arguments]) [As Type] [Throws Name, ...]
    [Statements]
  End Function
  ...


  [Static] [Public | Protected | Private]
  Sub Name([Arguments]) [Throws Name, ...]
    [Statements]
  End Sub
  ...


  [Public | Protected | Private]
  Slot Name([Arguments])
    [Statements]
  End Slot
  ...


  [Public | Protected | Private]
  Signal Name([Arguments])
    [Statements]
  End Signal
  ...


End Class
```

Längeres Beispiel:

```
Class Salsa Inherits Waltz

  Public Enum class_enum
     Entry
     Entry2
     Security = Entry
  End Enum

  Public Type class_type
    element As Integer
```

```
  End Type

  Const classConst = 4

  Public publicInstanceVar As Integer
  Private privateInstanceVar As Integer
  Protected protectedInstanceVar As Integer

  Static Public publicClassVar As Integer
  Dim publicModuleType As module1.module_type
  Dim publicModuleType2 As module_type

  ' parent constructor call inside constructor

  Sub meExplicit()
    Dim localVar = Me.publicInstanceVar ' it is the same with parent
    Dim localVar2 = Me.publicClassVar
    Dim localVar3 = Salsa.publicClassVar
    Dim localVar4 = Salsa.classConst
    Dim localVar5 = Me.classConst
'
    Dim localVar6 As class_enum
    localVar6 = Salsa.class_enum.Entry
'    Dim localVar7 As Me.class_enum    '  full type name not allowed
    Dim localVar8 As class_type
  End Sub

  Sub meImplicit()
    Dim localVar = publicInstanceVar
    Dim localVar2 = publicClassVar
    Dim localVar3 = classConst
    Dim localVar4 As class_enum
    Dim localVar5 As class_type

  End Sub

  Sub classSub()
    Const localConst = 6
    Dim n = localConst
  End Sub

  Sub classSubWithArgument(i As Integer)
    Dim localVar = i
  End Sub

  Function classFunction() As String
    Return "hello"
  End Function


  Static Public Sub test() Throws Waltz
    Throw New Waltz
  End Sub

  Private pvtFname As String
```

```
  Public Property Nickname As String

    Get
      Print "Hi"
    End Get

    Set ( ByVal Value As String )
      Print "Hi"
    End Set

  End Property

End Class
```

Module

Eine einfache Anwendung kann z. B. aus einem einzigen Formular bestehen, während sich der gesamte Code der Anwendung in einem eingebauten Formularmodul befindet. Wenn Ihre Awendungen jedoch größer und komplizierter werden, möchten Sie vielleicht denselben Code in verschiedenen Formularen verwenden . Dazu erstellen Sie ein separates globales Modul, das Prozeduren enthält, die Sie aus jedem Formular aufrufen können. Mit der Zeit können Sie eine Bibliothek Ihrer globalen Module aufbauen, die nützliche, häufig gebrauchte Prozeduren enthält.

Ihren KBasic-Code speichern Sie somit in einer KBasic-Anwendung in Modulen oder Klassen. Sie können Ihre Prozedur z. B. mit Hilfe von Modulen ordnen. Jedes Modul besitzt einen einzelnen Deklarationsbereich und die von Ihnen hinzugefügten Prozeduren.

Ein Modul kann enthalten:

- *Deklarationen*
  für Variablen und Konstanen, Typen und Aufzählungen.

- *Ereignisprozeduren*
  Hierbei handelt es sich um Sub-Prozeduren, die für ein bestimmtes Objekt gelten und als Reaktion auf ein Benutzer-oder Systemereignis ausgeführt werden, z. B. einen Mausklick. Ereignisprozeduren werden immer zusammen mit einem Formular im Formularmodul gespeichert.

- *Generelle Prozeduren*
  Dies sind Prozeduren, die nicht direkt einem bestimmten Objekt oder Ereignis zugeordnet sind. Sie können generelle Prozeduren in ein Formularmodul oder in ein globales Modul aufnehmen. Generelle Prozeduren können entweder Sub-Prozeduren (Prozeduren, die keinen Wert zurückgeben) oder Funktionen (Prozeduren, die einen Wert zurückgeben) sein.

Sie können in KBasic in einer Moduldatei oder Klassendatei mehrere Module oder Klassen unterbringen, davon wird aber dringend abgeraten.

Globale Module

Sie erstellen globale Module als eigenständige Objekte in ihrer Anwendung und verwenden sie zum Speichern von Code, den Sie von einer beliebigen Stelle der Anwendung aus ausführen möchten. Prozeduren in globalen Modulen können Sie aus Ausdrücken, Ereignisprozeduren oder Prozeduren in anderen globalen Modulen aufrufen.

Module werden im Gegensatz zu Prozeduren nicht ausgeführt

Ein neues Modul enthält lediglich den Deklarationsbereich und noch keine Prozeduren. Die einzelnen Prozeduren müssen Sie selbst erstellen. Ein Modul enthält kein Hauptprogram, sondern nur den Deklarationsbereich und Prozeduren. Sie führen nicht ein Modul, sondern die darin enthaltenen Prozeduren als Reaktion auf Ereignisse aus, oder Sie rufen diese Prozeduren aus Ausdrücken oder anderen Prozeduren auf.

Bearbeiten eines Moduls

Die Bearbeitung von Code in einem Modul unterscheidet sich kaum von der Textbearbeitung in einem beliebigen Texteditor. Der blinkende vertikale Strich, der Cursor, kennzeichnet die Stelle auf dem Bildschirm, an der geschriebender oder eingefügte Text erscheint. Das Quelltextfenster stellt ihnen Befehle, wie z. B. Suchen und Ersetzen als Hilfe bei der Quelltextbearbeitung zur Verfügung.

Syntax:

```
Module Name


  Dim Name As Type
  Public Name As Type
  Private Name As Type
  Const Name As Type
  Public Const Name As Type
  Private Const Name As Type
  ...


  [Public | Private]
  Enum Name
    Name As Type
    ...
  End Enum
  ...
```

```
  [Public | Private]
  Type Name
    Name As Type
    ...
  End Type
  ...

  [Public | Private]
  Function Name([Arguments]) [As Type] [Throws Name, ...]
    [Statements]
  End Function
  ...

  [Public | Private]
  Sub Name([Arguments]) [Throws Name, ...]
    [Statements]
  End Sub
  ...

End Module
```

Größeres Beispiel:

```
Module module1

  Public Type address
    age As Integer
  End Type

  Public Type module_type
    element AS integer
  End Type

  Public Enum module_enum
     Entry
     Entry2
     Security = Entry
  End Enum

  Const moduleConst = 7

  Public publicModuleVar As Integer
  Private privateModuleVar As Integer


  Sub moduleExplicit()
    Dim localVar = module1.publicModuleVar
    Dim localVar2 = module1.moduleConst
    Dim localVar3 As module_enum
```

```
  localVar3 = module1.module_enum.Entry
End Sub

Sub moduleImplicit()
  Dim localVar = publicModuleVar
  Dim localVar2 = moduleConst
  Dim localVar3 As module_enum
  localVar3 = module_enum.Entry
  Dim localVar4 As module_type
End Sub

Sub moduleSubWithDefaultArgument(ko As Integer = 6)
  Dim localVar = ko
End Sub

Sub moduleSubWithOptionalArgument(Optional ko As Integer)
  If Not IsMissing(ko) Then
    Dim localVar = ko
  End If
End Sub

Sub moduleSub()
  Const localConst = 6
  Dim n = localConst
End Sub

Sub moduleSubWithArgument(i As Integer)
  Dim localVar = i
End Sub

Sub moduleSubWithArgumentShadowing(i2 As Integer)
  Dim localVar = i2
  Dim i2 = localVar + 99
  Dim i3 = i2
End Sub

Sub subOverloading ( )
  Print "sub1"
End Sub

Sub subOverloading ( i As Integer = 1)
  Print "sub2"
End Sub

Function moduleFunction() As String

  subOverloading()
  subOverloading(88)

  Return "hello"
End Function

Function moduleFunctionRecursive(ByRef i As Integer) As  Integer
  If i > 6 Then Return 1''i

  ''i = i + 1
```

```
    Return moduleFunctionRecursive(1)''i)
  End Function

End Module
```

Fehler-Behandlung

Entwickler in jeder (Programmier-)Sprache wollten schon immer fehlerfreie Programme schreiben, Programme, die nie abstürzen, Programme, die mit jeder Situation vernünftig umgehen können und die sich von ungewöhnlichen Situationen erholen können, ohne dem Anwender unangebrachten Streß zu bereiten. Gute Vorsätze beiseite, solche Programme gibt es nicht. In realen Programmen kommen Fehler vor, entweder weil der Programmierer nicht jede Situation vorhersehen konnte, in die sein Code kommt (oder er hatte nicht die Zeit, sein Programm genügend zu testen), oder auf Grund von Situationen außerhalb der Kontrolle des Programmierers, falsche Daten des Anwenders, verfälschte Felder, die nicht die richtigen Daten enthalten usw.

Drei mögliche Fehlerquellen gibt es:

- *Fehler während des Kompilierens*
  Dieser Fehler tritt auf, wenn Anweisungen falsch geschrieben sind, das bedeutet, Sie müssen die korrekte Syntax benutzen. Vielleicht haben Sie ein Schlüsselwort falsch geschrieben oder am falschen Platz falsch verwendet, oder nicht alle benötigten Zeichen zu gesetzt (Klammern, Punkte usw.)
- *Laufzeit-Fehler (Fehler zur Laufzeit)*
  Dieser Fehler tritt auf, wenn KBasic einen Fehler zur Laufzeit findet, also wenn Ihr Programm ausgeführt wird. Ein Beispiel für einen solchen Fehler ist das ‚Dividieren durch Null', das in keinen mathematischen Ausdruck erlaubt ist.
- *Logischer Fehler*
  Dieser Fehler tritt auf, wenn Syntax und Laufzeitverhalten in Ordnung sind, das Programm aber nicht arbeitet wie beabsichtigt oder erwartet. Sie können nur sicher sein, dass Ihr Programm arbeitet, wie Sie wollen, wenn Sie jedes Ergebnis prüfen und analysieren.

Wenn ein Laufzeitfehler eintritt, wird KBasic die Ausführung Ihres Programms anhalten, wenn keine Fehlerbehandlung an der Stelle definiert wurde, wo der Fehler auftrat.

Die alte Fehlerbehandlungssyntax wird unterstützt wie ‘On Error GoTo', aber sie sollte die neue Fehlerbehandlungs-Syntax benutzen (Exceptions / Ausnahmen).

Ausnahmen

In KBasic werden solche Arten von Ereignissen/Fehlern, die ein Programm veranlassen, fehl zu schlagen, Ausnahmen genannt. Die Ausnahme-Behandlung ist eine wichtige Eigenschaft von KBasic. Eine Ausnahme ist ein Signal, das anzeigt, dass eine nicht-normale Phase erreicht ist (wie ein Fehler). Eine Ausnahme zu erzeugen bedeutet, diese spezielle Phase anzuzeigen. Eine Ausnahme zu erfassen bedeutet, eine Ausnahme zu behandeln, einige Befehle zu geben, die sich mit dieser speziellen Phase befassen mit der Aufgabe, den normalen Zustand wieder herzustellen.

Beispiel:

```
Public Sub tt2() Throws samba

  Try
    test()
  Catch (b As rumba)
    Print "tt2: got you!"
    b.dance()
  Finally
    Print "tt2: will be always executed, whatever happend"
  End Catch

End Sub
```

Ausnahmen laufen vom Ursprung zur nächsten Kontroll-Struktur (Aufruf der aktuellen Prozedur). Als erstes wird geprüft, ob eine ‚Try – Catch' – Anweisung oder eine ‚Throw' – Anweisung im aktuellen Bereich (Prozedur etc.) definiert wurde. Wenn nicht, sucht Kbasic den Aufruf des aktuellen Bereiches und versucht wieder, eine ‚Try – Catch' – oder ‚Throw' – Anweisung zu finden usw. Wenn keine gefunden wurde, zeigt KBasic dem Anwender den Fehler auf und die Ausführung des Programms wird beendet.

Ausnahme - Objekte

Eine Ausnahme ist ein Objekt, das eine Instanz von 'Object' ist oder jeder anderen Klasse. Und weil Ausnahmen Objekte sind, können sie Daten und Methoden enthalten wie jede andere Klasse oder (jedes andere) Objekt auch.

Ausnahme - Behandlung

Die 'Try – Catch' – Anweisung wird benötigt, um eine Ausnahme abzufangen. ‚Try' schließt den normalen Code ein, der gut laufen sollte. ´Catch' enthält die Befehle, die ausführt werden, wenn eine Ausnahme auftritt. In ´Finally' stehen einige Befehle, die immer ausgeführt werden, ob eine Ausnahme stattfindet oder nicht.

Try

´Try' schließt den normalen Code ein, der gut laufen sollte. ´Try' arbeitet nicht allein. Es tritt immer mit mindestens einer ´Catch' - oder ´Finally' – Anweisung auf.

Catch

Nach 'Try' und seinen Befehlen können Sie so viele 'Catch' – Anweisungen definieren, wie Sie wollen. ´Catch' – Anweisungen sind ähnlich einer Variablen-Deklaration.

Finally

'Finally' ist hilfreich, wenn Sie Befehle haben wie Datei-Zugriffe oder das Schließen einer Datenbank, die immer ausgeführt werden müssen, wenn eine Ausnahme eingetreten ist.
Wenn Sie eine ´Catch' – Anweisung und eine ´Finally' – Anweisung definiert haben und die passende Ausnahme ist eingetreten, wird zuerst die ´Catch' – Anweisung ausgeführt und danach die ´Finally' – Anweisung.

Deklaration von Ausnahmen

KBasic verlangt, dass jede Prozedur, die eine Ausnahme hervorrufen kann, eine Ausnahme mit einer ´Throw' – Anweisung in seiner Deklaration definieren muss.

Beispiel:

```
Sub mySub() Throws myException1, myException2
' statements which can raise myException1 or myException2
...
End Sub
```

Definieren und Erzeugen von Ausnahmen

Sie können Ihre eigene Ausnahme hervorrufen, indem Sie die ´Throw' – Anweisung benutzen. Nach ´Throw' müssen Sie das Throw-Objekt angeben. Häufig erzeugen Sie dies Objekt mit Hilfe der ´New' – Anweisung, bevor es ausgelöst wird.

```
Throw New MyException1()
```

Syntax:

```
Throw ExceptionObject
```

Wenn eine Ausnahme ausgelöst wurde, wird die normale Programm-Ausführung verlassen und KBasic versucht, für diese Ausnahme eine passende ´Catch' - Anweisung zu finden. Es durchsucht alle Anweisungen, die die Stelle einschließen, an der die Ausnahme ausgelöst wurde. Alle ´Finally' – Anweisungen werden auf diesem Weg ausgeführt. Ausnahmen zu benutzen ist ein geschaffener Weg, um mit Ausnahmen umzugehen, es ist leicht zu handhaben und klar zu lesen. Oft können Sie ein einfaches Fehler-Objekt verwenden, manchmal aber ist es besser, ein eigenes, selbst-erzeugtes Fehler-Objekt zu benutzen.

Syntax:

```
Try
  [Statements]
Catch (Name As Exception)
  [Statements]
End Catch

Try
  [Statements]
Catch (Name As Exception)
  [Statements]
Catch (Name As Exception)
  [Statements]
End Catch

Try
  [Statements]
Catch (Name As Exception)
  [Statements]
Finally
  [Statements]
End Catch
```

Ausführliches Beispiel:

```
Class rumba

  Sub dance
    Print "rumba.dance"
```

```
  End Sub

End Class

Class samba

  Sub dance
    Print "samba.dance"
  End Sub

End Class


Public Sub test() Throws rumba, samba
  Throw New rumba  ' return rumba = new rumba
End Sub

Public Sub tt2() Throws samba

  Try
    test()
  Catch (b As rumba)
    Print "tt2: got you!"
    b.dance()
  Finally
    Print "tt2: will be always executed, whatever happend"
  End  Catch

End Sub


Public Sub tt()

  tt2()

Catch (c As samba)
  Print "tt: got you!"
  c.dance()
Finally
  Print "tt: will be always executed, whatever happend"
End Sub


tt()
```

Ausnahmen am Ende einer Prozedur

Es ist möglich, am Ende einer Prozedur eine Ausnahme abzufangen, die die ganze Prozedur schützt. Das macht es leichter, Ihren Code zu lesen und ähnelt dem alten 'On Error GoTo'. Schreiben Sie die erforderlichen ´Catch' – und ´Finally' – Anweisungen am Ende der Prozedur. Eine ´Try' – Anweisung ist nicht erforderlich, da die ganze Prozedur gemeint ist.

Beispiel:

```
Public Sub tt()

  tt2()

Catch (c As samba)
  Print "tt: got you!"
  c.dance()
Finally
  Print "tt: will be always executed, whatever happend"
End  Sub
```

Syntax:

```
Catch (Name As Exception)
  [Statements]
Finally
  [Statements]
End Catch
```

Die KBasic Entwicklungsumgebung

Die KBasic Entwicklungs-Umgebung enthält Fenster, Werkzeugleisten und Editoren, die das Entwickeln Ihrer KBasic-Anwendungen leicht machen. Es ist eine Íntegrierte Entwicklungs-Umgebung (Integrated Development Environment (IDE)). Wenn von einer Entwicklungsumgebung gesagt wird, sie sei integriert, so bedeutet dies, dass die Werkzeuge in der Umgebung zusammenarbeiten. Der Kompiler zum Beispiel mag einen Fehler im Quell-Code finden. Zusätzlich zur Anzeige des Fehlers öffnet KBasic die Quell-Datei im Text-Editor und springt genau zu der Zeile im Quell-Code, in der der Fehler aufgetreten ist. Ein großer Teil der Anwendungsentwicklung bringt das Hinzufügen und Anordnen von Steuerelementen auf Formularen mit sich. KBasic stellt Werkzeuge bereit, die das Entwerfen von Formularen zu einem einfachen Vorgang machen.

Fenster

KBasic Fenster sind die Werkzeuge auf Ihrem Bildschirm, den Sie zur Entwicklung Ihrer Programme benutzen, einschließlich der Überwachung des Status Ihres Projektes zu jeder Zeit. Diese Fenster beinhalten:

- Formular-Gestalter – ziehen Sie die Steuerelemente auf diese Hauptentwicklungsebene, legen Sie sie dort ab und ordnen Sie sie nach Ihren Vorstellungen an. Dies ist das wichtigste Werkzeug, das Sie nutzen, um Ihr Programm zu erzeugen.
- Projekt-Fenster – arbeiten Sie mit den unterschiedlichen Teilen Ihres Projektes in diesem Fenster. Angezeigt werden auch Objekte und Dateien.
- Eigenschaften-Fenster – zeigt die Eigenschaften Ihrer Steuerelemente an und lässt Sie diese verwalten. Sie können an Ihren Steuerelementen die Größe verändern, einen Namen angeben, die Sichtbarkeit ändern, Werte zuweisen, Farben und Zeichensätze verändern.
- Werkzeug-Fenster – der zentrale Ort, wo auf häufig benutzte Steuerelemente zugegriffen werden kann.

Quell-Code-Editor – hier ist es, wo Sie in jeder Phase der (Programm-)Entwicklung den Quell-Code für Ihr Projekt hinzufügen und bearbeiten.

Werkzeugleisten

KBasic stellt einen umfassenden Satz von Werkzeugleisten bereit. Werkzeugleisten können im KBasic-Fenster angekoppelt werden oder auf dem Bildschirm schweben.

Editor

KBasic hat einen Editor zum erzeugen und verwalten von KBasic-Projekten. Sie können diesen Editor benutzen, um die Entwicklung Ihrer Projekte zu steuern wie Quell-Codes bearbeiten und Klassen beeinflussen. Der Quell-Code-Editor ist ein Werkzeug zum Bearbeiten von Quell-Code.

Debugger

Eines der mächtigsten Werkzeuge in der KBasic-Umgebung ist der eingebaute Debugger (nur in der professional Version). Der Debugger macht es möglich, die Ausführung Ihres Programms zeilenweise zu betrachten. Während Ihr Programm ausgeführt wird, können Sie verschiedene Teile des Programms beobachten um zu sehen, wie sie sich verhalten. Sie können im Debugger beobachten:

- die Werte von Variablen
- welche Unterprogramme/Funktionen/Methoden aufgerufen werden
- die Reihenfolge, in der Programm-Ereignisse auftreten.

Klassen und Objekte von KBasic

Es gibt zwei Typen von Objekten in KBasic, sichtbare und unsichtbare. Ein sichtbares Objekt ist ein Steuerelement und sichtbar zur Laufzeit und lässt Benutzer und Anwendung miteinander reagieren; Es hat eine Bildschirmposition, eine Größe und eine Vordergrundfarbe. Beispiele für sichtbare Objekte sind Formulare und Knöpfe. Ein unsichtbares Objekt ist zur Laufzeit nicht sichtbar, wie ein Timer. Einige Objekte können andere Komponenten enthalten, so wie ein Anwendungs-Fenster einen Knopf enthält. In KBasic fügen Sie sichtbare Objekte/Steuerelemente Ihrem Formularen hinzu, um Anwendungen zusammenzubauen.

Projekte

Projekte halten Ihre Projekte zusammen. Wenn Sie eine Anwendung in KBasic entwickeln, arbeiten Sie hauptsächlich mit Projekten. Ein Projekt ist eine Ansammlung von Dateien, die Ihr KBasic-Projekt bilden. Sie erzeugen ein Projekt, um diese Dateien zu verwalten und zu ordnen. KBasic stellt ein leichtes aber durchdachtes System zur Verfügung, um die Sammlung von Dateien zu verwalten, die Ihr Projekt bilden. Das Projekt-Fenster zeigt jede Einzelheit in einem Projekt. Der Start einer neuen Anwendung in KBasic beginnt mit dem Erstellen eines Projektes. Bevor Sie also eine Anwendung mit KBasic bilden können, müssen Sie ein neues Projekt erzeugen.
Ein Projekt besteht aus vielen einzelnen Dateien, zusammengefasst in einem Projekt-Verzeichnis, indem eine Datei namens *.kbasic_project liegt und viele andere Dateien:

- *.kbasic_module
- *.kbasic_class
- *.kbasic_form

Formulare

In KBasic-Anwendungen sind Formulare nicht nur Schablonen für Eingaben und Ändern von Daten, sondern sie sind die grafische Schnittstelle Ihrer Anwendung. In den Augen des Betrachters sind sie die Anwendung!

Bei der Erstellung Ihrer Anwendung mittels Formularen steuern Sie den Programmfluss mit Ereignissen (events), die in den Formularen ausgelöst werden.

Formulare halten Ihr KBasic-Programm zusammen!

Zentrale Bedeutung von Formularen

Jedes Formular einer KBasic-Anwendung hat ein ´Formular-Modul' erhalten mit den Ereignis-Prozeduren. Diese Ereignis-Prozeduren reagieren auf Ereignisse, die in dem Formular ausgelöst werden. Zusätzlich kann jedes Modul andere, nicht-ereignisgesteuerte Prozeduren enthalten.
Ein Formular-Modul ist Teil jedes Formulars, und wenn Sie ein Formular kopieren, wird dessen Formular-Modul automatisch mit kopiert. Wenn Sie ein Formular löschen, wird sein Formular-Modul ebenfalls gelöscht. KBasic erzeugt ein Formular-Modul automatisch. So brauchen Sie nur die Ereignis-Prozeduren und andere Prozeduren zu schreiben.

Prozeduren in Formularen

Die Prozeduren in Formular-Modulen eines Formulars sind 'private'-Prozeduren dieses Formulars; Sie können sie nur innerhalb des Formular-Moduls benutzen. Und weil die Prozeduren innerhalb eines Formular-Modules 'private' sind, können Sie gleich-benannte Prozeduren in vielen Formularen verwenden. Sogar die Namen von Prozeduren in globalen Modulen dürfen gleich sein.

Es ist möglich, eine Prozedur in einem Formular-Modul zu erzeugen, die den selben Namen bekommen hat wie eine Prozedur in einem globalen Modul. In diesem Fall verwendet KBasic zwei Regeln, um die richtige Prozedur zu erkennen, die ausgeführt werden soll.

- KBasic durchsucht das aktuelle Formular-Modul als erstes, oder das globale Modul.
- Wenn KBasic keine Prozedur mit dem passenden Namen findet, dann durchsucht es die globalen Module (aber nicht die anderen Formular-Module) nach einem passenden Prozedur-Namen.

Kompatibilität von VB6 und KBasic

KBasic unterstützt zu 100% die Syntax von VB6. Einige Bestandteile sind ebenso gleich. Es ist möglich, eine GUI-Anwendung zu entwickeln mit der bekannten BASIC-Syntax in einer modernen Form. Es kommt mit wirklicher Java-ähnlicher Objekt-Orientierung und (rückwärtiger) Unterstützung für VB6 und Qbasic, da es zu 100% syntax-kompatibel ist. KBasic vereint die ausdrucksvolle Kraft einer objekt-orientierten Sprache wie C++ mit der Vertrautheit und dem leichten Gebrauch von VB6. Es erlaubt Entwicklern mit einer vorhandenen Grundlage von VB6-Anwendungen mit der Entwicklung einer gemischten Windows-, Mac OS X- und Linux-Umgebung zu beginnen, ohne einer steilen Lernkurve zu begegnen: KBasic verwendet das bekannte Vorbild der graphischen Darstellung und besitzt eine vollständige Realisierung der BASIC-Sprache.

Wechseln von VB6 zu KBasic

Im Folgenden sind einige Informationen aufgelistet, die helfen können zu wechseln:

- benutzen Sie nicht () für einen Array-Zugriff, sondern besser []
- benutzen Sie nicht 'Option Old Basic´ oder ´Very Old Basic'.
- benutzen Sie nicht 'On Error Goto' , besser 'Try Catch'

- benutzen Sie nicht 'Nothing', sondern, besser 'Null'

- vermeiden Sie den Gebrauch vom Daten-Typ 'Variant'

- verwenden Sie nicht 'Class initialize', besser benutzen Sie 'Constructor'. Dasselbe gilt für den 'Destructor'

- verwenden Sie nicht 'Call', um ein Unterprogramm oder eine Funktion aufzurufen

- benutzen Sie nicht 'Optional' und 'IsMissing' mit Argumenten in Unterprogrammen oder Funktionen, verwenden Sie besser den voreingestellten Wert eines Arguments.

- verwenden Sie möglichst 'Do While...Loop' und 'Do ...Loop While' anstelle der anderen Schleifen

- schreiben Sie viele Kommentare in Ihre Quell-Code

- verwenden Sie in Bedingungen 'AndAlso' und 'OrElse' anstelle der binären Operatoren 'And' und 'Or'

- vermeiden Sie den Gebrauch von 'ByRef' mit einfachen Daten-Typen, um schnelleren Code zu erzeugen

- benutzen Sie nicht ‘Data’ und ‘Def*’ wie ‘DefInt’
- verwenden Sie Enumerationen anstelle von vielen Integer-Konstanten
- setzen Sie Konstanten ein anstelle von vielen numerischen Literalen in Ihrem Code
- vermeiden Sie den Gebrauch von ‘GoSub’, verwenden Sie besser Funktionen oder Unterprogramme
- vermeiden Sie die Verwendung von ‘GoTo’, benutzen Sie besser Schleifen oder andere Kontrollfluss-Anweisungen
- ‘Let’ und ‘Set’ werden nicht benötigt
- verwenden Sie ‘For Each’, wenn es möglich ist
- verwenden Sie 'For i As Integer = 0 To 10 Next' und deklarieren Sie die Zählervariable nicht außerhalb der Schleife
- benennen Sie Variable ohne Suffixe
- benutzen Sie immer ( ), wenn Sie ein Unterprogramm oder eine Funktion aufrufen

KBasics virtuelle Maschine

Die KBasic virtuelle Maschine ist eine Umgebung, in der KBasic-Programme ausgeführt werden. Sie ist eine Art allgemeiner Computer und bestimmt die Befehle, die ein KBasic-Programm verwenden darf. Diese Befehle werden p-code oder pseudo code genannt. Im Allgemeinen bedeutet das, KBasic-pcode ist dasselbe für die virtuelle Maschine wie der Maschinencode für die CPU (den Prozessor). Ein pcode ist ein Befehl mit 2 Bytes Länge, der vom KBasic-Kompiler erzeugt wird und vom KBasic-Interpreter übersetzt wird. Wenn der Kompiler ein KBasic-Programm übersetzt, erzeugt er eine Reihe von pcodes und speichert sie. Der KBasic-Interpreter ist in der Lage, diese Pseudo-Codes auszuführen. Es ist wichtig zu wissen, dass der Name 'KBasic' viel mehr bedeutet als eine Programmiersprache. Es bedeutet auch eine vollständige Computer-Umgebung. Die Begründung dafür: KBasic enthält zwei Einheiten: KBasic-Entwicklung (Die Programmiersprache) und KBasic-Laufzeit (die virtuelle Maschine).

Zusammengefasst:
Die KBasic Maschine ist ein fiktiver Computer, in dem KBasic-Programme ausgeführt werden. Diese Umgebung kann besondere Befehle ausführen, Pseudo-Codes genannt, die von einem KBasic-Programm durch den KBasic-Compiler erzeugt werden.

Welches sind die Hauptfunktionen der virtuellen Maschine?

Die virtuelle Maschine (VM) führt u.a. folgende Tätigkeiten aus:

- ordnet den reservierten Speicherplatz dem erzeugten Objekt zu
- gibt den Speicher automatisch frei
- verwaltet den Stapelspeicher (stack) und Registrierung von Variablen

Lexikon

Die folgenden Informationen beschreiben festgelegte Bezeichnungen und Schlüsselworte von KBasic. Einige davon sind für die Zukunft reserviert.

Schlüsselworte

Ein Schlüsselwort ist eine festgelegte Bezeichnung, das eine besondere Bedeutung für KBasic hat und das beteutet, es darf nicht verändert werden. Die folgende Liste enthält alle KBasic-Schlüsselworte.

```
$Dynamic #Else $End #ExternalSource #If #Region $Static Absolute
Abstract AddressOf Alias Ansi As Assembly Auto Base ByRef ByVal Call
CallByName Case Catch Chain Choose Class Class_Initialize
Class_Terminate COM Common Compare Const Constructor Data Database
Decimal Declare Def Default DefBool DefByte DefCur DefDate DefDbl
DefInt DefLng DefObj DefSng DefStr DefVar Delegate Destructor Dim
DirectCast Do Each Else ElseIf Empty End EndIf Enum Erase Event Exit
Explicit Finally For Friend Function Global GoSub GoTo Handles If
IIf Implements Imports In Inherits Interface Is Iterate KBasic Key
LBound Let Lib Like Loop LSet Me Mid Module MustInherit MustOverride
MyBase MyClass NameSpace New Next Nothing NotInheritable
NotOverridable Null Off OldBasic On Option Optional Overloads
Overriddable Overrides ParamArray Parent Pen Play Preserve Private
Property Protected Public Range Read ReadOnly ReDim Rem /** /* */ '
Repeat Restore Resume Return RSet Run Select Set Shadows Shared
Signal SizeOf Slot Static Step Stop STRIG Structure Sub Swap Switch
SynClock System Text Then Throw Throws Timer To TROFF TRON Try Type
TypeDef UBound UniCode Until VARPTR VARPTR$ VARSEG VeryOldBasic Wait
Wend While With WithEvents WriteOnly
```

Eingebaute Funktionen

```
__Class__ __File__ __IsClass__ __IsLinux__ __IsMacOS__ __IsModule__
__IsSub__ __IsWindows__ __Line__ __Module__ __Scope__ __Sub__ Abs
Access Acs AddHandler AppActivate Append Array Asc Asn Atn Beep Bin
Bin$ Binary BLOAD BSAVE CBCD CBool CByte CChar CCur CDate CDbl CDec
CEXT CFIX ChDir ChDrive Chr Chr$ CInt Circle Clear CLng Close CLS
CObj Color Command Command$ Cos CQUD CreateObject CShort CSng CsrLin
CType CurDir CurDir$ CVD CVDMBF CVERR CVI CVL CVS CVSMBF Date Date$
DateAdd DateDiff DatePart DateSerial DateValue Day DDB Deg
DeleteSetting Dir Dir$ DoEvents DOF Draw Environ Environ$ EOF ErDev
ErDev$ Erl Err Error Error$ Exp Fact Field FileAttr FileCopy
FileDateTime FileLen Files Filter Fix FN Format Format$
FormatCurrency FormatDateTime FormatNumber FormatPercent Frac FRE
FreeFile FV Get GetAllSettings GetAttr GetAutoServerSettings
GetObject GetSetting GetType Hex Hex$ Hour Hypot IMEStatus Inkey
Inkey$ Inp Input Input$ InputBox InStr InStRev Int IOCtl IOCtl$ IPMT
IRR IsArray IsBoolean IsByte IsCharacter IsCollection IsCString
IsCurrency IsDate IsDouble IsEmpty IsError IsInt16 IsInt32 IsInt64
IsInteger IsMissing IsNull IsNumeric IsObject IsShort IsSingle
IsUInt16 IsUInt32 IsUInt64 IsLong IsString IsVariant Join Kill LCase
LCase$ Left Left$ Len Line Ln Load LoadPicture LoadResData
LoadResPicture LoadResString Loc Locate Lock LOF Log Logb LPos
LPrint LTrim LTrim$ Max Mid$ Min Minute MIRR MKD$ MkDir MKDMBF$ MKI$
```

```
MKL$ MKS MKS$ MKSMBF$ Month MonthName MsgBox MTIMER Name Now NPER
NPV Nz Oct Oct$ Open Out Output Paint Palette Partition PCopy Peek
PMAP PMT Point Poke Pos PPMT Preset Print PSet Put PV QBCOLOR Rad
Raise RaiseEvent RaiseSignal Random Randomize Rate RemoveHandler
Replace Reset RGB Right Right$ RmDir RND Round RTrim RTrim$
SavePicture SaveSetting Screen Sec Second Seek Seg SendKeys SetAttr
Sgn Shell Sin Sleep Sln Sound Space Space$ Spc Split Sqr Stick Str
Str$ StrComp StrConv String String$ StrReverse SYD Tab Tan Time
Time$ TimeSerial TimeValue Trim Trim$ TypeName TypeOf UCase UCase$
UnLoad UnLock Using Val VarType View Weekday WeekdayName Width
Window Write Year
```

Operatoren

```
<< >> Inc Dec += -= /= *= BitAnd BitOr BitXor BitNot + - * / Mod =
<> >= <= > < And AndAlso Or OrElse Not ^ & Xor \ Eqv Imp
```

(Daten-)Typen

```
Boolean Byte Character Collection CString Currency Date Double
Integer Long Object Short Single String Variant
```

ASCII-Codes (codes 0 - 127)

000		(nul)	016	?	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	?	(soh)	017	?	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	?	(stx)	018	?	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	?	(etx)	019	?	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	?	(eot)	020	¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	?	(enq)	021	§	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	?	(ack)	022	?	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	•	(bel)	023	?	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	?	(bs)	024	?	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009		(tab)	025	?	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010		(lf)	026		(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	?	(vt)	027	?	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	?	(np)	028	?	(fs)	044	,	060	<	076	L	092	\	108	l	124	|
013		(cr)	029	?	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	?	(so)	030	?	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	¤	(si)	031	?	(us)	047	/	063	?	079	O	095	_	111	o	127	¦

Anhang

Argument
Ein Wert, der einer Prozedur, Funktion oder Methode übergeben wird. Siehe auch Parameter.

Arithmetische Operationen
Sind mathematische Operationen wie z. B. Addition, Multiplikation, Subtraktion, und Division, die numerische Ergebnisse liefern.

Array
Eine Variable, die eine Serie von Werten speichern, auf die mittels einem Index zugegriffen werden kann.

BASIC
Beginner’s All-Purpose Symbolic Instruction Code, die Programmiersprache, auf der KBasic basiert.

Bit
Die kleinste Informationeinheit eines Computers. Ein Bit kann den Wert 1 oder 0 haben.

Boolean
Ein Datentyp: kann den Wert 'Wahr' oder 'Falsch' annehmen.

Boolscher Ausdruck
Eine Verknüpfung von Ausdrücken, deren Rechenergebnis enweder 'True' oder 'False' ist. Ein Beispiel: (x=5) AND (y=6).

Verzweigung
Wenn die Programm-Ausführung von einem Punkt im Programm zu einem anderem springt, anstatt die Ausführung der Anweisungen in der genauen Reihenfolge fortzusetzen. Siehe auch Bedingte Verzweigung und Unbedingte Verzweigung.

Haltepunkt
Die Stelle in einem Programm, in der der normale Programmablauf durch einen programmierten bedingten Halt angehalten wird. In der Entwicklungs- und Testphase von Programmen erleichtern solche Haltepunkte dem Entwickler das Auffinden von Fehlern, da er nach dem Anhalten des Programms Informationen über dessen Status abfragen kann. Siehe auch Debugging (Fehlerbeseitigung).

Byte
Eine Informationseinheit: besteht aus acht Bits. Siehe auch Bit.

Kompiler
Ein Programmier-Werkzeug, das den Quellcode eines Programms umwandelt in eine ausführbare Datei. KBasic verwendet automatisch einen Kompiler, wenn Sie ein Programm ausführen oder den Menü-Befehl “Ausführen/Übersetzen” auswählen, um ein Programm in eine ausführbare Datei umzuwandeln (nur Professional Version). Siehe auch Interpreter.

Verknüpfen
Das Zusammenfügen zweier Text-Stings zu einem einzigen. Zum Beispiel ist der String “OneTwo” eine Verknüpfung der zwei Strings “One” und “Two”.

Bedingte Verzweigung
Wenn die Programmausführung zu einer anderen Stelle im Programm verzweigt als Folge einer Art von Bedingung. Ein Beispiel für eine bedingte Verzweigung ist die If/Then – Anweisung, die das Programm veranlasst zu einer Progammzeile zu verzweigen als Folge eine Bedingung wie >> If (x=5) Then <<.

Konstante
Ein vordefinierter Wert, der sich nie ändert.

Datentyp
ie verschiedenen Arten von Werten, die ein Programm speichern kann. Diese Werte umfassen Integer, Long, String, Single, Double und Boolean.

Fehlerbeseitigung
Das Finden und Beheben von Fehlern in einem Programm.

Verminderung
Reduziert den Wert einer Variablen, für gewöhnlich um 1. Siehe auch Erhöhung.

Double
Ein Datentyp: Er repräsentiert den höchst- genauen Gleitkomma-Wert, auch bekannt als Gleitkommawert doppelter Genauigkeit. Siehe auch Gleitkomma(Zahl) und Single (einfache Genauigkeit)

Leere Zeichenkette

Eine Zeichenkette die die Länge 0 (=Null) hat, gekennzeichnet in einem Programm durch zwei doppelte Anführungszeichen. Das folgende Beilspiel weist der Variablen 'str1' eine leere Zeichenkette (Leerstring) zu: >> str1 = “” <<.

Ereignis

Eine Nachricht, die zu einem Programm gesendet wird als Ergebnis einer Kommunikation zwischen dem Anwender und dem Programm.

Ausführbare Datei

Eine Datei, für gewöhnlich eine Anwendung, die der Komputer laden und ausführen kann. Die meisten ausführbaren Dateien besitzen eine (entsprechende) Erweiterung, in Windows zum Beispiel .exe

Datei

Ein bezeichneter Satz von Daten auf einer Diskette

Gleitkomma

Ein Zahlenwert, der einen dezimalen Anteil hat. Zum Beispiel sind 12.75 und 235.7584 Gleitkomma-Werte (-Zahlen). Siehe auch Double und Single

Formular

Das KBasic-Objekt, das ein Anwendungs-Fenster darstellt. Das Formular ist das Gehäuse, in das Sie die Steuerelemente plazieren, die die Benutzer-Schnittstelle Ihres Programms darstellt.

Funktion

Ein Unterprogramm, das Daten irgendwie bearbeitet und einen einzelnen Wert zurückgibt, der das Ergebnis dieser Bearbeitung ist.

Globale Variable

Ein bezeichneter Wert (eine Variable), auf die von überall innerhalb eines Programm-Moduls zugegriffen werden kann.

Erhöhung

Erhöhen des Wertes einer Variablen, für gewöhnlich um 1. Siehe auch Verminderung.

Endlosschleife

Eine Schleife, die nie enden kann, weil ihre Abbruchbedingung nie eintritt. Eine Endlosschleife kann man nur durch Abbrechen des Programms beenden. Siehe auch Schleife und Steuerungs-Variable.

Initialisieren

Das Setzen des Startwertes einer Variablen

Integer

Ein Datentyp: Er stellt ganze Zahlen dar im Bereich von –2,147,483,648 bis 2,147,483,647. Die Werte 240, -128 und 2 sind Beispiele für Integer-Zahlen. Siehe auch long.

Interpreter

Ein Programmier-Werkzeug, das den Quell-Code übersetzt und ausführt, und zwar Zeile für Zeile. Ein Interpreter kann kein eigenständiges Programm erzeugen. Siehe auch Kompiler.

Literal

Eine konstante Zahl oder Zeichenwert. 'x' , '2' und 1.22 sind Literale. Der Wert eines Literals ist durch die verwendete Sprache eindeutig bestimmt und ist nicht in einer Variablen gespeichert.

Lokale Variable

Eine Variable, auf die nur innerhalb des Unterprogramms zugegriffen werden kann, in dem sie deklariert ist. Siehe auch Globale Variable und Variablen Bereich.

Logischer Fehler

Ein Programmierfehler, der eintritt, wenn ein Programm eine andere Aufgabe durchführt als der Programmierer dachte, die er zur Durchführung programmiert hätte. So wäre zum Beispiel in der Programmzeile >> If X = 5 Then Y = 6 << ein logischer Fehler, wenn die Variable X nie 5 werden kann. Siehe auch Laufzeitfehler.

Logischer Operator

Ein Symbol, das zwei Ausdrücke miteinander vergleicht. Das Ergebnis ist ein boolscher Wert (True oder False / Wahr oder Falsch). Zum Beispiel ist in der Zeile >> If X = 5 AND Y = 10 Then Z = 1 << 'AND' der logische Operator. Siehe auch Vergleichsoperator.

Long

Ein Datentyp: Er stellt Integer-Werte dar im Bereich von -2^{32} - 1 bis $+2^{32}$. Siehe auch Integer.

Schleife

Ein Quellcode-Block, der mehrfach ausgeführt wird bis eine bestimmte Bedingung eingetroffen ist.

Steuerungs-Variable

Eine Variable, die den Wert hält, der bestimmt, ob eine Schleife weiterhin ausgeführt wird.

Maschinensprache

Die einzige Sprache, die ein Komputer wirklich versteht. Der ganze Programm-Quell-Code muss in die Maschinensprache umgewandelt werden bevor der Komputer das Programm ausführen kann.

Mathematischer Ausdruck

Eine Reihe von Ausdrücken, welche Rechenoperatoren benutzen, um einen Zahlenwert zu erzeugen. Die Ausdrücke >> (X+ 17)/(Y+ 22) << stellen zum Beispiel einen mathematischen Ausdruck dar. Siehe auch Rechenoperationen.

Methode

Eine Prozedur, die zu einem Objekt oder Steuerelement gehört, und die eine Fähigkeit eines Objektes oder Steuerelementes darstellt. Zum Beispiel positioniert die 'Move' – Methode eines Befehlsknopfes diesen neu auf dem Formular.

Zahlen Literal (Zahl)

Ein buchstäblicher (elementarer) Wert, der eine Zahl darstellt, wie zum Beispiel 125 oder 34.87. Siehe auch Literal und Zeichenketten - Literal.

Zahlenwert

Ein Wert, der eine Zahl darstellt. Dieser Wert kann ein Literal sein, eine Variable oder das Ergebnis einer Berechnung.

Objekt

Generell ein Teil der Daten eines Programms. Speziell in KBasic ein Satz von Eigenschaften und Methoden, der sozusagen irgendein Objekt aus der realen Welt darstellt oder eine allgemeine Idee.

Reihenfolge der Operationen

Die Reihenfolge, in der KBasic Rechenoperationen löst. Ein Beispiel: In dem mathematischen Ausdruck >> (X+5)/(Y+") << wird KBasic erst die beiden Additionen (in den Klammern) ausführen und dann die Division. Werden die Klammern entfernt wie in dem Ausdruck >> X+5/Y+2 << würde KBasic erst 5 durch Y teilen und dann die verbleibenden Additionen durchführen.

Parameter

Oftmals besagt es dasselbe wie "Argument", trotzdem unterscheiden einige Leute zwischen Argument und Parameter, wobei ein Argument der Wert ist, der einer Prozedur oder Funktion übergeben wird, und ein Parameter ist die Variable innerhalb der Funktion oder Prozedur, die das Argument erhält (empfängt). Siehe auch Argument.

Prozedur

Ein Unterprogramm, das eine (Teil-)Aufgabe in einem Programm ausführt, jedoch keinen Wert zurückgibt. Siehe auch Funktion.

Programm

Eine Liste von Anweisungen für einen Komputer.

Programmiersprache

Eine Reihe von englisch-ähnlichen Schlüsselworten und Symbolen, die einen Programmierer in die Lage versetzen, ein Programm zu schreiben ohne die Maschinensprache zu verwenden.

Programmfluss

Die Reihenfolge, in welcher ein Komputer Programm-Anweisungen ausführt.

Eigenschaft

Ein Wert, der ein Merkmal von einem Objekt oder Steuerelement repräsentiert.

Ausschließlich lesbare Eigenschaft

Eine Eigenschaft, deren Wert nicht durch das Programm verändert werden kann. Allerdings kann ein Programm einen (solchen) Eigenschaften-Wert lesen.

Vergleichsoperator

Ein Symbol, das die Beziehung zwischen zwei Ausdrücken festlegt. In dem Beispiel "X>10" ist '>' der Vergleichsoperator, der "Größer als" bedeutet. Siehe auch Logischer Operator.

Rückgabe-Wert

Der Wert, den eine Funktion an die Anweisung zurückgibt, die die Funktion aufgerufen hat. Siehe auch Funktion.

Laufzeitfehler

Ein Systemfehler, der eintritt, während das Programm läuft. Ein Beispiel ist ein "Teilen-durch-Null"-Fehler oder eine Unverträglichkeit der Typen. Ohne irgendeine Art von Fehlerbehandlung, wie sie zum Beispiel durch die 'Try/Catch'-Anweisung sichergestellt wird, führt ein Laufzeitfehler häufig zu einem Programm-Absturz.

Sichtbarkeitsbereich
Siehe Variablen Bereich

Single (Einfache Genauigkeit)
Ein Datentyp: Er repräsentiert den am wenigsten-genauen Gleitkomma-Wert, auch bekannt als Gleitkomma-Wert (-Zahl) einfacher Genauigkeit.

Quell-Code
Die Zeilen von Befehlen, die ein Programm bilden.

Zeichenkette
Ein Datentyp: Er stellt einen oder mehr Text-Schriftzeichen dar. Zum Beispiel muss in der Zuweisungs-Anweisung >> str1 = “I'm a string” << die Variable 'str1' vom Datentyp 'String' (oder 'Variant') sein.

Zeichenketten-Literal
Ein oder mehr Text-Schriftzeichen, eingeschlossen in doppelte Anführungszeichen.

Unterprogramm
Ein Block von Quellcodes, der einen speziellen Teil einer größeren Aufgabe ausführt. In KBasic ist ein Unterprogramm entweder eine Prozedur oder eine Funktion. Siehe auch Funktion und Prozedur.

Unbedingte Verzweigung
Wenn die Programmausführung an eine andere Stelle verzweigt ungeachtet jeder Bedingung. Ein Beispiel einer unbedingten Verzweigung ist das 'GoTo'.

Benutzer Schnittstelle
Die sichtbare Darstellung einer Anwendung, für gewöhnlich bestehend aus verschiedenen Typen von Steuerelementen, die es dem Benutzer ermöglichen, mit der Anwendung zu kommunizieren.

Variable
Ein benannter Wert innerhalb eines Programms. Diesem Wert kann wieder und wieder ein Wert eines passenden Datentyps zugewiesen werden.

Variablen Bereich
Der Bereich eines Programms, in dem eine Variable zugänglich gemacht wird. Zum Beispiel ist der Bereich einer globalen Variablen irgendwo innerhalb des Programms, wogegen der Bereich einer lokalen Variablen begrenzt ist auf die Prozedur oder Funktion, die die Variable deklariert. Siehe auch Globale Variable und Lokale Variable.

Variant
Ein besonderer Datentyp: Er kann jeden Datentyp repräsentieren. Genauer, KBasic verwaltet den Datentyp eines Variant-Wertes für Sie. Dieser Datentyp ist sehr unrationell und sollte vermieden werden.

Kontakt/Impressum

www.kbasic.com
email: info@kbasic.com

Qualität von Bernd Noetscher

Made in Germany (European Union)

Eingetragene Warenzeichen

Linux® ist ein eingetragenes Warenzeichen von Linus Torvalds. Alle andere Produktnamen, die in diesem Text genannt werden, sind eingetragene Warenzeichen ihrer jeweiligen Eigentümer.