Andrew Pitonyak

OpenOffice.org-Makros Erklärt

Ins Deutsche übertragen von Volker Lenhardt

Lfg. 1 (Kapitel 1 - 4)

Letzte Änderung

Freitag, 1. Juli 2011 um 18:59:02

Die amerikanische Originalausgabe erscheint unter dem Titel

OpenOffice.org Macros Explained, 3. ed.

ODT-Version: http://www.pitonyak.org/OOME_3_0.odt

PDF-Version: http://www.pitonyak.org/OOME_3_0.pdf

Allgemeine Information

Ich hatte eine allerdings nie veröffentlichte zweite Auflage geschrieben. Daher bezeichne ich dieses Buch als die dritte Auflage.

Andrew D. Pitonyak

Anmerkung des Übersetzers

Diese Übersetzung wird nach und nach in der Reihenfolge der Originalkapitel vervollständigt und somit noch eine ganze Weile als Torso erscheinen.

Ich habe vor, nach jedem neu übersetzten Kapitel eine Aktualisierung der Downloadfassung vorzunehmen. Die Leser mögen also in gewissen Abständen den aktuellen Stand überprüfen.

Für die Übersetzung verwende ich zur Zeit OpenOffice.org 3.2.1 (Originalversion) und LibreOffice 3.3.1 (openSUSE-Brand) unter dem Linux-Betriebssystem openSUSE 11.3 (64 Bit) und der Benutzeroberfläche KDE 4.5.5.

Die Menübefehle und Bildschirmfotos der Originalfassung sind durch die der deutschen Benutzeroberfläche ersetzt, und zwar die von OpenOffice.org, falls sie sich von LibreOffice unterscheiden. LibreOffice-Nutzer müssen im allgemeinen nur „LibreOffice“ an die Stelle von „OpenOffice.org“ setzen.

Ich habe in diesem Buch alle Variablennamen in Makros im allgemeinen in ihrer englischen Originalform belassen und habe sie kommentiert, wenn deren Bedeutung aus dem Zusammenhang nicht direkt erschlossen werden konnte. Wenn es mir aber notwendig schien, habe ich sie übersetzt.

Es gibt eine Reihe von Gründen, weshalb es sinnvoll ist, auch im muttersprachlichen Umfeld für Variablen englische Bezeichner zu wählen: Verben ohne Flexionen, einsilbige Wörter, harmonischer Einklang mit den englischsprachigen Anweisungen und Objektmethoden und -eigenschaften, sowie – last but not least – internationale Hilfemöglichkeiten im Falle, dass man im Internet nachfragen muss, weil etwas nicht so funktioniert, wie man geglaubt hat.

Hier und da können Zahlen mit Tausenderpunkt und Dezimalzahlen für Verwirrung sorgen. Alle Zahlen in Basic-Anweisungen sind ohne Tausendertrenner und mit einem Dezimalpunkt zu schreiben. Die Ausgabe mit Print oder MsgBox verwendet jedoch die lokalisierte Form mit Dezimalkomma. Im laufenden Text schreibe ich Zahlen auf deutsche Weise mit Dezimalkomma und falls angebracht mit Tausenderpunkt, beim Zitieren von Basic-Anweisungen jedoch mit Dezimalpunkt.

Volker Lenhardt

Inhaltsverzeichnis

1. Einführung und Organisation

Am Anfang stand die erste Auflage von OpenOffice.org Macros Explained (OOME). Ein paar Jahre später stellte ich die zweite Auflage fertig, aktualisiert als Anpassung an die OpenOffice.org (OOo)-Version 2.x. Diese zweite Auflage wurde jedoch nie veröffentlicht. Nun denke ich, es wird Zeit für die dritte Auflage.

Der größte Teil des Inhalts der vorherigen Auflagen ist erhalten geblieben. Die ersten Kapitel, die die Sprachsyntax behandeln, sind im wesentlichen unverändert bis auf die neu hinzugekommenen Sprachelemente.

Seit der letzten Veröffentlichung hat sich die Anzahl der von OOo unterstützten Services mehr als verdoppelt, und die Funktionalität ist erheblich erweitert worden. Leider ist der Leistungsumfang größer, als ich Zeit oder Raum habe zu dokumentieren. So umfangreich dieses Buch auch ist, es fehlt leider noch viel. Sie sollten dieses Buch daher als Nachschlagewerk mit einer Vielzahl an Beispielen nutzen, aber immer daran denken, dass OOo in einem kontinuierlichen Wandel steckt und immer wieder neue Funktionalitäten unterstützt.

Das Dokument enthält Schaltflächen zum Starten der Makros, die im Text vorgestellt werden. Das ist zwar fantastisch, wenn man den Originalquelltext liest, produziert aber beim Ausdrucken unerwünschte Artefakte. Tut mir leid.

1.1. In eigener Sache

Ich bin der Hauptautor dieses Dokuments, ich bestreite meinen Lebensunterhalt nicht mit der Arbeit mit OOo, und nichts in diesem Buch hat mit meinem Hauptberuf zu tun. Mit anderen Worten, ich bin einfach irgendein Mitglied der OOo-Gemeinschaft, der dies hier weitgehend ohne Entlohnung tut.

Ich erhalte zahllose Bitten um Hilfe, weil ich in der OOo-Gemeinschaft prominent bin. Unglücklicherweise ist meine Zeit schon über Gebühr beansprucht, und es ist schwierig, allen persönlich zu Hilfe zu kommen. Ich helfe gerne in meiner nicht vorhandenen Freizeit, aber bitte, nutzen Sie nach Möglichkeit schon vorhandenes Material, Mailinglisten und Foren. Gelegentlich biete ich Lösungen auf Vergütungsbasis an, aber für größere Projekte fehlt mir einfach die Zeit.

Ich begrüße Kommentare und Bug-Reports. Wenn Sie glauben, etwas Interessantes sollte mit aufgenommen werden, lassen Sie es mich wissen. Wenn Sie einen ganzen Abschnitt oder ein Kapitel schreiben wollen, tun Sie es. Wahrscheinlich werde ich Ihr Werk heftig bearbeiten. Ich bitte um Ihr Feedback und Anregungen.

1.2. OpenOffice.org-Bug

OOo nutzt Formatvorlagen. Wenn sich etwas in der Formatierung ändert, können Sie daher davon ausgehen, dass eine andere Vorlage dahinter steckt. Ich verwende sorgfältig formatierten Programmcode, aber hinter jedem Wechsel des Formats steht ein Wechsel der Formatvorlage. Es gibt einen OOo-Bug, der OOo abstürzen lässt, wenn die Formatvorlage mehr als 64K-mal wechselt (s. http://www.openoffice.org/issues/show_bug.cgi?id=84159). Aus diesem Grund stürzt OOo ab, wenn AndrewMacro.odt geschlossen wird. Ich nehme an, dass dieses Dokument unter demselben Problem leiden wird. Wenn ich an diese Grenze stoße, wird das Dokument weitgehend unbenutzbar sein. Falls Sie also Votes abgeben können, votieren Sie für Issue 84159.

Tipp In der OOo-Version 3.4 soll der Bug gefixt sein. Ich möchte speziell Bartosz danken, einem Mitglied der Gemeinschaft, und all den anderen, die sich für die Behebung dieses Bugs eingesetzt haben.

1.3. Arbeitsumgebung und Kommentare

Die Hauptarbeit an diesem Buch wurde mit der offiziellen 64-Bit-Linuxversion unter Fedora Linux geleistet. Es fanden sich einige Bugs der 64-Bit-Distribution. Der Text enthält wahrscheinlich Fehler, und ich bin sehr daran interessiert, sie zu erfahren.

2. Die Grundlagen

In OpenOffice.org (OOo) werden Makros und Dialoge in Dokumenten und Bibliotheken gespeichert. Die integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) dient zum Erstellen und zum Debuggen von Makros und Dialogen. Dieses Kapitel führt in die Grundkonzepte des Starts der IDE und der Makroerstellung ein. Es zeigt schrittweise den Aufbau eines einfachen Makros, das den Text „Hallo Welt“ auf dem Bildschirm anzeigt.

Ein Makro ist eine für den späteren Gebrauch gespeicherte Folge von Anweisungen oder Tastenkombinationen. Ein Beispiel für ein einfaches Makro wäre eines, das Ihre Adresse ausgibt. Makros unterstützen Anweisungen für eine Vielzahl weiterreichender Funktionen: Entscheidungen zu treffen (wenn zum Beispiel die Differenz kleiner ist als Null, zeige sie in Rot, falls nicht, zeige sie in Schwarz), Schleifen zu durchlaufen (solange die Differenz größer ist als Null, subtrahiere 10), und sogar mit einer Person zu kommunizieren (den Nutzer zum Beispiel nach einer Zahl zu fragen). Einige dieser Anweisungen basieren auf der Programmiersprache BASIC. (BASIC ist ein Akronym für Beginner’s All-purpose Symbolic Instruction Code.) Üblicherweise bindet man ein Makro an eine Tastenkombination oder ein Werkzeugleisten-Symbol, um es schnell starten zu können.

Ein Dialog – oder Dialogfenster – ist ein Fenstertyp für den „Dialog“ mit einem Nutzer. Der Dialog mag dem Nutzer Informationen bieten oder eine Eingabe vom Benutzer erwarten. Sie können Ihre eigenen Dialoge erschaffen und sie in einem Modul mit Ihren Makros speichern.

Die OpenOffice.org-Makrosprache ist sehr flexibel. Sie erlaubt die Automatisierung sowohl einfacher wie auch komplexer Aufgaben. Obwohl es viel Spaß machen kann, Makros zu schreiben und die internen Abläufe von OpenOffice.org kennenzulernen, ist es nicht immer der beste Ansatz. Makros sind vor allem nützlich, wenn dieselbe Aufgabe immer wieder durchgeführt werden muss, oder wenn man einen einzigen Tastendruck wünscht für etwas, das normalerweise mehrere Schritte benötigt. Hier und da wird man auch ein Makro schreiben, um etwas zu tun, dass man ansonsten in OpenOffice.org nicht tun kann, aber in einem solchen Fall sollte man vorher gründlich recherchieren, um sich zu vergewissern, dass OOo es auch wirklich nicht kann. Zum Beispiel findet man in einigen OpenOffice.org-Mailinglisten regelmäßig die Bitte um ein Makro zum Entfernen leerer Absätze. Diese Funktionalität steht über die AutoKorrektur zur Verfügung (**Extras > AutoKorrektur-Optionen > Optionen**: *Leere Absätze entfernen*). Man kann auch reguläre Ausdrücke verwenden, um Leerzeichen zu suchen und zu ersetzen. Es gibt eine Zeit und einen Anlass für Makros und eine Zeit für andere Lösungen. Dieses Kapitel wird Sie auf die Zeiten vorbereiten, wenn ein Makro das Mittel der Wahl ist.

2.1. Makrospeicherung

In OpenOffice.org werden logisch zusammengehörende Prozeduren in einem Modul gespeichert. Ein Modul könnte zum Beispiel Prozeduren enthalten zum Auffinden typischer Fehler, die weitere Bearbeitung erfordern. Logisch zusammenhängende Module werden in einer Bibliothek gespeichert, und Bibliotheken wiederum werden in Bibliothekscontainern gespeichert. Die OpenOffice.org-Anwendung kann, wie auch jedes OOo-Dokument, als Bibliothekscontainer fungieren. Zusammenfassend gesagt, die OpenOffice-Anwendung und jedes OpenOffice-Dokument können Bibliotheken, Module und Makros enthalten.

Container	Ein Bibliothekscontainer enthält keine, eine oder mehrere Bibliotheken.
Bibliothek	Eine Bibliothek enthält keine, eine oder mehrere Module und Dialoge.
Modul	Ein Modul enthält keine, eine oder mehrere Subroutinen oder Funktionen.

2.1.1. Bibliothekscontainer

Ein OOo-Dokument ist ein Bibliothekscontainer, genauso wie die Anwendung als ganze. Wenn ein bestimmtes Dokument ein Makro benötigt, speichert man das Makro praktischerweise in dem Doku-

ment. Das hat den Vorteil, dass das Makro beim Dokument bleibt. So kann man auch ganz einfach Makros versenden.

Wenn jedoch verschiedene Dokumente dasselbe Makro benötigen und jedes Dokument eine Kopie davon hätte und Sie dann Änderungen an dem Makro vornehmen wollten, müssten Sie es in jedem Dokument ändern, das dieses Makro enthält. In einem Dokument gelagerte Makros sind nur für dieses Dokument sichtbar. Es ist daher nicht einfach, ein Makro in einem Dokument von außerhalb des Dokuments aufzurufen.

Tipp Speichern Sie keine Makros in einem Dokument, die von außerhalb des Dokuments aufgerufen werden (von seltenen Ausnahmen abgesehen), denn Makros in einem Dokument sind nur für dieses Dokument sichtbar.

Der Bibliothekscontainer der Anwendung besteht aus zwei Hauptkomponenten: mit OOo installierte Makros und von Ihnen selbst erstellte Makros. Der OOo-Makrodialog zeigt Ihre Makros in einem Container mit dem Namen „Meine Makros“ und die mitgelieferten als „OpenOffice.org Makros“ (s. Bild 1). OpenOffice.org-Makros werden in einem Verzeichnis der Anwendungsinstallation gespeichert, „Meine Makros“ hingegen in Ihrem Nutzerverzeichnis.

Über **Extras > Makros > Makros verwalten > OpenOffice.org Basic** öffnen Sie den Dialog für OOo-Basic-Makros (s. Bild 1). Die Bibliothekscontainer sind die Objekte der obersten Stufe im Bereich „Makro aus“.

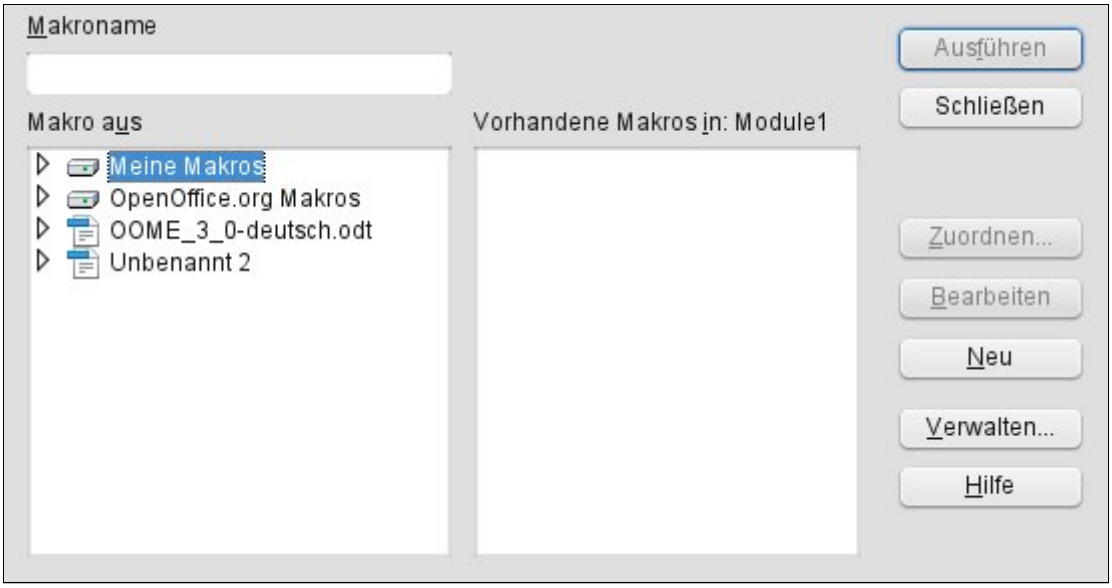


Bild 1. *Im Ooo-Makro-Dialog legen Sie neue Makros an und organisieren die Bibliotheken.*

2.1.2. Bibliotheken

Ein *Bibliothekscontainer* enthält eine oder mehrere Bibliotheken, und eine *Bibliothek* enthält eine oder mehrere Module und Dialoge. Doppelklicken Sie auf einen Bibliothekscontainer im Bild 1, um die enthaltenen Bibliotheken zu sehen. Doppelklicken Sie auf eine Bibliothek, um die Bibliothek zu laden und die enthaltenen Module und Dialoge zu sehen.

Der OOo-Makrodialog zeigt geladene Bibliotheken mit einem anderen Symbol an. Im Bild 2 sind Standard, XMLProcs, XrayTool und Depot geladen, die anderen Bibliotheken nicht.

Tipp Die Symbole und Farben auf Ihrem Rechner können sich von denen der Bildschirmfotos unterscheiden. Unterschiedliche OOo-Versionen können unterschiedliche Symbole und Farben nutzen, und es wird mehr als nur ein Symbol-Set unterstützt. Über **Extras > Optionen > OpenOffice.org > Ansicht** können Sie Größe und Stil der Symbole einstellen.

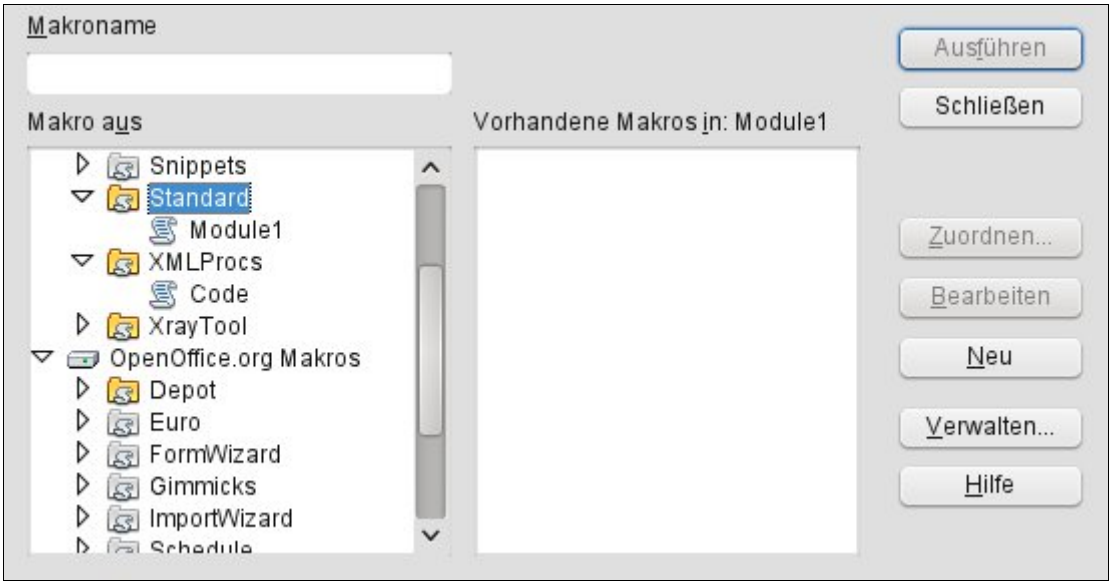


Bild 2. *Geladene Bibliotheken werden anders angezeigt.*

2.1.3. Module und Dialoge

In einem Modul werden typischerweise ähnliche Funktionalitäten auf einer niedrigeren Stufe als einer Bibliothek gruppiert. Die Makros werden in den Modulen gespeichert. Um ein neues Modul anzulegen, markieren Sie eine Bibliothek und klicken auf Neu.

2.1.4. Kernpunkte

Zu beachten:

- Sie können Bibliotheken von einem Bibliothekscontainer in einen anderen importieren.
- Sie importieren ein Modul dadurch, dass Sie die Bibliothek importieren, die das Modul enthält. Es ist über das GUI nicht möglich, einfach ein einzelnes Modul zu importieren.
- Geben Sie den Bibliotheken, Modulen und Makros anschauliche Namen. Anschauliche Namen reduzieren die Wahrscheinlichkeit von Namenskollisionen, die den Bibliotheksimport behindern.
- Die Standardbibliothek spielt eine Sonderrolle: sie wird automatisch geladen, so dass die enthaltenen Makros immer verfügbar sind.
- Die Standardbibliothek wird von OOo automatisch angelegt und kann nicht importiert werden.
- Makros in einer Bibliothek sind erst verfügbar, nachdem die Bibliothek geladen ist.
- Über den Makro-Verwaltungsdialog kann man neue Module anlegen, aber keine neuen Bibliotheken.

Diese Kernpunkte habe gewisse Konsequenzen. Zum Beispiel speichere ich Makros selten in der Standardbibliothek, weil ich die Bibliothek nicht woanders hin importieren kann. Ich verwende die Standardbibliothek normalerweise für Makros, die über Schaltflächen in einem Dokument gestartet werden. Die Makros in der Standardbibliothek laden dann die eigentlichen Arbeitsmakros in anderen Bibliotheken und starten sie.

2.2. Neue Module und Bibliotheken anlegen

Im Makrodialog erstellt die Schaltfläche Neu das Gerüst einer neuen Subroutine in der ausgewählten Bibliothek (s. Bild 1 und Bild 2). Wenn die Bibliothek noch kein Modul enthält, wird nach einer Namensabfrage ein neues angelegt.

Über die Schaltfläche Verwalten öffnen Sie die OOo-Makroverwaltung (s. Bild 3). Die Registerkarten Module und Dialoge sind nahezu identisch. Verwenden Sie die Registerkarten Module und Dialoge, um Module oder Dialoge anzulegen, zu löschen oder umzubenennen.

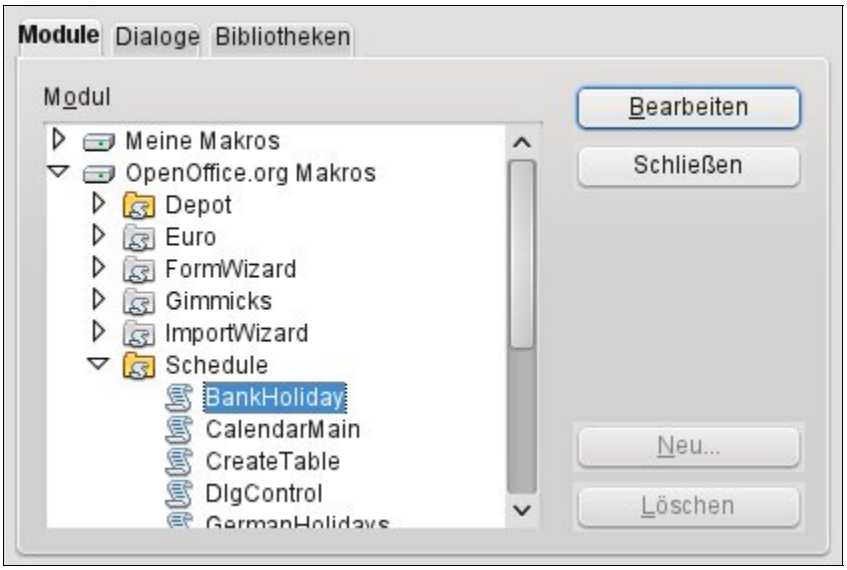


Bild 3. *Die Registerkarte Module der OOo-Makro-Verwaltung.*

Verwenden Sie die Registerkarte Bibliotheken (s. Bild 4), um Bibliotheken anzulegen, zu löschen, umzubenennen, zu importieren oder zu exportieren.

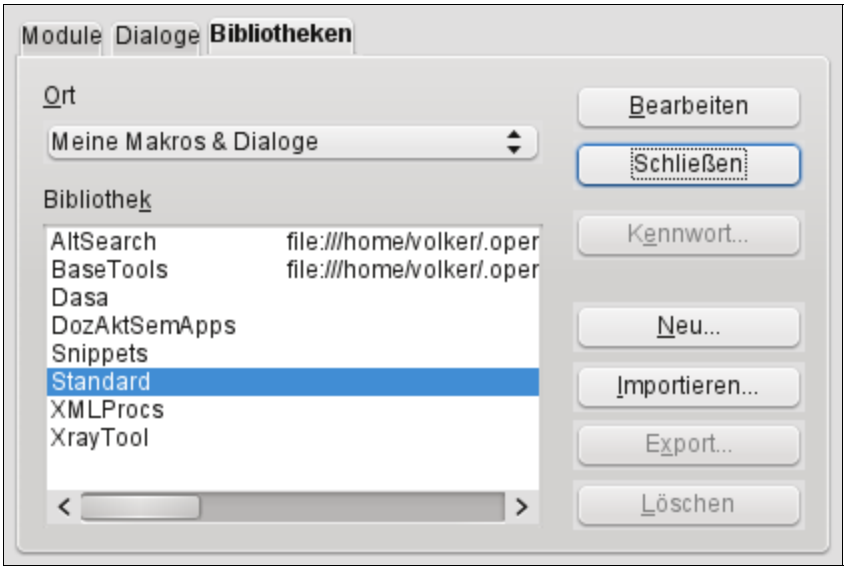


Bild 4. *Die Registerkarte Bibliotheken der OOo-Makro-Verwaltung.*

Aus der Ort-Aufklappliste wählen Sie den gewünschten Bibliothekscontainer. Um eine Bibliothek umzubenennen, doppelklicken Sie auf die Bibliothek und ändern den Namen direkt.

Tipp Ich habe das Umbenennen von Modulen und Bibliotheken in der Makroverwaltung immer als frustrierend erlebt. Um eine Bibliothek umzubenennen, klicken Sie doppelt oder dreifach auf den Namen und warten dann ein paar Sekunden. Versuchen Sie es wieder. Und noch einmal. Wechseln Sie die Bibliothek. Klicken Sie wieder doppelt oder dreifach auf den Namen. Sie merken schon, worum es geht.

Den einfachsten Weg, einen Modulnamen zu ändern, finden Sie in der IDE. Rechtsklicken Sie auf den Modulnamen am unteren Rand und wählen Umbenennen.

2.3. Makrosprache

Die OpenOffice.org-Makrosprache basiert auf der Programmiersprache BASIC. Die Standardmakrosprache heißt offiziell StarBasic, wird aber auch als OOo Basic oder nur Basic bezeichnet. Viele andere Programmiersprachen können zur Automatisierung von OOo genutzt werden. OOo bringt praktische Unterstützung für Makros mit, die in Basic, JavaScript, Python und BeanShell geschrieben sind. In diesem Dokument liegt mein Hauptinteresse auf Basic.

2.4. Ein Modul in einem Dokument anlegen

Jedes OOo-Dokument ist ein Bibliothekscontainer, der Makros und Dialoge enthalten kann. Wenn ein Dokument die Makros enthält, die es benutzt, bedeutet folglich der Besitz des Dokuments auch den Besitz der Makros. Das ist eine praktische Methode für die Weitergabe und Speicherung. Senden Sie das Dokument an jemand anderen oder an einen anderen Speicherort, sind die Makros immer noch erreichbar und nutzbar.

1) Um einem OOo-Dokument ein Makro hinzuzufügen, müssen Sie das Dokument zur Bearbeitung öffnen. Öffnen Sie als erstes ein neues Textdokument, das den Namen „Unbenannt 1“ erhält – vorausgesetzt, dass momentan kein anderes Dokument ohne Namen geöffnet ist.
2) Über **Extras > Makros > Makros verwalten > OpenOffice.org Basic** öffnen Sie den Dialog für OOo-Basic-Makros (s. Bild 1).
3) Klicken Sie auf Verwalten, um die OOo-Makro-Verwaltung zu öffnen, und klicken dann auf die Registerkarte Bibliotheken (s. Bild 4).
4) Wählen Sie „Unbenannt 1“ aus der Ort-Aufklappliste.

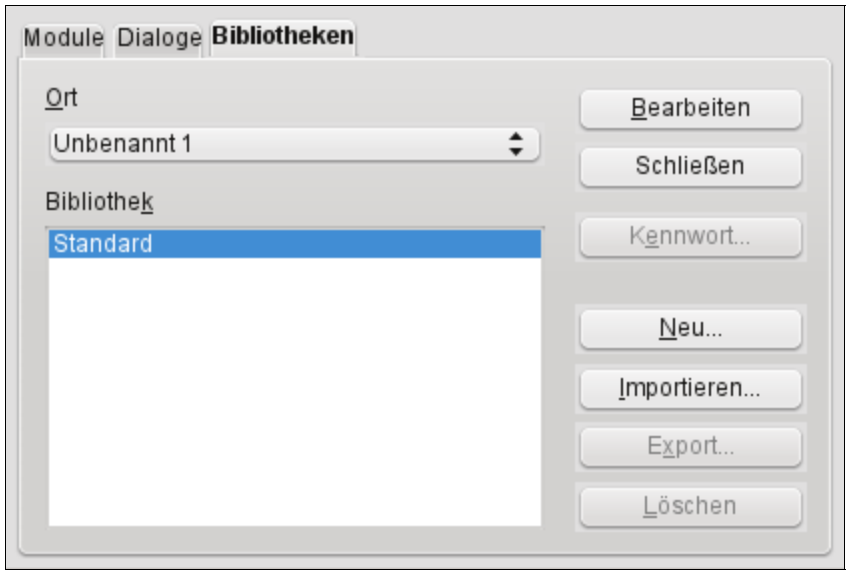


Bild 5. *Die Registerkarte Bibliothekender OOo-Makro-Verwaltung.*

5) Klicken Sie auf Neu, um den Dialog für eine neue Bibliothek zu öffnen.

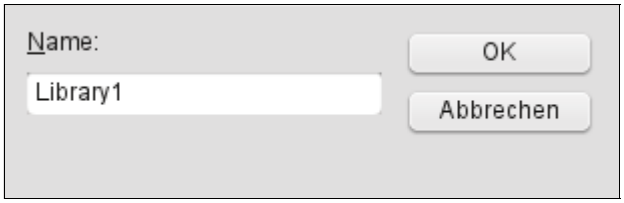


Bild 6. *Dialog Neue Bibliothek.*

6) Der Standardname ist Library1, nicht gerade aussagekräftig. Verwenden Sie einen aussagekräftigen Namen und klicken OK. Die neue Bibliothek steht nun in der Liste. Für dieses Beispiel habe ich die Bibliothek „HelloWorld“ genannt.

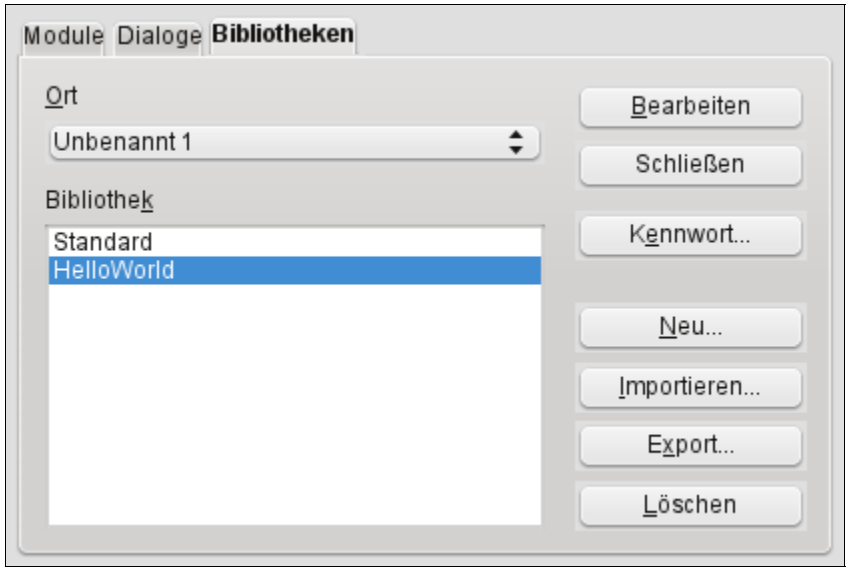


Bild 7. *Die neue Bibliothek steht in der Liste.*

7) In der Registerkarte Module wählen Sie die Bibliothek HelloWorld. OOo hat das Modul namens „Module1“ beim Anlegen der Bibliothek erstellt.

Tipp Obwohl Module1 gleichzeitig mit der Bibliothek erstellt wird, kann ein Bug in OOo 3.2 verhindern, dass das Modul angezeigt wird, bevor der Dialog geschlossen und neu geöffnet wird.

Bild 8. *Die neue Bibliothek steht in der Liste.*

8) Mit einem Klick auf Neu öffnen Sie den Dialog Neues Modul. Der Standardname ist Module2, weil Module1 schon existiert.

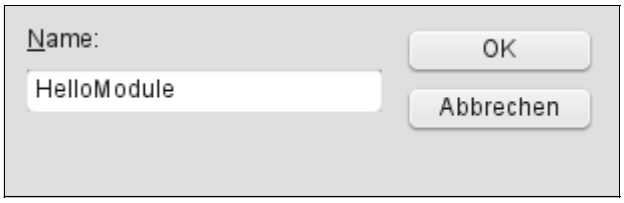


Bild 9. *Der Dialog Neues Modul.*

9) Geben Sie einen aussagekräftigen Namen ein klicken auf OK. Zusammen mit dem neu erstellten Modul wird schließlich auch Module1 angezeigt (Bug in 3.2.0).

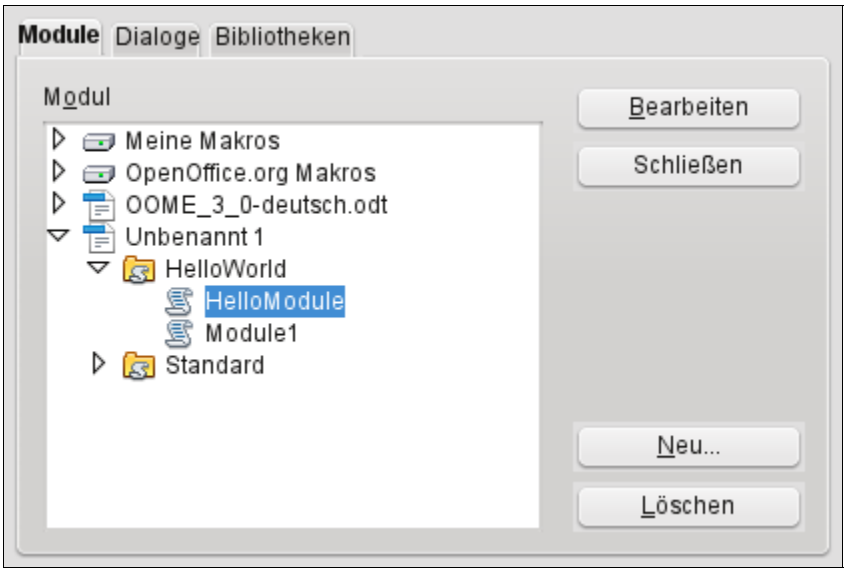


Bild 10. *Das neue Modul steht in der Liste.*

10) Markieren Sie HelloModule und klicken auf Bearbeiten.

11) An diesem Punkt habe ich das Dokument unter dem Namen „DelMeDoc“ gespeichert, weil ich es löschen werde, wenn ich mit dem Beispiel fertig bin. Nennen Sie es nach Ihren Wünschen. Wenn Sie dann den Dialog im Bild 10 neu öffnen, wird statt „Unbenannt 1“ der Dokumentenname angezeigt.

Jetzt wird die Integrierte Entwicklungsumgebung (Integrated Debugging Environment, IDE) zum Editieren des Makros geöffnet.

2.5. Integrierte Entwicklungsumgebung (Integrated Debugging Environment)

In der Integrierten-Entwicklungsumgebung (IDE) für Basic erstellen Sie Makros und führen sie aus (s. Bild 11). Die IDE bietet wesentliche Funktionalitäten auf kleinem Raum. Die Symbole der Werkzeugleiste sind in der Tabelle 1 erläutert. Links über dem Bearbeitungsfenster finden Sie eine Aufklappliste 

mit der Anzeige der aktuellen Bibliothek. Der Bibliothekscontainer steht in eckigen Klammern, gefolgt von der Bibliothek. So kann man schnell eine Bibliothek auswählen.

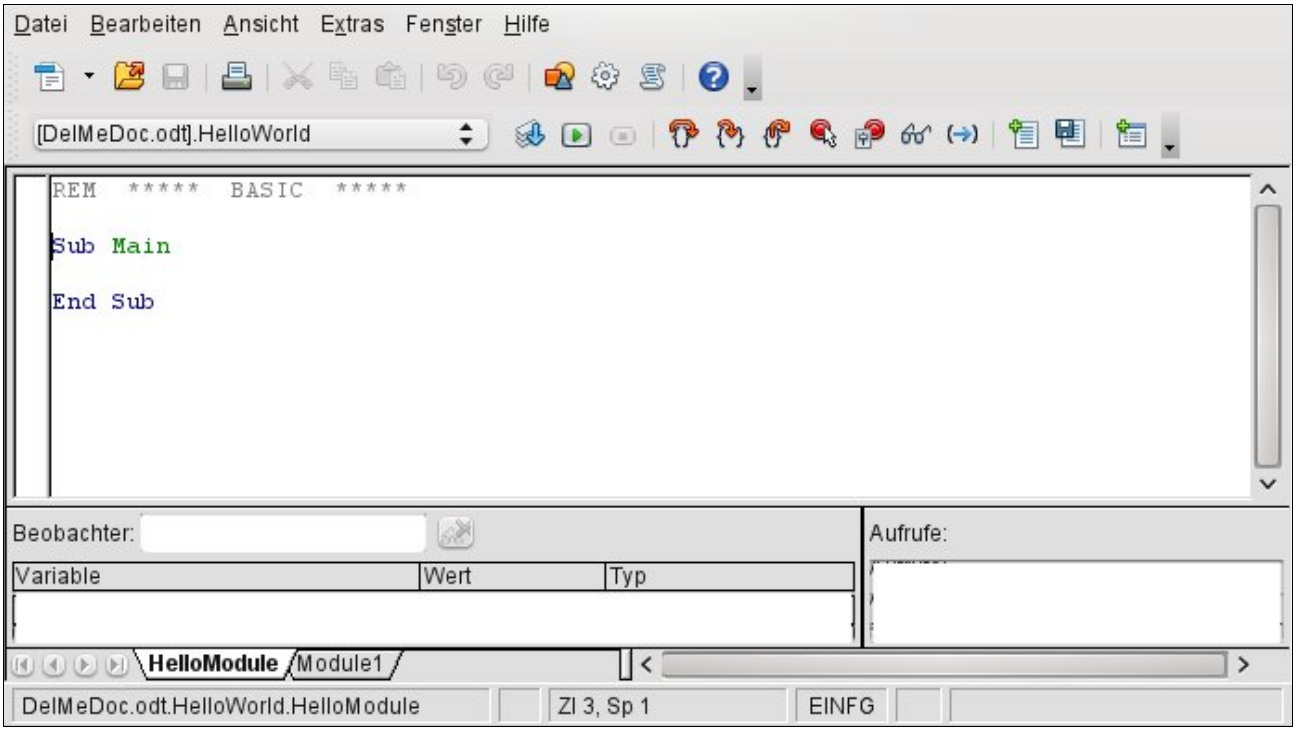


Bild 11. *Die Integrierte-Entwicklungsumgebung für Basic.*

Lassen Sie den Mauscursor ein paar Sekunden auf einem Symbol der Werkzeugleiste ruhen, um den beschreibenden Text zu lesen, der einen Hinweis auf die dahinter liegende Funktion gibt.

Tabelle 1. *Symbole der Werkzeugleiste in der Basic-IDE.*

Symbol	Taste	Beschreibung
	Ctrl+N	Neues Ooo-Dokument anlegen.
	Ctrl+O	Vorhandenes OOo Dokument öffnen.
	Ctrl+S	Aktuelle Bibliothek speichern. Wenn die Bibliothek zu einem Dokument gehört, dann wird das ganze Dokument gespeichert.
	Ctrl+P	Makro zum Drucker senden.
	Ctrl+V	Zwischenablage einfügen..
	Ctrl+Z	Letzte Aktivität rückgängig machen.
	Ctrl+V	Letzte rückgängig gemachte Aktivität wiederherstellen.
		Objektkatalog öffnen (s. Bild 12). Doppelklicken Sie auf das ausgewählte Makro.
		OOo-Makrodialog öffnen (s. Bild 2). Wählen Sie ein Makro aus und klicken Sie auf Bearbeiten oder Ausführen. Shortcut für **Extras > Makros > Makros verwalten > OpenOffice.org Basic**.
		Modul auswählen. Öffnet die OOo-Makro-Verwaltung und zeigt die Registerkarte Module an (s. Bild 3). Wählen Sie ein Modul und klicken auf Bearbeiten.
		OOo-Hilfe öffnen. Sie enthält eine Vielzahl nützlicher Beispiele für Basic.
		Kompilierungssymbol zum Prüfen von Syntaxfehlern. Eine Meldung erscheint nur, wenn ein Fehler gefunden wurde. Das Kompilierungssymbol kompiliert nur das aktuelle Modul.

Symbol	Taste	Beschreibung
	F5	Ausführen des ersten Makros im aktuellen Modul. Ist es angehalten (von einem Haltepunkt oder bei Einzelschritten), wird es fortgesetzt. Um ein anderes Makro auszuführen, nutzen Sie  . Damit öffnen Sie den OOo-Makrodialog. Wählen Sie das gewünschte Makro und klicken auf Ausführen.
	Shift+F5	Stoppen Sie das gerade laufende Makro.
	Shift+F8	Prozedurschritt zur nächsten Anweisung. Wenn ein Makro an einem Haltepunkt steht, wird die aktuelle Anweisung ausgeführt. Dient auch zum Start eines Makros im Einzelschrittmodus.
	F8	Einzelschritt. Wie Prozedurschritt, außer dass der Schritt, wenn die aktuelle Anweisung ein anderes Makro aufruft, in dieses Makro hineinführt, so dass die Beobachtung dort weitergeht.
		Rücksprung. Führt das Makro bis zum Ende der aktuellen Subroutine oder Funktion aus.
	Shift+F9	Haltepunkt ein/aus an der aktuellen Cursorposition in der IDE. Links neben der Zeile wird ein Symbol () platziert, das zeigt, dass für diese Zeile ein Haltepunkt gesetzt ist. Sie können auch in diesen Haltepunktbereich doppelklicken, um einen Haltepunkt ein- oder auszuschalten.
		Öffnet den Dialog zur Verwaltung der Haltepunkte (s. Bild 17), mit dem Sie wahlweise Haltepunkte ein- oder ausschalten können. Zusätzlich können Sie einstellen, nach wie vielen Durchläufen ein Haltepunkt erstmalig ausgelöst wird.
	F7	Beobachter-Symbol. Fügt die Variable unter dem aktuellen Cursor in der IDE in das Beobachterfenster ein. Sie können alternativ den Variablennamen in die Beobachter-Einfügezeile eintragen und Enter drücken.
		Klammernsuche.
		Basic-Quellcode einfügen. Öffnet einen Dialog zur Suche einer Datei mit Basic-Code, der dann eingefügt wird.
		Das aktuelle Modul als Textdatei speichern. OOo speichert Module auf der Festplatte in einem speziellen Format. Auf diesem Weg erstellte Dateien sind Standardtextdateien. Diese Form eignet sich ausgezeichnet für Makro-Backups oder zum Versenden an andere Personen. Im Gegensatz dazu wird über das Speichern-Symbol die gesamte Bibliothek oder das Dokument mit dem Modul gespeichert.
		Import eines Dialogs aus einem anderen Modul.

Die Modulnamen sind entlang dem unteren Ende der IDE aufgeführt. Links von den Namen sind Schaltflächen zur Modulnavigation  . Die Schaltflächen führen zum ersten, vorherigen, folgenden und letzten Modul der aktuellen Bibliothek. Rechtsklicken Sie auf einen Modulnamen, um:

- Ein neues Modul oder einen neuen Dialog einzufügen.
- Ein Modul oder einen Dialog zu löschen.
- Ein Modul umzubenennen. Ich kenne keinen einfacheren Weg, ein Modul oder einen Dialog umzubenennen
- Ein Modul oder einen Dialog auszublenden.
- Die OOo-Basic-Makroverwaltung zu öffnen.

Mit  öffnen Sie den Objektkatalog (s. Bild 12). Doppelklicken Sie auf das Makro, das Sie bearbeiten wollen.

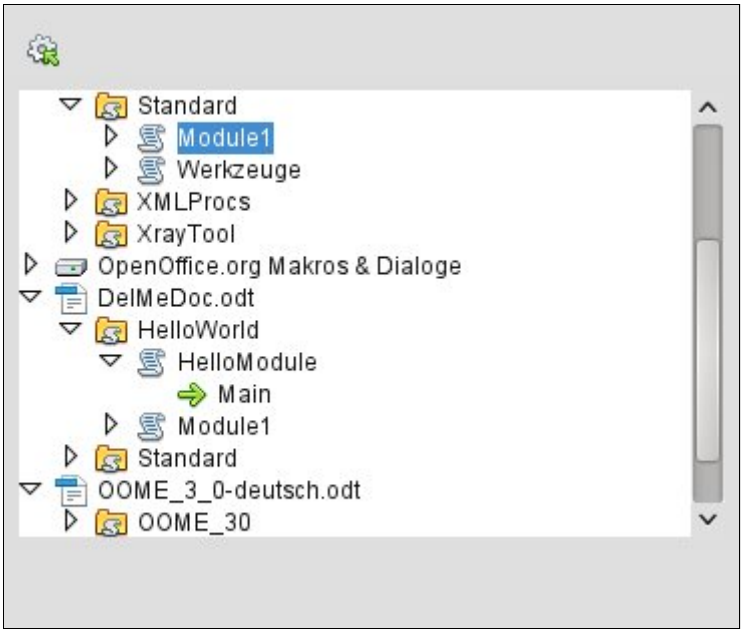


Bild 12. *Objektkatalog.*

2.6. Das Makro eingeben

Überschreiben Sie den Text in der IDE gemäß Listing 1. Klicken Sie auf das Ausführen-Symbol.

Listing 1. *Das Makro Hallo Welt*

```
REM  *****  BASIC  *****
Option Explicit

Sub Main
  Print "Hallo Welt"
End Sub
```

Tabelle 2. *Erläuterung der Zeilen in Listing 1.*

Zeile	Erläuterung
`REM ***** BASIC *****`	Basic-Kommentar, diese Zeile wird ignoriert. Ein Kommentar kann auch mit einem einfachen Anführungszeichen (= Apostroph) beginnen.
`Option Explicit`	Teilt dem Basic-Übersetzer mit, dass es ein Fehler ist, eine Variable zu verwenden, die nicht explizit deklariert ist. Schreibfehler in Variablennamen können leicht zu einem Laufzeitfehler führen.
`Sub Main`	Beginn der Definition der Subroutine mit dem Namen Main.
`Print "Hallo Welt"`	Print-Anweisung.
`End Sub`	Ende der-Subroutine Main.

2.7. Ein Makro ausführen

Das Ausführen-Symbol startet immer das erste Makro des aktuellen Moduls. Daraus folgt, dass ein anderer Weg gebraucht wird, wenn das Modul mehr als ein Makro enthält. Folgende Optionen stehen zur Auswahl:

- Schieben Sie das Makro an den Anfang des Moduls und klicken dann auf das Ausführen-Symbol.
- Rufen Sie mit dem ersten Makro das gewünschte Makro auf. Ich nutze diese Methode gerne während der Entwicklung. Ich habe immer ein Main-Makro als erstes, das nichts tut. Aber während der Entwicklung ändere ich es dahin, dass es das gerade relevante Makro aufruft.

Ganz allgemein lasse ich das Main-Makro ganz oben das am meisten ausgeführte Makro aufrufen.

- Gehen Sie über den Makrodialog (s. Bild 2), um eine beliebige Routine im Modul auszuführen.
- Fügen Sie Ihrem Dokument oder einer Werkzeugleiste eine Schaltfläche zu, über die Sie das Makro ausführen.
- Binden Sie das Makro an eine Tastenkombination. Öffnen Sie den Anpassen-Dialog über **Extras > Anpassen**. In der Registerkarte Tastatur finden Sie die Makrobibliotheken ganz unten in der Liste Kategorien. Ein anderer Weg geht über **Extras > Makros > Makros verwalten > OpenOffice.org Basic**. Wählen Sie das Makro aus und klicken auf Zuordnen. Damit öffnet sich der Anpassen-Dialog, in dem Sie über entsprechende Registerkartekarten das Makro als Menüpunkt, von einer Werkzeugleiste oder als Systemereignis ausführen lassen können.

Über den Makrodialog eine beliebige Subroutine in einem Modul auszuführen, geht so:

1. Über **Extras > Makros > Makros verwalten > OpenOffice.org Basic** öffnen Sie den Makrodialog (s. Bild 2).
2. Suchen Sie das Dokument mit dem gewünschten Modul in der Liste „Makro aus“.
3. Doppelklicken Sie auf eine Bibliothek, um deren enthaltene Module anzuzeigen.
4. Wählen Sie das Modul aus. In der Liste „Vorhandene Makros in: <selektierter Modulname>“ finden Sie die enthaltenen Subroutinen und Funktionen.
5. Wählen Sie die zum Ausführen gewünschte Subroutine oder Funktion aus – zum Beispiel HelloWorld.
6. Klicken Sie zum Ausführen der Subroutine oder der Funktion auf die Schaltfläche Ausführen.

2.8. Makrosicherheit

Je nachdem, wie OOo konfiguriert ist, kann es möglich sein, dass Sie keine Makros in einem Dokument ausführen dürfen. Wenn ich ein Dokument öffne, das ein Makro enthält, erscheint die Warnung Bild 13. Wenn Sie kein Makro erwarten oder der Quelle nicht trauen, deaktivieren Sie die Makros. Bedenken Sie, schließlich bin ich in der Lage, ein Makro zu schreiben, das Ihren Rechner zerstört.

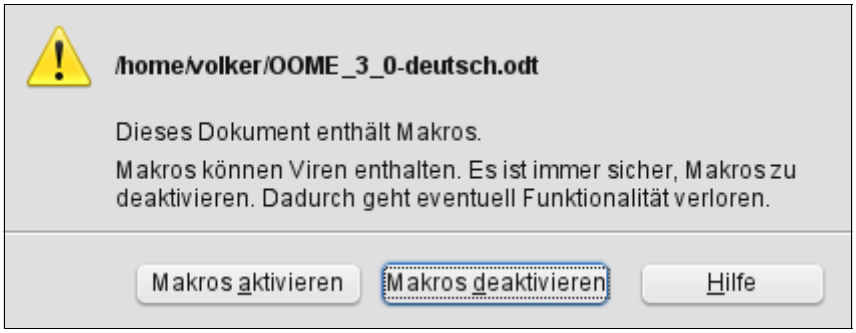


Bild 13. *Das geöffnete Dokument enthält ein Makro.*

Über **Extras > Optionen > OpenOffice.org > Sicherheit** öffnen Sie das Sicherheitsblatt des Optionen-Dialogs.

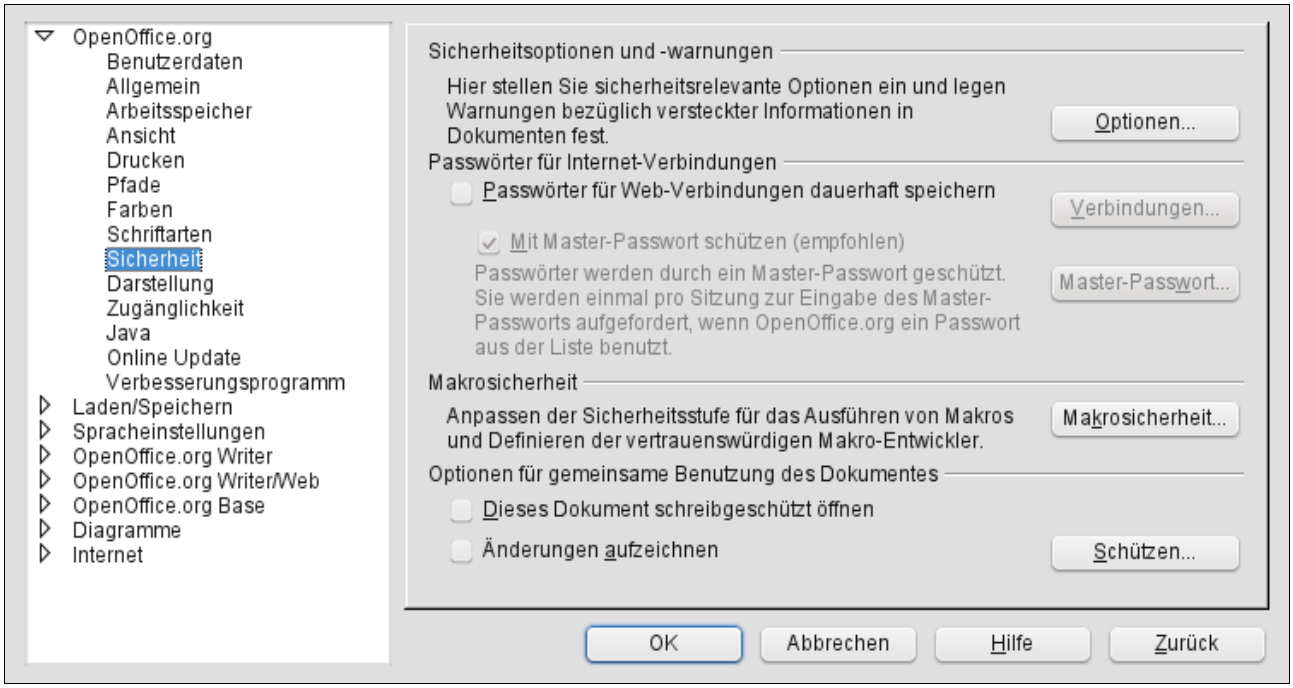


Bild 14. *Optionen-Dialog, Sicherheitseinstellungen.*

Klicken Sie auf die Schaltfläche Makrosicherheit, um den Makrosicherheitsdialog zu öffnen. Wählen Sie eine Sicherheitsstufe, bei der Ihnen wohl ist. Die mittlere Sicherheitsstufe verwendet die im Bild 13 gezeigte Rückfrage. Das ist unaufdringlich und relativ schnell erledigt.

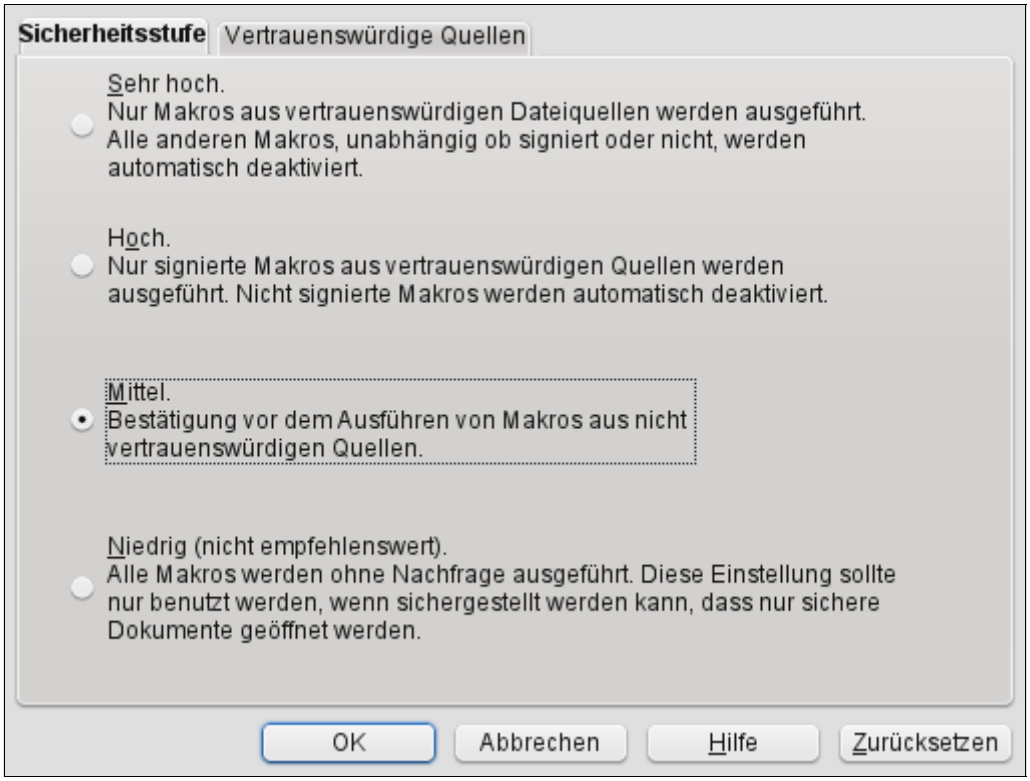


Bild 15. *Dialog Makrosicherheit, Registerkarte Sicherheitsstufe.*

Sie können vertrauenswürdige Quellen und Zertifikate eintragen, um Dokumente ohne Sicherheitsabfrage zu laden, entweder je nach Speicherort oder nach dem für das Dokument ausgestellten Zertifikat.

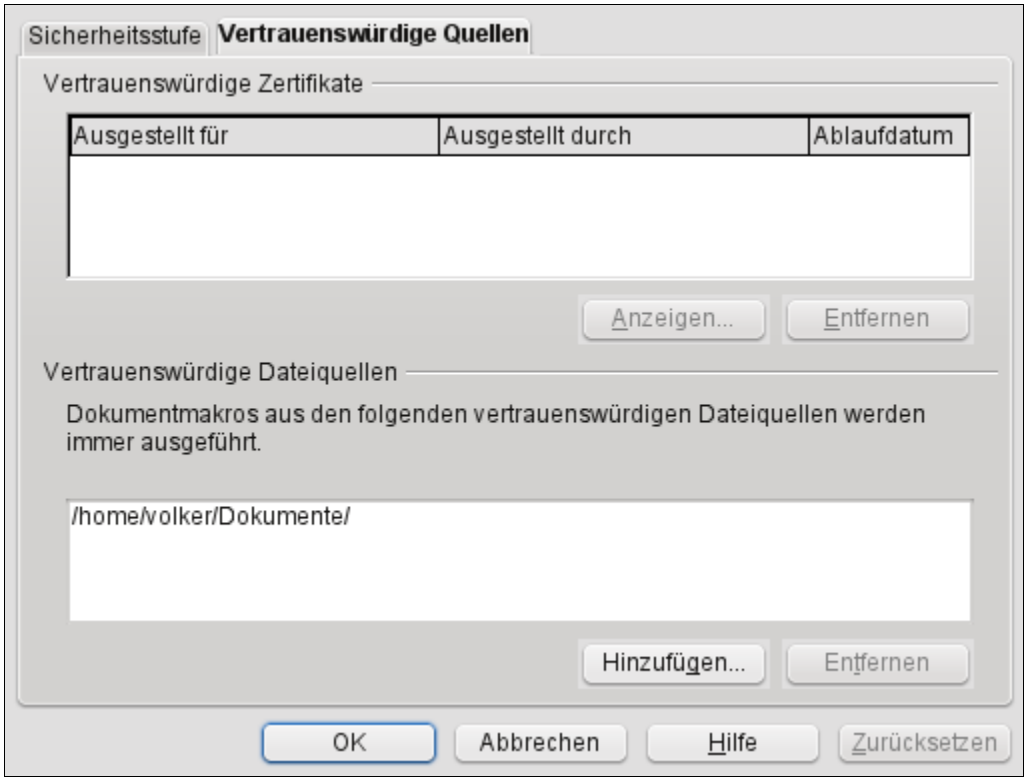


Bild 16. *Dialog Makrosicherheit, Registerkarte Vertrauenswürdige Quellen.*

2.9. Haltepunkte einsetzen

Wenn Sie im Quelltext einen Haltepunkt setzen, bleibt das Makro bei der Ausführung an dieser Stelle stehen. Sie können dann Variablenwerte überprüfen, die Ausführung des Makros komplett oder in Einzelschritten fortsetzen. Wenn ein Makro fehlerhaft arbeitet und Sie nicht wissen warum, erlaubt Ihnen der Einzelschritt-Modus (jeweils 1 Anweisungsschritt), das Makro bei der Arbeit zu beobachten. Sie wissen dann also, wie es zu dem Fehler kam. Falls aber vor der problematischen Stelle eine Menge Anweisungszeilen stehen, kann es sehr beschwerlich sein, in Einzelschritten dorthin zu gelangen. In solchen Fällen setzen Sie an oder in der Nähe der Stelle einen Haltepunkt, an der der Fehler auftritt. Das Programm hält an der Stelle an, so dass Sie nun in Einzelschritten das weitere Verhalten des Makros beobachten können.

Das Symbol Haltepunkt-Ein/Aus setzt einen Haltepunkt an die Anweisung, auf dem der Cursor steht. Ein rotes Stoppschild kenzeichnet die Zeile in der Haltepunktspalte. Doppelklicken Sie in die Haltepunktspalte, um einen Haltepunkt an der Anweisungszeile zu setzen oder zu entfernen. Ein Rechtsklick auf einen Haltepunkt in der Haltepunktspalte aktiviert oder deaktiviert ihn.

Mit dem Symbol zur Haltepunktverwaltung  gelangen Sie zu dem Dialog, in dem alle aktiven Haltepunkte im aktuellen Modul mit der Zeilennummer aufgelistet sind. Um einen weiteren Haltepunkt zu setzen, geben Sie eine Zeilennummer in das Eingabefeld ein und klicken auf Neu. Um einen Haltepunkt zu löschen, markieren Sie ihn in der Liste und klicken auf Löschen. Um den markierten Haltepunkt zu deaktivieren, ohne ihn zu löschen, nehmen Sie den Haken aus dem Ankreuzfeld Aktiv. Das Eingabefeld Durchlauf zeigt an, wie oft ein Haltepunkt erreicht werden muss, bis er aktiviert wird. Bei der Durchlaufzählung 4 wird die Anweisung mit dem Haltepunkt nicht ausgeführt, wenn die Zeile zum vierten Mal erreicht wird: das Makro wird angehalten. Das ist außerordentlich wichtig, wenn ein Makrobereich erst dann einen Fehler produziert, wenn er mehrfach aufgerufen wurde.

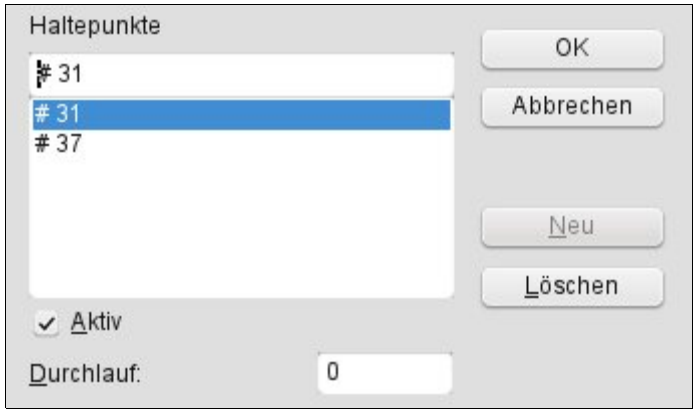


Bild 17. *Dialog zur Verwaltung der Haltepunkte.*

Ein Haltepunkt kann aus zwei Gründen ignoriert werden: eine Durchlaufzählung größer als Null und wenn der Haltepunkt in der Haltepunkt-Verwaltung als „nicht aktiv“ gekennzeichnet ist. Zu jedem Haltepunkt gehört eine Zählung, die beim Durchlauf in Richtung Null heruntergezählt wird. Wenn der Wert Null erreicht ist, wird der Haltepunkt aktiv und bleibt es, weil die Durchlaufzählung danach auf Null bleibt. Die Durchlaufzählung wird beim Beenden oder Neustart des Makros nicht auf den Originalwert zurückgesetzt.

2.10. Wie Bibliotheken gespeichert werden

Makrobibliotheken werden als XML-Dateien gespeichert, die man leicht mit einem einfachen Texteditor bearbeiten kann. Mit anderen Worten, es ist ganz einfach, darin herumzustöbern und die Dateien unbrauchbar zu machen. Dieses Kapitel ist eigentlich für Fortgeschrittene, so dass Sie es vielleicht überschlagen wollen. Wenn Sie nichts von XML verstehen und nicht wissen, weshalb in der Datei > anstatt > steht, sollten Sie die Datei vielleicht nicht bearbeiten. Obwohl es im allgemeinen als töricht gilt, die externen Bibliotheken manuell zu bearbeiten, hatte ich wenigstens einmal den Fall, wo es doch notwendig war, weil OOo ein Modul wegen eines darin enthaltenen Syntaxfehlers nicht laden konnte.

Jede Bibliothek wird in einem einzelnen Verzeichnis gespeichert, und jedes Modul wie auch jeder Dialog in einer einzelnen Datei. Die mit OpenOffice.org installierten globalen Bibliotheken werden in einem von allen Nutzern verwendeten Verzeichnis basic unterhalb des Verzeichnisses gespeichert, in dem OOo installiert ist. Beispiele:

```
C:\Programme\OpenOffice3.2\share\basic       'Eine Windows-Installation
/opt/openoffice.org/basis3.2/share/basic     'Eine Linux-Installation
```

OpenOffice.org speichert nutzerspezifische Daten in einem Verzeichnis unterhalb dessen Home-Verzeichnisses. Der genaue Ort hängt vom Betriebssystem ab. Über **Extras > Optionen > OpenOffice.org > Pfade** können Sie nachlesen, wo andere Konfigurationsdaten gespeichert sind. Hier sind einige Beispiele, wo meine Basic-Makros liegen:

```
C:\Documents and Settings\andy\Application Data\OpenOffice.org\3\user\basic  'Windows
XP
C:\Users\pitonyaka\AppData\Roaming\OpenOffice.org\3\user\basic              'Windows 7
/home/andy/OpenOffice.org/3/user/basic                                      'Linux
```

Nutzermakros werden in OpenOffice.org\3\user\basic gespeichert. Jede Bibliothek befindet sich in einem eigenen Verzeichnis unterhalb des Verzeichnisses basic. Das Verzeichnis user enthält zwei Dateien und je ein Verzeichnis für jede Bibliothek (s. Tabelle 3).

Tabelle 3. *Dateien und ein paar Verzeichnisse in meinem Verzeichnis user/basic.*

Eintrag	Beschreibung
dialog.xlc	XML-Datei, die auf jede Dialogdatei zeigt, die diesem Nutzer in OpenOffice.org bekannt ist.
script.xlc	XML-Datei, die auf jede Bibliotheksdatei zeigt, die diesem Nutzer in OpenOffice.org bekannt ist.
Standard	Verzeichnis der Standardbibliothek.
Pitonyak	Verzeichnis der Bibliothek mit dem Namen Pitonyak.
PitonyakDialogs	Verzeichnis der Bibliothek mit dem Namen PitonyakDialogs.

Die Dateien dialog.xlc und script.xlc enthalten je einen Zeiger auf alle Dialoge und Bibliotheken, die OOo kennt. Wenn diese Dateien überschrieben werden, weiß OOo nichts mehr von Ihren persönlichen Bibliotheken, auch wenn sie noch existieren. Sie können jedoch die Dateien manuell bearbeiten, oder besser noch, mit dem OOo-Makro-Verwalter die Bibliotheken importieren (weil Sie eine Bibliothek importieren können, die in einem Verzeichnis liegt).

Das Verzeichnis einer Bibliothek enthält je eine Datei für jedes Modul und für jeden Dialog der Bibliothek. Das Verzeichnis enthält außerdem die Dateien dialog.xlb und script.xlb, die auf die Dialoge bzw. Module zeigen.

2.11. Wie Dokumente gespeichert werden

Die Standard-OOo-Dateiformate speichern alle Komponenten als Standard-ZIP-Dateien. Mit jedem Programm, das ZIP-Dateien lesen und auspacken kann, ist die Untersuchung von OOo-Dokumenten möglich – für manche Programme müssen Sie jedoch die Dateinamenergänzung in ZIP ändern.

Nach dem Auspacken eines OOo-Dokuments erhalten Sie Dateien, die den Hauptinhalt, die Formatvorlagen und die Einstellungen enthalten, und außerdem noch drei Verzeichnisse. Das Verzeichnis META-INF zeigt auf alle anderen Dateien, eingebetteten Grafiken, Bibliotheken mit Makroquelltext und Dialoge. Das Verzeichnis Dialogs enthält alle eingebetteten Dialoge, und das Verzeichnis Basic enthält alle eingebetteten Bibliotheken.

Der entscheidende Punkt daran ist, dass Sie im Notfall die XML-Daten manuell einsehen und möglicherweise Probleme beheben können.

2.12. Fazit

Makros und Dialoge werden in Modulen gespeichert, Module in Bibliotheken, und Bibliotheken in Bibliothekscontainern. Die Anwendung ist ein Bibliothekscontainer, ebenso jedes Dokument. Mit Hilfe der IDE werden Makros und Dialoge entwickelt und getestet.

Zum Schreiben von Makros für OpenOffice.org haben Sie nun einen der schwierigsten Schritte getan: Sie haben Ihr erstes Makro geschrieben! Jetzt sind sie so weit, andere Makrobeispiele zu untersuchen und auszuprobieren, sowie ein paar eigene zu entwickeln.

3. Sprachstrukturen

Die Makrosprache von OpenOffice.org ist der von Microsoft Office ähnlich, weil beide auf BASIC beruhen. Beide Makrosprachen greifen auf die Strukturen der zugrunde liegenden Anwendung zu, die sich ganz wesentlich unterscheiden und daher nicht kompatibel sind. Dieses Kapitel greift die Teile der Sprache heraus, die nicht auf die zugrunde liegende Anwendung zugreifen.

Dieses Kapitel zeigt, wie verschiedene Komponenten zusammenkommen, um ein OOo-Makro zu produzieren, das übersetzt (kompiliert) und ausgeführt werden kann. In einem Wort: Syntax. Korrekte Syntax bedeutet nicht unbedingt, dass das Makro tut, was Sie wollen, nur dass die Einzelteile in korrekter Art und Weise zusammengefügt sind. Die Fragen „Kann ich fahren?“ und „Darf ich fahren?“ sind beide syntaktisch korrekt. Die erste Frage bezieht sich auf die Fähigkeit, die zweite Frage auf die Erlaubnis. In der Umgangssprache können beide Fragen dieselbe Bedeutung haben. Der Computer andererseits tut nur genau, was man ihm sagt, und nicht, was man meint.

Die syntaktischen Grundkomponenten eines OpenOffice.org-Makros sind Anweisungen, Variablen, Subroutinen, Funktionen und Programmflusskontrollen. Eine Anweisung ist ein kleiner, ausführbarer Teil des Codes, der im allgemeinen eine einzelne Textzeile umfasst. Variablen sind Behälter und enthalten Werte, die sich während der Makroausführung ändern können. Subroutinen und Funktionen unterteilen ein Makro in benannte Gruppen von funktional zusammengehörenden Anweisungen und ermöglichen so eine bessere organisatorische Struktur. Die Programmflusskontrollen steuern, welche Anweisungen ausgeführt werden und wie oft.

OOo führt Makros zeilenweise aus. Jede Makrozeile wird genau so begrenzt, wie es klingt: durch eine Neue-Zeile-Kennung (s. Listing 2).

Listing 2. *Makro aus zwei Zeilen*

```
Print "Dies ist Zeile eins"
Print "Dies ist Zeile zwei"
```

Überlange Zeilen können über mehr als eine Textzeile gehen, indem ein Unterstrich (_) an das Ende der Textzeile gefügt wird (s. Listing 3). Damit der Unterstrich zur Zeilenverbindung wird, muss er das letzte Zeichen der Textzeile sein. Der Unterstrich hat keine Sonderbedeutung, wenn er nicht am Zeilenende steht, er kann innerhalb von Zeichenfolgen und Variablennamen genutzt werden. Wenn er als Zeilenverbinder dient, können ihm Leerzeichen vorausgehen, in manchen Fällen sind Leerzeichen sogar notwendig, um den Zeileninhalt vom Unterstrich zu trennen. Wenn Sie zum Beispiel die Zeile „a+b+c“ hinter dem b teilen, benötigen Sie ein Leerzeichen zwischen dem b und dem Unterstrich. Ansonsten würde der Unterstrich als Teil des Variablennamens interpretiert. Achtung bei Leerzeichen, die versehentlich einem Zeilenverbinder folgen, sie können der Grund für einen Fehler bei der Kompilierung sein. Unglücklicherweise sagt der Fehler nichts darüber aus, dass etwas dem Unterstrich folgt, sondern nur, dass die nächste Zeile ungültig ist.

Listing 3. *Ein Unterstrich am Zeilenende setzt die Zeile logisch mit der nächste Zeile fort.*

```
Print "Zeichenfolgen werden verkettet, indem sie " & _
      "mit dem Kaufmanns-Und verbunden werden"
```

Tipp Wenn auf ein Zeichen zur Zeilenverbindung irgendwelche Zeichen folgen, wird die nächste Zeile nicht als Fortsetzung erkannt. Wenn ich Quelltexte von Websites kopiere und sie in die IDE einfüge, wird manchmal ein Leerzeichen am Zeilenende eingefügt, das die Zeilenverbindung auflöst.

Sie können in einer Textzeile mehrere Anweisungen unterbringen, indem Sie sie mit Doppelpunkten abtrennen. Man macht das gewöhnlich zur besseren Lesbarkeit. Jede dieser kombinierten Anweisungen wirkt beim Testen des Makros in der Integrierten Entwicklungsumgebung (IDE) als eine eigene Codezeile. Die Zeile in Listing 4 verhält sich wie drei getrennte Anweisungen, wenn man in der IDE das Makro im Einzelschrittmodus testet.

Listing 4. *Die Variablen x, y und z werden auf Null gesetzt.*

```
x = 0 : y = 0 : z = 0
```

Sie sollten in allen Makros, die Sie schreiben, nicht an Kommentaren sparen. Denken Sie beim Schreiben daran, dass das, was heute klar ist, morgen nicht mehr so klar sein könnte, denn die Zeit vergeht, neue Projekte entstehen, und das Gedächtnis schwindet nur allzu schnell. Sie können einen Kommentar entweder durch ein einfaches Anführungszeichen (Apostroph) oder das Schlüsselwort REM (remark = Kommentar) einleiten. Der gesamte Text, der in dieser Zeile folgt, wird ignoriert. Kommentare gelten nicht als ausführbare Anweisungen. Sie werden auch im Einzelschrittmodus übergangen.

Listing 5. *Fügen Sie allen Makros, die Sie schreiben, Kommentare zu.*

```
REM Kommentare können mit dem Schlüsselwort REM beginnen.
ReM Groß- und Kleinschreibung spielen keine Rolle. Dies ist also auch ein Kommentar.
' Alles was dem Kommentarstart folgt, wird ignoriert.
X = 0 ' Ein Kommentar kann auch mit
      ' einem Apostroph beginnen.
z = 0 REM Alles was dem Kommentarstart folgt, wird ignoriert.
```

Tipp Groß- und Kleinschreibung der Schlüsselwörter, Variablen- und Routinennamen sind in OOo Basic nicht von Belang.

Daher gelten sowohl REM, Rem oder rEm alle als Start eines Kommentars.

Einem Zeichen zur Zeilenverbindung (_) darf kein weiteres Zeichen folgen, auch kein Kommentar. Alles was einem Kommentarstart folgt, wird ignoriert – auch ein Zeilenverbinder. Die logische Folge dieser beiden Regeln ist, dass ein Zeilenverbinder niemals auf derselben Zeile mit einem Kommentar vorkommen kann.

3.1. Kompatibilität mit Visual Basic

Bezogen auf die Syntax und die BASIC-Funktionalität sind sich OOo Basic und Visual Basic sehr nahe. Die beiden Basic-Dialekte haben ganz und gar nichts miteinander gemein, wenn es darum geht, Dokumente zu manipulieren, aber der Grundbestand der Befehle ist sehr ähnlich. Die allgemeine Kompatibilität zwischen den beiden Dialekten ist stufenweise verbessert worden. Mit OOo 2.0 wurden viele Erweiterungen eingeführt. Viele dieser Änderungen sind mit dem herkömmlichen Verhalten nicht kompatibel. Zur Behebung dieser Konflikte wurden eine neue Kompilierungsoption und ein neuer Laufzeitmodus eingerichtet, die das neue kompatible Verhalten bestimmen.

Die Kompilierungsoption „Option Compatible“ steuert einige Besonderheiten. Diese Option wirkt sich nur auf das Modul aus, in dem sie steht. Weil ein Makro während der Ausführung auch andere Module aufruft, können sowohl das alte wie auch das neue Verhalten resultieren, abhängig davon, ob in den jeweils aufgerufenen Modulen „Option Compatible“ enthalten ist. Die Option in einem Modul zu setzen, hat also keinen Effekt in einem anderen aufgerufenen Modul.

Die Laufzeitfunktion CompatibilityMode(True/False) bietet die Möglichkeit, Laufzeitfunktionen während der Makroausführung zu modifizieren. Somit erhält man die Flexibilität, das neue Laufzeitverhalten einzuschalten, einige Operationen vorzunehmen und dann das neue Laufzeitverhalten wieder auszuschalten. CompatibilityMode(False) hebt Option Compatible für das neue Laufzeitverhalten auf. Es ist zu hoffen, dass eine Methode zur Prüfung des aktuellen Laufzeitverhaltens geschaffen wird.

Visual Basic erlaubt alle im Zeichensatz Latin-1 (ISO 8859-1) enthaltenen Zeichen in Variablennamen, OOo nicht. Wenn Sie „Option Compatible“ setzen, wird „ä“ zu einem gültigen Variablennamen. Dies ist nur eine der Änderungen, die durch „Option Compatible“ ermöglicht werden. Die Funktion CompatibilityMode() kann die neuen weiter gefassten Bezeichner weder aktivieren noch deaktivie-

ren, weil CompatibilityMode() erst zur Laufzeit aufgerufen wird und Variablennamen schon beim Kompilieren erkannt werden.

Sowohl Visual Basic wie auch OOo Basic kennen die Anweisung rmdir(), um ein Verzeichnis zu löschen. VBA kann nur leere Verzeichnisse löschen, wohingegen OOo Basic einen gesamten Verzeichnisbaum rekursiv löschen kann. Wenn die Funktion CompatibilityMode(True) vor der Anweisung rmdir() aufgerufen wird, wird OOo Basic wie VBA handeln und eine Fehlermeldung ausgeben, wenn das spezifizierte Verzeichnis nicht leer ist. Dies ist nur eine der vielen Änderungen, die durch CompatibilityMode() ermöglicht werden.

StarBasic lässt viel mehr durchgehen als VBA. Es ist daher leichter, einfache Makros von VBA nach OOo Basic zu konvertieren. Dazu zwei Beispiele:

In OOo Basic ist der Zuweisungsbefehl „set“ optional. Daher ist „`set x = 5`“ in VBA wie auch in OOo Basic gültig, wogegen „`x = 5`“ in VBA ein Fehler ist, in OOo Basic aber funktioniert.

Das zweite Beispiel betrifft die Arraymethoden. Sie sind weit stabiler und toleranter in OOo als in VBA. So arbeiten die Funktionen zur Ermittlung der Arraygrenzen (LBound and UBound) problemlos mit leeren Arrays, VBA hingegen stürzt dabei ab.

3.2. Variablen

Variablen sind Behälter für Werte. OpenOffice.org unterstützt verschiedene Typen von Variablen für unterschiedliche Wertetypen. Dieses Kapitel zeigt, wie man Variablen erstellt, benennt und benutzt. Obwohl Sie von OOo Basic nicht gezwungen werden, Variablen zu deklarieren, sollten Sie dennoch jede Variable deklarieren, die Sie benutzen. Die Gründe dafür werden in diesem Kapitel verdeutlicht.

3.2.1. Namen für Konstanten, Subroutinen, Funktionen, Sprungmarken und Variablen

Geben Sie Ihren Variablen immer aussagekräftige Namen. Auch wenn Ihnen der Variablenname „var1“ viel Nachdenken erspart, ist doch „Vorname“ viel sprechender. Es gibt aber auch Variablennamen, die zwar nicht wirklich sprechend, doch bei den Programmierern weit verbreitet sind. Zum Beispiel wird „i“ als Kürzel für „index“ verwendet, und zwar für eine Variable zur Durchlaufzählung einer Schleife. Folgende Einschränkungen gelten in OOo Basic für Variablennamen:

- Ein Variablenname darf nicht mehr als 255 Zeichen enthalten. Na ja, *das heißt offiziell*. Ich habe Namen mit mehr als 300 Zeichen ohne Probleme getestet, aber das soll keine Empfehlung sein!
- Das erste Zeichen eines Variablennamens muss ein Buchstabe sein: A-Z oder a-z. Mit der Einstellung `Option Compatible` werden alle im Zeichensatz Latin-1 (ISO 8859-1) als Buchstaben definierten Zeichen als Teil eines Bezeichners akzeptiert.
- Die Ziffern 0-9 und der Unterstrich (_) dürfen in Variablennamen vorkommen, aber nicht als erstes Zeichen. Ein Unterstrich am Ende eines Variablennamens wird nicht als Zeilenverbinder missverstanden.
- Bei Variablennamen spielt die Groß- und Kleinschreibung keine Rolle. Sowohl „VorName“ wie auch „vorNAME“ verweisen auf dieselbe Variable.
- Variablennamen dürfen Leerzeichen enthalten. Dann müssen sie aber in eckige Klammern eingeschlossen werden, zum Beispiel „[Vor Name]“. Doch obwohl das erlaubt ist, gilt es als schlechte Programmierpraxis.

Tipp Diese Einschränkungen gelten auch bei Namen für Konstanten, Subroutinen, Funktionen und Sprungmarken.

3.2.2. Variablen deklarieren

Einige Programmiersprachen verlangen, dass Sie alle verwendeten Variablen ausdrücklich auflisten. Diesen Vorgang nennt man „Variablen deklarieren“. OOo Basic verlangt das nicht. Es steht Ihnen frei, Variablen zu verwenden, ohne sie zu deklarieren.

Obwohl es ganz praktisch ist, Variablen ohne Deklarierung zu verwenden, so ist das doch fehleranfällig. Wenn Sie sich bei einem Variablennamen verschreiben, entsteht daraus eine neue Variable statt einer Fehlermeldung. Wenn Sie daher wollen, dass OOo Basic nicht deklarierte Variablen als Laufzeitfehler behandelt, dann stellen Sie die Schlüsselwörter „Option Explicit“ ganz an den Anfang, vor den ausführbaren Code. Vor Option Explicit dürfen allenfalls noch Kommentare stehen, weil sie nicht ausführbar sind. Es wäre sicher besser, wenn OOo Basic solche Fehler zur Kompilierungszeit fände, tatsächlich aber werden werden alle Variablen und Routinen erst zur Laufzeit aufgelöst.

Listing 6. *Verwendung von Option Explicit vor der ersten ausführbaren Codezeile eines Makros.*

```
REM  *****  BASIC  *****
Option Explicit
```

Tipp Verwenden Sie „Option Explicit“ ganz am Anfang eines jeden Makros, das Sie schreiben. Sie werden damit viel Zeit bei der Fehlersuche in Ihrem Code sparen. Wenn ich gebeten werde, ein Makro zu debuggen, füge ich zuallererst „Option Explicit“ an den Anfang jedes Moduls.

Sie können eine Variable mit oder ohne Typ deklarieren. Eine Variable ohne explizite Typangabe wird als Typ Variant geführt, der Daten eines jeden Typs enthalten kann. Das heißt, dass Sie Variant verwenden können, um zum Beispiel einen numerischen Wert aufzunehmen und ihn dann in der nächsten Codezeile zum Beispiel mit einer Zeichenfolge zu überschreiben. Tabelle 4 Zeigt die von OOo Basic unterstützten Variablentypen, die Werte, die sie unmittelbar nach der Deklarierung enthalten („Anfangswert“), und die jeweils verwendete Anzahl an Bytes. In OOo Basic kann der Typ einer Variablen auch dadurch festgelegt werden, dass ein spezielles Zeichen bei der Deklarierung an das Namensende gefügt wird. Die Spalte Annex in der Tabelle 4 enthält die vorgesehenen Zeichen, die an einen Variablennamen bei der Deklarierung angehängt werden können.

Tabelle 4. *Verfügbare Variablentypen und ihre Anfangswerte .*

Typ	Annex	Anfangswert	Bytes	Konversion	Beschreibung
Boolean		False	1	CBool	True oder False
Currency	@	0.0000	8	CCur	Währung mit 4 Dezimalstellen
Date		00:00:00	8	CDate	Datum und Uhrzeit
Double	#	0.0	8	CDbl	Dezimalzahl im Bereich von +/-1,79769313486232 x 10E308
Integer	%	0	2	CInt	Ganzzahl von -32.768 bis 3.,767
Long	&	0	4	CLng	Ganzzahl von -2.147.483.648 bis 2.147.483.647
Object		Null	varies		Objekt
Single	!	0.0	4	CSng	Dezimalzahl im Bereich von +/-3,402823 x 10E38
String	$	""	varies	CStr	Text mit bis zu 65.536 Zeichen
Variant		Empty	varies	CVar	Kann Daten jedes Typs enthalten

Obwohl OOo Basic den Variablentyp Byte unterstützt, können Sie ihn nicht direkt deklarieren und verwenden. Wie später erläutert, gibt die Funktion CByte einen Byte-Wert zurück, der einer Variablen vom Typ Variant zugewiesen werden kann. Seit OOo 2.0 können Sie eine Variable mit Typ Byte

deklarieren. Diese Variable wird aber als ein extern definiertes Objekt vom Typ Byte behandelt und nicht als eine intern definierte Byte-Variable.

Mit dem Schlüsselwort DIM deklarieren Sie eine Variable vor dem ersten Gebrauch (s. Tabelle 5). Sie können mehrere Variablen in einer einzigen Zeile deklarieren, und Sie können jeder Variablen beim Deklarieren einen Typ zuweisen. Variablen ohne deklarierten Typ sind automatisch vom Typ Variant.

Tabelle 5. *Deklarierung einfacher Variablen.*

Deklarierung	Beschreibung
Dim Name	Name hat den Typ Variant, weil kein Typ festgelegt ist.
Dim Name As String	Name hat den Typ String, weil der Typ explizit festgelegt ist.
Dim Name$	Name$ hat den Typ String, weil Name$ mit einem $ endet.
Dim Name As String, Weight As Single	Name hat den Typ String und Weight den Typ Single.
Dim Width, Length	Width und Length haben den Typ Variant.
Dim Weight, Height As Single	Weight hat den Typ Variant und Height den Typ Single.

TIP Wenn mehrere Variablen in einer Zeile deklariert werden, muss für jede Variable der Typ gesondert angegeben werden. In der letzten Zeile der Tabelle 5 ist Weight vom Typ Variant, auch wenn es so aussieht, als ob es vom Typ Single wäre.

In einem Großteil der Literatur über OOo-Makroprogrammierung wird ein Namensschema für Variablen verwendet, das auf der Ungarischen Notation beruht. Dabei können Sie vom Namen auf den Typ einer Variablen schließen. Praktisch macht das jeder so oder anders und hält sich mehr oder weniger fest daran. Das ist eine Frage des Stils. Manche lieben es, und manche hassen es.

OOo Basic hat eine Reihe von Anweisungen zur Förderung der Ungarischen Notation: Def<Typ>. Die Def-Anweisungen sind lokal auf das jeweilige Modul begrenzt, in dem sie stehen. Sie weisen nicht-deklarierten Variablen abhängig von ihren Namen einen bestimmten Typ zu. Normalerweise sind alle nicht-deklarierten Variablen vom Typ Variant.

Der Def-Anweisung folgt eine durch Kommas getrennte Liste von Zeichenbereichen für den ersten Buchstaben der Namen (s. Listing 7).

Listing 7. *Deklarierung von typlosen Variablen, die mit i, j, k oder n beginnen, als Typ Integer.*

```
DefInt i-k,n
```

Tabelle 6 enthält je ein Beispiel für die verfügbaren Def-Anweisungen. Wie Option-Anweisungen werden auch Def-Anweisungen vor alle anderen ausführbaren Zeilen oder Variablendeklarierungen eines Moduls platziert. Die Def-Anweisung erzwingt nicht für jede Variable, die mit einem bestimmten Buchstaben beginnt, einen spezifischen Typ, sondern stellt einen Ersatztyp statt Variant für Variablen zur Verfügung, die verwendet, aber nicht deklariert werden. Ich habe bisher die Def-Anweisungen noch nie in Gebrauch gesehen und kann sie auch nicht empfehlen.

Tipp Wenn Sie „Option Explicit“ nutzen, und das sollten Sie auch, müssen Sie alle Variablen deklarieren. Das macht die Def<Typ>-Anweisungen nutzlos, weil sie nur auf nicht deklarierte Variablen wirken.

Tabelle 6. *Beispiele für die verfügbaren Def-Anweisungen in OpenOffice.org.*

Def-Anweisung	Typ
DefBool b	Boolean
DefDate t	Date
DebDbl d	Double
DefInt i	Integer
DefLng l	Long
DefObj o	Object
DefVar v	Variant

3.2.3. Variablen einen Wert zuweisen

Der Zweck einer Variablen ist, einen Wert aufzunehmen. Um einer Variablen einen Wert zuzuweisen, schreiben Sie den Namen der Variablen, wahlweise Leerzeichen, ein Gleichheitszeichen, wahlweise Leerzeichen und den Wert, den die Variable erhalten soll. Etwa so:

```
X = 3.141592654
y = 6
```

Das optionale Schlüsselwort Let darf dem Variablennamen vorangehen, dient aber ausschließlich der Lesbarkeit. Das ähnliche Schlüsselwort Set, für Objektvariablen, dient auch nur der Lesbarkeit. Der Gebrauch dieser Schlüsselwörter ist sehr selten.

3.2.4. Boolsche Variablen sind entweder True oder False

Boolsche Variablen zeigen nur zwei Zustände: wahr oder falsch, als Wertrepräsentation True oder False. Intern sind es die Werte -1 beziehungsweise 0. Jeder einer boolschen Variablen zugewiesene numerische Wert, der nicht genau 0 ergibt, wird zu True konvertiert. Das Makro in Listing 8 führt einige neue Konzepte ein. Eine String-Variable, s, akkumuliert die Berechnungsergebnisse, die dann in einem Dialog angezeigt werden (s. Bild 18). Um einen Zeilenumbruch im Dialog zu erzeugen, wird CHR$(10) hinzugefügt. Die Rechenergebnisse in einer Zeichenfolge zusammenzufügen, braucht leider ein etwas kompliziertes Makro als einfache Anweisungen wie „`Print CBool(5=3)`“, aber die Resultate sind leichter zu verstehen(s. Bild 18). In der OOo-Version dieses Dokuments finden Sie häufig eine Schaltfläche, über das Sie das Makro direkt ausführen können.

Listing 8. *Demonstration der Konvertierung zum Typ Boolean.*

```
Sub ExampleBooleanType
  Dim b as Boolean
  Dim s as String
  b = True

  b = False
  b = (5 = 3) REM Ergibt False
  s = "(5 = 3) => " & b
  b = (5 < 7) REM Ergibt True
  s = s & CHR$(10) & "(5 < 7) => " & b
  b = 7       REM Ergibt True, weil 7 nicht 0 ist.
  s = s & CHR$(10) & "(7) => " & b
  MsgBox s
End Sub
```

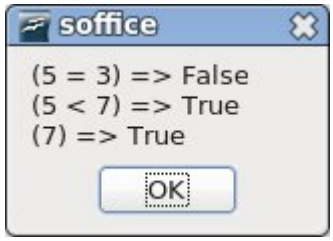


Bild 18. *Der von Listing 8 angezeigte Dialog.*

Intern wird True als -1 binär so dargestellt, dass alle Bits auf 1 gesetzt sind. Bei der binären Darstellung von False als 0 sind alle Bits auf 0.

3.2.5. Numerische Variablen

Numerische Variablen enthalten Zahlen. OOo Basic unterstützt Ganzzahlen, Fließkommazahlen und Währungszahlen. Ganzzahlen können hexadezimal (Basis 16), oktal (Basis 8) oder in der Standardform dezimal (Basis 10) repräsentiert werden. In der Praxis verwenden OOo-Nutzer fast ausschließlich Dezimalzahlen, aber der Vollständigkeit halber werden hier auch die anderen Typen vorgestellt.

Eine Diskussion der anderen Zahlensysteme ist deswegen wichtig, weil Computer ihre Daten intern binär speichern. Es ist einfach, zwischen binär, hexadezimal und oktal zu konvertieren, und für menschliche Wesen mag es einfacher sein, Binärzahlen visuell in anderen Zahlensystemen zu erkennen.

Dezimalzahlen, Basis 10, bestehen aus den 10 Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9. Addieren Sie dezimal 1 zu 9 und Sie erhalten 10. Im Computerbereich werden häufig auch Zahlen in binärer (Basis 2), oktaler (Basis 8) und hexadezimaler (Basis 16) Darstellung verwendet. Oktalzahlen bestehen aus den 8 Ziffern 0, 1, 2, 3, 4, 5, 6 und 7. Addieren Sie oktal 1 zu 7 und Sie erhalten 10 (Basis 8). Hexadezimalzahlen bestehen aus den 16 Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E und F. Binärzahlen bestehen aus den beiden Ziffern 0 und 1. Tabelle 7 enthält die Zahlen von 0 bis 18 in dezimaler Form, mit ihren binären, oktalen und hexadezimalen Entsprechungen.

Tabelle 7. *Zahlen in verschiedenen Darstellungssystemen.*

Dezimal	Binär	Oktal	Hexadezimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Dezimal	Binär	Oktal	Hexadezimal
17	10001	21	11
18	10010	22	12

Es wird vorausgesetzt, dass Ganzzahlen in dezimaler Form erscheinen. Kommas sind nicht erlaubt. Hexadezimalzahlen haben das Präfix „&H“ und Oktalzahlen das Präfix „&O“ (Buchstabe O, keine Null). Leider existiert kein einfacher Weg, Binärzahlen einzugeben. Tabelle 8 bietet einige einfache Richtlinien zur Zahleneingabe.

Tabelle 8. *Einige einfache Richtlinien zur Zahleneingabe in OOo Basic.*

Beispiel	Beschreibung
Schreiben Sie 1000, nicht 1.000	Schreiben Sie Zahlen ohne Tausenderkomma, verwenden Sie überhaupt keine Kommas.
+ 1000	Zwischen einem Plus- oder Minuszeichen und der Zahl ist ein Leerzeichen erlaubt.
&HFE ist dasselbe wie 254	Beginnen Sie eine Hexadezimalzahl mit &H.
&O11 ist dasselbe wie 9	Beginnen Sie eine Oktalzahl mit &O.
Schreiben Sie 3.1415, nicht 3,1415	Verwenden Sie keine Kommas als Dezimaltrenner.
6.022E23	In wissenschaftlicher Darstellung kann das „e“ groß oder klein geschrieben sein.
Schreiben Sie 6.6e-34, nicht 6.6e -34	Leerzeichen innerhalb von Zahlen sind nicht erlaubt. Mit dem Leerzeichen wird 6.6 - 34 = -27.4 errechnet.
6.022e+23	Vor dem Exponenten kann ein Plus- oder Minuszeichen stehen.
1.1e2.2 ergibt 1.1e2	Der Exponent muss eine Ganzzahl sein. Der Bruchteil wird ignoriert.

Wenn man einer numerischen Variablen eine Zeichenfolge zuweist, wird die Variable im Allgemeinen auf Null gesetzt. Es wird kein Fehler generiert. Wenn die ersten Zeichen des Strings aber eine Zahl ergeben, wird der String zu einer Zahl konvertiert, und der nicht numerische Rest des Strings wird ignoriert – mit der Möglichkeit eines numerischen Überlaufs.

Typ Integer

Eine Integer-Zahl ist eine Ganzzahl, die positiv, negativ oder gleich Null sein kann. Integer-Variablen sind eine gute Wahl für Zahlen ohne Bruchanteile, wie Lebensalter oder Kinderzahl. In OOo Basic sind Integer-Variablen 16-Bit-Zahlen im Bereich von -32.768 bis 32.767. Bei der Zuweisung einer Fließkommazahl zu einer Integer-Variablen wird zur nächsten Ganzzahl gerundet. Einem Variablennamen ein „%“ anzuhängen ist die Kurzform der Deklarierung als Typ Integer.

Listing 9. *Demonstration von Integer-Variablen.*

```
Sub ExampleIntegerType
  Dim i1 As Integer, i2%  REM i1 und i2 sind beide Integer-Variablen
  Dim f2 As Double
  Dim s$
  f2= 3.5
  i1= f2     REM i1 wird gerundet zu 4

  s = "3.50 => " & i1
  f2= 3.49
  i2= f2     REM i2 wird gerundet zu 3
  s = s & CHR$(10) & "3.49 => " & i2
  MsgBox s
End Sub
```

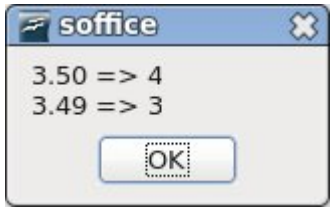


Bild 19. *Demonstration von Integer-Variablen im Listing 9.*

Typ Long Integer

„Long“ ist eine Ganzzahl mit einem größeren Bereich als der Typ Integer. Long-Variablen sind 32-Bit-Zahlen im Bereich von -2.147.483.648 bis 2.147.483.647. Long-Variablen benötigen daher zweimal so viel Speicherplatz wie Integer-Variablen, aber sie können Zahlen aufnehmen, die um ein *Vielfaches* größer sind. Bei der Zuweisung einer Fließkommazahl zu einer Long-Variablen wird zum nächsten Long-Ganzzahlwert gerundet. Einem Variablennamen ein „&“ anzuhängen ist die Kurzform der Deklarierung als Typ Long. Die Ausgabe von Listing 10 ist dieselbe wie von Listing 9 (s. Bild 19).

Listing 10. *Demonstration von Long-Variablen.*

```
Sub ExampleLongType
  Dim NumberOfDogs&, NumberOfCats As Long  ' Beide Variablen sind Long
  Dim f2 As Double
  Dim s$
  f2= 3.5
  NumberOfDogs = f2   REM Gerundet zu 4
  s = "3.50 => " & NumberOfDogs
  f2= 3.49
  NumberOfCats = f2   REM Gerundet zu 3
  s = s & CHR$(10) & "3.49 => " & NumberOfCats
  MsgBox s
End Sub
```

Typ Currency

Variablen vom Typ Currency sind, wie der Name schon sagt, für Finanzwerte gedacht. Der Typ Currency wurde ehemals eingeführt, um Rundungsfehler der Fließkommatypen Single und Double zu vermeiden. Visual Basic .NET ersetzte den Typ Currency durch den Typ Decimal.

Currency-Variablen sind 64-Bit-Festkommazahlen, die mit vier Dezimalstellen und 15 Stellen vor dem Komma genau berechnet werden. Der Bereich erstreckt sich von -922.337.203.658.477,5808 bis +922.337.203.658.477,5807. Einem Variablennamen ein „@“ anzuhängen ist die Kurzform der Deklarierung als Typ Currency.

Listing 11. *Demonstration von Currency-Variablen.*

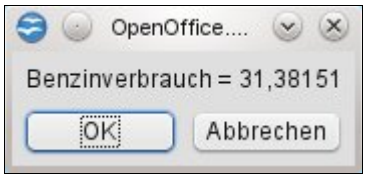
```
Sub ExampleCurrencyType
  Dim Income@, CostPerDog As Currency  'Einkommen und Hundehaltungskosten
  Income@ = 22134.37
  CostPerDog = 100.0 / 3.0
  REM Prints as 22134.3700
  Print "Einkommen = " & Income@
  REM Wird als 33.3333 ausgegeben
  Print "Kosten pro Hund = " & CostPerDog
End Sub
```

Typ Single

Variablen vom Typ Single können im Gegensatz zu denen vom Typ Integer einen Bruchanteil haben. Sie werden „Fließkommazahlen“ genannt, weil im Gegensatz zum Typ Currency die erlaubte Dezimalstellenanzahl nicht festgelegt ist. Single-Variablen sind 32-Bit-Zahlen mit einer Genauigkeit von etwa sieben angezeigten Ziffern, was für mathematische Operationen mittlerer Genauigkeit ausreicht. Sie umfassen positive und negative Werte von 3,402823 x 10E38 bis 1,401298 x 10E-45. Jede Zahl, die kleiner ist als 1,401298 x 10E-45, wird zu Null. Einem Variablennamen ein „!“ anzuhängen ist die Kurzform der Deklarierung als Typ Single.

Listing 12. *Demonstration von Single-Variablen.*

```
Sub ExampleSingleType
  Dim GallonsUsed As Single, Miles As Single, mpg!
  REM Amerikanische Methode für den durchschnittlichen
  REM Benzinverbrauch: Meilen pro Gallone
  GallonsUsed = 17.3
  Miles = 542.9
  mpg! = Miles / GallonsUsed
  Print "Benzinverbrauch = " & mpg!
End Sub
```



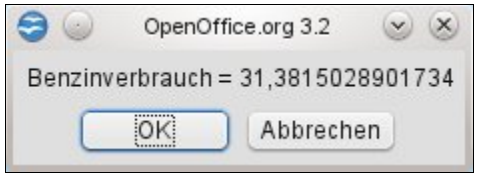
Typ Double

Variablen vom Typ Double ähneln denen vom Typ Single, außer dass sie 64 Bit lang sind und auf etwa 15 Ziffern genau sind. Sie eignen sich für mathematische Operationen hoher Genauigkeit. Sie umfassen positive und negative Werte von 1,79769313486232 x 10E308 bis 4,94065645841247 x 10E-324. Jede Zahl, die kleiner ist als 4,94065645841247 x 10E-324, wird zu Null. Einem Variablennamen ein „#“ anzuhängen ist die Kurzform der Deklarierung als Typ Double.

Listing 13. *Demonstration von Double-Variablen.*

```
Sub ExampleDoubleType
  Dim GallonsUsed As Double, Miles As Double, mpg#
  GallonsUsed = 17.3
  Miles = 542.9

  mpg# = Miles / GallonsUsed
  Print " Benzinverbrauch = " & mpg#
End Sub
```



3.2.6. String-Variablen enthalten Text

Variablen vom Typ String sind dazu gedacht, Text zu enthalten. In OOo wird Text im Standard von Unicode 2.0 gespeichert, wodurch eine Vielzahl von Sprachen unterstützt wird. Jede String-Variable kann bis zu 65.535 Zeichen aufnehmen. Einem Variablennamen ein „$“ anzuhängen ist die Kurzform der Deklarierung als Typ String.

Listing 14. *Demonstration von String-Variablen.*

```
Sub ExampleStringType
  Dim FirstName As String, LastName$
  FirstName = "Andrew"
  LastName$ = "Pitonyak"
  Print "Hallo " & FirstName & " " & LastName$
End Sub
```

Bild 20. *Demonstration von String-Variablen in Listing 14.*

Denken Sie immer daran, dass String-Variablen auf 65.535 Zeichen begrenzt sind. In einem Makro wurde die Anzahl der Zeichen in einem Textdokument auf folgende Art gezählt: das Dokument wurde in einen String umgewandelt, um dessen Länge zu entnehmen. Das Makro arbeitete aber nicht mehr, als mehr als 65.535 Zeichen im Dokument waren. In Visual Basic .NET können String-Variablen annähernd 2 Milliarden Unicode-Zeichen enthalten.

Um ein doppeltes Anführungszeichen in einen String einzufügen, setzen Sie es zweimal hintereinander.

```
S = "Sie sagte ""Hallo"""  REM Sie sagte "Hallo"
```

Mit „Option Compatible“ machen Sie Stringkonstanten von Visul Basic verfügbar (s. Tabelle 9). Sie müssen modulweit Option Compatible verwenden statt CompatibilityMode(True), denn Stringkonstanten werden zur Kompilierungszeit aufgelöst und nicht zur Laufzeit.

Tabelle 9. *Zu Visual Basic kompatible Stringkonstanten.*

Konstante	Wert	Beschreibung
vbCr	CHR$(13)	Zeilenrücklauf
vbCrLf	CHR$(13) & CHR$(10)	Kombination Zeilenrücklauf/Zeilenvorschub
vbFormFeed	CHR$(12)	Seitenvorschub
vbLf	CHR$(10)	Zeilenvorschub
vbNewLine	CHR$(13) & CHR$(10) oder CHR$(10)	Neue Zeile, betriebssystemabhängig – was gerade angebracht ist
vbNullChar	CHR$(0)	Zeichen mit dem ASCII-Wert 0
vbNullString	""	Leerer String. Ein String mit dem ASCII-Wert 0 am Ende.
vbTab	CHR$(9)	Waagerechter Tabulatorschritt
vbVerticalTab	CHR$(11)	Senkrechter Tabulatorschritt

Die Stringkonstanten in Tabelle 9 erlauben Ihnen, Strings mit Sonderzeichen zu definieren. Früher mussten Sie dafür die Funktion CHR$() aufrufen.

```
Option Compatible
Const sGreeting As String = "Hallo" & vbCr & "Johnny"  ' Inklusive Zeilenrücklauf (CR).
```

3.2.7. Date-Variablen

Variablen vom Typ Date enthalten Datums- und Uhrzeitwerte. OOo Basic speichert Date intern als Double. Date-Variablen werden wie alle numerischen Typen zu Null initialisiert. Das entspricht dem 30. Dezember 1899, 00:00:00 Uhr. Zu einem Date 1 zu addieren oder 1 von ihm zu subtrahieren entsprechen der Addition oder Subtraktion eines Tages. Eine Stunde, eine Minute und eine Sekunde entsprechen den Zahlen 1/24, 1/(24 * 60) und 1/(24 * 60 * 60). Die Datumsfunktionen von OOo Basic werden in Tabelle 10 vorgestellt und an späterer Stelle im einzelnen erörtert.

Listing 15. *Demonstration von Date-Variablen.*

```
Sub ExampleDateType
  Dim tNow As Date, tToday As Date
  Dim tBirthDay As Date
```

```
    tNow = Now()
    tToday = Date()
    tBirthDay = DateSerial(1776, 7, 4)
    Print "Heute = " & tToday
    Print "Jetzt = " & tNow
    Print "Insgesamt sind " & (tToday - tBirthDay) &_
      " Tage vergangen seit " & tBirthDay
End Sub
```

Negative Zahlen sind erlaubt und beziehen sich auf Datumswerte vor dem 30. Dezember 1899. Der 1. Januar 0001 wird also als Fließkommazahl -693.595 dargestellt. Weiter zurückgehend kommt man zu Datumswerten vor Christi Geburt (englische Abkürzung B.C = Before Christ) im Gegensatz zur heutigen Zeitzählung A.D. (Anno Domini). An späterer Stelle wird eingehend auf die Datumsbehandlung eingegangen.

Tabelle 10. *Funktionen und Subroutinen mit Bezug auf Datum und Uhrzeit.*

Funktion	Typ	Beschreibung
CDate(Ausdruck)	Date	Konvertiert Zeichenfolgen- oder numerische Ausdrücke in Datumswerte.
CDateFromIso(String)	Date	Gibt aus einer Zeichenkette mit einem Datum im ISO-Format eine Datumszahl im internen Format zurück.
CDateToIso(Date)	String	Gibt aus einer Datumszahl das Datum im ISO-Format zurück.
Date()	String	Gibt das aktuelle Systemdatum als Zeichenkette zurück.
DateSerial(Jahr, Monat, Tag)	Date	Erzeugt einen Datumswert für eine Datumsangabe aus den numerischen Werten Jahr, Monat und Tag.
DateValue(Date)	Date	Extrahiert das Datum aus einem Datum/Uhrzeit-Wert. Trennt dazu den Dezimalteil ab.
Day(Date)	Integer	Gibt aus einem Datumswert den Monatstag als Integer zurück.
GetSystemTicks()	Long	Gibt die vom Betriebssystem angegebene Anzahl von Systemzeit-Perioden (Systemticks) als Long zurück.
Hour(Date)	Integer	Gibt aus einem Datumswert die Stunde als Integer zurück.
IsDate(Wert)	Boolean	Ist dies ein Datumswert?
Minute(Date)	Integer	Gibt aus einem Datumswert die Minute als Integer zurück.
Month(Date)	Integer	Gibt aus einem Datumswert den Monat als Integer zurück.
Now()	Date	Gibt das aktuelle Systemdatum und die aktuelle Systemzeit als Date zurück.
Second(Date)	Integer	Gibt aus einem Datumswert die Sekunde als Integer zurück.
Time()	String	Gibt die aktuelle Systemzeit als Zeichenfolge zurück.
Timer()	Date	Gibt die seit Mitternacht vergangene Zeit in Sekunden zurück. Falls die Funktion in eine Long-Variable geschrieben wird, wird automatisch konvertiert.
TimeSerial(Stunde, Minute, Sekunde)	Date	Erzeugt einen Datumswert für eine Uhrzeitangabe aus den numerischen Werten Stunde, Minute und Sekunde.
WeekDay(Date)	Integer	Gibt aus einem Datumswert eine Zahl zwischen 1 und 7 zurück, entsprechend den Wochentagen Sonntag bis Samstag.
Year(Date)	Integer	Gibt aus einem Datumswert das Jahr als Integer zurück.

3.2.8. Eigene Datentypen erzeugen

In den meisten Implementierungen der Programmiersprache BASIC können Sie Ihre eigenen Datentypen erzeugen. OOo Basic erlaubt die Verwendung selbst definierter Datentypen.

Listing 16. *Demonstration benutzerdefinierter Typen.*

```
Type PersonType
  FirstName As String
  LastName As String
End Type

Sub ExampleCreateNewType
  Dim Person As PersonType
  Person.FirstName = "Andrew"
  Person.LastName  = "Pitonyak"
  PrintPerson(Person)
End Sub

Sub PrintPerson(x)
  Print "Person = " & x.FirstName & " " & x.LastName
End Sub
```

Tipp Obwohl benutzerdefinierte Typen nicht direkt ein Array enthalten können, so kann man es aber über den Typ Variant einrichten.

In OOo 3.2 gibt es drei Wege, eine Instanz eines benutzerdefinierten Typs zu deklarieren. Im folgenden Beispiel wird der Doppelpunkt verwendet, um zwei Anweisungen in einer Zeile zu platzieren.

```
Dim x As New PersonType                ' Der ursprüngliche Weg.
Dim y As PersonType                    ' New ist nun nicht mehr erforderlich.
Dim z : z = CreateObject("PersonType")  ' Das Objekt wird erstellt, wenn es nötig ist.
```

Wenn Sie Ihren eigenen Typ erstellen, verwenden Sie eine Struktur (häufig auch Struct genannt). OOo hat viele vordefinierte interne Strukturen. Eine häufig verwendete Struktur ist “com.sun.star.beans.PropertyValue”. Die internen OOo-Strukturen können genauso wie benutzerdefinierte Typen erstellt werden, aber auch mit der Funktion CreateUnoStruct (s. 10. Universal Network Objects (UNO)) auf der Seite 106.

```
Dim a As New com.sun.star.beans.PropertyValue
Dim b As New com.sun.star.beans.PropertyValue
Dim c : c = CreateObject("com.sun.star.beans.PropertyValue")
Dim d : d = CreateUnoStruct("com.sun.star.beans.PropertyValue")
```

Obwohl der Typ der Struktur „com.sun.star.beans.PropertyValue“ ist, ist es üblich, den Typnamen zum letzten Namensteil abzukürzen – in diesem Fall zu „PropertyValue“. Viele Objekte in OOo haben ähnlich lange sperrige Namen, die in diesem Buch auf ähnliche Weise abgekürzt werden.

Die meisten Variablen werden mit ihrem Wert kopiert. Das bedeutet, dass wenn ich eine Variable einer anderen zuweise, der Wert der einen in die andere kopiert wird. Sie verweisen nicht auf dieselben Daten, sondern sie enthalten je eine Kopie der Daten. Das gilt auch für benutzerdefinierte Typen und interne OOo-Strukturen. Auf diesem Wege deklarierte Variablen werden mit ihrem Wert kopiert. Andere intern von OOo genutzte Typen, so genannte Universal Network Objects, werden als Referenz kopiert. Obwohl sie erst an späterer Stelle erläutert werden, ist es jetzt schon wichtig, darüber nachzudenken, was geschieht, wenn eine Variable einer anderen zugewiesen wird. Wenn ich eine Variable einer anderen als Referenz zuweise, verweisen beide Variablen auf dieselben Daten. Wenn zwei Variablen auf dieselben Daten verweisen und ich eine ändere, ändere ich beide.

3.2.9. Variablen mit speziellen Typen deklarieren

Sie können mit den Schlüsselwörtern „As New“ eine Variable als bekanntes UNO-Struct definieren. Das Wort „Struct“ ist abgekürzt von dem Wort „Structure“, das häufig von Computerprogrammierern verwendet wird. Ein Struct enthält einen oder mehrere Datenelemente (so genannte Members), die

von unterschiedlichem Typ sein können. Structs werden eingesetzt, um zusammengehörende Daten zu gruppieren.

Option Compatible bietet eine neue Syntax zur Definition von bekannten und unbekannten Typen. Ein einfaches Beispiel deklariert eine Variable eines speziellen Typs, auch wenn OOo Basic den Typ nicht kennt.

```
Option Compatible          'Unterstützt in OOo 2.0
Sub Main
  Dim oVar1 As Object
  Dim oVar2 As MyType
  Set oVar1 = New MyType    ' Unterstützt in OOo 2.0
  Set oVar2 = New MyType    ' Unterstützt in OOo 2.0
  Set oVar2 = New YourType ' Fehler, als MyType deklariert, nicht als YourType.
```

Mit OOo 2.0 wurde eine neue OLE-Objekt-Factory geschaffen, mit der neue Typen erstellt werden können. Die neue Funktionalität macht es möglich, mit OOo Basic Microsoft-Word-Dokumente unter Microsoft Windows zu bearbeiten, vorausgesetzt Microsoft Office ist auch installiert.

```
Sub Main
  Dim W As Word.Application
  Set W = New Word.Application
  REM Dim W As New Word.Application        'Funktioniert in OOo 2.0
  REM W = CreateObject("Word.Application") 'Funktioniert in OOo 2.0
  W.Visible = True
End Sub
```

Zum Einsatz von CreateObject() benötigen Sie kein „Option Compatible“, weil diese Funktionalität von der neuen OLE-Objekt-Factory beginnend mit OOo 2.0 bereitgestellt wird.

3.2.10. Objekt-Variablen

Ein Object ist ein komplexer Datentyp, der mehr als ein einzelnes Datenelement enthalten kann. Der Code in Listing 16 zeigt ein Beispiel eines komplexen Datentyps. In OpenOffice.org dienen Object-Variablen dazu, mit OOo Basic erstellte und definierte komplexe Datentypen aufzunehmen. Bei der Deklarierung einer Variablen vom Typ Object wird sie mit dem Spezialwert Null initialisiert, der signalisiert, dass die Variable momentan keinen gültigen Wert enthält.

Wenn Sie auf interne OpenOffice.org-Objekte verweisen, verwenden Sie aber besser den Typ Variant, nicht Object. Warum, wird an späterer Stelle erörtert.

3.2.11. Variant-Variablen

Variablen vom Typ Variant können jeden Datentyp aufnehmen. Welche Daten ihnen auch immer zugewiesen werden, sie übernehmen deren Typ. Bei der Deklarierung einer Variant-Variablen wird sie zu dem Spezialwert Empty initialisiert, der signalisiert, dass die Variable momentan keinen Wert enthält. Eine Variant-Variable kann in der einen Anweisung einen Integer-Wert erhalten und in der nächsten einen Text. Wird einer Variant-Variablen ein Wert zugewiesen, wird auch nicht automatisch konvertiert. Sie erhält einfach nur den passenden Typ.

Wegen des chamäleonähnlichen Verhaltens der Variant-Variablen kann man sie wie jeden anderen Variablentyp verwenden. Diese Flexibilität hat aber seinen Preis: Zeit. Und schließlich hat man noch das Problem, dass nach einigen Zuweisungen nicht immer deutlich ist, welchen Typ eine Variant-Variable gerade repräsentiert.

Listing 17. *Demonstration von Variant-Variablen.*

```
Sub ExampleTestVariants
  DIM s As String
  DIM v As Variant
```

```
    REM v ist beim Start leer
    s = s & "1  : TypeName = " & TypeName(v) & " Wert = " & v & CHR$(10)
    v = "ab217" : REM v wird zu String
    s = s & "2  : TypeName = " & TypeName(v) & " Wert = " & v & CHR$(10)
    v = True    : REM v wird zu Boolean
    s = s & "3  : TypeName = " & TypeName(v) & " Wert = " & v & CHR$(10)
    v = (5=5)   : REM v wird zu Integer statt zu Boolean
    s = s & "4  : TypeName = " & TypeName(v) & " Wert = " & v & CHR$(10)
    v = 123.456 : REM Double
    s = s & "5  : TypeName = " & TypeName(v) & " Wert = " & v & CHR$(10)
    v =123      : REM Integer
    s = s & "6  : TypeName = " & TypeName(v) & " Wert = " & v & CHR$(10)
    v = 1217568942 : REM Es könnte Long sein, tatsächlich aber wird es Double
    s = s & "7  : TypeName = " & TypeName(v) & " Wert = " & v & CHR$(10)
    MsgBox s, 0, "Variant nimmt viele Typen an"
End Sub
```

Visual Basic .NET unterstützt den Typ Variant nicht. Nicht typisierte Variablen sind vom Typ Object.

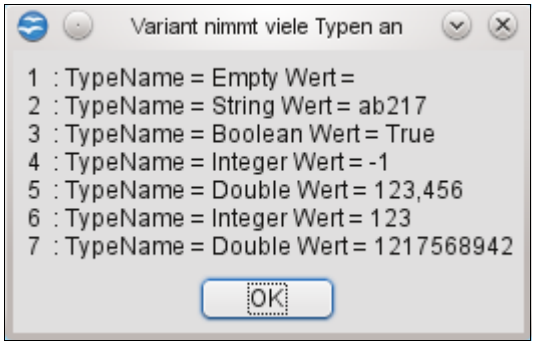


Bild 21. *Variant übernimmt den zugewiesenen Typ.*

Bei der Wertzuweisung in eine Variant-Variable wird der Wert nicht zu einem passenden Typ konvertiert. Stattdessen wird Variant zu dem Typ des Wertes. In Zeile 6 im Bild 21 ist Variant ein Integer. In Zeile 7 ist die Zahl zu groß für Integer, aber klein genug für Long. OOo Basic zieht es aber vor, alle Zahlen, die größer als Integer sind und alle Fließkommazahlen nach Double zu konvertieren, auch wenn sie als Single oder Long kodiert werden könnten.

3.2.12. Konstanten

Eine Konstante ist eine Variable ohne Typ, deren Wert nicht geändert werden kann. Die Variable ist als Platzhalter bestimmt, der durch den Ausdruck ersetzt wird, mit dem sie definiert wird. Konstanten werden mit dem Schlüsselwort Const definiert. Die Regeln für Konstantennamen sind dieselben wie für gültige Variablennamen.

```
Const ConstName=Expression
```

Konstanten bereichern Makros auf manche Weise. Nehmen Sie einmal eine Schwerkraftkonstante, wie sie oft in der Physik benötigt wird. Physiker werden sie als Erdbeschleunigung in Metern pro Sekundenquadrat erkennen.

```
Const Gravity = 9.81
```

Ein paar konkrete Vorteile bei der Verwendung von Konstanten:

- Konstanten verbessern die Lesbarkeit eines Makros. Das Wort Gravity (Schwerkraft) ist leichter zu erkennen als der Wert 9,81.
- Konstanten sind leicht zu pflegen. Wenn ich eine größere Genauigkeit brauche oder wenn sich die Anziehungskraft ändert, muss ich den Wert nur an einer einzigen Stelle ändern.

- Konstanten helfen, schwer zu findende Fehler dadurch zu vermeiden, dass Laufzeitfehler in Fehler zur Kompilierungszeit umgewandelt werden. Wenn Sie „Grevity“ schreiben statt „Gravity“, ist das ein Kompilierungsfehler, der Schreibfehler „9.18“ statt „9.81“ aber nicht.
- Ein Wert wie 9,81 mag für Sie unverwechselbar sein, für andere aber möglicherweise nicht, die Ihren Code später einmal lesen. Die Zahl wird zu dem, was Programmierer eine „magische Zahl“ nennen, und erfahrene Programmierer versuchen, solche magischen Zahlen unter allen Umständen zu vermeiden. Durch den Umstand, dass sie nicht erläutert sind, ergeben sich später Schwierigkeiten bei der Programmpflege, wenn nämlich der Programmautor nicht für eine Erklärung verfügbar ist – oder die Einzelheiten komplett vergessen hat.

Tipp OpenOffice.org definiert die Konstante Pi, eine mathematische Konstante mit einem Wert von angenähert 3,1415926535897932385.

3.3. Die With-Anweisung

Die Anweisung With vereinfacht den Zugriff auf komplexe Datentypen. Listing 16 Definiert einen Datentyp, der zwei verschiedene Variablen enthält: FirstName und LastName. Sie greifen auf diese Variablen zu, indem Sie einen Punkt zwischen Variablennamen und Datenelement setzen.

```
Sub ExampleCreateNewType
  Dim Person As PersonType
  Person.FirstName = "Andrew"
  Person.LastName  = "Pitonyak"
End Sub
```

Oder ... Die Anweisung With bietet Ihnen eine Kurzschreibweise des Zugriffs auf mehrere Datenelemente derselben Variablen.

```
Sub ExampleCreateNewType
  Dim Person As PersonType
  With Person
    .FirstName = "Andrew"
    .LastName  = "Pitonyak"
  End With
End Sub
```

Vergleichbar:

```
Dim oProp As New com.sun.star.beans.PropertyValue
oProp.Name  = "Person"      'Setzt die Eigenschaft Name
oProp.Value = "Boy Bill"    'Setzt die Eigenschaft Value
```

Mit With:

```
Dim oProp As New com.sun.star.beans.PropertyValue
With oProp
  .Name  = "Person"      'Setzt die Eigenschaft Name
  .Value = "Boy Bill"    'Setzt die Eigenschaft Value
End With
```

3.4. Arrays

Ein Array ist eine Datenstruktur, in der gleichartige Datenelemente in einer indexierten Listenstruktur angeordnet sind – zum Beispiele eine Spalte mit Namen oder eine Tabelle mit Zahlen, s. Tabelle 11. Ein Array bietet Ihnen die Möglichkeit, viele verschiedene Werte in einer einzigen Variablen zu speichern. Zur Definition der Array-Elemente und zum Zugriff auf die Elemente werden runde Klammern verwendet. OOo Basic kennt im Gegensatz zu anderen Sprachen wie zum Beispiel C oder Java keine eckigen Klammern.

Array-Variablen werden mit der Anweisung Dim deklariert. Stellen Sie sich ein eindimensionales Array als eine Wertespalte vor und ein zweidimensionales Array als eine Wertetabelle. Das System erlaubt auch höher dimensionierte Arrays, die man sich aber nur schwer bildhaft machen kann. Der Index eines Arrays ist vom Typ Integer, kann also den Wertebereich von -32.768 bis 32.767 abdecken.

Tabelle 11. *Es ist einfach, ein Array zu deklarieren!*

Definition	Elemente	Beschreibung
`Dim a(5) As Integer`	6	Von 0 bis 5 inklusive.
`Dim b(5 To 10) As String`	6	Von 5 bis 10 inklusive.
`Dim c(-5 To 5) As String`	11	Von -5 bis 5 inklusive.
`Dim d(5, 1 To 6) As Integer`	36	Sechs Reihen mit sechs Spalten von 0 bis 5 und 1 bis 6.
`Dim e(5 To 10, 20 To 25) As Long`	36	Sechs Reihen mit sechs Spalten von 5 bis 10 und 20 bis 25.

Tipp Bevor Sie Array-Variablen verwenden, müssen Sie sie deklarieren, auch wenn Sie „Option Explicit“ nicht verwenden.

Wenn das untere Bereichsende eines Arrays nicht angegeben ist, wird der Standardwert Null gesetzt – im Sprachgebrauch der Programmierer „nullbasiert“. Somit hat ein Array mit fünf Elementen die Indexnummerierung von a(0) bis a(4). Wenn Sie den Standard der unteren Arraybereichsgrenze auf 1 statt 0 ändern wollen, setzen Sie die Schlüsselwörter „Option Base 1“ vor alle anderen ausführbaren Anweisungen im Programm.

```
Option Base { 0 | 1 }
```

Tipp Geben Sie besser die untere Bereichsgrenze eines Arrays an, als auf das Standardverhalten zu setzen. Das ist übertragbar und ändert sich auch dann nicht, wenn Option Base verwendet wird.

Dim a(3) sieht vier Elemente vor: a(0), a(1), a(2) und a(3). Option Base bringt keine Änderung der Anzahl der Elemente, es ändert nur die Indexierung. Mit Option Base 1 ergibt dieselbe Deklarierung immer noch vier Elemente: a(1), a(2), a(3) und a(4). Dieses Verhalten ist für mich nicht gerade intuitiv. Ich kann daher den Gebrauch von Option Base nicht empfehlen. Wenn Sie spezifische Arraybereichsgrenzen benötigen, ist am besten, sie explizit zu deklarieren, zum Beispiel Dim a(1 To 4). Option Base hat seine Tücken, wenn es um die Erstellung einer sauberen Dokumentierung oder um eine sichere Portierung geht.

Visual Basic behandelt Option Base 1 anders als OOo Basic. VB setzt die untere Grenze auf 1, ändert aber die obere Grenze nicht. Visual Basic .NET unterstützt Option Base gar nicht mehr. Wenn Sie „Option Compatible“ verwenden, wird mit „Option Base 1“ die obere Grenze nicht um 1 angehoben. Mit anderen Worten, OOo Basic verhält sich wie VB.

Es ist einfach, lesend oder schreibend auf die Werte in einem Array zuzugreifen. Ein Array auf diesem Weg zu initialisieren ist aber mühselig.

Listing 18. *Demonstration eines einfachen Arrays.*

```
Sub ExampleSimpleArray1
  Dim a(2) As Integer, b(-2 To 1) As Long
  Dim m(1 To 2, 3 To 4)

  REM Erinnern Sie sich, dass mehrere Anweisungen
  REM auf einer Zeile stehen können, mit Doppelpunkt getrennt?
  a(0) = 0    : a(1) = 1   : a(2) = 2
  b(-2) = -2  : b(-1) = -1 : b(0) = 0 : b(1) = 1
```

```
  m(1, 3) = 3 : m(1, 4) = 4
  m(2, 3) = 6 : m(2, 4) = 8
  Print "m(2,3) = " & m(2,3)
  Print "b(-2) = " & b(-2)
End Sub
```

Zum schnellen Füllen eines Arrays vom Typ Variant dient die Funktion Array (s. Listing 19), die ein Variant-Array mit den aufgeführten Werten zurückgibt. Die Funktionen LBound und Ubound geben die untere und die obere Grenze des Arraybereichs zurück. Alle von OOo Basic bereitgestellten Array-Routinen sind in Tabelle 12 aufgelistet und werden an späterer Stelle ausführlich erörtert.

Listing 19. *Verwenden Sie Array(), um auf schnelle Art ein Array zu füllen.*

```
Sub ExampleArrayFunction
  Dim a, i%, s$
  a = Array("Null", 1, Pi, Now)
  Rem String, Integer, Double, Date
  For i = LBound(a) To UBound(a)
    s$ = s$ & i & " : " & TypeName(a(i)) & " : " & a(i) & CHR$(10)
  Next
  MsgBox s$, 0, "Beispiel für die Funktion Array"
End Sub
```

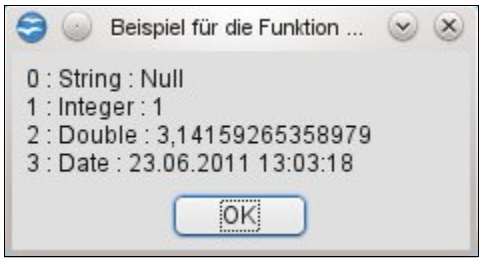


Bild 22. *Unterschiedliche Variablentypen in ein und demselben Array.*

Eine als Array definierte, aber nicht dimensionierte Variable, wie zum Beispiel Dim a(), heißt leeres Array. Prüfen Sie, ob ein Array leer ist, indem Sie die obere mit der unteren Grenze des Arraybereichs vergleichen. Das Array ist leer, das heißt, es ist nicht dimensioniert, wenn die obere Grenze kleiner ist als die untere. Ein dimensioniertes Array, wie Dim a(5), ist nicht leer.

Das Verhalten von Lbound und Ubound hat sich mit der Zeit geändert. Einige OOo-Versionen generieren für Ubound(b) einen Fehler, andere wiederum nicht. Alle Versionen sollten aber problemlos mit Ubound(b()) arbeiten. Zu dem Zeitpunkt, zu dem diese Zeilen geschrieben werden, gibt es einen Fehler bei der Ermittlung der oberen und unteren Grenze für c (in Listing 20), weil c ein leeres Objekt ist.

Listing 20. *Runde Klammern sind nicht immer erforderlich, aber immer erlaubt.*

```
Sub ArrayDimensionError
  On Error Goto ErrorHandler
  Dim a(), b(1 To 2), c
  Dim iLine As Integer
  Dim s$
  REM Gültige Konstrukte
  iLine = 1 : s = "a = (" & LBound(a()) & ", "
  iLine = 2 : s = s & UBound(a) & ")"
  iLine = 3 : s = s & CHR$(10) & "b = (" & LBound(b()) & ", "
  iLine = 4 : s = s & UBound(b) & ")"
  REM Ungültige Konstrukte
  iLine = 5 : s = s & CHR$(10) & "c = (" & LBound(c()) & ", "
  iLine = 6 : s = s & UBound(c) & ")"
  MsgBox s, 0, "LBound und UBound"
```

```
    Exit Sub
    ErrorHandler:
    s = s & CHR$(10) & "Fehler " & Err & ": " & Error$ & " (Zeile : " & iLine & ")"
    Resume Next
End Sub
```

Tabelle 12. *Liste der zu Arrays gehörenden Subroutinen und Funktionen.*

Funktion	Beschreibung
Array(args)	Gibt ein Variant-Array zurück, mit den Argumenten als Elemente.
DimArray(args)	Gibt ein leeres Variant-Array zurück, dimensioniert durch die Argumente.
IsArray(var)	Gibt True zurück, wenn die Variable ein Array ist, ansonsten False.
Join(array) Join(array, delimiter)	Gibt einen String zurück, der die einzelnen Array-Elemente hintereinander enthält, jeweils getrennt durch den optionalen Trennstring. Standardtrenner ist das Leerzeichen.
LBound(array) LBound(array, dimension)	Gibt die untere Bereichsgrenze des Arrays zurück. Die optionale Dimensionsangabe bestimmt die zu berücksichtigende Dimension. Die erste Dimension ist 1.
ReDim var(args) As Type	Ändert die Array-Dimensionen mit derselben Syntax wie Dim. Mit dem Schlüsselwort Preserve bleiben die bestehenden Werte erhalten – ReDim Preserve x(1 To 4) As Integer.
Split(str) Split(str, delimiter) Split(str, delimiter, n)	Splittet den String in ein Array von Strings. Der Standardtrenner ist ein Leerzeichen. Das optionale Argument „n“ begrenzt die Anzahl der entnommenen Stringelemente.
UBound(array) UBound(array, dimension)	Gibt die obere Bereichsgrenze des Arrays zurück. Die optionale Dimensionsangabe bestimmt die zu berücksichtigende Dimension. Die erste Dimension ist 1.

3.4.1. Die Dimensionen eines Arrays ändern

Die gewünschte Dimensionierung eines Arrays ist nicht immer von vornherein bekannt. Manchmal ist sie zwar bekannt, ändert sich aber periodisch, und der Code muss geändert werden. Eine Array--Variable kann mit oder ohne spezifizierte Dimensionen deklariert werden. OOo Basic bietet ein paar unterschiedliche Methoden, Array-Dimensionen zu setzen oder zu ändern.

Die Funktion Array erstellt ein Array vom Typ Variant, das schon Werte enthält. So hat man schnell ein Array initialisiert. Sie brauchen die Array-Dimensionen nicht zu setzen, aber wenn Sie es tun, werden sie sich ändern und die Dimensionen annehmen, die von der Funktion Array bestimmt sind.

```
Dim a()
a = Array(3.141592654, "PI", 9.81, "Schwerkraft")
```

Die an die Funktion Array übergebenen Argumente werden zu den Elementen des erzeugten Variant-Arrays. Die Funktion DimArray andererseits interpretiert die Argumente als Dimensionierung des zu erzeugenden Arrays (s. Listing 21). Die Argumente können aus Ausdrücken bestehen, somit kann die Dimension mit Hilfe einer Variablen gesetzt werden.

Listing 21. *Ein Array neu dimensionieren.*

```
Sub ExampleDimArray
  Dim a(), i%
  Dim s$
  a = Array(10, 11, 12)
  s = "" & LBound(a()) & " " & UBound(a())       Rem 0 2
  a() = DimArray(3)                              REM Dasselbe wie Dim a(3)
  a() = DimArray(2, 1)                           REM Dasselbe wie Dim a(2,1)
  i = 4
  a = DimArray(3, i)                             Rem Dasselbe wie Dim a(3,4)
  s = s & CHR$(10) & LBound(a(),1) & " " & UBound(a(),1)  Rem 0, 3
```

```
  s = s & CHR$(10) & LBound(a(),2) & " " & UBound(a(),2)  Rem 0, 4
  a() = DimArray()      REM Ein leeres Array
  MsgBox s, 0, "Beispiel für DimArray"
End Sub
```

Die Funktionen Array und DimArray geben beide ein Array von Variant-Elementen zurück. Die Anweisung ReDim ändert die Dimensionen eines bestehenden Arrays. Die Änderung kann sich gleichermaßen auf die einzelnen Dimensionen beziehen wie auch auf die Anzahl der Dimensionen. Die Argumente können aus Ausdrücken bestehen, denn die Anweisung ReDim wird zur Laufzeit ausgewertet.

```
Dim e() As Integer, i As Integer
i = 4
ReDim e(5) As Integer       REM Dimension ist 1, mit gültiger Größe 0 To 5.
ReDim e(3 To 10) As Integer REM Dimension ist 1, mit gültiger Größe 3 To 10.
ReDim e(3, i) As Integer    REM Dimension ist 2, mit gültiger Größe (0 To 3, 0 To 4).
```

Ein paar Tipps zu Arrays:

- LBound and UBound funktionieren mit leeren Arrays.
- Ein leeres Array hat nur eine Dimension, mit der unteren Grenze 0 und der oberen Grenze -1.
- Mit ReDim können Sie ein bestehendes Array leeren.

Die Anweisung ReDim kann mit dem Schlüsselwort Preserve aufgerufen werden. Damit werden nach Möglichkeit bei der Änderung der Dimensionen die Werte der Elemente erhalten. Bei einer Erweiterung der Dimensionierung bleiben alle Daten erhalten, bei einer Einschränkung gehen jedoch Daten verloren, da Elemente abgetrennt werden. Das kann an beiden Enden geschehen. Wenn ein Element des neuen Arrays im alten existierte, bleibt der Wert unverändert. Im Gegensatz zu manchen BASIC-Dialekten erlaubt OOo Basic die Änderung aller Dimensionen eines Arrays bei gleichzeitigem Datenerhalt.

```
Dim a() As Integer
ReDim a(3, 3, 3) As Integer
a(1,1,1) = 1 : a(1, 1, 2) = 2 : a(2, 1, 1) = 3
ReDim preserve a(-1 To 4, 4, 4) As Integer
Print "(" & a(1,1,1) & ", " & a(1, 1, 2) & ", " & a(2, 1, 1) & ")"
```

ReDim spezifiziert sowohl die Dimensionen wie auch einen optionalen Typ. Wenn der Typ mit angegeben ist, muss er dem Typ entsprechen, mit dem die Variable deklariert wurde. Ansonsten erzeugt OOo einen Fehler zur Kompilierungszeit.

Listing 22 ist eine Dienstfunktion, der ein einfaches Array übergeben wird und die einen String mit allen Elementen des Arrays zurückgibt. Der Code des ReDim-Beispiels, auch in Listing 22, verwendet ArrayToString.

Listing 22. *Dienstfunktion Array zu String.*

```
REM ArrayToString übernimmt ein einfaches Array und schreibt den Wert
REM eines jeden Elements des Arrays in einen String.
Function ArrayToString(a() As Variant) As String
  Dim i%, s$
  For i% = LBound(a()) To UBound(a())
    s$ = s$ & i% & " : " & a(i%) & CHR$(10)
  Next
  ArrayToString = s$
End Function

Sub ExampleReDimPreserve
  Dim a(5) As Integer, b(), c() As Integer
  a(0) = 0 : a(1) = 1 : a(2) = 2 : a(3) = 3 : a(4) = 4 : a(5) = 5
```

```
  Rem a ist dimensioniert von 0 bis 5, worin gilt a(i) = i
  MsgBox ArrayToString(a()), 0 , "a() zu Anfang"

  Rem a wird umdimensioniert von 1 bis 3, worin gilt a(i) = i
  ReDim Preserve a(1 To 3) As Integer
  MsgBox ArrayToString(a()), 0 , "a() nach ReDim"

  Rem Array() gibt den Typ Variant zurück.
  Rem b ist dimensioniert von 0 bis 9, worin gilt b(i) = i+1
  b = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  MsgBox ArrayToString(b()), 0 , "b() nach der ersten Zuweisung"

  Rem b ist dimensioniert von 1 bis 3, worin gilt b(i) = i+1
  ReDim Preserve b(1 To 3)
  MsgBox ArrayToString(b()), 0 , "b() nach ReDim"

  Rem Das folgende ist NICHT gültig, weil das Array schon
  Rem auf eine andere Größe dimensioniert ist.
  Rem a = Array(0, 1, 2, 3, 4, 5)

  Rem c ist dimensioniert von 0 bis 5, worin gilt c(i) = i
  Rem Wäre „ReDim" auf c angewendet worden, dann würde folgendes NICHT funktionieren.
  c = Array(0, 1, 2, "drei", 4, 5)
  MsgBox ArrayToString(c()), 0 , "Integer-Array c() dem Typ Variant zugewiesen"

  Rem Ironischerweise ist das folgende erlaubt, aber c wird keine Daten enthalten!
  ReDim Preserve c(1 To 3) As Integer
  MsgBox ArrayToString(c()), 0 , "ReDim Integer c() nach der Zuweisung zu Variant"
End Sub
```

In Visual Basic gibt es verschiedene Regelungen, die Dimensionen eines Arrays zu ändern, und diese Regelungen unterscheiden sich von Version zu Version von Visual Basic. Ganz allgemein ist OOo Basic flexibler.

3.4.2. Unerwartetes Verhalten von Arrays

Wenn man eine Integer-Variable einer anderen zuweist, wird der Wert kopiert, und die Variablen haben keinen weiteren Bezug mehr zueinander. Anders gesagt, wenn man den Wert der ersten Variablen ändert, ändert sich nichts am Wert der zweiten Variablen. Das gilt nicht für Array-Variablen. Wenn man eine Array-Variable einer anderen zuweist, wird statt einer Kopie eine so genannte Referenz auf das erste Array erstellt. Alle Änderungen an der einen Variablen werden automatisch auch von der anderen erkannt. Es ist unerheblich, welche der beiden geändert wird, immer sind beide betroffen. Das ist der Unterschied zwischen „Argumente übergeben als *Wert*“ (Integers) und „Argumente übergeben als *Referenz*“ (Arrays).

Listing 23. *Arrays werden als Referenz kopiert*

```
Sub ExampleArrayCopyIsRef
  Dim a(5) As Integer, c(4) As Integer, s$
  c(0) = 4 : c(1) = 3 : c(2) = 2 : c(3) = 1 : c(4) = 0
  a() = c()
  a(1) = 7
  c(2) = 10
  s$ = "**** a() *****" & CHR$(10) & ArrayToString(a()) & CHR$(10) &_
       CHR$(10) & "**** c() *****" & CHR$(10) & ArrayToString(c())
  MsgBox s$, 0 , "Ändern Sie eins, ändern Sie beide"
End Sub
```

Zur Verdeutlichung, dass Arrays als Referenz zugewiesen werden, erstellen wir drei Arrays – a(), b() und c() – wie in Bild 23 gezeigt. Intern erstellt OOo Basic drei Arrays, die durch a(), b() und c() referenziert werden.

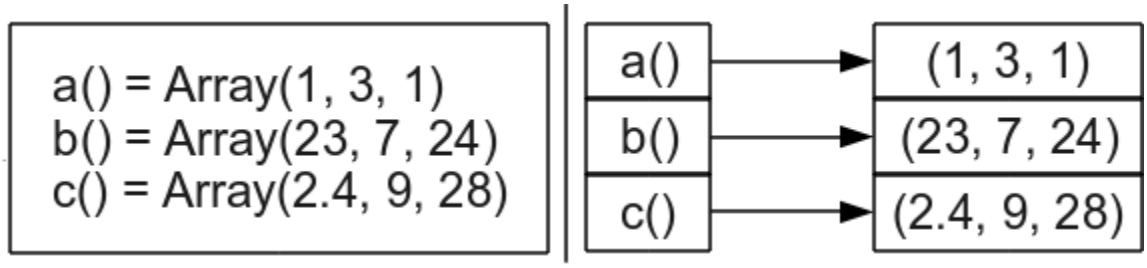


Bild 23. *Die Zuweisung eines Arrays weist eine Referenz zu.*

Weist man das Array a() dem Array b() zu, dann referenzieren a() und b() dieselben Daten. Es ist nicht so, dass die Variable a() die Variable b() referenziert, sondern sie referenziert dieselben Daten, die auch b() referenziert. (s. Bild 24). Daher ändert man mit a() gleichzeitig auch b(). Das von a() ursprünglich referenzierte Array wird nun nicht mehr referenziert.

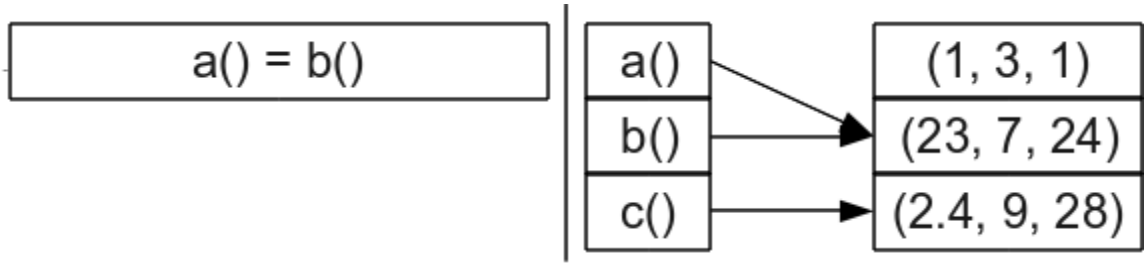


Bild 24. *Die Zuweisung eines Arrays weist eine Referenz zu.*

Weist man das Array b() dem Array c() zu, dann referenzieren b() und c() dieselben Daten. Die Variable a() bleibt unverändert, wie in Bild 25 zu sehen ist.

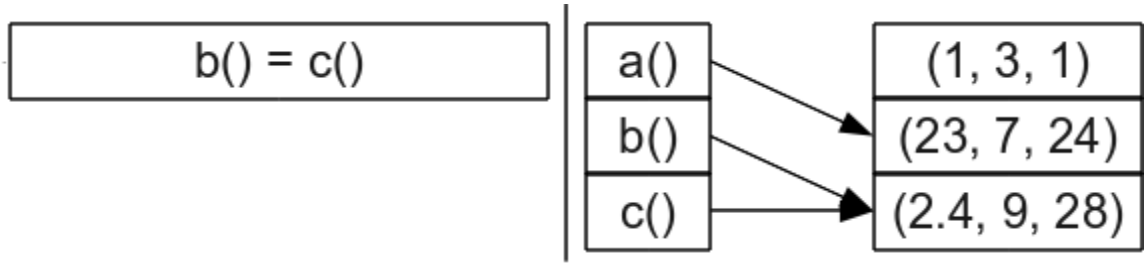


Bild 25. *Die Zuweisung eines Arrays weist eine Referenz zu.*

Tipp Wenn ein Array einem anderen zugewiesen wird, gibt es keine Typüberprüfung. Weisen Sie also keine Arrays unterschiedlichen Typs einander zu.

Weil es keine Typüberprüfung bei der Zuweisung eines Arrays zu einem anderen gibt, können unerwartete und undurchsichtige Probleme entstehen. Die Funktion Array gibt ein Variant-Array zurück und ist die schnellste Methode, mehrere Werte einer Array-Variablen zuzuweisen. Ein offensichtliches Problem ist, dass ein Integer-Array plötzlich String-Werte enthalten kann, wenn es ein Variant-Array referenziert. Ein weniger offensichtliches Problem ist, dass die Anweisung ReDim auf dem deklarierten Typ aufsetzt. Die Anweisung „ReDim Preserve“, angewendet auf ein Integer-Array, das einem Variant-Array zugewiesen wurde, versagt darin, die bestehenden Werte zu erhalten.

```
Dim a() As Integer                  REM Deklariert a() als Integer()
a() = Array(0, 1, 2, 3, 4, 5, 6)    REM Zuweisung eines Variant() zu einem Integer()
ReDim Preserve a(1 To 3) As Integer REM Leert das Array
```

Es braucht eine andere Methode zur sicheren Array-Zuweisung, wenn man den korrekten Datentyp erhalten möchte. Kopieren Sie jedes Element des Arrays einzeln. So wird auch verhindert, dass zwei Array-Variablen dasselbe Array referenzieren.

Listing 24. *Komplexeres Array-Beispiel.*

```
Sub ExampleSetIntArray
  Dim iA() As Integer
  SetIntArray(iA, Array(9, 8, "7", "sechs"))
  MsgBox ArrayToString(iA), 0, "Ein Variant-Array einem Integer-Array zuweisen"
End Sub

REM Das erste Array erhält dieselben Dimensionen wie das zweite.
REM Dann wird eine elementweise Kopie des Arrays vorgenommem.
Sub SetIntArray(iArray() As Integer, v() As Variant)
  Dim i As Long
  ReDim iArray(LBound(v()) To UBound(v())) As Integer
  For i = LBound(v) To UBound(v)
    iArray(i) = v(i)
  Next
End Sub
```

3.5. Subroutinen und Funktionen

Subroutinen sind Codezeilen, die zu sinnvollen Arbeitsabschnitten gruppiert sind. Eine Funktion ist eine Subroutine, die einen Wert zurückgibt. Der Einsatz von Subroutinen und Funktionen erleichtert die Fehlersuche, die Mehrfachverwendung und die Lesbarkeit des Codes. So werden Fehlerquellen reduziert.

Das Schlüsselwort Sub bestimmt den Beginn einer Subroutine, End Sub bestimmt das Ende.

```
Sub FirstSub
  Print "Führe FirstSub aus"
End Sub
```

Zum Aufruf einer Subroutine schreiben Sie den Namen der Subroutine in eine Zeile. Optional können Sie das Schlüsselwort Call davorsetzen.

```
Sub Main
  Call FirstSub ' Ruft das Sub FirstSub auf.
  FirstSub      ' Ruft das Sub FirstSub noch einmal auf.
End Sub
```

Die Namen von Subroutinen und Funktionen dürfen in einem Modul nur einmal vorkommen. Die Regeln zur Namensgebung sind dieselben wie bei Variablen, auch der Umgang mit Leerzeichen darin.

```
Sub One
  [Name mit Leerzeichen]
End Sub
Sub [Name mit Leerzeichen]
  Print "Hier bin ich"
End Sub
```

In Visual Basic darf einer Subroutine ein optionales Schlüsselwort wie Public oder Private vorangehen. Seit OOo 2.0 können Sie eine Routine als Public oder Private definieren, aber die Routine ist immer Public, außer wenn vorher CompatibilityMode(True) gesetzt ist.

Deklarieren Sie eine Subroutine als Private, indem Sie dem Schlüsselwort Sub das Schlüsselwort Private voranstellen.

```
Private Sub PrivSub
  Print "In Private Sub"
  bbxx = 4
End Sub
```

Option Compatible reicht nicht, es muss CompatibilityMode(True) gesetzt sein, um den Private-Status zu ermöglichen,.

```
Sub TestPrivateSub
  CompatibilityMode(False) 'Nur nötig, wenn vorher CompatibilityMode(True) gesetzt war.
  Call PrivSub()           'Dieser Aufruf funktioniert.
  CompatibilityMode(True)  'Erforderlich, auch wenn Option Compatible gesetzt ist.
  Call PrivSub()           'Laufzeitfehler (wenn PrivSub in einem anderen Modul steht).
End Sub
```

Mit dem Schlüsselwort Function wird eine Funktion deklariert, die wie eine Variable ihren Rückgabetyp definieren kann. Ohne deklarierten Typ wird der Standardtyp Variant zurückgegeben. Sie können den Rückgabewert innerhalb des Funktionscodes an jeder Stelle und so oft Sie wollen zuweisen. Der letzte zugewiesene Wert wird zurückgegeben.

```
Sub test
  Print "Die Funktion gibt " & TestFunc & " zurück."
End Sub
Function TestFunc As String
  TestFunc = "Hallo"
End Function
```

3.5.1. Argumente

Eine Variable, die einer Routine übergeben wird, nennt man Argument. Argumente müssen deklariert werden. Für die Deklarierung von Argumenttypen gelten dieselben Regeln wie für Variablen.

Einer Routine kann optional ein Paar runder Klammern folgen, sowohl bei der Definition wie auch beim Aufruf. Bei einer Routine, die Argumente übernimmt, kann die Argumentliste optional in runden Klammern stehen. Die Argumentliste folgt dem Namen der Routine direkt in derselben Zeile. Zwischen Name und Argumentliste dürfen Leerzeichen stehen.

Listing 25. *Einfacher Argumentetest.*

```
Sub ExampleParamTest1()
  Call ParamTest1(2, "Zwei")
  Call ParamTest1 1, "Eins"
End Sub
Sub ParamTest1(i As Integer, s$)
  Print "Integer = " & i & " String = " & s$
End Sub
```

Übergabe als Referenz oder als Wert

Standardmässig werden Argumente als Referenz übergeben und nicht als Wert. Anders gesagt, wenn die aufgerufene Subroutine ein Argument verändert, sieht die aufrufende Subroutine die Änderung. Man kann mit dem Schlüsselwort ByVal dieses Verhalten dahin abwandeln, dass eine Kopie des Arguments (statt einer Referenz auf das Argument) versendet wird (s. Listing 26 und Bild 26).

Tipp Wenn der Wert von Konstanten, die als Referenz-Argumente übergeben werden, in der aufgerufenen Routine modifiziert wird, kann es zu unerwartetem Verhalten kommen. Der Wert kann innerhalb der aufgerufenen Routine beliebig zurückgehen. Ich hatte zum Beispiel eine Subroutine, die in einer Schleife ein Integer-Argument bis Null zurückzählen sollte. Das Argument kam nie auf Null.

Listing 26. *Argumente als Referenz und als Wert.*

```
Sub ExampleParamValAndRef()
  Dim i1%, i2%
  i1 = 1 : i2 = 1
  ParamValAndRef(i1, i2)
  MsgBox "Das als Referenz übergebene Argument war 1 and ist nun " & i1 & CHR$(10) &_
         "Das als Wert übergebene Argument war 1 and ist immer noch " & i2 & CHR$(10)
End Sub
Sub ParamValAndRef(iRef%, ByVal iVal)
  iRef = iRef + 1  ' Dies wird sich auf die aufrufende Routine auswirken.
  iVal = iVal - 1  ' Dies wird sich nicht auf die aufrufende Routine auswirken.
End Sub
```

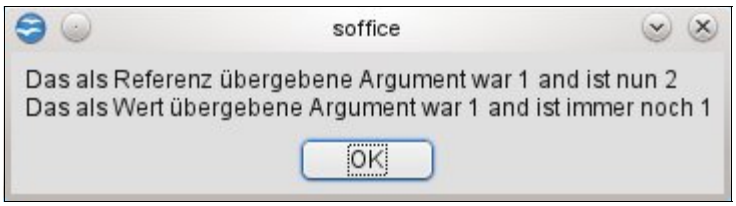


Bild 26. *Durch die Übergabe als Referenz können Änderungen an die aufrufende Routine zurückgegeben werden.*

Visual Basic unterstützt das optionale Schlüsselwort ByRef. Dieses Schlüsselwort wurde mit OOo 2.0 in OOo Basic eingeführt. Doch beachten Sie, dass die Übergabe als Referenz das Standardverhalten ist.

Optionale Argumente

Man kann Argumente als optional deklarieren, indem man ihnen das Schlüsselwort Optional voranstellt. Alle auf ein optionales Argument folgenden Argumente müssen auch optional sein. Mit der Funktion IsMissing finden Sie heraus, ob ein optionales Argument fehlt oder nicht.

Listing 27. *Optionale Argumente.*

```
REM Testaufrufe mit optionalen Argumenten.
REM Aufrufe mit Argumenten vom Typ Integer und Variant sollten
REM dieselben Ergebnisse zeigen.
REM Leider tun sie es nicht.
Sub ExampleArgOptional()
  Dim s$
  s = "Variant-Argumente () => "     & TestOpt()      & CHR$(10) &_
      "Integer-Argumente () => "     & TestOptI()     & CHR$(10) &_
      "-------------------------------------------" & CHR$(10) &_
      "Variant-Argumente (,,) => "   & TestOpt(,,)    & CHR$(10) &_
      "Integer-Argumente (,,) => "   & TestOptI(,,)   & CHR$(10) &_
      "-------------------------------------------" & CHR$(10) &_
      "Variant-Argumente (1) => "    & TestOpt(1)     & CHR$(10) &_
      "Integer-Argumente (1) => "    & TestOptI(1)    & CHR$(10) &_
      "-------------------------------------------" & CHR$(10) &_
      "Variant-Argumente (,2) => "   & TestOpt(,2)    & CHR$(10) &_
      "Integer-Argumente (,2) => "   & TestOptI(,2)   & CHR$(10) &_
      "-------------------------------------------" & CHR$(10) &_
      "Variant-Argumente (1,2) => "  & TestOpt(1,2)   & CHR$(10) &_
      "Integer-Argumente (1,2) => "  & TestOptI(1,2)  & CHR$(10) &_
      "-------------------------------------------" & CHR$(10) &_
      "Variant-Argumente (1,,3) => " & TestOpt(1,,3)  & CHR$(10) &_
      "Integer-Argumente (1,,3) => " & TestOptI(1,,3) & CHR$(10)
```

```
  MsgBox s, 0, "Optionale Argumente vom Typ Variant oder Integer"
End Sub

REM Gibt einen String zurück, der jedes Argument enthält.
REM Wenn das Argument fehlt, wird ein M an seine Stelle gesetzt.
Function TestOpt(Optional v1, Optional v2, Optional v3) As String
  TestOpt = "" & IIF(IsMissing(v1), "M", Str(v1)) &_
                 IIF(IsMissing(v2), "M", Str(v2)) &_
                 IIF(IsMissing(v3), "M", Str(v3))
End Function

REM Gibt einen String zurück, der jedes Argument enthält.
REM Wenn das Argument fehlt, wird ein M an seine Stelle gesetzt.
Function TestOptI(Optional i1%, Optional i2%, Optional i3%) As String
  TestOptI = "" & IIF(IsMissing(i1), "M", Str(i1)) &_
                  IIF(IsMissing(i2), "M", Str(i2)) &_
                  IIF(IsMissing(i3), "M", Str(i3))
End Function
```

Sie können jedes optionales Argument weglassen. Listing 27 zeigt zwei Funktionen, die optionale Argumente akzeptieren. Die Funktionen sind identisch bis auf die Argument-Typen. Jede Funktion gibt einen String zurück, der die Werte der Argumente aneinanderfügt. Weggelassene Argumente werden durch den Buchstaben „M“ repräsentiert. Obwohl die Rückgabestrings von TestOpt und TestOpt1 für dieselbe Argumentliste identisch sein sollten, sind sie es nicht (s. Bild 27). Dies ist ein Bug.

Tipp Die Funktion IsMissing gibt falsche Ergebnisse für Variablen, die nicht vom Typ Variant sind, wenn dem fehlenden Argument ein Komma folgt.

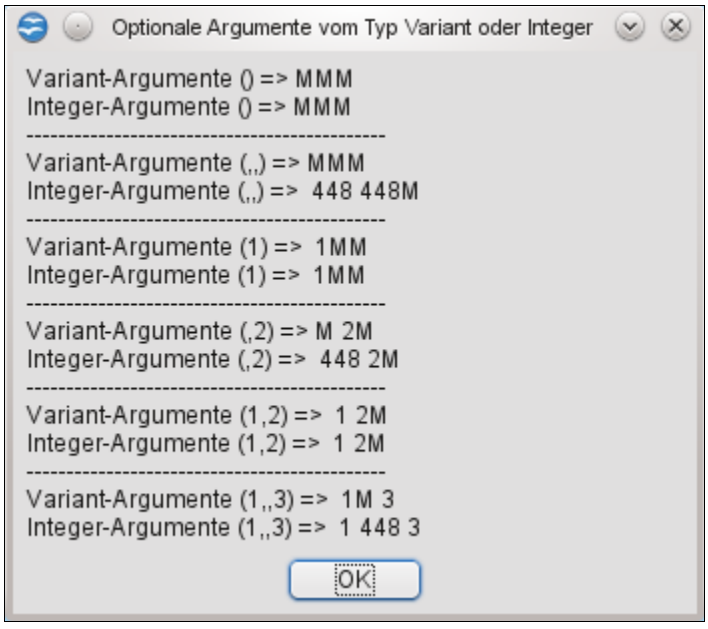


Bild 27. *In seltenen Fällen gibt es Fehler bei optionalen Argumenten, die nicht Typ Variant sind.*

Vorgegebene Argumentwerte

Seit OOo 2.0 sind vorgegebene Werte für weggelassene Argumente möglich. Das heißt, man kann einen Wert vorgeben für den Fall, dass ein optionales Argument fehlt. Vorgabewerte funktionieren aber nur, wenn „Option Compatible“ gesetzt ist.

```
Option Compatible
Sub DefaultExample(Optional n as Integer=100)
```

```
  REM If IsMissing(n) Then n = 100 'Das brauche ich nie mehr!
  Print n
End Sub
```

3.5.2. Rekursive Routinen

Eine rekursive Routine ruft sich selber auf. Betrachten Sie einmal die Berechnung der mathematischen Funktion Fakultät für positive Ganzzahlen. Die übliche Definition ist rekursiv.

Listing 28. *Rekursive Berechnung der Fakultät.*

```
Sub DoFactorial
  Print "Rekursive Fakultät = " & RecursiveFactorial(4)
  Print "Iterative Fakultät = " & IterativeFactorial(4)
End Sub

Function IterativeFactorial(ByVal n As Long) As Long
  Dim answer As Long
  answer = 1
  Do While n > 1
    answer = answer * n
    n = n - 1
  Loop
  IterativeFactorial = answer
End Function

' Dies funktioniert endlich seit Version 1.1
Function RecursiveFactorial(ByVal n As Long) As Long
  RecursiveFactorial = 1
  If n > 1 Then RecursiveFactorial = n * RecursiveFactorial(n-1)
End Function
```

Computer verwenden eine Datenstruktur, die Stapel heißt. Zuhause habe ich einen Stapel Bücher, die ich noch lesen will. Wenn ein neues Buch hinzukommt, lege ich es oben auf den Stapel. Wenn ich Zeit zum Lesen finde, nehme ich das Buch oben vom Stapel. So ähnlich funktioniert die Datenstruktur im Computer: ein Stapel ist ein Speicherbereich zur temporären Speicherung, in dem der zuletzt gespeicherte Wert der erste ist, der zurückgeholt wird. Stapel werden üblicherweise eingesetzt, wenn ein Computer eine Routine aufruft und Argumente übergibt. Im Folgenden eine typische Prozedur:

1. Die aktuelle Programmposition wird auf den Stapel geschoben.
2. Jedes Argument wird auf den Stapel geschoben.
3. Die gewünschte Funktion oder Subroutine wird aufgerufen.
4. Die aufgerufene Routine verwendet die Argumente vom Stapel.
5. Die aufgerufene Routine nutzt häufig den Stapel zur Speicherung der eigenen Variablen.
6. Die aufgerufene Routine entfernt die Argumente vom Stapel.
7. Die aufgerufene Routine entfernt und sichert die Position der aufrufenden Routine vom Stapel.
8. Wenn die aufgerufene Routine eine Funktion ist, wird der Rückgabewert auf den Stapel geschoben.
9. Die aufgerufene Routine kehrt über die vom Stapel gesicherte Position zur aufrufenden Routine zurück.
10. Wenn die aufgerufene Routine eine Funktion ist, wird der Rückgabewert vom Stapel geholt.

Obwohl eine Reihe von Optimierungen wirken, gibt es immer einen Überhang im Zusammenhang mit dem Aufruf von Subroutinen und Funktionen. Der Überhang besteht in der Verarbeitungszeit und im benötigten Speicherplatz. Die rekursive Fakultät-Version ruft sich kontinuierlich selber auf. Bei der Berechnung der Fakultät von vier (4!) enthält der Stapel zu einem Zeitpunkt gleichzeitig die Informationen für die Aufrufe für 4, 3, 2 und 1. Für manche Funktionen – zum Beispiel für die Fibonacci-Reihe – dürfte sich dieses Verhalten verbieten und stattdessen einen nicht-rekursiven Algorithmus erfordern.

3.6. Gültigkeitsbereich von Variablen, Subroutinen und Funktionen

Bei dem Konzept des Gültigkeitsbereichs geht es um die Lebenszeit und Sichtbarkeit von Variablen, Subroutinen und Funktionen in OOo Basic. Der Gültigkeitsbereich hängt ab vom Ort der Deklarierung und den Schlüsselwörtern Public, Private, Static und Global. Dim ist äquivalent zu Private, aber Variablen sind nur Private, wenn CompatibilityMode(True) gesetzt ist.

3.6.1. Lokale Variablen, in einer Subroutine oder Funktion deklariert

Variablen, die innerhalb einer Subroutine oder Funktion deklariert werden, heißen lokale Variablen. Häufig wird auch gesagt, dass eine Variable zu der Routine gehört, in der sie deklariert wird.

Sie können eine Variable mit Hilfe des Schlüsselworts Dim innerhalb einer Subroutine oder Funktion deklarieren. Solche Variablen sind nur innerhalb der Routine sichtbar. Es ist nicht möglich, auf eine in einer Routine deklarierte Variable von außerhalb der Routine direkt zuzugreifen. Es ist jedoch möglich, von innerhalb einer Routine auf eine Variable zuzugreifen, die außerhalb jeglicher Routine deklariert wurde – zum Beispiel im Kopf eines Moduls. Wenn innerhalb einer Routine der Name einer Variablen oder einer Routine erscheint, sucht OOo Basic in folgender Reihenfolge nach der Variablen oder Routine: aktuelle Routine, Modul, Bibliothek und andere offene Bibliotheken. Mit anderen Worten, Basic startet innen und arbeitet sich nach außen vor.

In einer Routine definierte Variablen werden jedesmal geschaffen und initialisiert, wenn die Routine gestartet wird. Jedes Mal, wenn die Routine verlassen wird, werden die Variablen entfernt, weil die Routine nicht mehr existiert. Wenn allerdings eine Routine verlassen wird, um eine weitere Routine aufzurufen, ist das kein Anlass zur Reinitialisierung der Variablen.

Mit dem Schlüsselwort Static ändern Sie den Zeitpunkt, zu dem eine Variable erstellt und entfernt wird, auf den Moment, in dem das Makro startet beziehungsweise endet. Auch wenn die Variable immer noch nur innerhalb der aktuellen Routine sichtbar ist, wird sie nur einmal initialisiert, nämlich wenn das Makro gestartet wird, und sie behält ihren Wert über mehrfache Aufrufe dieser Routine hinweg. Sagen wir einmal, Sie starten, wenn noch kein Makro läuft. Wenn nun eine Subroutine oder eine Funktion, die darin enthaltene statische Variablen nutzt, zum ersten Mal aufgerufen wird, erhalten die Variablen ihre Initialwerte gemäß ihren Typen. Die statischen Variablen behalten ihren Wert zwischen den Aufrufen, solange das Makro als Ganzes nicht beendet ist. Die Syntax des Schlüsselworts Static ist dieselbe wie bei dem Schlüsselwort Dim, und es ist nur innerhalb einer Subroutine oder Funktion gültig. Listing 29 ruft eine Routine auf, die eine statische Variable verwendet.

Listing 29. *Beispiel für Static.*

```
Sub ExampleStatic
  ExampleStaticWorker()
  ExampleStaticWorker()
End Sub

Sub ExampleStaticWorker
  Static iStatic1 As Integer
  Dim iNonStatic As Integer

  iNonStatic = iNonStatic + 1
  iStatic1 = iStatic1 + 1
```

```
    Msgbox "iNonStatic = " & iNonStatic & CHR$(10) &_
           "iStatic1 = " & iStatic1
End Sub
```

3.6.2. In einem Modul definierte Variablen

Die Anweisungen Dim, Global, Public oder Private werden für die Variablendeklarierung im Kopf eines Moduls verwendet. Global, Public und Private haben dieselbe Syntax wie Dim, können aber keine Variablen innerhalb einer Subroutine oder Funktion deklarieren. Jede Variablenart hat eine andere Lebensdauer, wie Tabelle 13 zeigt.

Die Schlüsselwörter Static, Public, Private und Global sind keine Modifizierer für das Schlüsselwort Dim, sie werden anstelle von Dim verwendet.

Obwohl es manchmal notwendig ist, eine Variable im Modulkopf zu deklarieren, sollten Sie aber nach Möglichkeit davon absehen. Im Kopf deklarierte Variablen sind auch in anderen Modulen sichtbar, wo sie vielleicht nicht erwartet werden. Es ist nicht einfach herauszufinden, warum sich der Kompilierer beschwert, dass eine Variable schon definiert sei, wenn sie in einer anderen Bibliothek oder einem anderen Modul deklariert ist. Schlimmer noch könnten zwei aktive Bibliotheken die Arbeit wegen Namenskonflikten ganz einstellen.

Tabelle 13. *Lebensdauer einer Variablen, die im Kopf eines Moduls definiert ist.*

Schlüsselwort	Initialisiert	Stirbt	Gültigkeitsbereich
Global	Kompilierung	Kompilierung	Alle Module und Bibliotheken.
Public	Makrostart	Makroende	Bibliothekscontainer, in der die Deklarierung erfolgt
Dim	Makrostart	Makroende	Bibliothekscontainer, in der die Deklarierung erfolgt
Private	Makrostart	Makroende	Modul, in der die Deklarierung erfolgt

Global

Eine als Global deklarierte Variable ist für jedes Modul in jeder Bibliothek erreichbar. Die Bibliothek, die die globale Variable enthält, muss geladen sein, damit die Variable sichtbar ist.

Wenn eine Bibliothek geladen wird, wird sie automatisch kompiliert und gebrauchsfertig gemacht. Zu diesem Zeitpunkt wird die globale Variable initialisiert. Änderungen an einer globalen Variablen werden von jedem Modul gesehen und bleiben erhalten, auch nachdem das Makro beendet ist. Globale Variablen werden zurückgesetzt, wenn die besitzende Bibliothek kompiliert wird. OpenOffice.org zu schließen und neu zu starten bewirkt die Neukompilierung aller Bibliotheken und die Initialisierung aller globalen Variablen. Änderungen am Quelltext des Moduls, das die globale Definition enthält, erzwingt gleichermaßen die Neukompilierung des Moduls.

```
Global iNumberOfTimesRun
```

Als global deklarierte Variablen ähneln denen, die als Static deklariert sind. Der Unterschied liegt darin, dass Static nur für lokale Variablen gilt und Global nur für Variablen, die im Kopf deklariert werden.

Public

Mit Public deklarierte Variablen sind für alle Module des Bibliothekscontainers sichtbar, in dem sie deklariert sind. Außerhalb dieses Bibliothekscontainers sind Public-Variablen nicht sichtbar. Public-Variablen werden bei jedem Aufruf eines Makros initialisiert.

Eine Anwendungsbibliothek ist eine Bibliothek, die im Bibliothekscontainer „OpenOffice.org“ aufgeführt ist. Sie ist verfügbar, wenn OOo läuft, ist in ihrem eigenen Verzeichnis gespeichert, und jedes Dokument kann sie sehen. Bibliotheken auf Dokumentbasis sind in OOo-Dokumenten gespeichert.

Die Bibliotheken werden als Teil des Dokuments gespeichert und sind von außerhalb des Dokuments nicht sichtbar.

Public-Variablen, die in einer Anwendungsbibliothek deklariert werden, sind in jeder OOo-dokumentbasierten Bibliothek sichtbar. Public-Variablen, die in einer Dokumentbibliothek deklariert werden, sind in Anwendungsbibliotheken nicht sichtbar. Mit der Deklarierung einer Public-Variablen in einer Dokumentbibliothek wird eine in einer Anwendungsbibliothek deklarierte Public-Variable effektiv verborgen. Schlicht und einfach (s. Tabelle 14), wenn Sie eine Public-Variable in einem Dokument deklarieren, ist sie nur im Dokument sichtbar und wird eine Public-Variable verbergen, die mit demselben Namen außerhalb des Dokuments deklariert wurde. Eine in der Anwendung deklarierte Public-Variable ist überall sichtabr – falls nicht eine Variablendeklarierung mit eher lokalem Gültigkeitsbereich die Oberhand über die Deklarierung mit eher globalem Gültigkeitsbereich gewinnt.

```
Public oDialog As Object
```

Tabelle 14. *Der Gültigkeitsbereich einer Public-Variablen hängt davon ab, wo sie deklariert wird.*

Ort der Deklarierung	Gültigkeitsbereich
Anwendung	Überall sichtbar.
Dokument	Sichtbar nur im Dokument der Deklarierung.
Anwendung und Dokument	Makros im Dokument können die Variable der Anwendungsebene nicht sehen.

Private oder Dim

Eine mit Private oder Dim in einem Modul deklarierte Variable ist in anderen Modulen nicht sichtbar. Private-Variablen werden wie Public-Variablen bei jedem Start eines Makros initialisiert. Ein und derselbe Variablenname kann in zwei verschiedenen Modulen als jeweils eigener Name verwendet werden, wenn die Variable als Private deklariert wird.

```
Private oDialog As Variant
```

- Die Deklarierung einer Variablen mit Dim ist äquivalent zur Deklarierung mit Private.
- Private-Variablen sind jedoch nur privat nach der Anweisung CompatibilityMode(True).
- Option Compatible hat keinen Effekt auf Private-Variablen.

Eine Private-Variable ist außerhalb des deklarierenden Moduls sichtbar, es sei denn, CompatibilityMode(True) ist gesetzt. Schauen Sie selbst, erstellen Sie zwei Module – Modul1 und Modul2 – in derselben Bibliothek. Fügen Sie in Modul1 die Deklarierung „Private priv_var As Integer“ ein. Makros in Modul2 können auf die Variable „priv_var“ zugreifen. Sogar wenn Modul2 in einer anderen Bibliothek desselben Dokuments liegt, ist die Variable „priv_var“ sichtbar und nutzbar. Wenn jedoch CompatibilityMode(True) gesetzt ist, dann ist die Private-Variable nicht mehr außerhalb des deklarierenden Moduls sichtbar.

Deklarieren Sie in Modul1 die Variable „Private priv_var As Double“. In Modul2 wird eine Variable mit demselben Namen deklariert, aber als Integer-Variable. Jedes Modul sieht seine eigene private Variable. Wenn man diese Variablen nicht als Private, sondern als Public deklariert, tritt eine unschöne Situation ein: nur eine dieser Variablen ist sichtbar und nutzbar, aber man weiß nicht welche, außer man führt einen Test durch. Weisen Sie der Variablen 4.7 zu und schauen Sie, ob es Integer oder Double wird.

3.7. Operatoren

Ein Operator ist ein Symbol, das eine mathematische oder logische Operation kennzeichnet oder durchführt. Ein Operator gibt wie eine Funktion ein Resultat zurück. Zum Beispiel addiert der Operator + zwei Zahlen. Die Argumente des Operators heißen Operanden. Operatoren haben Prioritäten.

Ein Operator mit der Priorität 1 steht sozusagen auf der höchsten Prioritätsstufe. Schließlich ist es die Nummer 1.

Tipp Beim Druck mathematischer Gleichungen wird das Minuszeichen durch das Unicodezeichen U+2212 (−) dargestellt. In OOo Basic muss stattdessen das ASCII-Zeichen 45 (-) verwendet werden.

In OOo Basic (s. Tabelle 15) werden Operatoren von links nach rechts ausgewertet, mit der Einschränkung, dass ein Operator mit einer höheren Priorität vor einem Operator mit einer niedrigeren Priorität wirkt. 1 + 2 * 3 ergibt 7, weil die Multiplikation eine höhere Priorität hat als die Addition. Durch die Verwendung runder Klammern können Sie die Reihenfolge ändern. Zum Beispiel ergibt (1 + 2) * 3 den Wert 9, weil der Ausdruck innerhalb der runden Klammern zuerst ausgewertet wird.

Tabelle 15. *Operatoren in OpenOffice.org Basic.*

Priorität	Operator	Typ	Beschreibung
1	NOT	Unär	Logisches oder bitweises NOT
1	-	Unär	Minus als Vorzeichen, Negation
1	+	Unär	Plus als Vorzeichen
2	^	Binär	Numerische Potenzierung. In der Mathematik hätte die Potenzierung eine höhere Priorität als die Negation.
3	*	Binär	Numerische Multiplikation
3	/	Binär	Numerische Division
4	MOD	Binär	Numerischer Rest nach Division
5	\	Binär	Ganzzahlige Division
6	-	Binär	Numerische Subtraktion
6	+	Binär	Numerische Addition und String-Verkettung
7	&	Binär	String-Verkettung
8	IS	Binär	Referenzieren beide Operandent dasselbe Objekt?
8	=	Binär	Gleich
8	<	Binär	Kleiner als
8	>	Binär	Größer als
8	<=	Binär	Kleiner als oder gleich
8	>=	Binär	Größer als oder gleich
8	<>	Binär	Ungleich
9	AND	Binär	Bitweises UND für Zahlen, logisches UND für Boolean
9	OR	Binär	Bitweises ODER für Zahlen, logisches ODER für Boolean
9	XOR	Binär	Exklusives ODER, bitweise für Zahlen, logisch für Boolean
9	EQV	Binär	Äquivalenz, bitweise für Zahlen, logisch für Boolean
9	IMP	Binär	Implikation, bitweise für Zahlen, logisch für Boolean

Tipp OOo-Prioritäten richten sich nicht unbedingt nach dem mathematischen Standard. Zum Beispiel sollte die Negation eine niedrigere Priorität haben als die Potenzierung: -1^2 sollte -1 ergeben, nicht 1.

Visual Basic hat andere Prioritäten für Operatoren – zum Beispiel ist die Reihenfolge der numerischen Potenzierung und Negation umgekehrt, wie auch der ganzzahligen Division und des Rests nach Division.

Das Wort „Binär“ bezeichnet etwas, das auf zwei Sachen basiert. „Unär“ bezeichnet etwas, das auf nur einer Sache basiert. Ein binärer Operator, nicht zu verwechseln mit einer binären Zahl, steht zwischen zwei Operanden. Zum Beispiel 1+2: da verwendet der Additionsoperator zwei Operanden. In OOo Basic werden binäre Operatoren immer von links nach rechts ausgewertet, nach Maßgabe der Operatorenprioritäten. Ein unärer Operator benötigt nur einen Operanden, und zwar direkt rechts neben dem Operator, zum Beispiel – (1+3). Notwendigerweise wird eine Reihe von unären Operatoren von rechts nach links ausgewertet. Zum Beispiel muss in + - (1+3) der rechtsstehende Negationsoperator zuerst ausgewertet werden, so dass sich der Wert -4 ergibt.

3.7.1. Mathematische und String-Operatoren

Mathematische Operatoren können mit allen numerischen Datentypen verwendet werden. Wenn Operanden unterschiedlichen Typs gemischt vorkommen, wird konvertiert, um Genauigkeitsverluste zu minimieren. Zum Beispiel bewirkt 1 + 3.443 die Konversion zu einer Fließkommazahl und nicht zu einer Ganzzahl. Wenn der erste Operand eine Zahl ist und der zweite ein String, wird der String zu einer Zahl konvertiert. Wenn der String nun keine gültige Zahl enthält, wird Null zurückgegeben, ohne Fehlermeldung. Einen String direkt einer numerischen Variablen zuzuweisen, ergibt die Zuweisung von Null, wiederum ohne Fehlermeldung.

Listing 30. *Strings werden automatisch zu Zahlen konvertiert, wenn es nötig ist.*

```
  Dim i As Integer
  i = "abc"         'Die Zuweisung eines Strings ohne Zahl ergibt Null, keine
Fehlermeldung
  Print i           '0
  i = "3abc"        'Zuweisung von 3, automatisch konvertiert, so gut es geht
  Print i           '3
  Print 4 + "abc"   '4
```

OOo Basic versucht die automatische Typkonvertierung. Es gibt keine Fehlermeldungen, wenn ein String verwendet wird, wo eine Zahl benötigt wird. Dazu mehr an späterer Stelle.

Unäres Plus (+) und Minus (-)

OOo Basic erlaubt Leerzeichen zwischen unären Operatoren und dem Operanden (s. Tabelle 8). Unäre Operatoren haben auch die höchste Priorität und werden von rechts nach links ausgewertet. Ein Plus als Vorzeichen ist sicherlich nutzlos – es unterstreicht, dass eine Konstante nicht negativ ist, wird aber ansonsten schlicht ignoriert. Ein Minus als Vorzeichen steht für numerische Negation.

Potenzierung (^)

Die numerische Potenzierung unterstützt ganzzahlige und Fließkommaexponenten. Der Potenzierungsoperator kann nur dann mit einer negativen Zahl verwendet werden, wenn der Exponent ganzzahlig ist.

```
result = number^exponent
```

Ein positiver ganzzahliger Exponent arbeitet nach einem einfachen Prinzip. Die Zahl wird mit sich selbst Exponent-mal multipliziert, zum Beispiel 2^4 = 2 * 2 * 2 * 2.

1. OOo folgt nicht unbedingt den mathematischen Standardregeln für die Potenzierung:
2. Potenzierung hat eine niedrigere Priorität als Negation, so wird -1^2 fälschlicherweise zu 1 ausgewertet. OOo Basic wertet mehrere Exponenten (2^3^4) von links nach rechts aus ((2^3)^4), wohingegen der mathematische Prioritätenstandard von rechts nach links geht (2^(3^4)).

Listing 31. *Demonstration der Potenzierung.*

```
Sub ExampleExponent
  Dim s$
  s = "2^3 = " & 2^3                        REM 2*2*2 = 8
  s = s & CHR$(10) & "3^2 = " & 3^2         REM 3 *3 = 9
  s = s & CHR$(10) & "-3^2 = " & -3^2       REM (-3) * (-3) = 9
  s = s & CHR$(10) & "2^3^2 = " & 2^3^2     REM 2^3^2 = 8^2 = 64
  s = s & CHR$(10) & "4^0.5 = " & 4^.5      REM 2
  s = s & CHR$(10) & "4^-0.5 = " & 4^-.5    REM .5
  s = s & CHR$(10) & "-1^2 = " & -1^2       REM 1
  s = s & CHR$(10) & "-(1^2) = " & -(1^2)   REM -1
  MsgBox s
End Sub
```

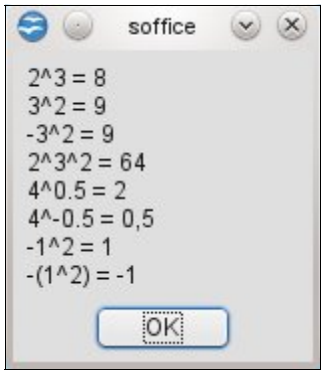


Bild 28. *Der Gebrauch des Potenzierungsoperator.*

Multiplikation (*) und Division (/)

Multiplikation und Division haben dieselbe Priorität.

Listing 32. *Demonstration der Multiplikation und der Division.*

```
Sub ExampleMultDiv
  Print "2*3 = " & 2*3        REM 6
  Print "4/2.0 = " & 4/2.0    REM 2
  Print "-3/2 = " & -3/2      REM -1.5
  Print "4*3/2 = " & 4*3/2    REM 6
End Sub
```

Rest nach Division (MOD)

Der Operator MOD wird auch „Rest nach Division“ genannt. Zum Beispiel hat 5 MOD 2 das Ergebnis 1, weil 5 dividiert durch 2 gleich 2 ist mit einem Rest von 1. Alle Operanden werden zu Ganzzahlen gerundet, bevor die Rechnung durchgeführt wird.

Listing 33. *Definition des Operators MOD für die ganzzahligen Operanden x and y.*

```
x MOD y = x - (y * (x\y))
```

Listing 34. *Demonstration des Operators mod.*

```
REM x MOD y kann auch so geschrieben werden:
REM CInt(x) - (CInt(y) * (CInt(x)\CInt(y)))
REM CInt ist nötig, weil die Zahlen gerundet werden müssen,
REM bevor die Berechnungen ausgeführt werden.
Sub ExampleMOD
  Dim x(), y(), s$, i%
  x() = Array (4, 15, 6, 6.4, 6.5, -15, 15, -15)
```

```
  y() = Array (15, 6, 3, 3,   3,     8, -8,  -8)
  For i = LBound(x()) To UBound(x())
    s = s & x(i) & " MOD " & y(i) & " = " & (x(i) MOD y(i)) & CHR$(10)
  Next
  MsgBox s, 0, "Operator MOD"
End Sub
```

Bild 29. *Der Gebrauch des Operators MOD.*

Tipp Die Operanden für MOD werden vor der Division zu ganzzahligen Werten gerundet.

Ganzzahlige Division (\)

Bei einer normalen Division werden Operanden vom Typ Double erwartet, und das Ergebnis ist wiederum ein Double. Zum Beispiel ergibt 7.0 / 4.0 den Wert 1.75. Ganzzahlige Division andererseits erwartet die Division zweier Integer-Werte, und das Ergebnis wird auch ein Integer sein. Im Beispiel 7.2 \ 4.3 werden die Operanden zu 7 \ 4 konvertiert, und als Ergebnis kommt 1 heraus. Konstante numerische Operanden des Operators der ganzzahligen Division werden zu Integer-Werten beschnitten, bevor die ganzzahlige Division erfolgt. Das Resultat ist ein beschnittener, nicht gerundeter Wert. Listing 35 zeigt den Unterschied zwischen einer ganzzahligen und normalen Division.

Listing 35. *Demonstration einer ganzzahligen Division.*

```
Sub ExampleIntDiv
  Dim f As Double
  Dim s$
  f = 5.9
  s = "5/2 = " & 5/2        REM 2.5
  s = s & CHR$(10) & "5\2 = " & 5\2        REM 2
  s = s & CHR$(10) & "5/3 = " & 5/3        REM 1.666666667
  s = s & CHR$(10) & "5\3 = " & 5\3        REM 1
  s = s & CHR$(10) & "5/4 = " & 5/4        REM 1.25
  s = s & CHR$(10) & "5\4 = " & 5\4        REM 1
  s = s & CHR$(10) & "-5/2 = " & -5/2      REM -2.5
  s = s & CHR$(10) & "-5\2 = " & -5\2      REM -2
  s = s & CHR$(10) & "-5/3 = " & -5/3      REM -1.666666667
  s = s & CHR$(10) & "-5\3 = " & -5\3      REM -1
  s = s & CHR$(10) & "-5/4 = " & -5/4      REM -1.25
  s = s & CHR$(10) & "-5\4 = " & -5\4      REM -1
  s = s & CHR$(10) & "17/6 = "   & 17/6    REM 2.83333333333333
  s = s & CHR$(10) & "17\6 = "   & 17\6    REM 2
  s = s & CHR$(10) & "17/5.9 = " & 17/5.9  REM 2.88135593220339
  s = s & CHR$(10) & "17\5 = " & 17\5      REM 3
  s = s & CHR$(10) & "17\5.9 = " & 17\5.9  REM 3, weil 5.9 zu 5 beschnitten wurde.
  s = s & CHR$(10) & "17\f = " & 17\f      REM 2, weil f zu 6 aufgerundet wurde.
  s = s & CHR$(10) & "17\(11.9/2) = " & 17\(11.9/2) REM 3, weil 11.9/2 zu 5
                                                    REM beschnitten wurde.
```

```
    MsgBox s
End Sub
```

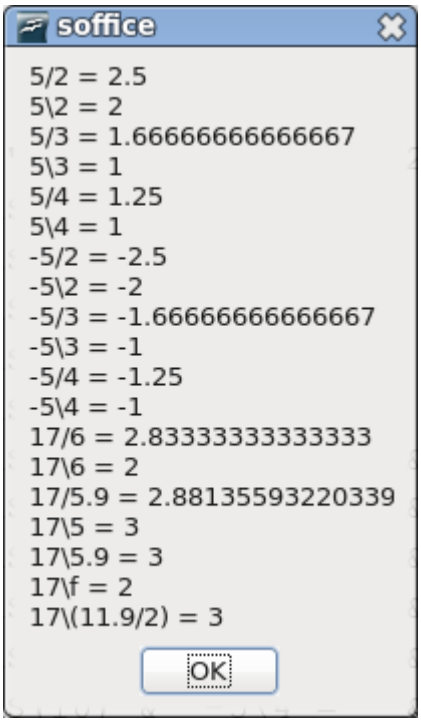


Bild 30. *Ganzzahlige Division.*

Bei der ganzzahligen Division werden konstante numerische Operanden zu Integer-Werten beschnitten, bevor die Division erfolgt. Numerische Variablen und Resultate von Operationen werden jedoch vor der Division zu Integer-Werten gerundet.

Addition (+), Subtraktion (-) und String-Verkettung (& and +)

Addition und Subtraktion haben dieselbe Priorität, die höher ist als der Operator zur String-Verkettung. Beim Addieren numerischer Werte ist Vorsicht geboten, denn der Operator + kann auch zur String-Verkettung genutzt werden. Wenn der erste Operand des Operators + eine Zahl ist und der zweite ein String, wird der String zu einer Zahl konvertiert. Wenn der erste Operand des Operators + ein String ist und der zweite eine Zahl, wird die Zahl in einen String konvertiert.

```
Print 123 + "3"      REM 126  (Numerisch)
Print "123" + 3      REM 1233 (String)
```

Der Verkettungsoperator versucht, die Operanden zu Strings zu konvertieren, wenn wenigstens ein Operator ein String ist.

```
Print 123 & "3"     REM 1233 (String)
Print "123" & 3     REM 1233 (String)
Print 123 & 3       REM Um zu funktionieren, muss wenigsten ein String dabei sein!
```

String-Manipulationen und numerische Operationen miteinander zu vermengen, kann zu seltsamen Ergebnissen führen, vor allem, weil die String-Verkettung mit dem Operator & geringere Priorität hat als der Operator +.

```
Print 123 + "3" & 4  '1264  Erst die Addition, dann die Stringkonvertierung
Print 123 & "3" + 4  '12334 Erst die Addition, aber der erste Operand ist ein String
Print 123 & 3 + "4"  '1237  Erst die Addition, aber der erste Operand ist ein Integer
```

3.7.2. Logische und bitweise Operatoren

Jeder logische Operator stellt eine einfache Frage und gibt eine Antwort zurück, die True oder False ist. Ist es zum Beispiel wahr, dass (Sie Geld haben) AND (Sie mein Buch kaufen wollen)?. Derartige

Operationen sind einfach und werden häufig in OOo Basic benötigt. Erheblich seltener genutzt, und hier nur der Vollständigkeit halber zur Beglückung der Computerprofis erwähnt, sind die bitweisen Operatoren. Bitweise arbeitende Operatoren sind nicht schwierig, aber wenn Sie sie nicht verstehen, so wird Ihr Umgang mit OOo Basic höchstwahrscheinlich gar nicht beeinträchtigt.

Ein logischer Operator wird im allgemeinen mit den Werten True und False in Verbindung gebracht. In OOo Basic führen logische Operatoren aber auch bitweise Operationen auf Integer-Werte aus. Das bedeutet, dass jedes Bit des ersten Operanden mit dem entsprechenden Bit des zweiten Operanden verglichen wird, um dann des entsprechende Bit im Ergebnis zu setzen. Zum Beispiel wird bei den binären Operanden 01 und 10 von der 01 die 0 und von der 10 die 1 für das erste Bit des Resultats verwendet.

Ungewöhnlich bei den logischen und bitweisen binären Operatoren in OOo Basic ist, dass die Priorität dieselbe ist. In anderen Sprachen hat AND normalerweise größere Priorität als OR.

Tabelle 16 listet die logischen und bitweisen Operatoren auf, die von OOo unterstützt werden. True und False stehen für logische Werte, 0 und 1 für Bitwerte.

Tabelle 16. *Wahrheitstabelle für logische und bitweise Operatoren.*

x	y	x AND y	x OR y	x XOR y	x EQV y	x IMP y
True	True	True	True	False	True	True
True	False	False	True	True	False	False
False	True	False	True	True	False	True
False	False	False	False	False	True	True
1100	1010	1000	1110	0110	1001	1011

Intern werden die Operanden der logischen Operatoren zum Typ Long konvertiert. Dabei kommt es bei einem Fließkomma-Operanden zu dem unerwarteten Nebeneffekt, dass auch er zu Long konvertiert wird, was einen numerischen Überlauf zur Folge haben kann. Die Konversion von einer Fließkommazahl zu einer Ganzzahl vom Typ Long geht mit Runden, nicht mit Beschneiden des Wertes einher. Das funktioniert, weil der Wert für True -1 ist und der für False 0. Doch mit zwei Boolean-Operanden ist der Rückgabewert manchmal noch vom Typ Long.

Listing 36. *Logische Operanden sind ganzzahlig vom Typ Long.*

```
Sub LogicalOperandsAreLongs
  Dim v, b1 As Boolean, b2 As Boolean
  b1 = True : b2 = False
  v = (b1 OR b2)
  Print TypeName(v)    REM Long, weil Operanden zu Long konvertiert werden.
  Print v              REM -1,   weil der Rückgabetyp Long ist.
  Print (b2 OR "-1")   REM -1,   weil "-1" zu Long konvertiert wird.
End Sub
```

Bei manchen logischen Ausdrücken müssen nicht alle Operanden ausgewertet werden. Zum Beispiel wird der Ausdruck (False AND True) schon beim Operator AND nach dem ersten Operanden als False erkannt. So etwas kennt man als Kurzschluss-Auswertung. Leider leider gibt es das nicht in OOo Basic, es werden alle Operanden ausgewertet.

Tipp OOo Basic unterstützt keine Kurzschluss-Auswertung, somit bewirkt (x <> 0 AND y/x > 3) den Fehler „Division durch Null“, falls x gleich Null ist.

Die bitweisen Operatoren werden alle auf dieselbe Art veranschaulicht. Zwei Arrays werden mit booleschen Werten gefüllt, dazu kommen zwei Integer-Variablen mit ganzzahligen Werten.

```
xi% = 12 : yi% = 10
x() = Array(True, True, False, False)
y() = Array(True, False, True, False)
```

Die dezimale Zahl 12 wird auf der Basis 2 als 1100 repräsentiert, in Übereinstimmung mit den Werten in x(). Die dezimale Zahl 10 wird auf der Basis 2 als 1010 repräsentiert, in Übereinstimmung mit den Werten in y(). Der Operator wird dann nacheinander angewendet auf „x(0) op y(0)“, „x(1) op y(1)“, „x(2) op y(2)“, „x(3) op y(3)“ und „xi op yi“. Das Result wird in einer Meldung ausgegeben. Die Ganzzahlen werden auf der Basis 2 dargestellt, um die bitweise Operation zu verdeutlichen. Listing 37 demonstriert, wie ein Integer-Wert in eine Reihe von Bits konvertiert wird. Hierbei werden viele Techniken angewendet, die später in diesem Kapitel behandelt werden.

Listing 37. *Konversion eines Integer-Wertes in eine Binärzahl.*

```
Sub TestIntoToBinary
  Dim s$
  Dim n%
  Dim x%
  x = InputBox("Bitte geben Sie eine ganze Zahl ein")
  If x <> 0 Then
    n = Log(Abs(x)) / Log(2) + 1
    If (x < 0) Then
      n = n + 4
    End If
  Else
    n = 1
  End If
  print "s = " & IntToBinaryString(x, n)
End Sub

REM Konversion eines Integer-Wertes in einen String der Bits
REM x ist die zu konvertierende Ganzzahl
REM n ist die Anzahl der zu konvertierenden Bits
REM Es wäre leichter, wenn ich das niedrigste Bit herausschieben (shift)
REM und gleichzeitig das Vorzeichenbit der Zahl erhalten könnte, aber das geht nicht.
REM Ich bilde es dadurch nach, dass ich durch zwei dividiere, was aber bei negativen
REM Zahlen fehlschlägt. Um dieses Problem zu vermeiden, drehe ich alle Bits um,
REM wenn die Zahl negativ ist, und mache dadurch aus ihr eine positive Zahl.
REM Dann baue ich ein invertiertes Resultat.
Function IntToBinaryString(ByVal x%, ByVal n%) As String
  Dim b1$                 'Bitwert 1 für positive Zahlen
  Dim b0$                 'Bitwert 0 für positive Zahlen
  If (x >= 0) Then        'Nicht negativ, also funktioniert der Algorithmus
    b1 = "1" : b0 = "0"   'mit den Standard-Bitwerten.
  Else                    'Eine negative Zahl, also
    x = NOT x             'werden alle Bits umgedreht,
    b1 = "0" : b0 = "1"   'auch die Standard-Bitwerte.
  End If
  Dim s$                  'Die Bits werden im String s$ aneinandergereiht
  Do While n > 0          'n ist die zurückzugebende Anzahl an Bits
    If (x AND 1) = 1 Then 'AND mit 1, um den Wert von Bit 1 herauszufinden
      s = b1$ & s         'Bit 1 ist gesetzt, es wird 1 hinzugefügt (0 bei negativem x)
    Else
      s = b0$ & s         'Bit 1 ist frei, also wird 0 hinzugefügt (1 bei negativem x)
    End If
    x = x\2               'Ganzzahldivision durch 2
```

```
    n = n - 1              'Vermindere n um 1 für jedes erledigte Bit.
  Loop                     'Zurück zum Start der While-Schleife
  IntToBinaryString = s    'Zuweisung des Rückgabewertes
End Function
```

AND

Operiert als logisches AND auf booleschen Werten und als bitweises AND auf numerischen Werten. Nehmen Sie den Satz „Sie können ins Kino gehen, wenn Sie Geld haben AND wenn Sie über ein Fahrzeug verfügen“. Beide Bedingungen müssen wahr sein, bevor Sie ins Kino gehen können. Wenn beide Operanden True sind, ist das Ergebnis True, andernfalls ist das Ergebnis False.

Listing 38. *Operator AND.*

```
Sub ExampleOpAND
  Dim s$, x(), y(), i%, xi%, yi%
  xi% = 12 : yi% = 10
  x() = Array(True, True, False, False)
  y() = Array(True, False, True, False)
  For i = LBound(x()) To UBound(x())
    s = s & x(i) & " AND " & y(i) & " = " & CBool(x(i) AND y(i)) & CHR$(10)
  Next
  s = s & IntToBinaryString(xi, 4) & " AND " & IntToBinaryString(yi, 4) &_
     " = " & IntToBinaryString(xi AND yi, 4) & CHR$(10)
  MsgBox s, 0, "Beispiel für Operator AND"
End Sub
```

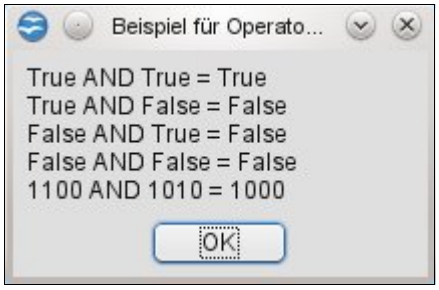


Bild 31. *Gebrauch des Operators AND.*

OR

Operiert als logisches OR auf booleschen Werten und als bitweises OR auf numerischen Werten. Nehmen Sie den Satz „Sie können den Kauf tätigen, wenn Sie das Geld haben OR wenn Ihr Freund das Geld hat“. Es spielt keine Rolle, wer das Geld hat. Wenn einer von beiden Operanden True ist, ist das Ergebnis True, andernfalls ist das Ergebnis False.

Listing 39. *Operator OR.*

```
Sub ExampleOpOR
  Dim s$, x(), y(), i%, xi%, yi%
  xi% = 12 : yi% = 10
  x() = Array(True, True, False, False)
  y() = Array(True, False, True, False)
  For i = LBound(x()) To UBound(x())
    s = s & x(i) & " OR " & y(i) & " = " & CBool(x(i) OR y(i)) & CHR$(10)
  Next
  s = s & IntToBinaryString(xi, 4) & " OR " & IntToBinaryString(yi, 4) &_
     " = " & IntToBinaryString(xi OR yi, 4) & CHR$(10)
  MsgBox s, 0, "Beispiel für Operator OR"
End Sub
```

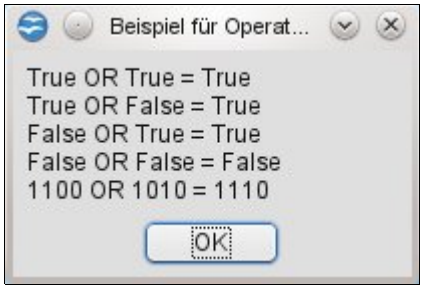


Bild 32. *Gebrauch des Operators OR.*

XOR

Der Operator XOR wird auch „exklusives Oder“ genannt. Er fragt nach Antivalenz oder Ungleichwertigkeit. Das Ergebnis ist True, wenn die Operanden unterschiedliche Werte haben. Das Ergebnis ist False, wenn beide Operanden denselben Wert haben. Das logische XOR operiert auf booleschen Werten und das bitweise XOR auf numerischen Werten.

Listing 40. *Operator XOR.*

```
Sub ExampleOpXOR
  Dim s$, x(), y(), i%, xi%, yi%
  xi% = 12 : yi% = 10
  x() = Array(True, True, False, False)
  y() = Array(True, False, True, False)
  For i = LBound(x()) To UBound(x())
    s = s & x(i) & " XOR " & y(i) & " = " & CBool(x(i) XOR y(i)) & CHR$(10)
  Next
  s = s & IntToBinaryString(xi, 4) & " XOR " & IntToBinaryString(yi, 4) &_
     " = " & IntToBinaryString(xi XOR yi, 4) & CHR$(10)
  MsgBox s, 0, "Beispiel für Operator XOR"
End Sub
```

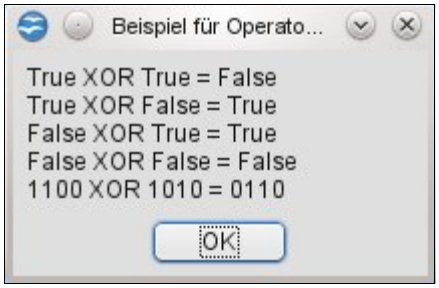


Bild 33. *Der Gebrauch des Operators XOR.*

EQV

Der Operator EQV fragt nach Äquivalenz oder Gleichwertigkeit. Sind beide Operanden gleich? Das logische EQV operiert auf booleschen Werten und das bitweise EQV auf numerischen Werten. Wenn beide Operanden denselben Wert haben, ist das Ergebnis True. Wenn die Operanden nicht denselben Wert haben, ist das Ergebnis False.

Listing 41. *Operator EQV.*

```
Sub ExampleOpEQV
  Dim s$, x(), y(), i%, xi%, yi%
  xi% = 12 : yi% = 10
  x() = Array(True, True, False, False)
  y() = Array(True, False, True, False)
  For i = LBound(x()) To UBound(x())
    s = s & x(i) & " EQV " & y(i) & " = " & CBool(x(i) EQV y(i)) & CHR$(10)
```

```
  Next
  s = s & IntToBinaryString(xi, 4) & " EQV " & IntToBinaryString(yi, 4) &_
     " = " & IntToBinaryString(xi EQV yi, 4) & CHR$(10)
  MsgBox s, 0, "Beispiel für Operator EQV"
End Sub
```

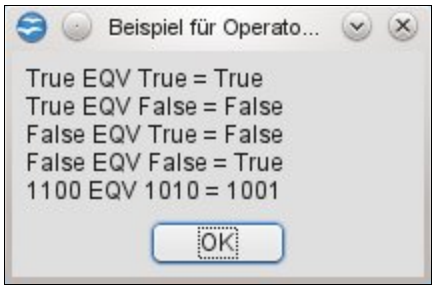


Figure 34. *Der Gebrauch des Operators EQV.*

IMP

Der Operator IMP führt eine logische Schlussfolgerung durch, eine Implikation. Das logische IMP operiert auf booleschen Werten und das bitweise IMP auf numerischen Werten. Wie der Name schon aussagt, fragt „x IMP y“, ob die Aussage„x impliziert y“ wahr ist. Zum Verständnis einer logischen Implikation definieren Sie x und y folgendermaßen:

```
x = Der Himmel ist wolkig
y = Die Sonne ist nicht zu sehen
If x Then y
```

Wenn sowohl x wie auch y wahr sind – der Himmel ist wolkig, und die Sonne ist nicht zu sehen –, kann die Aussage als wahr betrachtet werden. Die Aussage sagt nichts über y aus, wenn x nicht wahr ist. Mit anderen Worten, wenn der Himmel nicht wolkig ist, impliziert die Aussage nicht, dass die Sonne zu sehen oder nicht zu sehen ist. Es könnte zum Beispiel eine klare Nacht sein, oder Sie könnten (wie ein guter Computer-Fex) sich in einem Zimmer ohne Fenster befinden. Daraus folgt, dass die gesamte Aussage immer als wahr gewertet wird, wenn x falsch ist. Wenn schließlich x wahr ist und y nicht, wird die gesamte Aussage als falsch gewertet. Wenn der Himmel wolkig ist und die Sonne zu sehen ist, kann die Aussage unmöglich korrekt sein. Ein wolkiger Tag könnte nicht implizieren, dass die Sonne sichtbar ist.

Listing 42. *Operator IMP.*

```
Sub ExampleOpIMP
  Dim s$, x(), y(), i%, xi%, yi%
  xi% = 12 : yi% = 10
  x() = Array(True, True, False, False)
  y() = Array(True, False, True, False)
  For i = LBound(x()) To UBound(x())
    s = s & x(i) & " IMP " & y(i) & " = " & CBool(x(i) IMP y(i)) & CHR$(10)
  Next
  s = s & IntToBinaryString(xi, 4) & " IMP " & IntToBinaryString(yi, 4) &_
     " = " & IntToBinaryString(xi IMP yi, 4) & CHR$(10)
  MsgBox s, 0, "Beispiel für Operator IMP"
End Sub
```

Bild 35. *Der Gebrauch des Operators IMP.*

NOT

Das logische NOT operiert auf booleschen Werten und das bitweise NOT auf numerischen Werten. Das heißt, dass „Not True“ False ist und „Not False“ True ist. Bei bitweisen Operationen wird eine 1 zu 0 und eine 0 zu 1.

```
Print NOT True   REM 0, das False ist
Print NOT False  REM -1, das True ist
Print NOT 2      REM -3, die Bits 0010 wurden zu 1101
```

3.7.3. Vergleichsoperatoren

Die Vergleichsoperatoren funktionieren mit numerischen Datentypen, sowie mit den Typen Date, Boolean und String.

```
Print 2  = 8/4 AND 0 <  1/3 AND 2  > 1    '-1=True
Print 4 <= 4.0 AND 1 >=   0 AND 1 <> 0    '-1=True
```

Stringvergleiche basieren darauf, dass Strings intern als Zahlen vorliegen, und unterscheiden Groß- und Kleinschreibung. Der Buchstabe „A“ ist kleiner als der Buchstabe „B“. Alle Großbuchstaben sind kleiner als die Kleinbuchstaben.

```
Dim a$, b$, c$
a$ = "A" : b$ = "B" : c$ = "B"
Print a$  < b$   'True
Print b$  = c$   'True
Print c$ <= a$   'False
```

Es kann zu seltsamen Problemen führen, wenn alle Operanden String-Konstanten sind. Wenn wenigstens ein Operand eine Variable ist, gibt es das erwartete Ergebnis. Das hat wahrscheinlich mit der Art und Weise zu tun, wie Operanden erkannt und zum Einsatz konvertiert werden.

```
Print "A" < "B"     '0=False, das ist nicht korrekt
Print "B" < "A"     '-1=True, das ist nicht korrekt
Print  3  = "3"     'False, aber das ändert sich bei der Verwendung einer Variablen
```

Werden Variablen an Stelle von String-Konstanten verwendet, werden die numerischen Wert für den Vergleich zum Typ String konvertiert.

```
Dim a$, i%, t$
a$ = "A" : t$ = "3" : i% = 3
Print  a$ < "B"     'True,  String-Vergleich
Print "B" <  a$     'False, String-Vergleich
Print  i% = "3"     'True,  String-Vergleich
Print  i% = "+3"    'False, String-Vergleich
Print  3  =  t$     'True,  String-Vergleich
Print  i% < "2"     'False, String-Vergleich
Print  i% > "22"    'True,  String-Vergleich
```

Tipp Wenn man Operanden unterschiedlichen Typs vergleicht, vor allem in der Konstellation numerischer Typ und String, ist eine explizite Tpykonversion sicherer. Konvertieren Sie entweder den String zu einer Zahl oder die Zahl zum String. Die dafür vorgesehenen Funktionen sehen Sie an späterer Stelle.

OOo erkennt die Visual-Basic-Anweisung Option Compare { Binary | Text}, aber die Anweisung bewirkt nichts, jedenfalls in OOo 3.20. Das aktuelle Verhalten ist der binäre Vergleich von Strings.

3.8. Ablaufsteuerung

Ablaufsteuerung handelt davon, welche Codezeile als nächste ausgeführt wird. Der Aufruf einer Subroutine oder einer Funktion ist eine einfache Form nicht bedingter Ablaufsteuerung. Zu komplexeren Ablaufsteuerungen gehören Verzweigungen und Schleifen. Ablaufsteuerung ermöglicht verwickelte Abläufe in Makros, die sich je nach aktueller Datenlage ändern.

Verzweigungsanweisungen bewirken eine Änderung des Programmflusses. Der Aufruf einer Subroutine oder einer Funktion ist eine nicht bedingte Verzweigung. OOo Basic unterstützt bedingte Verzweigungsanweisungen wie „wenn x, dann tu y“. Schleifenanweisungen bewirken, dass das Programm Codebereiche wiederholt. Mit Schleifenanweisungen kann ein Bereich bestimmte Male wiederholt werden oder bis eine spezifische „exit“-Bedingung erreicht wird.

3.8.1. Definition eines Labels als Sprungmarke

Einige Anweisungen zur Ablaufsteuerung wie GoSub, GoTo und On Error benötigen eine markierte Stelle im Code, ein Label. Label-Namen gehorchen denselben Regeln wie Namen von Variablen. Auf einen Label-Namen folgt direkt ein Doppelpunkt. Sie erinnern sich, dass ein Doppelpunkt auch als Trenner von Anweisungen dient, um mehrere Anweisungen auf einer Zeile zu ermöglichen. Ein Leerzeichen zwischen Label und Doppelpunkt würde den Doppelpunkt als Anweisungstrenner ausweisen, wodurch das Label nicht definiert wäre. Die folgenden Zeilen sind gültiger OOo-Basic-Code.

```
    <Anweisungen>
    i% = 5 : z = q + 4.77

MyCoolLabel:
    <weitere Anweisungen>

JumpTarget: <weitere Anweisungen> REM Zwischen Label und Doppelpunkt ist kein Leerraum
```

Tipp Ein zwischen Label und Doppelpunkt eingefügtes Leerzeichen macht aus dem Doppelpunkt einen Anweisungstrenner, und das Label ist nicht definiert.

3.8.2. GoSub

Die Anweisung GoSub bewirkt, dass das Programm an einem definierten Label innerhalb der aktuellen Routine fortgesetzt wird. Ein Sprung an eine Stelle außerhalb der Routine ist nicht möglich. Beim Erreichen der Anweisung Return wird das Programm an der Stelle der GoSub-Anweisung fortgesetzt. Eine Return-Anweisung ohne vorheriges GoSub produziert einen Laufzeitfehler. Mit anderen Worten, Return ist kein Ersatz für Exit Sub oder Exit Function. Im allgemeinen sollten Funktionen und Subroutinen aber verständlicheren Code liefern als GoSub oder GoTo.

Tipp GoSub ist ein hartnäckiges Überbleibsel von alten BASIC-Dialekten, beibehalten aus Gründen der Kompatibilität. Von der Verwendung von GoSub wird dringend abgeraten, weil es dazu verführt, nicht lesbaren Code zu produzieren. Schreiben Sie stattdessen eine Subroutine oder eine Funktion. Tatsächlich unterstützt Visual Basic .NET das Schlüsselwort GoSub nicht mehr.

Listing 43. *Beispiel für GoSub.*

```
Sub ExampleGoSub
  Dim i As Integer
  GoSub Line2       REM Springt zu Line2 und kehrt zurück, i ist dann 1.
  GoSub [Line 1]    REM Springt zu [Line 1] und kehrt zurück, i ist dann 2.
  MsgBox "i = " + i, 0, "Beispiel für GoSub"  REM i ist nun 2
  Exit Sub          REM Beendet die aktuelle Subroutine.
[Line 1]:           REM Dieses Label enthält ein Leerzeichen.
  i = i + 1         REM Addiert 1 zu i.
  Return            REM Kehrt zur aufrufenden Stelle zurück.
Line2:              REM Dies ist ein typischeres Label, kein Leerzeichen.
  i = 1             REM Setzt i zu 1.
  Return            REM Kehrt zur aufrufenden Stelle zurück.
End Sub
```

3.8.3. GoTo

Die Anweisung GoTo bewirkt, dass das Programm an einem definierten Label innerhalb der aktuellen Routine fortgesetzt wird. Ein Sprung an eine Stelle außerhalb der Routine ist nicht möglich. Im Gegensatz zur Anweisung GoSub weiß GoTo nicht, von wo es aufgerufen wird. GoTo ist ein hartnäckiges Überbleibsel von alten BASIC-Dialekten, beibehalten aus Gründen der Kompatibilität. Von der Verwendung von GoTo wird dringend abgeraten, weil es dazu verführt, nicht lesbaren Code zu produzieren. Schreiben Sie stattdessen eine Subroutine oder eine Funktion.

Listing 44. *Beispiel für GoTo.*

```
Sub ExampleGoTo
  Dim i As Integer
  GoTo Line2        REM Okay, das sieht leicht genug aus,
Line1:              REM aber jetzt gerate ich doch ein wenig durcheinander.
  i = i + 1         REM Ich wünschte, ich hätte GoTo nicht verwendet.
  GoTo TheEnd       REM Das ist doch verrückt, das sieht irgendwie nach Spaghetti aus,
Line2:              REM ein Gewirr von losen Fäden: Spaghetti-Code.
  i = 1             REM Falls Sie es so machen mussten, dann
  GoTo Line1        REM haben Sie wohl etwas schlecht gemacht.
TheEnd:             REM Verwenden Sie kein GoTo.
  MsgBox "i = " + i, 0, "Beispiel für GoTo"
End Sub
```

3.8.4. On GoTo und On GoSub

Mit diesen Anweisungen wird das Programm abhängig von einem numerischen Ausdruck N zu einem Label verzweigt. Wenn N Null ist, wird nicht verzweigt. Der numerische Ausdruck N muss im Bereich von 0 bis 255 liegen. Manchmal wird das als „berechnetes GoTo“ bezeichnet, weil eine Berechnung als Grundlage für den Programmfluss dient. Ein Sprung an eine Stelle außerhalb der Routine ist nicht möglich.

Syntax: `On N GoSub Label1[, Label2[, Label3[,...]]]`
Syntax: `On N GoTo  Label1[, Label2[, Label3[,...]]]`

In der Aufzählung wird die Arbeitsweise deutlich: wenn N = 1 ist, dann verzweige zum Label 1, wenn N = 2 ist, dann verzweige zu Label 2 ... Wenn N kleiner als 1 oder größer als die Anzahl der Labels ist, dann wird gar nicht verzweigt, die Anweisung wird einfach ignoriert.

Listing 45. *Beispiel für On GoTo.*

```
Sub ExampleOnGoTo
  Dim i As Integer
  Dim s As String
  i = 1
```

```
  On i+1 GoSub Sub1, Sub2
  s = s & Chr(13)
  On i GoTo Line1, Line2
  REM Mit exit beenden wir die Subroutine, wenn wir den Rest nicht ausführen wollen.
  Exit Sub
Sub1:
  s = s & "In Sub 1" : Return
Sub2:
  s = s & "In Sub 2" : Return
Line1:
  s = s & "Am Label 1" : GoTo TheEnd
Line2:
  s = s & "Am Label 2"
TheEnd:
  MsgBox s, 0, "Beispiel für On GoTo"
End Sub
```

3.8.5. If Then Else

Mit der If-Konstruktion wird abhängig von einem Ausdruck ein Codeblock ausgeführt. Auch wenn man mit GoTo oder GoSub aus einem If-Block heraussprungen kann, so kann man doch nicht in einen If-Block hineinspringen. Die einfachste If-Anweisung hat die folgende Form:

```
If Bedingung Then Anweisung
```

Die Bedingung kann aus jedem Ausdruck bestehen, der entweder True oder False ergibt – oder in diese Werte konvertierbar ist. Mit einer etwas komplexeren Version können Sie mehr als eine einzige Bedingung abfragen.

```
If Bedingung Then
  Anweisungsblock
[ElseIf Bedingung Then]
  Anweisungsblock
[Else]
  Anweisungsblock
End If
```

Wenn die erste Bedingung True ergibt, läuft der erste Block. Die Anweisung ElseIf erlaubt mehrere If-Anweisungen für eine Abfolge von Tests. Der Anweisungsblock für die erste wahre Bedingung läuft. Wenn keine andere Bedingung True ergibt, läuft der Else-Block.

Listing 46. *Beispiel für If.*

```
Sub ExampleIf
  Dim i%
  i% = 4
  If i = 4 Then Print "i ist vier"
  If i <> 3 Then
    Print "i ist nicht drei"
  End If
  If i < 1 Then
    Print "i ist kleiner als 1"
  elseif i = 1 Then
    Print "i ist 1"
  elseif i = 2 Then
    Print "i ist 2"
  else
    Print "i ist größer als 2"
  End If
End Sub
```

If-Anweisungen können verschachtelt sein.

```
If i <> 3 THEN
  If k = 4 Then Print "k ist vier"
  If j = 7 Then
     Print "j ist sieben"
  End If
End If
```

3.8.6. IIf

Die Funktion Iif („Immediate If“ = unmittelbares If) gibt abhängig von einer Bedingung einen von zwei Werten zurück.

```
Syntax:  object = IIf (Bedingung, AusdruckWennWahr, AusdruckWennFalsch)
```

Das entspricht annähernd dem folgenden Code:

```
If Bedingung Then
  object =  AusdruckWennWahr
Else
  object =  AusdruckWennFalsch
End If
```

Es ist de facto eine wunderbare einzeilige If-Then-Else-Anweisung.

```
max_alter = IIf(johns_alter > bills_alter, johns_alter, bills_alter)
```

3.8.7. Choose

Die Anweisung Choose wählt abhängig von einem Indexwert aus einer Werteliste aus.

```
Syntax: obj = Choose (Ausdruck, Select_1[, Select_2, ... [,Select_n]])
```

Die Anweisung Choose gibt Null zurück, wenn der Index-Ausdruck kleiner ist als 1 oder größer ist als die Anzahl der Argumente in der Liste. Choose gibt „Select_1“ zurück, wenn der Ausdruck 1 ergibt, und „Select_2“, wenn der Ausdruck 2 ergibt. Genauso kann man auch die Listenwerte in einem Array mit der Bereichsuntergrenze 1 speichern und per Index darauf zugreifen.

Listing 47. *Ein Division-durch-Null-Fehler entsteht, obwohl 1/(i-2) zurückgegeben werden sollte.*

```
i% = 3
Print Choose (i%, 1/(i+1), 1/(i-1), 1/(i-2), 1/(i-3))
```

Selektionen können Ausdrücke sein, auch mit Funktionsaufrufen. Bei der Anweisung Choose wird jede Funktion der Argumentenliste aufgerufen und jeder Ausdruck ausgewertet. Der Code in Listing 47 bewirkt einen Division-durch-Null-Fehler, weil jedes Argument ausgewertet wird, nicht nur das zurückzugebende Argument. Listing 48 ruft die Funktionen Choose1, Choose2, und Choose3 auf.

Listing 48. *Beispiel für die Anweisung Choose.*

```
Sub ExampleChoose
  Print Choose(2, "Eins", "Zwei", "Drei")           'Zwei
  Print Choose(2, Choose1(), Choose2(), Choose3())  'Zwei
End Sub
Function Choose1$()
  Print "Ich bin in Choose1"
  Choose1 = "Eins"
End Function
Function Choose2$()
  Print "Ich bin in Choose2"
  Choose2 = "Zwei"
End Function
Function Choose3$()
```

```
  Print "Ich bin in Choose3"
  Choose3 = "Drei"
End Function
```

Tipp In der Anweisung Choose werden alle Argumente ausgewertet. Werden innerhalb der Argumente Funktionen genutzt, werden sie alle ausgeführt.

3.8.8. Select Case

Die Anweisung Select Case ähnelt einer If-Anweisung mit mehreren Else-If-Blöcken. Es wird aber nur ein Bedingungsausdruck definiert, der dann mit mehreren Werten auf Gleichheit geprüft wird:

```
Select Case bedingungs-ausdruck
  Case case_ausdruck1
    Anweisungsblock1
  Case case_ausdruck2
    Anweisungsblock2
  Case Else
    Anweisungsblock3
End Select
```

Der Ausdruck bedingungs-ausdruck wird in jedem Case-Ausdruck verglichen. Der auf den ersten Treffer folgende Block von Anweisungen wird ausgeführt. Der optionale Block Case Else wird ausgeführt, wenn keine Bedingung zutrifft. Es ist kein Fehler, wenn nichts zutrifft und kein Case Else definiert ist. Dann gibt es einfach nichts zu tun.

Case-Ausdrücke

Der Bedingungsausdruck wird nur einmal ausgewertet und dann der Reihe nach mit jedem Case-Ausdruck verglichen, bis ein Treffer erfolgt. Ein Case-Ausdruck ist häufig nur eine Konstante wie „Case 4“ oder „Case "hello"“.

```
Select Case 2
  Case 1
    Print "Eins"
  Case 3
    Print "Drei"
End Select
```

Sie können mehrere Werte gleichzeitig angeben, wenn Sie sie mit Kommas trennen: „`Case 3, 5, 7`“. Das Schlüsselwort `To` kontrolliert einen Wertebereich – zum Beispiel „`Case 5 To 10`“. Bereiche mit einem offenen Ende werden als „`Case < 10`“ oder als „`Case IS < 10`“ kontrolliert.

Tipp Die Anweisung Case IS hat nichts mit dem Operator IS zu tun, der prüft, ob zwei Objekte gleich sind.

Jede Case-Anweisung in der Form „Case op expression“ ist eine Kurzform für „Case IS op expression“. Die Form „Case expression“ ist die Kurzform von „`Case IS = expression`“. Zum Beispiel ist „`Case >= 5`“ äquivalent zu „`Case IS >= 5`“, und „`Case 1+3`“ ist äquivalent zu „`Case IS = 1+3`“.

```
Select Case i
  Case 1, 3, 5
    Print "i ist eins , drei oder fünf"
  Case 6 To 10
    Print "i ist ein Wert von 6 bis 10"
  Case < -10
    Print "i ist kleiner als -10"
  Case IS > 10
```

```
      Print "i ist größer als 10"
    Case Else
      Print "Keine Ahnung, was i ist"
End Select
```

Eine Case-Anweisung kann, durch Kommas getrennt, eine Liste von Ausdrücken enthalten. Jeder Ausdruck kann einen zu einer Seite offenen Bereich einschließen. Jeder Ausdruck kann die Anweisung Case IS nutzen (s. Listing 49).

Listing 49. *Das Schlüsselwort IS ist optional.*

```
Select Case i%
  Case 6, Is = 7, Is = 8, Is > 15, Is < 0
    Print "" & i & " passt"
  Case Else
    Print "" & i & " ist außerhalb des Bereichs"
End Select
```

Wenn Case-Anweisungen so einfach sind, warum sind sie so oft fehlerhaft?

Ich sehe häufig fehlerhafte Beispiele von Case-Anweisungen. Es ist sehr lehrreich, sich anzusehen, was immer wieder falsch gemacht wird. Schauen Sie sich die Beispiele in der Tabelle 17 an. Die korrekte Form der fehlerhaften Beispiele finden Sie in der Tabelle 19.

Tabelle 17. *Case-Anweisungen sind häufig falsch definiert.*

Beispiel	Gültig	Beschreibung
`Select Case i` `  Case 2`	Korrekt	Der Case-Ausdruck 2 wird zur Zahl 2 ausgewertet und mit i verglichen.
`Select Case i` `  Case Is = 2`	Korrekt	Der Case-Ausdruck 2 wird zur Zahl 2 ausgewertet und mit i verglichen.
`Select Case i` `  Case Is > 7`	Korrekt	Der Case-Ausdruck 7 wird zur Zahl 7 ausgewertet und mit i verglichen.
`Select Case i` `  Case 4, 7, 9`	Korrekt	Der Bedingungsausdruck i wird nacheinander mit 4, 7 und 9 verglichen.
`Select Case x` `  Case 1.3 TO 5.7`	Korrekt	Sie können einen Bereich definieren und Fließkommazahlen verwenden.
`Select Case i` `  Case i = 2`	Falsch	Der Case-Ausdruck (i=2) wird als True oder False ausgewertet. Dieser boolesche Wert wird mit i verglichen. Kurzform für „IS = (i=2)“.
`Select Case i` `  Case i<2 OR i>9`	Falsch	Der Case-Ausdruck (i<2 OR 9<i) wird als True oder False ausgewertet. Dieser boolesche Wert wird mit i verglichen. Kurzform für „IS = (i<2 OR 9<i)“.
`Select Case i%` `  Case i%>2 AND i%<10`	Falsch	Der Case-Ausdruck (i%>2 AND i% < 10) wird als True oder False ausgewertet. Dieser boolesche Wert wird mit i verglichen. Kurzform für „IS = (i%>2 AND i% < 10)“.
`Select Case i%` `  Case IS>8 And i<11`	Falsch	Wiederum wird i% mit True und False verglichen. Es ist die Kurzform für „IS > (8 AND i<11)“. Die Prioritätsregeln machen daraus „IS > (8 AND (i<11))“. Es ist nicht anzunehmen, dass dies gewünscht ist.
`Select Case i%` `  Case IS>8 And IS<11`	Falsch	Kompilierungsfehler. Das Schlüsselwort IS muss direkt auf Case folgen.

Ich habe OOo-Beispiele mit falschen Definitionen wie „`Case i > 2 AND i < 10`“ gesehen. Das geht schief. Glauben Sie so etwas nicht, auch wenn Sie es gedruckt lesen. Wenn Sie verstehen, warum es falsch ist, haben Sie die Case-Anweisungen gemeistert.

Das vorletzte unkorrekte Beispiel in der Tabelle 17 ist repräsentativ für die meisten Fehler in Case-Ausdrücken, die ich gesehen habe. Listing 50 behandelt den Fall, dass i kleiner als 11 ist, und den

Fall, dass i größer als oder gleich 11 ist. Schlicht und einfach bewirkt `IS>8 AND i<11`, dass der Case-Ausdruck den Wert von i mit dem Ergebnis eines booleschen Ausdrucks vergleicht, der nur 0 oder -1 sein kann. Das große Problem mit Case-Anweisungen besteht darin, dass sie wie If-Anweisungen aussehen, die nach True oder False fragen, aber Case-Anweisungen suchen einen bestimmten Wert, der mit dem Bedingungswert verglichen wird. Dafür sind 0 oder -1 nicht gerade hilfreich.

Nehmen wir den zweiten Fall im Listing 50, `i >= 11`. Der Operator < hat eine höhere Priorität als der Operator `AND`, wird also zuerst ausgeführt. Der Ausdruck `i<11` ergibt False (wegen der Annahme, dass `i>=11`). False wird intern als 0 repräsentiert. Da bei Null keine Bits gesetzt sind, ergibt `8 AND 0` den Wert Null. Für die Fälle, dass i größer als oder gleich 11 ist, wird der gesamte Ausdruck gleichbedeutend mit „`IS > 0`“. Mit anderen Worten, die Case-Anweisung wird für `i = 45` unerwünschterweise als Treffer angenommen.

Eine ähnliche Argumentation für Werte von i kleiner als 11 – dem Leser zur Übung überlassen – zeigt, dass die Case-Anweisung gleichbedeutend ist mit „`Case IS > 8`“. Daher werden Werte von i kleiner als 11 korrekt ausgewertet, nicht aber Werte von i größer als oder gleich 11.

Listing 50. *„Case IS > 8 AND i < 11“ wird in unerwarteter Weise reduziert.*

```
IS > (8 AND i < 11) => IS > (8 AND -1) => IS > 8 'Korrekt, wenn i <  11
IS > (8 AND i < 11) => IS > (8 AND  0) => IS > 0 'Falsch, wenn i >= 11
```

Wie man fehlerfreie Case-Ausdrücke schreibt

Nachdem Sie ein paar einfache Beispiele kennengelernt haben, wird es Ihnen leicht fallen, fehlerfreie Case-Ausdrücke zu schreiben. Tabelle 18 listet die Varianten theoretisch auf, und Listing 54 zeigt sie in konkreten Beispielen.

Tabelle 18. *Einfache Case-Varianten.*

Beispiel	Beschreibung
`Case IS Operator Ausdruck`	Dieser Fall ist zugleich der einfachste wie auch der schwierigste. Wenn der Ausdruck aus einer Konstanten besteht, ist der Fall einfach. Wenn der Ausdruck aber komplexer ist, besteht der schwierige Teil darin, den Ausdruck zu definieren.
`Case Ausdruck`	Dies ist eine Kurzform von „Case IS Operator Ausdruck“, wenn der Operator auf Gleichheit prüft.
`Case Ausdruck TO Ausdruck`	Prüft einen begrenzten Bereich. Wird gewöhnlich richtig gemacht.
`Case Ausdruck, Ausdruck,`	Jeder Ausdruck wird verglichen. Wird gewöhnlich richtig gemacht.

In den schwierigen Fällen reicht es, einen Ausdruck zu finden, der zu dem Bedingungswert ausgewertet wird, wenn es ein Treffer sein soll, und irgendetwas anderes ergibt, wenn es kein Treffer sein soll. Mit anderen Worten, im Falle von Select Case 4 muss der Ausdruck 4 ergeben, wenn der Anweisungsblock ausgeführt werden soll.

Listing 51. *Wenn x ein String ist, funktioniert das Folgende mit jedem booleschen Ausdruck.*

```
Select Case x
  Case IIF(Boolean ausdruck, x, x&"1") ' Für den Fall, dass x ein String ist
```

In Listing 51 wird x zurückgegeben, wenn der boolesche Ausdruck True ergibt. Der Ausdruck x=x ist True, also ist die Case-Anweisung ein Treffer. Wenn der boolesche Ausdruck False ergibt, wird x&“1“ zurückgegeben. Dieser String ist nicht mehr derselbe wie x, also ist die Case-Anweisung kein Treffer. Ähnlich macht man das auch mit numerischen Werten.

Listing 52. *Wenn x numerisch ist, funktioniert das Folgende mit jedem booleschen Ausdruck.*

```
Select Case x
  Case IIF(Boolean ausdruck, x, x+1) ' Für den Fall, dass x numerisch ist
```

In Listing 52 wird x zurückgegeben, wenn der boolesche Ausdruck True ergibt. Der Ausdruck x=x ist True, also ist die Case-Anweisung ein Treffer. Wenn der boolesche Ausdruck False ergibt, wird x+1 zurückgegeben. Für numerische Werte ergibt x=x+1 nicht True, also ist die Case-Anweisung kein Treffer. Im allgemeinen funktioniert das, doch es besteht das Risiko des numerischen Überlaufs. Von Bernard Marcelly, einem Mitglied des Projekts Französischsprachiges OOo, kommt eine brillante und elegantere Lösung für numerische Werte.

```
Case x XOR NOT (Boolean ausdruck)
```

Dies setzt voraus, dass Case ein Treffer sein soll, wenn der boolesche Ausdruck True (-1) ergibt, und kein Treffer, wenn False (0) herauskommt.

Listing 53. *Dieser Code nutzt XOR und NOT in einer Case-Anweisung.*

```
x XOR NOT(True)  = x XOR NOT(-1) = x XOR  0 =  x
x XOR NOT(False) = x XOR NOT( 0) = x XOR -1 <> x
```

Zuerst war ich ziemlich verwirrt, bemerkte aber dann, wie brillant das tatsächlich ist. Es entsteht kein Überlaufproblem und funktioniert mit allen Integer-Werten von x. Vereinfachen Sie es nicht zur fehlerhaften Verkürzung „x AND (Boolean expression)“, denn das versagt, wenn x Null ist.

Listing 54. *Beispiel für Select Case.*

```
Sub ExampleSelectCase
  Dim i%
  i = Int((20 * Rnd) -2) 'Rnd liefert eine Zufallszahl zwischen Null und Eins
  Select Case i%
    Case 0
      Print "" & i & " ist null"
    Case 1 To 5
      Print "" & i & " ist eine Zahl von 1 bis 5"
    Case 6, 7, 8
      Print "" & i & " ist die Zahl 6, 7 oder 8"
    Case IIf(i > 8 And i < 11, i, i+1)
      Print "" & i & " ist größer als 8 und kleiner als 11"
    Case i% XOR NOT(i% > 10 AND i% < 16)
      Print "" & i & " ist größer als 10 und kleiner als 16"
    Case Else
      Print "" & i & " ist außerhalb des Bereichs 0 bis 15"
  End Select
End Sub
```

ExampleSelectCase in Listing 54 liefert bei jedem Durchlauf eine Zufallszahl als Ganzzahl von -2 bis 18. Starten Sie das Makro mehrfach hintereinander, so dass Sie jede Case-Anweisung ausgeführt bekommen. Jeder dieser Fälle könnte das IIF-Konstrukt verwendet haben.

Nachdem ich nun die verschiedenen Methoden des Umgangs mit Bereichen erläutert habe, wird es Zeit, die fehlerhaften Fälle in Tabelle 17 zu überarbeiten. Die Lösungen in Tabelle 19 sind nicht die einzig möglichen Lösungen, nutzen aber einige der präsentierten Muster.

Tabelle 19. *Die fehlerhaften Beispiele der Tabelle 17 - nun korrigiert.*

Falsch	Richtig	Beschreibung
`Select Case i` `  Case i = 2`	`Select Case i` `  Case 2`	Die Variable i wird mit 2 verglichen.
`Select Case i` `  Case i = 2`	`Select Case i` `  Case IS = 2`	Die Variable i wird mit 2 verglichen.

Falsch	Richtig	Beschreibung
`Select Case i` `  Case i<2 OR i>9`	`Select Case i` `  Case IIf(i<2 OR i>9, i, i+1)`	Funktioniert auch, wenn i keine Ganzzahl ist.
`Select Case i%` `  Case i%>2 AND i %<10`	`Select Case i%` `  Case 3 TO 9`	i% ist eine Ganzzahl, also geht der Bereich von 3 bis 9.
`Select Case i%` `  Case IS>8 And i<11`	`Select Case i%` `  Case i XOR NOT(i>8 AND i< 11)`	Funktioniert, weil i% eine Ganzzahl ist.

3.8.9. While ... Wend

Mit der Anweisung While ... Wend wiederholen Sie einen Anweisungsblock, solange eine Bedingung wahr ist. Dieses Konstrukt hat gewisse Nachteile gegenüber der Schleifenanweisung Do While ... Loop, bietet aber auch keine Vorteile. While ... Wend unterstützt keine Exit-Anweisung, und Sie können While ... Wend nicht mit GoTo verlassen.

```
While Bedingung
  Anweisungsblock
Wend
```

Visual Basic .NET kennt das Schlüsselwort Wend nicht, ein weiterer Grund, stattdessen Do While ... Loop zu verwenden.

3.8.10. Do ... Loop

Der Loop-Mechanismus kennt verschiedene Formen und wird eingesetzt, wenn ein Anweisungsblock wiederholt ausgeführt werden soll, solange oder bis eine Bedingung wahr ist. In der gebräuchlichsten Form wird die Bedingung geprüft, bevor die Schleife startet, und der Anweisungsblock wird so lange wiederholt, wie die Bedingung wahr bleibt. Wenn die Eingangsbedingung falsch ist, wird die Schleife gar nicht ausgeführt.

```
Do While Bedingung
  Block
  [Exit Do]
  Block
Loop
```

In ähnlicher, aber nicht so häufig vorkommender Form wird der Block so lange wiederholt, wie die Bedingung falsch ist. Mit anderen Worten, der Code wird ausgeführt, bis die Bedingung wahr wird. Wenn sich die Bedingung schon zu Beginn als wahr zeigt, wird die Schleife gar nicht ausgeführt.

```
Do Until Bedingung
  Block
  [Exit Do]
  Block
Loop
```

Sie können die Prüfung der Bedingung an das Schleifenende setzen, so dass der Anweisungsblock wenigstens einmal ausgeführt wird. In der folgenden Form läuft die Schleife wenigstens einmal durch und wird danach so lange wiederholt, wie die Bedingung wahr ist:

```
Do
  Block
  [Exit Do]
  Block
Loop While Bedingung
```

In der folgenden Form führen Sie die Schleife wenigstens einmal aus und danach immer wieder, solange die Bedingung falsch ist:

```
Do
  Block
  [Exit Do]
  Block
Loop Until Bedingung
```

Aussteigen aus der Do-Schleife

Die Anweisung Exit Do bewirkt das sofortige Schleifenende. Sie ist nur zwischen Do und Loop zulässig. Das Programm wird mit der Anweisung fortgesetzt, die der innersten Loop-Anweisung folgt. Die Subroutine ExampleDo im Listing 55 zeigt eine Do-While-Schleife, mit der eine Zahl in einem Array gesucht wird.

Listing 55. *Beispiel für Do Loop.*

```
Sub ExampleDo
  Dim a(), i%, x%
  a() = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30)
  x = Int(32 * Rnd)                  REM Zufällige Ganzzahl zwischen 0 und 32
  i = LBound(a())                    REM i ist die Bereichsuntergrenze des Arrays.
  Do While a(i) <> x                 REM Solange a(i) ungleich x ist,
   i = i + 1                         REM wird i um 1 hochgezählt.
   If i > UBound(a()) Then Exit Do   REM Ausstieg, sobald i zu groß ist.
  Loop                               REM Die Schleife kehrt zu Do While zurück.
  If i <= UBound(a()) Then           REM Wenn i nicht zu groß ist, wurde x gefunden.
    MsgBox "Habe " & x & " gefunden an der Position " & i, 0, "Beispiel für Do"
  Else
    MsgBox "Konnte " & x & " nicht im Array finden", 0, " Beispiel für Do"
  End If
End Sub
```

Welche Do-Loop-Form ist zu wählen?

OOo Basic unterstützt vier Varianten der Do-Loop-Konstruktion. Der Einsatzzweck und Zeitpunkt bestimmt die jeweilige Variante. Das häufigste Problem liegt darin, dass die Schleife einmal zu viel oder einmal zu wenig durchlaufen wird, weil der Bedingungsausdruck am falschen Ende platziert wurde.

Zur Entscheidung, wo der Bedingungsausdruck einer Do-Loop-Schleife stehen soll, stellen Sie sich folgende Frage: „Muss die Schleife mindestens einmal durchlaufen werden?“ Bei der Antwort Nein muss der Bedingungsausdruck nach oben. Das verhindert, dass die Anweisungen innerhalb der Schleife ausgeführt werden, wenn der Vergleich fehlschlägt. Stellen Sie sich vor, Sie wollen alle Elemente eines Arrays unbekannter Größe ausdrucken. Das Array könnte leer sein und gar keine Elemente enthalten. In diesem Fall sollte der Code innerhalb der Schleife gar nicht ausgeführt werden (s. Tabelle 20).

Tabelle 20. *Schleifen mit While (solange) und Until (bis) sind sehr ähnlich.*

Do While	Do Until

`i% = LBound(a())`
`Do While i% <= UBound(a())`
`  Print a(i%)`
`  i% = i% + 1`
`Loop`
	`i% = LBound(a())`
	`Do Until i% > UBound(a())`
	`  Print a(i%)`
	`  i% = i% + 1`
	`Loop`

In beiden Fällen der Tabelle 20 wird die Schleife gar nicht durchlaufen, wenn das Array leer ist. Vor der Auswertung der Bedingung wird i% auf den Wert der Bereichsuntergrenze gesetzt. In beiden Fällen läuft die Schleife, solange i% nicht größer ist als die Bereichsobergrenze.

Den Unterschied zwischen einer While-Schleife und einer Until-Schleife kann man an einem einfachen Beispiel sehen. Solange (While) Sie im Auto Gas geben, können Sie es fahren. Bis (Until) Sie im Auto kein Gas mehr geben, können Sie es fahren. Der Hauptunterschied zwischen While und Until ist das Wort NOT. Die vorige Aussage würde in OOo Basic eher lauten „Until NOT (es wird Gas gegeben)“. Die Wahl zwischen While und Until richtet sich danach, welches Sie ohne NOT schreiben können.

Wenn die Schleife mindestens einmal durchlaufen werden soll, stellen Sie den Bedingungsausdruck an das Ende von Do Loop. Gesetzt den Fall, Sie fordern eine Nutzereingabe, bis ein gültiger Wert geliefert wird. Der Bedingungsausdruck steht dabei natürlicherweise am Ende.

```
Dim s$, x As Double
Do
  s$ = InputBox("Geben Sie bitte eine Zahl zwischen 1 und 5 ein")
  x = CDbl(s$)      'Konvertierung des Strings zu Double
Loop Until x >= 1 AND x <= 5
```

Die Schleife braucht wenigstens einen Durchlauf, so dass wenigstens eine Zahl eingegeben werden kann. Die Schleife wird so lange wiederholt, bis eine gültige Zahl eingegeben wird. Zur Übung können Sie diese Schleife einmal mit While schreiben.

3.8.11. For ... Next

Die Anweisung For ... Next wiederholt einen Anweisungsblock für eine bestimme Anzahl von Durchläufen.

```
For counter=start To end [Step schrittWert]
  Anweisungsblock1
  [Exit For]
  Anweisungsblock2
Next [counter]
```

Der numerische Zähler „counter“ wird auf den Wert von „start“ initialisiert. Wenn das Programm zur Anweisung Next gelangt, wird counter durch den Schrittwert „step“ heraufgezählt. Fehlt der Schrittwert, wird um 1 heraufgezählt. Wenn counter kleiner oder gleich dem Wert von „end“ ist, wird der Anweisungsblock durchlaufen. Im folgenden eine äquivalente Do-While-Schleife:

```
counter = start
Do While counter <= end
  Anweisungsblock1
  [Exit Do]
  Anweisungsblock2
  counter = counter + step
Loop
```

Die Angabe von „counter“ hinter der Next-Anweisung ist optional. Next bezieht sich automatisch auf die zuletzt vorgekommene For-Anweisung.

```
For i = 1 To 4 Step 2
  Print i ' Gibt erst 1, dann 3 aus.
Next i    ' In dieser Anweisung ist das i optional.
```

Mit der Anweisung Exit For verlassen Sie unmittelbar den For-Block, und zwar den nach der zuletzt vorgekommenen For-Anweisung. Listing 56 zeigt das mit einer Sortier-Routine. Ein Array wird mit zufälligen Ganzzahlen gefüllt und dann mit Hilfe von zwei verschachtelten Schleifen sortiert. Diese Technik heißt „modifiziertes Bubblesort (Blasensortierung)“.

Zuerst wird iOuter auf die letzte Zahl des Arrays gesetzt. Die innere Schleife nutzt iInner und vergleicht jede Zahl mit der nachfolgenden. Wenn die erste Zahl größer als die zweite ist, werden die zwei Zahlen vertauscht. Die zweite Zahl wird nun mit der dritten verglichen. Nach dem Ende der inneren Schleife ist sichergestellt, dass die größte Zahl im Array die letzte Position einnimmt.

Danach wird iOuter um 1 heruntergezählt. Die innere Schleife ignoriert diesmal die letzte Zahl des Arrays und vergleicht wiederum jede Zahl mit der ihr folgenden. Nach dem Ende der inneren Schleife ist die zweitgrößte Zahl an der zweitletzten Stelle im Array.

Mit jedem Wiederholungsschritt wird eine weitere Zahl an die richtige Stelle geschoben. Wenn keine Zahlen vertauscht werden, ist die Liste sortiert.

Listing 56. *Modifiziertes Bubblesort.*

```
Sub ExampleForNextSort
  Dim iEntry(10) As Integer
  Dim iOuter As Integer, iInner As Integer, iTemp As Integer
  Dim bSomethingChanged As Boolean  'True, wenn eine Änderung erfolgte

  ' Das Array wird mit Ganzzahlen zwischen -10 und 10 gefüllt.
  For iOuter = LBound(iEntry()) To Ubound(iEntry())
    iEntry(iOuter) = Int((20 * Rnd) -10)
  Next iOuter

  ' iOuter erhält nacheinander den Arrayindex von hinten nach vorne.
  For iOuter = UBound(iEntry()) To LBound(iEntry()) Step -1
    'Setzt voraus, dass das Array schon sortiert ist.
    'Prüft, ob es wirklich zutrifft.
    bSomethingChanged  = False
    For iInner = LBound(iEntry()) To iOuter-1
      If iEntry(iInner) > iEntry(iInner+1) Then
        iTemp = iEntry(iInner)
        iEntry(iInner) = iEntry(iInner+1)
        iEntry(iInner+1) = iTemp
        bSomethingChanged  = True
      End If
    Next iInner
    'Wenn das Array schon sortiert ist, wird die Schleife verlassen!
    If Not bSomethingChanged Then Exit For
  Next iOuter
  Dim s$
  For iOuter = LBound(iEntry()) To Ubound(iEntry())
    s = s & iOuter & " : " & iEntry(iOuter) & CHR$(10)
  Next iOuter
  MsgBox s, 0, "Sortiertes Array"
End Sub
```

3.8.12. Exit Sub und Exit Function

Mit der Anweisung Exit Sub wird eine Subroutine abrupt beendet, gleichermaßen eine Funktion durch Exit Function. Das Makro fährt mit der Anweisung fort, die der Anweisung folgt, mit der die Routine aufgerufen wurde. Die Exit-Anweisungen beenden nur die aktuell laufende Routine und gelten nur für die entsprechenden Typen, zum Beispiel können Sie Exit Sub nicht in einer Funktion verwenden.

3.9. Fehlerbehandlung mit On Error

Fehler fallen gewöhnlich in drei Kategorien – beim Kompilieren, zur Laufzeit sowie logische Fehler. Fehler beim Kompilieren sind üblicherweise Syntaxfehler wie zum Beispiel nicht vorhandene Anfüh-

rungszeichen, die das Kompilieren Ihres Makros verhindern. Mit Fehlern zur Kompilierungszeit ist am einfachsten umzugehen, weil sie sofort gefunden werden, denn die IDE zeigt Ihnen, welche Zeile das Problem verursacht hat. Laufzeitfehler werden ordentlich kompiliert, treten aber auf, wenn das Makro läuft. Wenn zum Beispiel durch eine Variable dividiert wird, die an einem bestimmten Punkt den Wert Null annimmt, wird ein Laufzeitfehler produziert. Der dritte Typ, logische Fehler, sind Irrtümer in der konzeptionellen Logik des Programms. Sie werden kompiliert, laufen fehlerlos, liefern aber die falschen Antworten. Das sind die schlimmsten Fehler, denn Sie müssen sie selbst finden – der Computer kann Ihnen dabei nicht helfen. Dieser Abschnitt handelt von Laufzeitfehlern: wie man mit ihnen umgeht und wie man sie behebt.

Eine Fehlerbehandlungsroutine ist ein Programmteil, der beim Auftritt eines Fehlers gestartet wird. Die Standard-Fehlerbehandlung zeigt eine Fehlermeldung und bricht das Makro ab. OOo Basic bietet einen Mechanismus, dieses Verhalten zu beeinflussen (s. Tabelle 21). Das erste Format, On Error Resume Next, veranlasst OOo, alle Fehler zu ignorieren: Was auch immer passiert, mach weiter und tu so, als wäre alles in Ordnung. Das zweite Format, On Error GoTo 0, deaktiviert die aktuelle Fehlerbehandlung. Ungeachtet der an späterer Stelle erläuterten Bereichsaspekte der Fehlerroutine betrachten Sie On Error GotTo 0 als Mittel, die Standardmethode der Fehlerbehandlung wiederherzustellen: Brich das Makro ab und zeige eine Fehlermeldung. Das letzte Format, On Error GoTo LabelName, erlaubt Ihnen, Code zu schreiben, der Fehler nach Ihren eigenen Wünschen behandelt. Sie erstellen einen „Error-Handler“.

Tabelle 21. *Unterstützte On Error ...-Formate.*

Format	Einsatzbereich
`On Error Resume Next`	Ignoriert Fehler und setzt das Makro mit der folgenden Zeile fort.
`On Error GoTo 0`	Bricht die aktuelle Fehlerbehandlung ab.
`On Error GoTo LabelName`	Verzweigt zur genannten Sprungmarke.

Wenn ein Fehler auftritt, wird der aktuell ausgeführte Code abgebrochen und die Kontrolle dem aktuellen Error-Handler übertragen. Den Error-Handlers stehen die Funktionen in Tabelle 22 zur Verfügung, um die Ursache und die Position des Fehlers herauszufinden. Visual Basic setzt ein Error-Objekt ein und unterstützt die Funktionen der Tabelle 22 nicht.

Tabelle 22. *Variablen und Funktionen im Zusammenhang mit Fehlerroutinen.*

Funktion	Verwendung
CVErr	Konvertiert einen Ausdruck in ein Error-Objekt.
Erl	Nummer der Zeile, in der der letzte Fehler auftrat.
Err	Nummer des zuletzt aufgetretenen Fehlers.
Error	Fehlermeldung des zuletzt aufgetretenen Fehlers.

Alle Error-Handler müssen in einer Routine definiert werden und gehören somit zu eben dieser Subroutine oder Funktion. Wenn ein Fehler auftritt, arbeitet sich OOo Basic rückwärts durch den Aufrufstapel, bis ein Error-Handler gefunden ist. Falls keiner zu finden ist, wird der Standard-Handler eingesetzt, der eine Fehlermeldung ausgibt und das Programm abbricht. Die Fehlerinformation Erl zeigt die Nummer der Zeile in der aktuellen Routine an, die den Fehler verursacht hat. Wenn zum Beispiel die aktuelle Routine in der Zeile 34 die Funktion b() aufruft und ein Fehler innerhalb von b() auftritt, wird ein Fehler gemeldet, der in der Zeile 34 aufgetreten ist. Listing 57 enthält dafür ein Beispiel, und Bild 36 zeigt den Aufrufstapel. Ein weiteres Beispiel finden Sie in Listing 60.

Listing 57. *Eigener Error-Handler.*

```
  x = x + 1                 'Angenommen, dies ist Zeile 33
  Call b()                  'Zeile 34:
                            'Fehler in b() oder in einer von b() aufgerufenen Routine
  Exit Sub                  'Abbruch der Subroutine
ErrorHandler:               'Kein weiterer Error-Handler zwischen hier und dem Fehler
  Print "Fehler in der Zeile " & Erl  'Gibt die Zeilennummer 34 aus
```

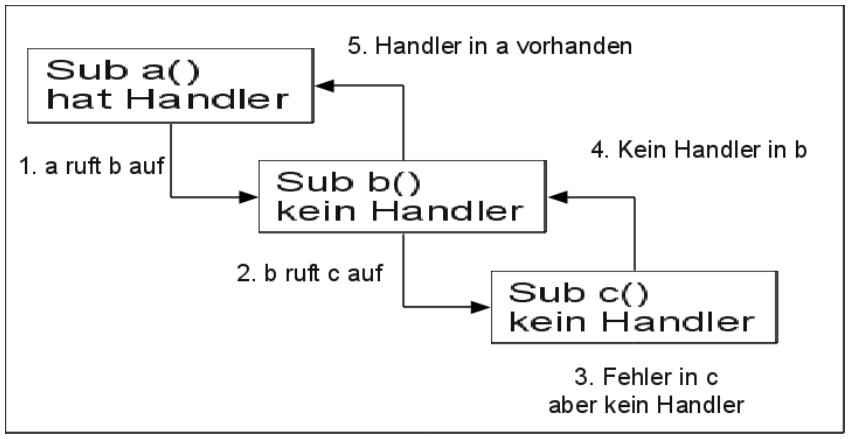


Bild 36. *Der Weg durch den Aufrufstapel auf der Suche nach einem Error-Handler.*

Sie können keine Fehler abfangen, die in einer DLL auftreten. Prüfen Sie stattdessen den Rückgabecode der aufgerufenen DLL.

3.9.1. Fehler ignorieren mit On Error Resume Next

Fehlerbehandlung bedeutet in manchen Fällen Fehlerignorierung. Die Anweisung On Error Resume zwingt OOo Basic im Falle eines Standardfehlers, den Fehler zu ignorieren und mit der nächsten Zeile im Makro weiterzumachen (s. Listing 58). Die Fehlerinformation wird gelöscht, so dass es nach der verursachenden Anweisung nicht mehr möglich ist, zu prüfen, ob ein Fehler aufgetreten ist.

Listing 58. *Mit der Anweisung Resume Next ist der Fehler gelöscht.*

```
Private zero%
sub ExampleErrorResumeNext
  On Error Resume Next
  Print 1/Zero%
  If Err <> 0 Then Print Error$ & " in der Zeile " & Erl 'Err wurde gelöscht
End Sub
```

3.9.2. Mit On Error GoTo 0 einen Error-Handler ausschalten

Verwenden Sie On Error GoTo 0, um einen definierten Error-Handler auszuschalten, üblicherweise innerhalb eines Error-Handlers oder nach dem Code, für den wirken soll. Wenn ein Fehler innerhalb eines Error-Handlers auftritt, wird er nicht abgefangen und das Makro bricht ab.

Listing 59. *DerError-Handler wird mit der Anweisung On Error GoTo 0 ausgeschaltet.*

```
Private zero%
sub ExampleErrorResumeNext
  On Error Resume Next
  Print 1/Zero%
  On Error GoTo 0
  ...
End Sub
```

In einigen Versionen von Visual Basic wird auch On Error GoTo -1 als äquivalent zu On Error GoTo 0 akzeptiert.

3.9.3. Mit On Error GoTo Label einen eigenen Error-Handler definieren

Ihren eigenen Error-Handler richten Sie mit On Error GoTo Label ein. In OOo Basic definieren Sie ein Label, indem Sie in einer gesonderten Zeile Text eingeben, dem ein Doppelpunkt folgt. Zeilenlabel dürfen im Makro unter demselben Namen mehrfach vorkommen, innerhalb einer Routine müssen sie aber eindeutig sein. Somit können Fehlerbehandlungsroutinen immer gleich benannt werden, statt für jeden Error-Handler einen ganz eigenen Namen finden zu müssen (s. Listing 60 und Bild 37). Im Falle eines Fehlers wird der Programmfluss zu dem Label verzweigt.

Listing 60. *Fehlerbehandlung.*

```
Private zero%
Private error_s$
Sub ExampleJumpErrorHandler
  On Error GoTo ExErrorHandler
  JumpError1
  JumpError2
  Print 1/Zero%
  MsgBox error_s, 0, "Sprung zum Error-Handler"
  Exit Sub
ExErrorHandler:
  error_s = error_s & "Fehler in MainJumpErrorHandler in der Zeile " & Erl() &_
        " : " & Error() & CHR$(10)
  Resume Next
End Sub
Sub JumpError1
  REM Bewirkt einen Sprung zum Handler in ExampleJumpErrorHandler.
  REM Der Haupt-Error-Handler zeigt an, dass die Fehlerposition
  REM beim Aufruf von JumpError1 liegt und nicht innerhalb von JumpError1.
  Print 1/zero%
  error_s = error_s & "Hey, ich bin in JumpError1" & CHR$(10)
End Sub

Sub JumpError2
  On Error GoTo ExErrorHandler
  Print 1/zero%
  Exit Sub
ExErrorHandler:
  error_s = error_s & "Fehler in JumpError2 in der Zeile " & Erl() &_
        " : " & Error() & CHR$(10)
  Resume Next
End Sub
```

Bild 37. *Der zuletzt eingerichtete Error-Handler wird genutzt.*

Eine Routine kann mehrmals die Anweisung On Error enthalten. Jede dieser Anweisungen kann Fehler auf andere Weise behandeln. Die Error-Handler in Listing 60 benutzen alle Resume Next, ignorieren damit den Fehler und führen das Programm mit der auf den Fehler folgenden Zeile fort. Mit mul-

tiplen Error-Handlern ist es möglich, im Falle eines Fehlers Codebereiche zu überspringen (s. Listing 61).

Tipp Im Hilfetext von OOo 3.20 wird immer noch fälschlich behauptet, dass die Fehlerbehandlung am Anfang einer Routine eingefügt werden müsse.

Listing 61. *Überspringen von Codebereichen, wenn ein Fehler auftritt.*

```
  On Error GoTo PropertiesDone  'Ignoriert alle Fehler in diesem Bereich.
  a() = getProperties()         'Wenn getProperties() nicht funktioniert, dann wird ein
  DisplayStuff(a(), "Eigenschaften")  'Fehler verhindern, dass diese Zeile erreicht
                                      ' wird.
PropertiesDone:
  On Error GoTo MethodsDone        'Ignoriert alle Fehler in diesem Bereich.
  a() = getMethods()
  DisplayStuff(a(), "Methoden")
MethodsDone:
  On Error Goto 0                  'Schaltet alle aktuellen Error-Handler ab.
```

Wenn Sie einen Error-Handler schreiben, müssen Sie eine Entscheidung treffen, wie Sie die Fehler behandeln wollen. Mit Hilfe der Funktionen in der Tabelle 22 werden Fehler diagnostiziert und Fehlermeldungen ausgegeben oder in Logdateien geschrieben. Sie müssen aber auch den Programmablauf kontrollieren. Es folgen einige Leitfäden zur Fehlerbehandlung:

Verlassen Sie die Subroutine oder Funktion mit Exit Sub oder Exit Function.

Lassen Sie das Makro weiterlaufen und ignorieren Sie den Fehler (s. Listing 61).

Setzen Sie mit Resume Next das Makro mit der dem Fehler folgenden Zeile fort (s. Listing 62 und Bild 38).

Wiederholen Sie mit Resume die Ausführung der den Fehler erzeugenden Anweisung. Wenn das Problem aber nicht behoben ist, wird der Fehler wieder auftreten, mit der Folge einer Endlosschleife.

Setzen Sie mit Resume LabelName das Programm an einer benannten Position fort.

Listing 62. *Error-Handler mit Resume Next.*

```
Sub ExampleResumeHandler
  Dim s$, z%
  On Error GoTo Handler1                'Gibt eine Meldung aus, macht bei Spot1 weiter.
  s = "(0) 1/z = " & 1/z & CHR$(10)     'Division durch Null, also Sprung zu Handler1.
Spot1:                                  'Von Handler1 hierher gekommen.
  On Error GoTo Handler2                'Handler2 verwendet Resume.
  s = s & "(1) 1/z = "&1/z & CHR$(10)   'Fehler beim ersten Mal, korrekt beim zweiten
Mal.
  On Error GoTo Handler3                'Handler3 nimmt mit der nächsten Zeile wieder
auf.
  Z = 0                                 'Ermöglicht eine weitere Division durch Null.
  s = s & "(2) 1/z = "&1/z & CHR$(10)   'Fehler und Sprung zu Handler3
 MsgBox s, 0, "Error-Handler mit Resume"
 Exit Sub
Handler1:
  s = s & "Handler1 aufgerufen von der Zeile " & Erl() & CHR$(10)
  Resume Spot1
Handler2:
  s = s & "Handler2 aufgerufen von der Zeile " & Erl() & CHR$(10)
  z = 1   'Behebt den Fehler und führt die Zeile noch einmal aus.
  Resume
Handler3:
  s = s & "Handler3 aufgerufen von der Zeile " & Erl() & CHR$(10)
```

```
    Resume Next
End Sub
```

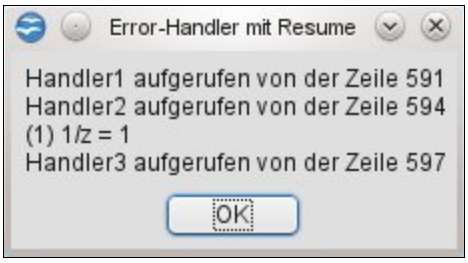


Bild 38. *Der jeweils zuletzt deklarierte Error-Handler wird verwendet.*

Tipp	In einem Error-Handler auftretende Fehler werden nicht behandelt. Das Makro bricht einfach ab.

3.9.4. Error-Handler – wozu?

Wenn ich ein Makro ausführe und es abstürzt, verstehe ich für gewöhnlich die manchmal kryptischen Fehlermeldungen und erkenne, wie ich damit umgehen muss. Wenn andere meine Makros ausführen und Fehler auftreten, informieren sie mich normalerweise darüber, weil sie nicht wissen, wie sie sie behandeln sollen. Das ist dann ein guter Indikator dafür, dass keine angemessene Fehlerbehandlung eingebaut habe.

Sie brauchen nicht für jede Routine einen Error-Handler. Wenn die aktuelle Routine keinen hat, aber die sie aufrufende Routine, wird deren Error-Handler aktiviert. Gesetzt den Fall, Sub1 hat einen Error-Handler und ruft Sub2 auf, das keinen hat. Wenn in Sub2 ein Fehler auftritt, wird der Error-Handler in Sub1 aktiviert.

Wenn Sie einen Error-Handler einsetzen, bestimmen Sie, wie und wann der Nutzer über einen Fehler informiert wird. Es gibt auch andere Gründe, einen Error-Handler zu nutzen, als die Kontrolle über die Fehlermeldungen auszuüben. Mit Bedacht eingesetzt können Error-Handler den Umfang Ihres Makrocodes reduzieren. Nehmen Sie einmal die mathematischen Operationen. Es ist lästig, jede mathematische Operation vor ihrem Gebrauch zu testen.

```
If x <> 0 Then y = z / x
If x > 0 Then y = Log(x)
If i% < 32767 Then i% = i% + 1
```

Trotz meines paranoiden Codes, die Argumente zu testen, könnte ich einen numerischen Überlauf produzieren. Schlimmer noch, nichts geschieht im Falle eines Fehlers, das Makro wird normal weiter ausgeführt. Manchmal können Sie auch gar nichts testen, um den Fehler zu vermeiden. Vor OOo 2.0 gab zum Beispiel die Funktion DimArray ein ungültiges leeres Array zurück. Die Funktionen LBound und UBound generieren Ausnahmefehler bei diesen ungültigen leeren Arrays. Mit einer passenden Fehlerbehandlung werden LBound und UBound auch im Falle eines Fehlers sicher eingesetzt. Betrachten Sie folgende Fälle:

- Das Argument ist kein Array.
- In einem leeren Array ist zum Beispiel UBound < LBound: -1 und 0.
- Es gibt kein Problem, wenn das Array kein ungültiges leeres Array ist.
- Sollte die optionale Dimensionierung berücksichtigt werden?

Der Code in Listing 63 zeigt einen einfachen Error-Handler, der schlicht Fehler ignorieren kann. Die Funktion gibt True zurück, wenn die untere Dimensionsgrenze kleiner als oder gleich groß wie die obere ist – mit anderen Worten, ob das Array Daten enthält. Wenn ein Fehler auftritt, sei es, dass das Argument kein Array oder ein ungültiges leeres Array ist, wird die Zeile nicht bis zum Ende ausgeführt, so dass es gar keine Wertzuweisung gibt. Wenn die Zuweisung nicht erfolgt, wird der originale

Standardwert False zurückgegeben. Das ist dann nämlich die korrekte Antwort. Dem Leser bleibt es als Übung, eine sichere Routine für die obere und untere Grenze zu schreiben – allerdings sind die sicheren Versionen seit OOo 2.0 nicht mehr nötig, weil die Funktionen UBound und LBound gefixt sind.

Listing 63. *Ermittelt, ob ein Array Daten enthält.*

```
Sub ExampleArrayHasStuff
  Dim a(), b(3), v
  Print ArrayHasStuff(a())         'False, weil leer
  Print ArrayHasStuff(v)           'False, kein Array, der Error-Handler greift ein.
  Print ArrayHasStuff(DimArray()) 'False, ungültiges Array, der Error-Handler greift
                                   'ein.
  Print ArrayHasStuff(DimArray(3))'True,  nicht leer
  Print ArrayHasStuff(b())         'True,  nicht leer
End Sub

Function ArrayHasStuff(v) As Boolean
  REM Der Standardwert für Boolean ist False, also ist die Standardantwort False.
  REM Wenn ein Fehler auftritt, wird die Anweisung nicht beendet!
  REM Dies ist eine guter Fall für On Error Resume Next
  On Error Resume Next
  ArrayHasStuff = CBool(LBound(v) <= UBound(v))
End Function
```

Ein Error-Handler kann auch interaktiv sein. Der Code in Listing 64 versucht, eine nicht existierende Datei an eine nicht existierende Adresse zu kopieren. Keine Frage, es wird ein Fehler generiert. Eine Fehlermeldung wird ausgegeben mit der Frage an den Nutzer, ob die Kopie noch einmal versucht werden soll. Der Nutzer hat dadurch die Gelegenheit, Fehler zu korrigieren und weiterzumachen.

Listing 64. *Kopiert eine Datei.*

```
Sub ExampleCopyAFile()
  CopyAFile("/ich/bin/nicht/vorhanden.txt", "/ich/auch/nicht.txt")
End Sub

Sub CopyAFile(Src$, Dest$)
  On Error GoTo BadCopy:     'Richtet den Error-Handler ein.

  FileCopy(Src$, Dest$)      'Generiert den Fehler.

AllDone:                     'Ohne Fehler geht es hier weiter.
  Exit Sub                   'Rückkehr vom Handler zum Label AllDone.

BadCopy:                     'Ausgabe eines Fehlerdialogs.
  Dim rc%                    'Antwort auf die Frage nach einem neuen Versuch.
  rc% = MsgBox("Missglückte Kopie von " & Src$ & " zu " & Dest$ & CHR$(10) &_
         "Ursache: "&CHR$(10) & Error() & " In der Zeile " & Erl & CHR$(10) &_
         "Neuer Versuch?", (4 OR 32), "Fehler beim Kopieren")
  If rc% = 6 Then    'Ja, neuer Versuch
    Resume
  End If
  If rc% = 7 Then    'Nein, Sprung zum Label AllDone.
    Resume AllDone
  End If
End Sub
```

3.10. Fazit

Zum Aufbau eines nennenswerten OOo-Makroprogramms ist das Verständnis der Syntax von OOo Basic unerlässlich. Dieses Kapitel deckte die wesentlichen Elemente ab:

- Die Syntax des Makros bestimmt den gültigen oder ungültigen Aufbau.
- Die Logik eines Makros bestimmt, was das Makro tut.
- Die Ablaufsteuerung lenkt die Abfolge der Anweisungen während der Laufzeit.
- Die Fehlerbehandlung lenkt das Makro, wenn es etwas Unvorhergesehenes tut.

Ein vollständiges, gut gebautes Makro, das eine wesentliche Funktion erbringt, wird höchstwahrscheinlich alle diese Merkmale der OOo-Programmierung erfüllen. Welche spezifischen Elemente aus dem OOo-Fundus für den Aufbau eines spezifischen Programms genutzt werden, hängt von der Anwendung ab, vom angestrebten logischen Verhaltens des Programms und vom besten Ermessen des Programmierers. Ein wesentlicher Teil des erfolgreichen Programmierens besteht darin, Erfahrung zu gewinnen, um die Ideen dieses Kapitels möglichst wirkungsvoll anzuwenden.

4. Numerische Routinen

In diesem Kapitel werden die Subroutinen und Funktionen vorgestellt, die OpenOffice.org Basic im Zusammenhang mit Zahlen bereitstellt – eingeschlossen mathematische Funktionen, Konversionsroutinen, Formatierung von Zahlen als String sowie Zufallszahlen. Außerdem werden in diesem Kapitel auch alternative Zahlensysteme behandelt.

Numerische Subroutinen und Funktionen sind Routinen, die mathematische Operationen durchführen. Wenn Sie schon einmal eine Tabellenkalkulation genutzt haben, sind Sie möglicherweise schon vertraut mit mathematischen Funktionen wie „SUMME“, mit der Gruppen von Zahlen addiert werden, oder vielleicht sogar mit „IKV“, womit der interne Zinsfuß einer Investition berechnet wird. Die von OOo Basic bereitgestellten numerischen Routinen (s. Tabelle 23) sind in ihrer Form einfacher und arbeiten normalerweise mit nur einem oder zwei Argumenten und nicht mit ganzen Zahlengruppen.

Tabelle 23. *Mit Zahlen und numerischen Operationen verbundene Subroutinen und Funktionen.*

Funktion	Beschreibung
ABS(Zahl)	Absolutwert der Zahl.
ATN(Zahl)	Der Winkel, in Radiant, dessen Tangens die angegebene Zahl ist, im Bereich -Pi/2 bis Pi/2.
CByte(Ausdruck)	Rundet den numerischen oder String-Ausdruck zu einem Byte.
CCur(Ausdruck)	Konvertiert den Ausdruck zum Typ Currency.
CDbl(Ausdruck)	Konvertiert den numerischen oder String-Ausdruck zum Typ Double.
CDec(Ausdruck)	Konvertiert den numerischen oder String-Ausdruck zum Typ Decimal (nur unter Windows).
CInt(Ausdruck)	Rundet den numerischen oder String-Ausdruck zur nächsten Ganzzahl.
CLng(Ausdruck)	Rundet den numerischen oder String-Ausdruck zum nächsten Long-Wert.
COS(Zahl)	Kosinus des angegebenen Winkels (in Radiant).
CSng(Ausdruck)	Konvertiert den numerischen oder String-Ausdruck zum Typ Single.
Exp(Zahl)	Potenziert die Basis des natürlichen Logarithmus um die angegebene Zahl.
Fix(Zahl)	Schneidet den Dezimalteil der Zahl ab.
Format(Zahl, Format)	Benutzerdefinierte Zahlenformatierung, s. Kapitel 6, „String-Routinen“.
Hex(Zahl)	Gibt die hexadezimale Darstellung der Zahl als String zurück.
Int(Zahl)	Rundet die Zahl in Richtung minus Unendlich.
Log(Zahl)	Der natürliche Logarithmus der Zahl. In Visual Basic .NET kann diese Methode überladen werden, um entweder den natürlichen Logarithmus (Basis e) oder den Logarithmus zu einer bestimmten Basis zurückzugeben.
Oct(Zahl)	Gibt die oktale Darstellung der Zahl als String zurück.
Randomize(Zahl)	Initialisiert den Zufallsgenerator. Wird die Zahl weggelassen, wird der aktuelle Wert des Systemzeitgebers verwendet.
Rnd	Zufallszahl als Double zwischen 0 und 1.
Sgn(Zahl)	Ganzzahl, die dem Vorzeichen der Zahl entspricht (-1, 0, 1).
SIN(Zahl)	Sinus des angegebenen Winkels (in Radiant).
Sqr(Zahl)	Quadratwurzel der Zahl.
Str(Zahl)	Konvertiert die Zahl zu einem String (ohne Lokalisierung).
TAN(Zahl)	Tangens des angegebenen Winkels (in Radiant).
Val(Text)	Konvertiert den String zu einer Zahl vom Typ Double. Sehr tolerant bei nicht-numerischem Text.

Die mathematischen Funktionen dieses Kapitels sind gut bekannt und vertraut bei Mathematikern, Ingenieuren und anderen, die jede Gelegenheit nutzen, im täglichen Leben Rechenformeln einzusetzen. Wenn Sie nicht dazugehören – wenn Sie den Rechenschieber vielleicht *nicht* für die coolste Erfindung seit dem Schnittbrot halten –, dann geraten Sie aber bitte nicht in Panik, wenn die Beschreibung naturgemäß ins Mathematische übergeht. Ich bemühe mich, die Information verständlich zu halten und gleichzeitig denen tiefer gehende Informationen zu bieten, die sie benötigen. Die Routinen sind thematisch in Unterkapitel gruppiert, so dass Sie die Unterkapitel überschlagen können, von denen Sie wissen, dass Sie sie nicht brauchen.

Die numerischen Routinen arbeiten mit numerischen Daten. OOo Basic versucht vor der eigentlichen Berechnung, die Argumente in den passenden Typ zu konvertieren. Es ist aber sicherer, mit den in diesem Kapitel vorgestellten Konvertierungsfunktionen die Datentypen explizit zu konvertieren, als sich auf das Standardverfahren zu verlassen. Wenn ein ganzzahliges Argument erwartet wird und eine Fließkommazahl vorliegt, wird die Zahl standardgemäß gerundet. Im Beispiel „16.8 MOD 7“ wird 16.8 vor der Berechnung zu 17 aufgerundet. Bei dem Operator zur ganzzahligen Division werden die Operanden jedoch beschnitten. Im Beispiel „Print 4 \ 0.999“ wird 0.999 zu 0 beschnitten mit der Folge eines Division-durch-Null-Fehlers.

Tipp Tabelle 23 enthält Subroutinen und Funktionen, keine Operatoren wie MOD, + oder \. Operatoren finden Sie im Kapitel 3, Sprachstrukturen.

4.1. Trigonometrische Funktionen

Trigonometrie ist die Lehre von den Eigenschaften der Dreiecke und trigonometrischen Funktionen und ihren Anwendungen. Die Behandlung trigonometrischer Funktionen bezieht sich normalerweise auf Dreiecke mit einem 90-Grad-Winkel, so genannte rechtwinklige Dreiecke (s. Bild 39). Es gibt einen Satz fester Beziehungen zwischen den trigonometrischen Funktionen, den Seitenlängen eines rechtwinkligen Dreiecks und den entsprechenden Winkeln. Wenn Sie diese Beziehungen kennen, können Sie die trigonometrischen Funktionen benutzen, um trigonometrische Augaben zu lösen.

In der Praxis benötigen Sie Trigonometrie, wenn Sie den Abstand zu einem Stab oder Turm bekannter Höhe schätzen wollen. Sie messen den Beobachtungswinkel vom Erdboden zur Spitze des Stabs, und da die Höhe des Stabs bekannt ist, errechnet sich Ihr Abstand zum Stab aus der Stabhöhe dividiert durch den Tangens des gemessenen Winkels. Dieses Verfahren eignet sich für Golf, Segeln oder Wandern, immer da, wo Sie den Abstand zu einem Fixpunkt bestimmen wollen (die Golffahne zum Beispiel oder ein Funkturm).

Die grundlegenden trigonometrischen Funktionen sind Sinus, Kosinus und Tangens. Jede ist definiert als das Verhältnis zweier Seiten eines rechtwinkligen Dreiecks. Die Werte dieser Funktionen für jede Größe des Winkels x entsprechen den Quotienten der Seiten eines rechtwinkligen Dreiecks mit diesem Winkel x. Für einen gegebenen Winkel bestimmen die trigonometrischen Funktionen die Seitenlängen eines rechtwinkligen Dreiecks. Gleichermaßen kann man, wenn die Längen von zwei beliebigen Seiten eines rechtwinkligen Dreiecks bekannt sind, mit Hilfe der inversen trigonometrischen Funktionen die Größe des Winkels berechnen.

OOo Basic benutzt Radiant als Maßeinheit für Winkel. Die meisten wissenschaftlichen Laien denken jedoch in der Einheit Grad. Ein Winkel von 90 Grad, so die Ecke eines Quadrats, hat Pi/2 Radiant.

Tipp Die eingebaute Konstane Pi hat den Wert von etwa 3,1415926535897932385. Pi ist eine fundamentale Konstante, die in wissenschaftlichen Berechnungen weit verbreitet ist. Sie ist definiert als Verhältnis des Umfangs eines Kreises zu seinem Durchmesser. Die Winkelsumme im Dreieck – natürlich auch im rechtwinkligen Dreieck – ist 180 Grad oder Pi Radiant. Eine gewaltige Menge eleganter und praktischer mathematischer Methoden beruht auf dieser Beziehung zwischen Dreieck und Kreis. Alle Beschreibungen periodischer Bewegung fußen auf dieser Grundlage. Es gilt:Trigonometrie ist eine fundamentale und äußerst nützliche Gruppe mathematischer Hilfsmittel.

Mit Hilfe der Beziehung zwischen Grad (engl. degree) und Radiant (engl. radian) ist es einfach, zwischen beiden Winkelmaßeinheiten zu konvertieren.

```
degrees = (radians * 180) / Pi
radians = (degrees * Pi) / 180
```

Um den Sinus eines Winkels von 45 Grad zu berechnen, müssen Sie erst den Winkel in Radiant umrechnen. Das geht so:

```
radians = (45° * Pi) / 180 = Pi / 4 = 3,141592654 / 4 = 0,785398163398
```

Sie können diesen Wert direkt in der trigonometrischen Funktion SIN verwenden.

```
Print SIN(0.785398163398)    ' 0,707106781188
```

Zur Ermittlung des Winkels mit dem Tangens 0,577350269189 dient die Funktion Arkustangens. Der Rückgabewert ist in der Einheit Radiant, muss also in Grad zurückgerechnet werden.

```
Print ATN(0.577350269189) * 180 / Pi   '29,9999999999731
```

Tipp Rundungsfehler beeinträchtigen diese Beispiele, wie wir später noch sehen. Mit unendlicher Genauigkeit würde das vorige Beispiel eine Antwort von 30 Grad ergeben anstatt 29,9999999999731.

Die Antwort ist nahe 30 Grad. Das Dreieck im Bild 39 hilft uns, die trigonometrischen Funktionen zu erläutern.

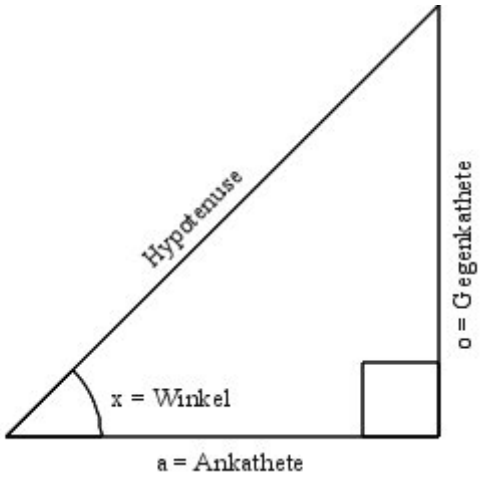


Bild 39. *Ein rechtwinkliges Dreieck hat einen 90-Grad-Winkel.*

Tabelle 24. *Trigonometrische Funktionen in OOo Basic.*

OOo Basic	VB	VB .NET	Rückgabewert
ATN	ATN	Math.Atan	Der Winkel, in Radiant, dessen Tangens die angegebene Zahl ist, im Bereich -Pi/2 bis Pi/2.
COS	COS	Math.Cos	Kosinus des angegebenen Winkels (in Radiant).
SIN	SIN	Math.Sin	Sinus des angegebenen Winkels (in Radiant).
TAN	TAN	Math.Tan	Tangens des angegebenen Winkels (in Radiant).

Tabelle 24 zeigt die von OOo Basic unterstützten trigonometrischen Funktionen, bildlich dargestellt in dem rechtwinkligen Dreieck im Bild 39. Sie erwarten einen Ausdruck als einziges Argument, dessen Wert vor der Berechnung in eine Zahl doppelter Genauigkeit (Typ Double) konvertiert wird.

COS(x) = Ankathete / Hypotenuse

SIN(x) = Gegenkathete / Hypotenuse

TAN(x) = Gegenkathete / Ankathete = SIN(x) / COS(x)

ATN(Gegenkathete / Ankathete) = x

Der Code in Listing 65 löst eine Reihe von geometrischen Aufgaben mit Hilfe der trigonometrischen Funktionen. Der Code geht von einem rechtwinkligen Dreieck aus (s. Bild 39) mit der Länge 3 für die Gegenkathete und der Länge 4 für die Ankathete. Der Tangens wird leicht mit 3/4 errechnet, und die Funktion ATN berechnet die Größe des Winkels. Dazu kommen noch ein paar andere Berechnungen, wie die Bestimmung der Länge der Hypotenuse mit Hilfe der Funktionen SIN einerseits und COS andererseits. Siehe auch Bild 40

Listing 65. *Trigonometrisches Beispiel*

```
Sub ExampleTrigonmetric
  Dim OppositeLeg As Double      REM Gegenkathete
  Dim AdjacentLeg As Double      REM Ankathete
  Dim Hypotenuse As Double
  Dim AngleInRadians As Double   REM Winkel in Radiant
  Dim AngleInDegrees As Double   REM Winkel in Grad
  Dim s As String
  OppositeLeg = 3
  AdjacentLeg = 4
  AngleInRadians = ATN(3/4)
  AngleInDegrees = AngleInRadians * 180 / Pi
  s = "Gegenkathete = " & OppositeLeg & CHR$(10) &_
      "Ankathete = " & AdjacentLeg & CHR$(10) &_
      "Winkel in Grad von ATN = " & AngleInDegrees & CHR$(10) &_
      "Hypotenuse von COS = " & AdjacentLeg/COS(AngleInRadians) & CHR$(10) &_
      "Hypotenuse von SIN = " & OppositeLeg/SIN(AngleInRadians) & CHR$(10) &_
      "Gegenkathete von TAN = " & AdjacentLeg * TAN(AngleInRadians)
  MsgBox s, 0, "Trigonometrische Funktionen"
End Sub
```

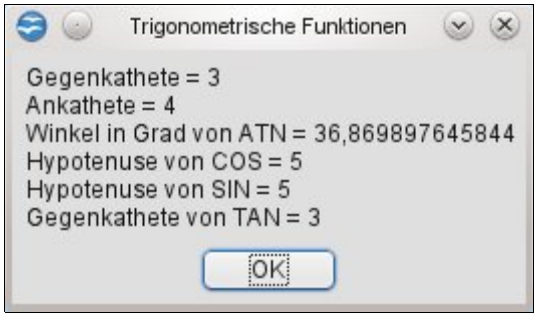


Bild 40. *Mit den trigonometrischen Funktionen werden Dreiecksaufgaben gelöst.*

4.2. Rundungsfehler und Genauigkeit

Ein Computer oder ein Taschenrechner führt numerische Rechnungen mit nur einer begrenzten Anzahl von Ziffern aus. So entstehen Rundungsfehler. Dieses Problem besteht nicht bei Ganzzahlen. Die Zahl 1/3 wird in dezimaler Form zu 0,33333333, aber eigentlich gehörte eine unendliche Anzahl von Dreien hinter das Dezimalkomma. Mit vier Genauigkeitsstellen wird die Zahl als 0,3333 geschrieben. Das ist ungenau in der Darstellung und den Rechenergebnissen.

```
1/3 + 1/3 + 1/3 = 3/3 = 1          'Das korrekte Ergebnis ist 1
0,3333 + 0,3333 + 0,3333 = 0,9999  'Das Ergebnis mit begrenzter Genauigkeit,
                                   'ein wenig daneben
```

Das einfache Makro in Listing 66 zeigt das Problem. Der Wert 0,2 wird so oft zur Variablen *num* addiert, bis sie den Wert 5 erreicht. Wenn wir unendliche Genauigkeit hätten oder wenn der Computer

die Zahl 0,2 intern exakt repräsentierte, wäre die Schleife beendet, wenn die Variable *num* den Wert 5 erhält. Die Variable wird jedoch niemals genau gleich dem Wert 5 sein, daher wird die Schleife nie beendet. Der Wert 5 wird zwar ausgegeben, aber nur, weil die Anweisung Print den Wert 4,9999999 für die Ausgabe auf 5 rundet.

Listing 66. *Rundungsfehler und begrenzte Genauigkeit verhindern das Schleifenende.*

```
Dim num As Single
Do
  num = num + 0.2
  If num > 4.5 Then Print num  'Ausgabe: 4,6, 4,8, 5, 5,199999...
Loop Until num = 5.0
Print num
```

Im Computer wirken komplexe Rundungsalgorithmen im Bestreben, die Auswirkungen der begrenzten Genauigkeit zu reduzieren – begrenzte Genauigkeit bedeutet, dass zur Darstellung einer Zahl nur eine begrenzte Menge an Ziffern und Speicherplatz zur Verfügung steht. Trotz dieser unterstützenden Maßnahmen zeigt Listing 66 deutlich, dass die internen komplexen Rundungsalgorithmen das Problem nicht beheben. Wenn Sie zwei Fließkommazahlen auf Gleichheit prüfen, ist es sicherer, sie mit einem Wertebereich zu vergleichen. Der Code in Listing 67 stoppt die Schleife, wenn die Variable größer als oder gleich 5 ist.

Listing 67. *Vermeidung von Rundungsfehlern durch >= (größer als oder gleich).*

```
Dim num As Single
Do
  num = num + 0.2
Loop Until num >= 5.0
Print num  '5,199999
```

Der Code in Listing 67 funktioniert irgendwie, aber Sie wollen möglicherweise, dass die Schleife endet, wenn die Variable *num* den Wert 4,9999999 hat, und nicht bei 5,1999999. Das kann man erreichen, indem man prüft, ob die zwei Zahlen nahe beieinander anstatt gleich sind. Die große Frage dabei ist, wie nahe die zwei Zahlen einander sein müssen, dass Sie als gleich gewertet werden. Aufgrund Ihrer Kenntnis der vorliegenden Aufgabe können Sie normalerweise eine einfache Schätzung vornehmen. Variablen vom Typ Single können etwa 8 genaue Stellen enthalten, die vom Typ Double etwa 16. Versuchen Sie nicht, von den Variablen eine höhere Genauigkeit zu erwarten, als sie leisten können. Der Code in Listing 67 verwendet Variablen einfacher Genauigkeit (Single), womit Sie eine Genauigkeit von ungefähr sieben Stellen erwarten können. Der Code in Listing 68 gibt den Unterschied zwischen 5 und *num* aus – beachten Sie, dass etwa sechs Stellen korrekt sind.

Listing 68. *Vergleich der Variablen mit einem Wertebereich.*

```
Dim num As Single
Do
  num = num + 0.2
Loop Until 4.99999 < num AND num < 5.00001
Print 5 - num  '4,76837158203125E-07  = 0,000004768371582031 25
```

Die Funktion ABS gibt den Absolutwert einer Zahl zurück. Mit ihrer Hilfe vereinfachen Sie den Prüfvorgang, ob eine Zahl sehr nahe bei einer anderen ist.

```
If ABS(num - 5) < 0.00001 Then
```

Mit ABS und Subtraktion finden Sie heraus, wie nahe zwei Zahlen einander sind, aber das mag nicht ausreichend sein. Das Licht bewegt sich zum Beispiel 299.792.458 Meter pro Sekunde. Diese Zahl hat 9 Ziffern. Eine Zahl mit einfacher Genauigkeit ist nur bis zu etwa sieben Stellen genau. Siehe Listing 69.

Listing 69. *Variablen vom Typ Single sind nur auf sieben oder acht Stellen genau.*

```
Dim c1 As Single  'In der Wissenschaft wird der Buchstabe c als Zeichen
Dim c2 As Single  'für die Lichtgeschwindigkeit verwendet.
c1 = 299792458    'Lichtgeschwindigkeit in Metern pro Sekunde: 9 Stellen
c2 = c1 + 16      'Addition von 16 zur Lichtgeschwindigkeit
If c1 = c2 Then   'Beide sind gleich, weil nur die ersten sieben
  Print "Gleich"  'oder acht Stellen signifikant sind.
End If
```

Der Code in Listing 69 addiert zur Lichtgeschwindigkeit 16 hinzu, doch das ändert den Wert nicht, und zwar, weil nur die ersten sieben oder acht Ziffern signifikant sind. Der Code in Listing 70 verwendet eine Zahl, die zwar in der Größenordnung kleiner ist, aber dieselbe Anzahl an Ziffern hat. Die Addition von 1 würde eine signifikante Stelle ändern, aber die Addition einer viel kleineren Zahl führt wiederum dazu, dass die Zahlen gleich sind.

Listing 70. *Variablen vom Typ Single sind nur auf sieben oder acht Stellen genau.*

```
Dim c1 As Single   'In der Wissenschaft wird der Buchstabe c als Zeichen
Dim c2 As Single   'für die Lichtgeschwindigkeit verwendet.
c1 = 299.792458    'Dies sind neun Ziffern, aber es ist nicht die Lichtgeschwindigkeit
c2 = c1 + .0000016 'Man muss eine kleine Zahl addieren, damit beide gleich bleiben
If c1 = c2 Then    'Beide sind gleich, weil nur die ersten sieben
  Print "Equal"    'oder acht Stellen signifikant sind.
End If
```

Fließkommazahlen können in ihrer Größe sehr unterschiedlich sein – das betrifft die reine Größe der Zahl auf der Zahlenskala, aber nur unwesentlich die Anzahl an relevanten Stellen. Beim Test, ob zwei Zahlen ungefähr gleich sind, können sich große Zahlen um einen größeren Betrag unterscheiden als kleine Zahlen. Die größte akzeptierte Differenz hängt von der reinen Zahlengöße ab, Mathematiker nennen das den „relativen Fehler“. Siehe Listing 71.

Listing 71. *Vergleich zweier Zahlen.*

```
REM Die Zahl n1 ist von primärem Interesse
REM n2 wird mit n1 relativ verglichen
REM rel_diff ist die gewünschte relative Differenz
REM rel_diff wird als nicht negativ angenommen
Function AreSameNumber(n1, n2, rel_diff) As Boolean
  AreSameNumber = False                 'Ausgangsannahme: sie unterscheiden sich.
  If n1 <> 0 Then                       'Keine Division durch n1, wenn es den Wert 0
hat.
    If ABS((n1-n2)/n1) <= rel_diff Then 'Division der Differenz durch n1
      AreSameNumber = True              'für den relativen Vergleich.
    End If                              'Wenn n1, die Zahl, um die es geht,
  ElseIf ABS(n2) <= rel_diff Then       'null ist, dann wird die Größe von n2
verglichen.
    AreSameNumber = True
  End If
End Function
```

Der Code in Listing 71 dividiert den Unterschied zweier Zahlen durch eine der beiden. Der Code in Listing 72 prüft, ob Zahlen unterschiedlicher Größe dieselbe Zahl sind.

Listing 72. *Test, ob die Zahlen gleich sind.*

```
Sub CheckSameNumber
  Dim s1 As Single
  Dim s2 As Single
  Print AreSameNumber(299792458, 299790000, 1e-5)  'True: fünf gleiche Ziffern
```

```
    Print AreSameNumber(299792458, 299700000, 1e-5)  'False: vier gleiche Ziffern
    s1 = 299792458                                   's1 erhält einen anderen Wert
    s2 = 299792448                    'zugewiesen als s2, ist aber dieselbe Zahl.
    Print AreSameNumber(s1, s2, 0.0)  'True: dieselbe Zahl in einfacher Genauigkeit.
    Print AreSameNumber(299.792458, 299.790000, 1e-5)'True: fünf gleiche Ziffern
    Print AreSameNumber(2.99792458, 2.99700000, 1e-5)'False: vier gleiche Ziffern
End Sub
```

Es ist viel geschrieben und geforscht worden über die negativen Aspekte im Zusammenhang mit Fließkommazahlen. Eine vollständige Behandlung dieses Themas übersteigt daher bei weitem den Rahmen dieses Buches. Für den Normalgebrauch sind die Probleme kaum so störend, aber wenn sie auftauchen, können sie sehr verwirrend sein, wenn Sie sich der Problematik nicht bewusst sind.

4.3. Mathematische Funktionen

Die mathematischen Funktionen in OOo Basic erwarten ein numerisches Argument. Vor der Berechnung werden alle Standardtypen zu Double konvertiert. Strings dürfen aus Hexadezimal- und Oktalzahlen bestehen. Die Funktionen sind dieselben wie in Visual Basic (s. Tabelle 25).

Tabelle 25. *Mathematische Funktionen in OOo Basic.*

OOo Basic	VB	VB .NET	Rückgabewert
ABS	ABS	Math.Abs	Absolutwert der angegebenen Zahl.
Exp	Exp	Math.Exp	Potenziert die Basis des natürlichen Logarithmus um die angegebene Zahl.
Log	Log	Math.Log	Der natürliche Logarithmus der angegebenen Zahl. In Visual Basic .NET kann diese Methode überladen werden, um entweder den natürlichen Logarithmus (Basis e) oder den Logarithmus zu einer bestimmten Basis zurückzugeben.
Sgn	Sgn	Math.Sign	Ganzzahl, die dem Vorzeichen der angegebenen Zahl entspricht (-1, 0, 1).
Sqr	Sqr	Math.Sqrt	Quadratwurzel der angegebenen Zahl.

Mit der Funktion ABS bestimmen Sie den Absolutwert einer Zahl. Sie können sich das so vorstellen, dass einfach das Vorzeichen (+ oder -) von der Zahl abgetrennt wird. Die geometrische Definition von ABS(x) ist der Abstand von x zu 0 auf einer geraden Linie.

```
ABS(23.33) = 23.33
ABS(-3)    = 3
ABS("-1")  = 1  'Beachten Sie, dass der String "-1" zu Double konvertiert wird.
```

Die Funktion Sgn bestimmt das Vorzeichen einer Zahl. Es wird ein Integer-Wert -1, 0 oder 1 zurückgegeben, wenn die Zahl entsprechend negativ, Null oder positiv ist.

```
Sgn(-37.4) = -1
Sgn(0)     =  0
Sgn("4")   =  1
```

Die Quadratwurzel von 9 ist 3, denn 3 mal 3 ergibt 9. Mit der Funktion Sqr erhalten Sie die Quadratwurzel einer Zahl. Die Funktion Sqr kann keine Quadratwurzel einer negativen Zahl errechnen – der Versuch endet in einem Laufzeitfehler.

```
Sqr(100) = 10
Sqr(16)  = 4
Sqr(2)   = 1.414213562371
```

Logarithmen wurden von John Napier erfunden, der von 1550 bis 1617 lebte. Napier entwickelte Logarithmen, um arithmetische Berechnungen dadurch zu vereinfachen, dass Addition und Subtraktion an die Stelle von Multiplikation und Division traten. Logarithmen haben folgende Eigenschaften:

```
Log(x*y) = Log(x) + Log(y)
Log(x/y) = Log(x) - Log(y)
Log(x^y) = y * Log(x)
```

Die Funktion Exp ist die Umkehrung der Funktion Log. Zum Beispiel ist Exp(Log(4)) = 4 und Log(Exp(2)) = 2. Konzeptionell wandeln Logarithmen Multiplikationsaufgaben in Additionsaufgaben. So kann man Logarithmen in ihrer ursprünglichen Gestaltung verwenden.

```
Print Exp(Log(12) + Log(3)) '36 = 12 * 3
Print Exp(Log(12) - Log(3)) ' 4 = 12 / 3
```

Logarithmen sind durch die Gleichung y=b^x definiert. Daher sagt man: der Logarithmus Basis b von y it x. Zum Beispiel mit dem Logarithmus Basis 10: 10^2 = 100. Der Logarithmus Basis 10 von 100 ist also 2. Häufig wird der Logarithmus mit der Basis e (ungefähr e=2,71828182845904523536) verwendet, weil er ein paar nette mathematische Eigenschaften besitzt. Dies ist der „natürliche Logarithmus“ und wird in OOo Basic genutzt. Visual Basic .NET ermöglicht Ihnen, Logarithmen auf anderer Basis zu berechnen. Das kann jedoch einfach mit der Formel erreicht werden, wonach der Logarithmus Basis b durch Log(x)/Log(b) bestimmt ist, ungeachtet der Basis des aktuell genutzten Logarithmus.

Logarithmen spielen heute als allgemeine Rechenvereinfachungen nicht mehr die Rolle, angesichts der Leistungsfähigkeit heutiger Computer. Allerdings beschreiben Logarithmen das Verhalten vieler natürlicher Phänomene. Zum Beispiel wird das Bevölkerungswachstum oft mit Logarithmen dargestellt, denn geometrisches Wachstum zeigt sich in einer logarithmischen Kurvendarstellung als Gerade. Exponentielle und logarithmische Kurven werden auch ausgiebig im Ingenieurswesen bei Berechnungen genutzt, die das dynamische Verhalten elektrischer, mechanischer und chemischer Systeme beschreiben.

Das Makro in Listing 73 berechnet den Logarithmus der Zahl x (erstes Argument) zur angegebenen Basis b (zweite Argument). Zum Beispiel berechnen Sie mit LogBase(8, 2) den Logarithmus Basis 2 von 8 (Ergebnis 3).

Listing 73. *LogBase.*

```
Function LogBase(x, b) As Double
  LogBase = Log(x) / Log(b)
End Function
```

4.4. Numerische Konvertierungen

OOo Basic versucht, Argumente vor der Operation zu einem passenden Typ zu konvertieren. Es ist jedoch sicherer, die Datentypen explizit mit Hilfe von Konvertierungsfunktionen – präsentiert in diesem Kapitel – umzuwandeln, als sich auf das Standardverhalten zu verlassen, denn es könnte nicht das Gewünschte erbringen. Wenn ein Integer-Argument benötigt wird und eine Fließkommazahl bereit steht, wird die Zahl standardmäßig gerundet. Bei „16.8 MOD 7“ zum Beispiel wird vor der Berechnung 16.8 zu 17 aufgerundet. Der Operator für die Ganzzahldivision schneidet die Operanden jedoch ab. Bei „Print 4\0.999“ zum Beispiel wird 0.999 zu 0 und verursacht einen Division-durch-Null-Fehler.

Es gibt viele verschiedene Methoden und Funktionen zur Konvertierung zu numerischen Typen. Die wesentlichen Konvertierungsfunktionen wandeln Zahlen um, die als Strings gemäß dem lokalen Gebietsschema vorliegen. Die Konvertierungsfunktionen in Tabelle 26 wandeln jeden String oder numerischen Ausdruck in eine Zahl um. Hexadezimal- oder Oktalzahlen in Strings müssen in der Standardnotation von OOo Basic vorliegen. Zum Beispiel muss die hexadezimale Zahl 2A als „&H2A“ angegeben werden.

Tabelle 26. *Konvertierung zu einem numerischen Typ.*

Funktion	Typ	Beschreibung
CByte(expression)	Byte	Rundet den numerischen oder String-Ausdruck zu einem Byte.
CCur(expression)	Currency	Konvertiert den numerischen oder String-Ausdruck zum Typ Currency. Das lokale Gebietsschema gilt für Dezimaltrenner und Währungssymbole.
CDec(expression)	Decimal	Konvertiert den Ausdruck zum Typ Decimal (nur unter Windows).
CInt(expression)	Integer	Rundet den numerischen oder String-Ausdruck zum nächsten Integer-Wert.
CLng(expression)	Long	Rundet den numerischen oder String-Ausdruck zum nächsten Long-Wert.
CDbl(expression)	Double	Konvertiert den numerischen oder String-Ausdruck zum Typ Double.
CSng(expression)	Single	Konvertiert den numerischen oder String-Ausdruck zum Typ Single.

Die Funktionen, die eine Ganzzahl zurückgeben, verhalten sich alle ähnlich. Numerische Ausdrücke werden gerundet und nicht abgeschnitten. Ein String-Ausdruck, der keine Zahl enthält, wird mit 0 gewertet. Es wird nur der Teil des Strings ausgewertet, der eine Zahl enthält. Siehe Listing 74.

Listing 74. *CInt und CLng ignorieren nicht-numerische Werte.*

```
Print CInt(12.2)       ' 12
Print CLng("12.5")     ' 12
Print CInt("xxyy")     '  0
Print CLng("12.1xx")   ' 12
Print CInt(-12.2)      '-12
Print CInt("-12.5")    '-12
Print CLng("-12.5xx") '-12
```

CLng und CInt verhalten sich ähnlich, aber nicht ganz gleich bei verschiedenen Arten von Überlaufbedingungen. Zu große dezimale Zahlen in Strings verursachen einen Laufzeitfehler. Es gibt zum Beispiel bei CInt("40000") und CLng("999999999999") einen Laufzeitfehler, aber nicht bei CLng("40000"). CLng verursacht nie einen Überlauf bei zu großen Hexadezimal- oder Oktalzahlen, es wird in stillschweigender Duldung Null zurückgegeben. CInt interpretiert jedoch Hexadezimal- und Oktalzahlen als Long und konvertiert sie dann zu Integer, mit dem Ergebnis, dass ein gültiger Long-Wert beim Konvertieren zu Integer einen Laufzeitfehler auslöst. Ein hexadezimaler Wert aber, der zu groß für einen gültigen Long-Wert ist, wird anstandslos zu Null und dann zu einem Integer umgewandelt (s. Listing 75).

Listing 75. *CInt wertet die Zahl als Long und konvertiert dann zu Integer.*

```
Print CLng("&HFFFFFFFFFFFE")   '0  Überlauf bei Long
Print CInt("&HFFFFFFFFFFFE")   '0  Überlauf bei Long,
                               '   danach Konvertierung zu Integer
Print CLng("&HFFFFE")          '1048574
Print CInt("&HFFFFE")          'Laufzeitfehler, konvertiert zu Long, danach Überlauf
```

Der Code in Listing 76 konvertiert eine Reihe von hexadezimalen Zahlen mit CLng zu Long. Zur Erläuterung der Ausgaben von Listing 76 s. Tabelle 27.

Listing 76. *Beispiel für CLng mit Hexadezimalzahlen.*

```
Sub ExampleCLngWithHex
  On Error Resume Next
  Dim s$, i%
  Dim v()
  v() = Array("&HF",      "&HFF",      "&HFFF",      "&HFFFF",_
              "&HFFFFF", "&HFFFFFF", "&HFFFFFFF", "&HFFFFFFFF",_
```

```
                "&HFFFFFFFFF",_
                "&HE",      "&HFE",     "&HFFE",     "&HFFFE",_
                "&HFFFFE",  "&HFFFFFE", "&HFFFFFFE", "&HFFFFFFFE",_
                "&HFFFFFFFFE")
    For i = LBound(v()) To UBound(v())
      s = s & i & " CLng(" & v(i) & ") = "
      s = s & CLng(v(i))
      s = s & CHR$(10)
    Next
    MsgBox s
  End Sub
```

Tabelle 27. *Ausgaben von Listing 76 mit Erläuterungen.*

Eingabe	CLng	Erläuterung
F	15	Korrekter hexadezimaler Wert.
FF	255	Korrekter hexadezimaler Wert.
FFF	4095	Korrekter hexadezimaler Wert.
FFFF	65535	Korrekter hexadezimaler Wert.
FFFFF	1048575	Korrekter hexadezimaler Wert.
FFFFFF	16777215	Korrekter hexadezimaler Wert.
FFFFFFF	268435455	Korrekter hexadezimaler Wert.
FFFFFFFF	??	Sollte -1 zurückgeben, könnte aber mit 64-Bit-Versionen einen Laufzeitfehler auslösen.
FFFFFFFFF	0	Überlauf gibt Null zurück, es sind neun hexadezimale Ziffern.
E	14	Korrekter hexadezimaler Wert.
FE	254	Korrekter hexadezimaler Wert.
FFE	4094	Korrekter hexadezimaler Wert.
FFFE	65534	Korrekter hexadezimaler Wert.
FFFFE	1048574	Korrekter hexadezimaler Wert.
FFFFFE	16777214	Korrekter hexadezimaler Wert.
FFFFFFE	268435454	Korrekter hexadezimaler Wert.
FFFFFFFE	ERROR	Laufzeitfehler, hat einmal -2 zurückgegeben.
FFFFFFFFE	0	Überlauf gibt Null zurück, es sind neun hexadezimale Ziffern.

Wenn Sie Zahlen schreiben, brauchen Sie keine führenden Nullen anzugeben. 3 und 003 sind zum Beispiel ein und dieselbe Zahl. Long Integer kann acht hexadezimale Ziffern enthalten. Wenn nur vier geschrieben werden, können Sie führende Nullen voraussetzen. Wenn die Hexadezimalzahl zu groß für Long ist, wird Null zurückgegeben. Die negativen Zahlen sind genauso leicht zu erklären. Intern repräsentiert der Computer eine negative Zahl durch ein gesetztes erstes Bit. Bei all den Hexadezimalziffern 8, 9, A, B, C, D, E und F ist in der binären Zahlendarstellung das höchste Bit gesetzt. Wenn die erste Hexadezimalziffer ein gesetztes höchstes Bit hat, ist der zurückgegebene Long-Wert negativ. Eine Hexadezimalzahl ist positiv, wenn sie weniger als acht hexadezimale Ziffern hat, und negativ, wenn sie aus acht hexadezimalen Ziffern besteht, deren erste Ziffer 8, 9, A, B, C, D, E oder F ist. Nun, so war es jedenfalls einmal.

Tipp

Von OOo 3.30 an lösen die 64-Bit-Versionen von CLng bei hexadezimal dargestellten negativen Zahlen einen Fehler aus! Hoffentlich wird das gefixt. Zwei Tests, die -1 ausgeben sollen: ??

```
print &HFFFFFFFF
print Clng("&HFFFFFFFF")   ' Löst einen Fehler aus!
```

Die FunktionCByte verhält sich wie CInt und CLng, wiewohl mit Besonderheiten. Der Rückgabetyp, Byte, wird als Zeichen behandelt, wenn nicht explizit in eine Zahl konvertiert wird. Ein Byte ist ein Kurz-Integer, das nur acht Bit einnimmt anstatt 16 Bit bei Integer.

```
Print CByte("65")          'A hat den ASCII-Wert 65
Print CInt(CByte("65xx")) '65 wird direkt zu einer Zahl konvertiert.
```

Tipp

Ein Integer ist in VB .NET äquivalent zu einem Long in OOo Basic.

VB hat abweichende Rundungsregeln. Zahlen werden zur nächsten geraden Zahl hin gerundet, wenn der Dezimalteil exakt 0,5 ist. Das nennt man IEEE-Rundung.

Die Funktionen, die eine Fließkommazahl zurückgeben, verhalten sich alle ähnlich. Numerische Ausdrücke werden zu dem naheliegendsten Wert konvertiert. Strings mit nicht-numerischen Anteilen lösen einen Laufzeitfehler aus. Zum Beispiel ergibt CDbl(“13.4e2xx”) einen Laufzeitfehler. CDbl und CSng lösen einen Laufzeitfehler aus für zu große Hexadezimal- und Oktalzahlen.

Listing 77. *CSng und CDbl mit String-Argumenten.*

```
Print CDbl(12.2)       ' 12.2
Print CSng("12.55e1") ' 125.5
Print CDbl("-12.2e-1")'-1.22
Print CSng("-12.5")    '-12.5
Print CDbl("xxyy")     ' Laufzeitfehler
Print CSng("12.1xx")   ' Laufzeitfehler
```

Die Funktionen CDbl und CSng scheitern an Strings mit nicht-numerischen Daten, die Funktion Val jedoch nicht. Nehmen Sie die Funktion Val, um einen String, der noch andere Zeichen enthalten kann, zum Typ Double zu konvertieren. Die Funktion Val schaut sich jedes Zeichen in der Zeichenkette an, ignoriert Leerzeichen, Tabulatorschritte und Zeilenwechselzeichen und stoppt erst, wenn sie auf ein Zeichen stößt, das nicht ein Teil einer Zahl ist. Symbole und Zeichen, die oft als Teil numerischer Werte gelten, wie Dollarzeichen und Kommas, werden nicht erkannt. Die Funktion erkennt allerdings Oktal- und Hexadezimalzahlen, die mit &O (oktal) und &H (hexadezimal) eingeleitet werden.

Die Funktion Val behandelt Leerzeichen anders als die anderen Funktionen. Zum Beispiel gibt Val(“ 12 34”) die Zahl 1234 zurück, CDbl und CSng produzieren einen Laufzeitfehler, und CInt gibt mit demselben Argument 12 zurück.

Listing 78. *Die Behandlung von Leerzeichen ist unterschiedlich.*

```
Sub NumsAreDifferent
  On Error GoTo ErrorHandler:
  Dim s$
  s = "Val(""  12  34"") = "
  s = s & Val("  12    34")
  s = s & CHR$(10) & "CInt(""  12  34"") = "
  s = s & CInt("  12    34")
  s = s & CHR$(10) & "CLng(""  12  34"") = "
  s = s & CLng("  12    34")
  s = s & CHR$(10) & "CSng(""  12  34"") = "
  s = s & CSng("  12    34")
  s = s & CHR$(10) & "CDbl(""  12  34"") = "
  s = s & CDbl("  12    34")
  MsgBox s
  Exit Sub
  ErrorHandler:
    s = s & " Fehler: " & Error
    Resume Next
End Sub
```

Tipp Die Funktion Val berücksichtigt bei der Zahlenkonvertierung das lokale Gebietsschema nicht. Als Dezimaltrenner gilt nur der Punkt, das Komma kann als Tausendertrenner dienen, aber niemals rechts vom Dezimaltrenner. Um lokalisierte Zahlen zu konvertieren, verwenden Sie CDbl oder CLng. Falls Sie es vergessen haben sollten: Das Gebietsschema ist eine weitere Methode, auf die Einstellungen zuzugreifen, die die Formatierungen abhängig von einem bestimmten Land beeinflussen. Siehe Listing 79.

Listing 79. *Die Funktion Val ist die Umkehrung der Funktion Str.*

```
Sub ExampleVal
  Print Val(" 12 34")  '1234
  Print Val("12 + 34") '12
  Print Val("-1.23e4") '-12300
  Print Val(" &FF")    '0
  Print Val(" &HFF")   '255
  Print Val("&HFFFF")  '-1
  Print Val("&HFFFE")  '-2
  Print Val("&H3FFFE") '-2, ja, es wird wirklich zu -2 konvertiert
  Print Val("&HFFFFFFFFFFFFF") '-1
End Sub
```

Seit der Version 1.1.1 verhält sich die Funktion Val beim Erkennen von Hexadezimal- oder Oktalzahlen seltsam genug, so dass ich von einem Bug rede. Intern werden Hexadezimal- und Oktalzahlen zu einem 32 Bit großen Long Integer, und die niedrigstwertigen 16 Bits dann zu einem Integer konvertiert. Das erklärt , warum in Listing 79 die Zahl H3FFFE zu -2 konvertiert wird, weil nämlich nur die niedrigstwertigen 16 Bits erkannt werden – falls Sie es vergessen haben sollten, es sind die ganz rechts angeordneten vier Hexadezimalziffern. Listing 80 demonstriert dieses seltsame Verhalten. Die Ausgabe ist in Tabelle 28 erläutert.

Listing 80. *Beispiel für Val mit Hexadezimalzahlen.*

```
Sub ExampleValWithHex
  Dim s$, i%
  Dim l As Long
  Dim v()
   v() = Array("&HF",      "&HFF",      "&HFFF",      "&HFFFF",_
               "&HFFFFF", "&HFFFFFF", "&HFFFFFFF", "&HFFFFFFFF",_
               "&HFFFFFFFFF",_
               "&HE",       "&HFE",      "&HFFE",      "&HFFFE",_
               "&HFFFFE",  "&HFFFFFE", "&HFFFFFFE", "&HFFFFFFFE",_
               "&HFFFFFFFFE", "&H111111111", "&H1111")
  For i = LBound(v()) To UBound(v())
    s = s & "Val(" & v(i) & ") = " & Val(v(i)) & CHR$(10)
  Next
  'Folgendes funktionierte in OOo 2.x,
  'misslingt aber in OOo 3.2.1
  'l = "&H" & Hex(-2)
  s = s & CHR$(10) & "Hex(-1) = " & Hex(-1) & CHR$(10)
  s = s & "Hex(-2) = " & Hex(-2) & CHR$(10)
  's = s & "l = &H" & Hex(-2) & " ==> " & l & CHR$(10)
  MsgBox s
End Sub
```

Tabelle 28. *Ausgabe vom Listing 80 mit Erläuterungen.*

Eingabe	Ausgabe	Erläuterung
F	15	Hexadezimales F ist 15.
FF	255	Hexadezimales FF ist 255.
FFF	4095	Hexadezimales FFF ist 4095.
FFFF	-1	Hexadezimales FFFF ist -1 für ein 16-Bit-Integer (zwei Bytes).
FFFFF	-1	Nur die vier Ziffern ganz rechts werden erkannt.
E	14	Hexadezimales E ist 14.
FE	254	Hexadezimales FE ist 254.
FFE	4094	Hexadezimales FFE ist 4094.
FFFE	-2	Hexadezimales FFFE ist -2 für ein 16-Bit-Integer (zwei Bytes).
FFFFE	-2	Nur die vier Ziffern ganz rechts werden erkannt; dies sind fünf Ziffern.
FFFFFE	-2	Nur die vier Ziffern ganz rechts werden erkannt; dies sind sechs Ziffern.
FFFFFFE	-2	Nur die vier Ziffern ganz rechts werden erkannt; dies sind sieben Ziffern.
FFFFFFFE	-2	Nur die vier Ziffern ganz rechts werden erkannt.
HFFFFFFFFE	-2	Nur die vier Ziffern ganz rechts werden erkannt.
111111111	4639	Korrekt, nur die vier Ziffern ganz rechts.
1111	4639	Korrekt.

Val konvertiert Hexadezimalzahlen, nutzt aber nur die vier Ziffern ganz rechts.

Die Funktionen CByte, CInt, CLng, CSng und CDbl konvertieren Zahlen, Strings oder Ausdrücke zu einem spezifischen numerischen Typ. Die Funktionen Int und Fix entfernen den dezimalen Anteil und geben den Typ Double zurück. Ein String-Ausdruck, der keine Zahl enthält, wird als 0 ausgewertet. Es wird nur der Teil berücksichtigt, der eine Zahl enthält, s. Tabelle 29.

Tabelle 29. *Entfernung des dezimalen Anteils einer Fließkommazahl.*

Funktion	Typ	Beschreibung
Int	Double	Rundet die Zahl in Richtung negativ Unendlich.
Fix	Double	Schneidet den dezimalen Anteil ab.

Die Funktionen Int und Fix unterscheiden sich nur in der Behandlung negativer Zahlen. Fix entfernt immer den dezimalen Teil, was der Rundung Richtung Null entspricht. Int rundet hingegen in Richtung negativ Unendlich. Als Beispiel: „Int(12.3)“ wird 12 und „Int(-12.3)“ wird -13.

```
Print Int(12.2)      ' 12
Print Fix(12.2)      ' 12
Print Int("12.5")    ' 12
Print Fix("12.5")    ' 12
Print Int("xxyy")    '  0
Print Fix("xxyy")    '  0
Print Int(-12.4)     '-13
Print Fix(-12.4)     '-12
Print Fix("-12.1xx")'-12
Print Int("-12.1xx")'-12
```

Die Funktion CCur konvertiert einen numerischen Ausdruck zu einem Währungsobjekt. Visual Basic .NET hat die Unterstützung für die Funktion CCur wie auch für den Datentyp Currency eingestellt. OOo Basic hält noch an dem Datentyp Currency fest.

4.5. Konvertierungen von Zahl zu String

Funktionen zur Stringkonvertierung, s. Tabelle 30, wandeln Daten, die keine Strings sind, in Strings um. In OOo wird Text gemäß Unicode 2.0 gespeichert, wodurch eine Vielzahl von Sprachen unterstützt wird. Jede String-Variable kann bis zu 65.535 Zeichen enthalten.

Tabelle 30. *Funktionen zur String-Konvertierung.*

Funktion	Beschreibung
Str	Konvertiert eine Zahl zu einem String ohne Lokalisierung.
CStr	Konvertiert jeden Typ zu einem String. Zahlen und Datumswerte werden gemäß dem lokalen Gebietsschema formatiert..
Hex	Gibt eine Zahl in hexadezimaler Darstellung als String zurück.
Oct	Gibt eine Zahl in oktaler Darstellung als String zurück.

4.6. Einfache Formatierung

Die Funktion CStr konvertiert ganz allgemein jeden Typ zu einem String. Der Rückgabewert hängt vom Datentyp des Arguments ab. Boolsche Werte resultieren in „True“ oder „False“. Datumswerte werden in das kurze Datumsformat des Systems umgesetzt. Zahlen werden zu einer Zeichenkette konvertiert. Siehe Listing 81.

Listing 81. *Die Ausgabe von CStr ist abhängig vom Gebietsschema; hier Deutsch (Deutschland).*

```
Dim n As Long, d As Double, b As Boolean
n = 999999999 : d = EXP(1.0) : b = False
Print "X" & CStr(b)   'XFalse
Print "X" & CStr(n)   'X999999999
Print "X" & CStr(d)   'X2,71828182845904
Print "X" & Cstr(Now)'X29.06.2011 20:24:24 (fast genau 8 Jahre nach der ersten Auflage)
```

Die Funktion CStr führt eine einfache Zahlenformatierung nach Kenntnis des lokalen Gebietsschemas durch. Mit Str gibt es eine einfache Konvertierung einer Zahl zu einem String. Obwohl die Funktion Str eigentlich speziell für numerische Werte gedacht ist, ähnelt die Ausgabe sehr der von CStr. Wenn die Funktion Str eine Zahl zu einem String konvertiert, wird immer für das Vorzeichen der Zahl ein Leerzeichen vorangestellt. Eine negative Zahl enthält das Minuszeichen, das Leerzeichen entfällt. Eine nicht-negative Zahl hingegen enthält ein führendes Leerzeichen. Str gibt eine Zahl nicht-lokalisiert zurück, als Dezimaltrenner steht immer der Punkt. Die Ausgabe eines Datums wird jedoch gemäß dem Gebietsschema formatiert. Siehe Listing 82.

Listing 82. *Die Ausgabe von Str ist unabhängig vom Gebietsschema (Ausnahme: Datum).*

```
Dim n As Long, d As Double, b As Boolean
n = 999999999 : d = EXP(1.0) : b = False
Print "X" & Str(b)   'XFalse
Print "X" & Str(n)   'X 999999999
Print "X" & Str(d)   'X 2.71828182845904
Print "X" & Str(Now)'X29.06.2011 20:28:48 (fast genau 8 Jahre nach der ersten Auflage)
```

Die Ausgabe vom Code in Listing 81 und Listing 82 sind gleich bis auf die führenden Leerzeichen vor den nicht-negativen Zahlen und dem Dezimaltrenner. Wenn Sie den Code mit einem anderen Gebietsschema ausführen, wie zum Beispiel Englisch (USA), ändert sich die Ausgabe von Listing 81.

Tipp Es gibt kaum einen Grund, Str anstatt CStr zu nutzen. Str mag wohl ein wenig schneller sein, aber CStr kennt Ihr aktuelles Gebietsschema.

Zur Demonstration, dass CStr abhängig vom Gebietsschema arbeitet, habe ich mein Gebietsschema auf Englisch (USA) umgestellt und den Code in Listing 81 noch einmal ausgeführt. Listing 83 zeigt, dass der Dezimaltrenner nun der Punkt ist und dass das Datum die Form DD/MM/YYYY hat.

Listing 83. *Die Ausgabe von CStr ist abhängig vom Gebietsschema; hier Englisch (USA).*

```
Dim n As Long, d As Double, b As Boolean
n = 999999999 : d = EXP(1.0) : b = False
Print "X" & CStr(b)  'XFalse
Print "X" & CStr(n)  'X999999999
Print "X" & CStr(d)  'X2.71828182845904
Print "X" & CStr(Now)'X06/29/2011 20:35:42
```

4.7. Zahlen auf anderer Basis, hexadezimal, oktal und binär

OOo Basic stellt die Funktionen Hex und Oct bereit zur Konvertierung einer Zahl in ihre hexadezimale und oktale Form. Allerdings fehlt originäre Unterstützung zur Konvertierung zu und von Binärzahlen. Und man kann die Ausgabe von Hex und Oct nicht direkt zur Rückkonvertierung zu einer Zahl verwenden, weil dem String das einleitende „&H“ und „&O“ fehlt.

```
Print Hex(447)              '1BF
Print CInt("&H" & Hex(747)) '747
Print Oct(877)              '1555
Print CInt("&O" & Oct(213)) '213
```

Der Quellcode für Kapitel 2 enthält die Funktion IntToBinaryString, die eine Ganzzahl zu einer Binärzahl konvertiert (Modul LanguageConstructs). Die Funktion ist sehr flexibel, aber nicht besonders schnell. In Listing 84 finden Sie eine schnellere Routine, die sich der Funktion Hex bedient.

Listing 84. *IntToBinaryString.*

```
Function IntToBinaryString(ByVal x As Long) As String
  Dim sHex As String
  Dim sBin As String
  Dim i As Integer
  sHex = Hex(x)
  For i=1 To Len(sHex)
    Select Case Mid(sHex, i, 1)
      Case "0"
        sBin = sBin & "0000"
      Case "1"
        sBin = sBin & "0001"
      Case "2"
        sBin = sBin & "0010"
      Case "3"
        sBin = sBin & "0011"
      Case "4"
        sBin = sBin & "0100"
      Case "5"
        sBin = sBin & "0101"
      Case "6"
        sBin = sBin & "0110"
      Case "7"
        sBin = sBin & "0111"
      Case "8"
        sBin = sBin & "1000"
      Case "9"
        sBin = sBin & "1001"
      Case "A"
        sBin = sBin & "1010"
```

```
      Case "B"
        sBin = sBin & "1011"
      Case "C"
        sBin = sBin & "1100"
      Case "D"
        sBin = sBin & "1101"
      Case "E"
        sBin = sBin & "1110"
      Case "F"
        sBin = sBin & "1111"
    End Select
  Next
  IntToBinaryString = sBin
End Function
```

Der Code in Listing 84 mag lang sein, ist aber sehr einfach. Es gibt eine Wechselbeziehung zwischen Hexadezimalziffern und Binärziffern: jede Hexadezimalziffer entspricht vier Binärziffern. Diese Beziehung besteht nicht mit Zahlen auf der Basis 10. Die Zahl wird mit der Funktion Hex zu einer Hexadezimalzahl konvertiert. Dann wird jede Hexadezimalziffer zu den entsprechenden Binärziffern konvertiert. Um eine Binärzahl in Stringform zu einem Integer zurück zu wandeln, nehmen Sie den Code in Listing 85.

Tipp Diese Routine versagt mit negativen Zahlen, weil in OOo 3.3.0 CLng() mit Hexadezimalzahlen versagt, die negative Werte repräsentieren.

Listing 85. *BinaryStringToLong.*

```
Function BinaryStringToLong(s$) As Long
  Dim sHex As String
  Dim sBin As String
  Dim i As Integer
  Dim nLeftOver As Integer 'Für den Rest nach Ganzzahldivision
  Dim n As Integer

  n = Len(s$)
  nLeftOver = n MOD 4
  If nLeftOver > 0 Then
    sHex = SmallBinToHex(Left(s$, nLeftOver))
  End If
  For i=nLeftOver + 1 To n Step 4
    sHex = sHex & SmallBinToHex(Mid(s$, i, 4))
  Next
  BinaryStringToLong = CLng("&H" & sHex)
End Function

Function SmallBinToHex(s$) As String
    If Len(s$) < 4 Then s$ = String(4-Len(s$), "0") & s$
    Select Case s$
      Case "0000"
        SmallBinToHex = "0"
      Case "0001"
        SmallBinToHex = "1"
      Case "0010"
        SmallBinToHex = "2"
      Case "0011"
        SmallBinToHex = "3"
      Case "0100"
        SmallBinToHex = "4"
```

```
        Case "0101"
          SmallBinToHex = "5"
        Case "0110"
          SmallBinToHex = "6"
        Case "0111"
          SmallBinToHex = "7"
        Case "1000"
          SmallBinToHex = "8"
        Case "1001"
          SmallBinToHex = "9"
        Case "1010"
          SmallBinToHex = "A"
        Case "1011"
          SmallBinToHex = "B"
        Case "1100"
          SmallBinToHex = "C"
        Case "1101"
          SmallBinToHex = "D"
        Case "1110"
          SmallBinToHex = "E"
        Case "1111"
          SmallBinToHex = "F"
      End Select
End Function
```

Zur Konvertierung eines Binärzahlstrings zu einem Integer wird die Zahl zuerst zu einer Hexadezimalzahl konvertiert. Ein Satz von vier Binärziffern entspricht einer einzelnen Hexadezimalziffer. Die Zahl wird links mit Nullen aufgefüllt, so dass der String in Blöcke von je vier Binärziffern aufgeteilt werden kann. Jeder dieser Blöcke von vier Binärziffern wird zu einer Hexadezimalziffer umgewandelt. Dann wird mit Hilfe der Funktion CLng die Hexadezimalzahl in die dezimale Form gebracht. Die Routine in Listing 86 zeigt den Gebrauch dieser Funktionen, s. auch Bild 41.

Listing 86. *ExampleWholeNumberConversions.*

```
Sub ExampleWholeNumberConversions
  Dim s As String
  Dim n As Long
  Dim nAsHex$, nAsOct$, nAsBin$
  s = InputBox("Zu konvertierende Zahl:", "Long in anderer Form", "1389")
  If IsNull(s) Then Exit Sub
  If Len(Trim(s)) = 0 Then Exit Sub
  n = CLng(Trim(s))  'Trim entfernt führende und nachgestellte Leerzeichen
  nAsHex = Hex(n)
  nAsOct = Oct(n)
  nAsBin = IntToBinaryString(n)
  s = "Originalzahl = " & CStr(n) & CHR$(10) &_
      "Hex(" & CStr(n) & ") = " & nAsHex & CHR$(10) &_
      "Oct(" & CStr(n) & ") = " & nAsOct & CHR$(10) &_
      "Binary(" & CStr(n) & ") = " & nAsBin &_
      " ==> " & BinaryStringToLong(nAsBin)
  MsgBox(s, 0, "Konvertierung einer ganzen Zahl")
End Sub
```

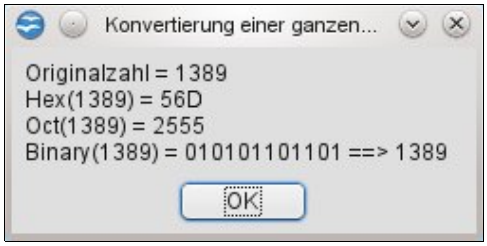


Bild 41. *Konvertierung einer ganzen Zahl in die hexadezimale, oktale und binäre Form.*

4.8. Zufallszahlen

OOo Basic erzeugt Zufallszahlen als Fließkommazahlen im Bereich von 0 bis 1. Von Computern erzeugte Zufallszahlen sind im allgemeinen nicht zufällig. Diese „Zufallszahlen“ werden durch einen Algorithmus erzeugt, der auf einer vorherigen Zufallszahl aufsetzt. Die erste Zahl, auf der alle weiteren Zufallszahlen basieren, heißt „Saat“ (engl. seed). Mit der Funktion Randomize wird der Wert des Seed festgelegt. Wird kein Wert angegeben, verwendet die Funktion den aktuellen Wert des Systemzeitgebers. Durch die Angabe eines spezifischen Startwerts haben Sie die Möglichkeit, identische Zufallszahlensequenzen zu erzeugen, um Programme zu testen.

Die Funktion Rnd erzeugt die Zufallszahlen zwischen 0 und 1. Die OOo-Hilfe behauptet, dass die Funktion Rnd ein Argument akzeptiere, dass „festlegt, wie Zufallszahlen erzeugt werden“. Ich habe den Quellcode der Funktion Rnd untersucht, der sich übrigens von der Version 1.x bis 3.2.1 nicht verändert hat: Das Argument wird ignoriert.

Tipp Die OOo-Hilfedateien behaupten fälschlich, dass die Funktion `Rnd` ein Argument akzeptiere, womit das Verhalten beeinflusst werde. Das Argument wird, und wurde schon immer, ignoriert.

```
Print Rnd()    'Eine Zahl von 0 bis 1
Print Rnd()    'Noch eine Zahl von 0 bis 1
```

Die erzeugte Zufallszahl ist irgendeine Zahl zwischen 0 und 1. Um einen anderen Bereich zu erhalten, braucht man ein paar mathematische Rechenoperationen. Zum Beispiel erhält man durch die Multiplikation einer Zahl zwischen 0 und 1 mit 10 eine Zahl zwischen 0 und 10. Für einen Bereich, der nicht mit 0 beginnen soll, addieren Sie einen passenden Offset. Siehe Listing 87.

Listing 87. *Rückgabe einer Zufallszahl innerhalb eines bestimmten Bereichs.*

```
Function RndRange(lowerBound As Double, upperBound As Double) As Double
  RndRange = lowerBound + Rnd() * (upperBound - lowerBound)
End Function
```

Mit einer entsprechenden Funktion, zum Beispiel CInt oder CLng, erhalten Sie eine ganze Zahl statt einer Fließkommazahl.

```
CLng(lowerBound + Rnd() * (upperBound - lowerBound))
```

Zur Ermittlung des GGT (des Größten Gemeinsamen Teilers) zweier Ganzzahlen hatte ich zwei Funktionen zur Auswahl. Ich wollte wissen, welche schneller ist. Ich erzeugte Zufallszahlen und rief jede Routine einige tausendmal auf. Bei Tests zur Laufzeitermittlung ist es wichtig, für jeden Durchlauf dieselben Daten zu benutzen. Ich konnte die Zufallszahlen aus dem Grund nehmen, weil die Anweisung Randomize mir die Möglichkeit gab, jedes Mal dieselben Zufallszahlen zu erzeugen.

```
Randomize(2)    'Der Zufallszahlengenerator wird auf einen definierten Stand gesetzt.
t1 = getSystemTicks()
For i = 0 To 30000
  n1 = CLng(10000 * Rnd())
  n2 = CLng(10000 * Rnd())
  call gcd1(n1, n2)
Next
```

```
total_time_1 = getSystemTicks() - t1

Randomize(2)    'Der Zufallszahlengenerator wird auf den definierten Stand
zurückgesetzt.
t1 = getSystemTicks()
For i = 0 To 30000
  n1 = CLng(10000 * Rnd())
  n2 = CLng(10000 * Rnd())
  call gcd2(n1, n2)
Next
total_time_2 = getSystemTicks() - t1
```

4.9. Fazit

Die mathematischen Standardfunktionen in OpenOffice.org bieten wenig Überraschendes. Die Konvertierungsfunktionen arbeiten gut, trotz der einen oder anderen Eigenart bei der Konvertierung von Strings zu Zahlen. Achten Sie darauf, die Funktion zu wählen, die mit dem Format und dem verwendeten Wertebereich umgehen kann. Besondere Aufmerksamkeit muss auch auf das Runden gelegt werden. Obwohl das Rundungsverhalten dokumentiert und konsistent ist, bewirken verschiedene Funktionen und Operatoren doch unterschiedliche Rundungsergebnisse.

5. Array-Routinen

6. Date-Routinen

7. String-Routinen

8. Dateiroutinen

9. Sonstige Routinen

10. Universal Network Objects (UNO)

11. Der Dispatcher

12. StarDesktop

13. Allgemeine Dokument-Methoden

14. Writer-Dokumente

Anhang A. Glossar

Begriff	***Definition***
Kompilieren	Im Kontext dieses Dokuments bedeutet kompilieren, Star-Basic-Makros in einen Code zu konvertieren, den der Computer direkt ausführen kann.