# 2 Erste Schritte mit VBScript



Fragen Sie in einer unbekannten Stadt nach dem Weg zum Hauptbahnhof, dann freuen Sie sich bestimmt über eine einfache und schnelle Auskunft, die Sie auf direktem Weg zum Ziel führt.

Wissenschaftlich betrachtet wäre es allerdings ergiebiger, sich mit derselben Frage an einen ambitionierten Geografie-Professor zu wenden. Der könnte Ihnen Hunderte verschiedener Wege unter besonderer Berücksichtigung der Hydro- und Orographie mit all ihren Vor- und Nachteilen aufzeigen und nebenbei vermutlich auch noch die optimale Durchschnittsgeschwindigkeit errechnen, um alle Fußgängerampeln bis zum Bahnhof bei Grün zu erreichen. Ob Sie anschließend mit rauchendem Kopf und all den Fakten den Bahnhof jemals finden würden, bliebe allerdings zweifelhaft. Ihren Zug hätten Sie inzwischen sowieso verpasst.

Deshalb beginnt dieses Kapitel sofort praktisch. Sie setzen schon auf den ersten Seiten Ihre ersten einfachen Skripts ein, die Sie sofort ausprobieren können. Sie lernen in kürzester Zeit die wesentlichen und wichtigen Grundelemente kennen, die in jedem Skript eine Rolle spielen. Aber Sie erfahren nicht alles. Noch nicht. In diesem Kapitel dreht sich alles um die wichtigen Grundbegriffe.

Zu fast allem, was Sie in diesem Kapitel erforschen und erlernen, könnte man noch unendlich viel mehr schreiben, aber das passiert nicht in diesem ersten Kapitel. Viele Details und technische Extralösungen werden in den folgenden Kapiteln wieder aufgenommen und weiter ausgearbeitet, aber nicht in diesem ersten Kapitel.

In diesem ersten Kapitel dürfen Sie in Ruhe ausprobieren und experimentieren. Die Skripts, die Sie hier entwickeln, lösen noch keine Alltagsprobleme. Sie lösen Verständnisprobleme und machen es einfach, VBScript zu verstehen. Wenn Sie also gleich eine Stoppuhr, einen Promille-

tester oder einen Mondphasenrechner basteln, dann wissen Sie jetzt, dass das nur der Anfang ist und in den folgenden Kapiteln noch viele weitere seriöse Aufgaben gelöst werden.

# Eine Stoppuhr programmieren

Ihr erstes Skript soll eine kleine Stoppuhr simulieren. Dafür brauchen Sie keine zusätzlichen Programme oder kostspielige Software. Die Windows-Bordmittel reichen für erste kleine Skripts völlig aus. Weil Skripts nichts weiter als einfache Textdateien sind, starten Sie deshalb als Erstes den eingebauten Windows-Editor:

Wählen Sie im Startmenü *Ausführen*, und geben Sie ein: *notepad* EINGABE. Der Windows-Editor öffnet sich.

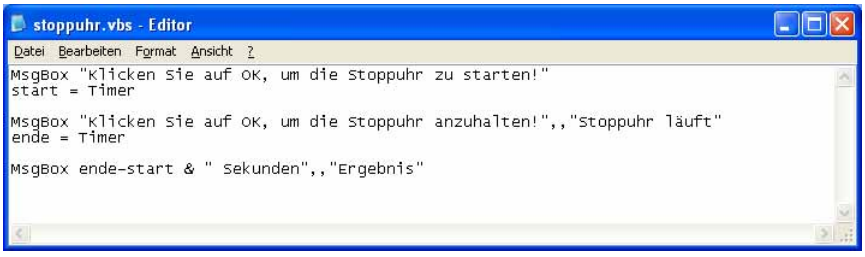


***Abbildung 2.1:*** *Der einfache Windows-Editor genügt, um erste Skripts zu verfassen*

Geben Sie nun das folgende Skript in den Editor ein:

```
MsgBox "Klicken Sie auf OK, um die Stoppuhr zu starten!"
start = Timer
MsgBox "Klicken Sie auf OK, um die Stoppuhr anzuhalten!",,"Stoppuhr läuft"
ende = Timer
MsgBox ende - start & " Sekunden",,"Ergebnis"
```

***Listing 2.1:*** *Eine einfache Stoppuhr*

Bis jetzt ist Ihr Skript einfacher Text. Damit dieser Text zu einem ausführbaren Skript wird, ist keine Magie nötig, sondern nur ein kleiner Kniff. Speichern Sie den Text unter einem beliebigen Namen und hängen Sie an den Namen die Erweiterung *.vbs* an. Speichern Sie den Text zum Beispiel als *stoppuhr.vbs*. Wenn Sie alles richtig gemacht haben, erscheint eine neue Datei namens *stoppuhr.vbs* dort, wo Sie das Skript gespeichert haben, und trägt ein Skriptsymbol.


***Abbildung 2.2:*** *Skriptdateien bekommen ein typisches Symbol*

Probieren Sie als Nächstes Ihre Stoppuhr aus. Dazu doppelklicken Sie auf die neue Datei *stoppuhr.vbs*. Es erscheint eine Meldung, dass Sie per Klick auf *OK* die Stoppuhr starten können. Sobald Sie ein zweites Mal auf *OK* klicken, wird die Stoppuhr wieder angehalten, und Sie sehen auf die Millisekunde genau, wie viel Zeit zwischen den beiden Klicks verstrichen ist.


***Abbildung 2.3:*** *Das Ergebnis Ihrer hochauflösenden Stoppuhr*

**TIPP** Sollte Ihr Skript nicht so funktionieren wie erwartet, sondern stattdessen eine Fehlermeldung erscheinen (Abbildung 2.4), dann haben Sie das Skript wahrscheinlich nicht richtig abgetippt.


***Abbildung 2.4:*** *Der Windows Script Host meldet einen Fehler und gibt Zeile und Zeichen (Spalte) an*

Bei Skripts kommt es auf jedes einzelne Zeichen an. Die Fehlermeldung stammt vom Windows Script Host, also demjenigen, der im Hintergrund Ihr Skript liest und ausführen soll. Sie zeigt an, dass der Windows Script Host Ihr Skript nicht richtig verstehen kann.

Macht aber nichts, per Rechtsklick auf die Datei *stoppuhr.vbs* können Sie im Kontextmenü *Bearbeiten* wählen und so das Skript wieder in den Windows-Editor laden. Die Fehlermeldung verrät Ihnen die Zeile und das Zeichen, wo der Fehler passiert ist.

Kontrollieren Sie diese Stelle. Haben Sie vielleicht ein Anführungs- oder ein Leerzeichen vergessen? Nachdem Sie den Fehler gefunden und korrigiert haben, speichern Sie das Skript mit *Datei/Speichern* neu und probieren das Skript danach einfach noch einmal aus.

## Das Skript Zeile für Zeile analysieren

Lassen Sie uns das kleine Skript nun Zeile für Zeile besprechen. Sie werden sehen, dass Ihre kleine Stoppuhr bereits viele wesentliche Grundelemente enthält, die Sie später in allen übrigen Skripts auch wieder finden. Beginnen wir mit der ersten Zeile:

```
MsgBox "Klicken Sie auf OK, um die Stoppuhr zu starten!"
```

In dieser Zeile findet sich das Wort *MsgBox* und ein Text in Anführungszeichen. Solcher Text wird künftig »String« genannt. Das Wort *MsgBox* ist Ihr erster Befehl. *MsgBox* liefert ein Dialogfeld mit einer *OK*-Schaltfläche. Den Text, den das Dialogfeld anzeigen soll, können Sie selbst bestimmen.


***Abbildung 2.5:*** *Dieses Dialogfeld stammt vom* MsgBox*-Befehl. Den Text können Sie selbst bestimmen*

**HINWEIS** Steht bei Ihrem Dialogfeld in der Titelleiste »VBScript«, und wundern Sie sich gerade, warum das in Abbildung 2.5 anders ist? Die Lösung ist ganz banal: Ab Windows XP mit Service Pack 2 zeigt *MsgBox* in den Dialogfeldern kein »VBScript« mehr an, und gleich werden Sie sehen, wie Sie eigene Texte in der Titelleiste unterbringen.

Probieren Sie aus, was passiert, wenn Sie den String hinter *MsgBox* ändern, das Skript neu speichern und dann noch einmal ausführen. Wie Sie sehen, bestimmen Sie allein, welcher Text im Dialogfeld angezeigt wird. Achten Sie nur penibel darauf, dass der String immer in Anführungszeichen steht und Sie nicht etwa das abschließende Anführungszeichen vergessen.

Schauen Sie sich die Zeile dann noch einmal an. Skripts werden von oben nach unten abgearbeitet, das ist klar. Aber wussten Sie, dass die einzelnen Zeilen nicht von links nach rechts gelesen werden, sondern von rechts nach links?

Tatsächlich also »sieht« der Windows Script Host, der das Skript später ausführt, zuerst den String, also den Text in Anführungszeichen. Danach erst wird der Befehl *MsgBox* gelesen, und weil es ein Befehl ist, wird ihm der String als so genanntes Argument übergeben. Befehle brauchen Argumente, damit sie wissen, was sie für Sie tun sollen. Woher sonst sollte *MsgBox* wissen, welchen Text es in seinem Dialogfeld anzeigen soll?

Die erste Zeile enthält also bereits die folgenden Grundelemente:

- **Befehl:** Die Zeile verwendet den *MsgBox*-Befehl, um ein Dialogfeld anzuzeigen. Im Verlaufe dieses Buchs werden Sie Hunderte weiterer Befehle kennen lernen, die alle im Grunde genauso verwendet werden wie *MsgBox*.
- **Argumente:** Damit ein Befehl weiß, was er genau für Sie tun soll, kann er Argumente entgegennehmen. Die Argumente stehen direkt hinter dem Befehl. Sie haben gesehen: Der *MsgBox*-Befehl möchte mindestens ein Argument erhalten, in dem steht, welchen Text das Dialogfeld anzeigen soll.
- **Literal:** Literale sind Werte, die Sie bei der Skripterstellung fest vorgeben und die sich nicht im Verlauf des Skripts ändern. Das Argument, das in der ersten Zeile hinter dem *MsgBox*-Befehl steht, ist ein solches Literal. In diesem Fall enthält das Literal einen String. Strings werden immer durch Anführungszeichen gekennzeichnet, damit klar ist, wo der Text beginnt und wo er aufhört.

Schauen Sie sich nun die zweite Zeile an, und denken Sie daran, die Zeile diesmal gleich von rechts nach links zu lesen.

```
start = Timer
```

Das auffälligste Merkmal in dieser Zeile ist das Gleichheitszeichen. Hier findet eine Zuweisung statt, es wird ein Wert von einem Ort an einen anderen geschoben. Weil die Zeile von rechts nach links gelesen wird, findet also eine Zuweisung von *Timer* an *start* statt. Das hört sich mächtig kompliziert an, ist es aber bei näherer Betrachtung gar nicht.

*Timer* ist genau wie *MsgBox* ein eingebauter Befehl und liefert die aktuelle Systemzeit zurück, fast wie ein eingebautes Uhrwerk. Tatsächlich liefert *Timer* die Anzahl der verstrichenen Sekunden seit Mitternacht, und zwar bis auf die Millisekunde genau. Damit ist diese Zeitangabe besonders hochauflösend und eignet sich perfekt dazu, um kurze Zeiträume zu stoppen.

Weil sich das Skript den momentanen Zeitpunkt merken möchte, weist es die aktuelle Systemzeit, die von *Timer* bei jedem Aufruf neu geliefert wird, *start* zu. Fragt sich nur: Was ist *start*?

*Start* ist eine Variable. Variablen kann man sich wie Karteikärtchen vorstellen, mit denen sich Ihr Skript Dinge merken kann. Dazu geben Sie der Variablen einfach einen Namen und weisen ihr

einen beliebigen Wert zu. In diesem Fall wird *start* das zugewiesen, was *Timer* liefert, nämlich die aktuelle Zeit. Während die Zeit, die *Timer* bei jedem Aufruf liefert, also munter weitertickt, hat sich das Skript den Startzeitpunkt in der Variablen *start* gemerkt.

Probieren Sie doch mal aus, was passiert, wenn Sie den Begriff *start* in Ihrem Skript durch den Begriff *ZeitpunktAnDemEsLosgeht* ersetzen. Wenn Sie das tun, dann müssen Sie allerdings auch den Begriff *start* in der letzten Zeile durch den neuen Namen ersetzen, damit alles wie gewohnt funktioniert.

Haben Sie alles richtig gemacht, dann funktioniert die Stoppuhr trotz der Änderung genau wie vorher. Es ist also egal, unter welchem Namen Sie die Information speichern, solange Sie in Ihrem Skript für dieselbe Information überall denselben Namen verwenden.

**TIPP** Gerade haben Sie gesehen, dass Sie Variablen so nennen dürfen, wie Sie es am besten finden – mit ganz wenigen Ausnahmen. Erlaubt sind alle Zeichen von »A« bis »Z«, der Unterstrich und alle Zahlen, solange die Zahlen nicht am Anfang stehen.

Sonderzeichen wie zum Beispiel die deutschen Umlaute sind nicht erlaubt. Leerzeichen sind ebenfalls tabu, denn Leerzeichen trennen verschiedene Begriffe voneinander, und Ihr Variablenname würde so in mehrere Teile zerfallen.

Und natürlich sollten Sie Ihre Variablen nicht so nennen wie die bereits vorhandenen Befehle. Nennen Sie Ihre Variable also nicht *MsgBox* oder *Timer*, denn dann wüsste niemand mehr, was Sie eigentlich genau meinen. Dafür dürfen Variablen bis zu 255 Zeichen lang sein – genug Platz für aussagekräftige Namen.

Wieder hat diese Zeile einige Grundlagen verwendet, die in fast jedem Skript vorkommen:

- **Zuweisung:** Mit dem Gleichheitszeichen werden einer Variablen neue Inhalte zugewiesen. Die Variable steht links vom Gleichheitszeichen, und der Wert, der künftig in der Variablen aufbewahrt werden soll, steht rechts vom Gleichheitszeichen.
- **Variablen:** Ihnen stehen so viele »Merkzettel« in Ihrem Skript zur Verfügung, wie Sie brauchen. Legen Sie so viele Variablen an wie nötig. Ihr Skript kann so später unter dem Namen der Variablen den Inhalt wieder abrufen, den Sie in der Variablen gespeichert haben. Das Skript kann also den Startzeitpunkt, zu dem die Stoppuhr gestartet wurde, unter dem Namen *start* jederzeit wieder abrufen und macht sich das ein paar Zeilen später zunutze, um die verstrichene Zeit auszurechnen.

Nun wird es Zeit für die dritte Skriptzeile:

```
MsgBox "Klicken Sie auf OK, um die Stoppuhr anzuhalten!",,"Stoppuhr läuft"
```

Diese Zeile sieht fast so aus wie die erste Zeile, und richtig, auch hier wird ein Dialogfeld angezeigt. Allerdings sieht das Argument hinter *MsgBox* sonderbar aus. Es besteht aus zwei Strings (Texte in Anführungszeichen, Sie erinnern sich? Beide Strings werden Literal genannt, denn beide Strings sind fest vorgegeben und werden sich nicht ändern). Zwischen den beiden Strings stehen zwei Kommas.


***Abbildung 2.6:*** MsgBox *kann auch Dialogfelder mit separatem Text in der Titelleiste anzeigen*

Tatsächlich werden dem *MsgBox*-Befehl in dieser Zeile nicht ein, sondern drei Argumente übergeben. Drei? Schauen Sie genau hin! Argumente werden immer durch ein Komma vom nächsten Argument getrennt.

Hinter dem ersten String folgt ein Komma, also ein weiteres Argument. Nach dem ersten Komma folgt allerdings gleich noch ein Komma und dann das dritte Argument. Und wo ist das zweite Argument geblieben, das Sie eigentlich zwischen dem ersten und dem zweiten Komma erwartet haben? Es fehlt! Es wurde einfach übersprungen. Wenn das Skript ein Argument überspringt, dann gilt für dieses Argument ein Standardwert. Dieser Standardwert wird vom jeweiligen Befehl selbst festgelegt.

Wie lauten also die drei Argumente, die *MsgBox* hier bekommt?

- **1. Argument:** "*Klicken Sie auf OK, um die Stoppuhr anzuhalten!*" (Text im Dialogfeld)
- **2. Argument:** nichts
- **3. Argument:** "*Stoppuhr läuft*" (Text in der Titelleiste des Dialogfelds)

Und welche Bedeutung hat das dritte Argument? Wenn Sie das Skript noch einmal starten, werden Sie den Unterschied zwischen dem ersten und dem zweiten Dialogfeld sicher schnell entdecken: Das dritte Argument legt den Text in der Titelleiste des Dialogfelds fest.

Was passiert in der vierten Skriptzeile?

```
ende = Timer
```

Diese Zeile sieht fast genauso aus wie die zweite Zeile: eine Zuweisung. Wieder wird die aktuelle Systemzeit vom *Timer*-Befehl erfragt und in einer zweiten Variablen namens *ende* aufbewahrt. Ihr Skript hat sich nun also in den Variablen *start* und *ende* zwei Zeiten gemerkt, nämlich den Start der Stoppuhr und den Zeitpunkt des Stopps. Interessant wird es deshalb nun in der letzten Skriptzeile:

```
MsgBox ende - start & " Sekunden",,"Ergebnis"
```

Hier sehen Sie wieder einen *MsgBox*-Befehl, der wieder drei Argumente übergeben bekommt:

- **1. Argument:** *ende - start &* " *Sekunden*"
- **2. Argument:** nichts
- **3. Argument:** "*Stoppuhr läuft*"

Wirklich neu ist der Inhalt des ersten Arguments. Hier findet sich diesmal nämlich nicht nur ein String in Anführungszeichen, sondern außerdem die beiden Variablen und zwei Operatoren – ein Minuszeichen und ein kaufmännisches Und-Zeichen (&).


***Abbildung 2.7:*** *Das dritte Dialogfeld zeigt das Ergebnis an*

Das letzte Dialogfeld meldet die Zeitdifferenz zwischen Start- und Endzeitpunkt. Den kann das Skript ganz einfach über die Berechnung *ende - start* ausrechnen. Das Ergebnis ist die Zeit in Sekunden, die zwischen *start* und *ende* liegt. Das &-Zeichen verknüpft diese errechnete Information mit einem weiteren String, sodass das Ergebnis nicht nur eine Zahl ist, sondern galant die verstrichenen Sekunden meldet. Mit dem &-Zeichen können Sie also verschiedene Informationen zu einem Gesamtstring zusammenfassen.

Mit dieser letzten Skriptzeile haben Sie wieder ein wichtiges Element kennen gelernt, das in den meisten Skripts vorkommt:

- **Operatoren:** Operatoren sind Zeichen, die die Begriffe links und rechts davon miteinander verknüpfen. Das können mathematische Operatoren sein wie zum Beispiel das Minuszeichen, mit dem das Skript die beiden Zeiten voneinander subtrahiert. Es können aber auch Verknüpfungsoperatoren sein, mit denen zwei Begriffe wie zum Beispiel das Rechenergebnis und ein fester String zu einem neuen String zusammengefügt werden.

Ihr erstes Skript zeigt nicht nur viele Grundelemente, die Sie später in allen künftigen Skripts so oder ähnlich wieder finden, auch die »Rechtschreibregeln« in VBScript sind wichtig:

- **Groß- und Kleinschreibung:** Es spielt keine Rolle, ob Sie Ihre Variable *start* oder *Start* oder *START* nennen. Und auch der Befehl *MsgBox* funktioniert noch, wenn Sie ihn als *MSGBOX* oder *msgbox* schreiben. VBScript unterscheidet nicht zwischen Groß- und Kleinschreibung und eliminiert damit eine ganz wesentliche Fehlerquelle von vornherein.
- **Zeilenbegrenzung:** Eine Zeile ist für VBScript dann zu Ende, wenn Sie EINGABE drücken und eine neue Zeile beginnen. Bei VBScript gibt es also anders als zum Beispiel bei JScript kein Zeilenendekennzeichen wie »;«, das man gern vergisst. Allerdings hat die Zeilenbegrenzung durch EINGABE einen Nachteil: Zeilen werden unter Umständen sehr lang und laufen dann rechts aus dem Editorfenster heraus, sodass Sie möglicherweise umständlich mit der Maus scrollen müssen. Zwar sind solche ultralangen Zeilen selten, aber auch dafür gibt es eine Lösung: Mit dem »_«-Zeichen (Unterstrich) lassen sich lange Zeilen in mehrere Zeilen aufteilen. Wenn VBScript also am Ende einer Zeile ein Leerzeichen und dann den Unterstrich entdeckt, dann setzt es die Ausführung dieser Zeile in der nächsten Zeile fort. Allerdings darf der Unterstrich nicht innerhalb von Anführungszeichen stehen, denn dann würde VBScript ihn als Teil des Textes ansehen, der in diesen Anführungszeichen steht.

```
MsgBox "Dies ist eine lange Zeile, die auf mehrere Zeilen verteilt wird!", , _
    "Titeltext des Dialogfelds"
```

**TIPP** Auch wenn es nicht empfohlen wird, soll der Hinweis auf ein besonderes Steuerzeichen in VBScript nicht fehlen: Der »:« wird von VBScript ebenfalls als Zeilenendekennzeichen verstanden. Er ist sozusagen der Gegenspieler zu »_« und sorgt dafür, dass mehr als eine Anweisung in einer einzelnen Zeile untergebracht werden kann. Das Ergebnis ist allerdings nicht besonders gut lesbar. Listing 2.1 könnte man also auch so schreiben:

```
MsgBox "Klicken Sie auf OK, um die Stoppuhr zu starten!":start = Timer
MsgBox "Klicken Sie auf OK, um die Stoppuhr anzuhalten!",,"Stoppuhr läuft":ende = Timer
MsgBox ende - start & " Sekunden",,"Ergebnis"
```

## Selbst nachschlagen: die WSH-Referenz nutzen

Gerade haben Sie gesehen, dass Ihr Skript zwei Befehle verwendet hatte: *MsgBox* und *Timer*. Woher aber weiß man, dass diese beiden Wörter zwei Befehle sind – und vor allen Dingen: Welche Befehle gibt es sonst noch?

Wenn Sie in Frankreich Brötchen und Käse bestellen, dann wissen Sie, dass Sie dafür besser französisch sprechen, und wenn Sie die Begriffe für Ihre Bestellung nicht im Kopf haben, dann schlagen Sie sie eben in einem französischen Wörterbuch nach. Genauso funktioniert das auch mit Skripts.

## Installation der WSH-Referenz

Legen Sie die Buch-CD ein und wechseln Sie in den Ordner *WSH\Dokumentation deutsch*. Darin befindet sich die Datei *scd56de.exe*. Doppelklicken Sie auf die Datei. Das Installationsprogramm startet und fragt, in welchem Ordner die Referenz installiert werden soll. Das Installationsprogramm schlägt den Ordner *%PROGRAMFILES%\Microsoft Windows Script\ScriptDocs* vor. Klicken Sie auf *OK*.


***Abbildung 2.8:*** *Die Sprachreferenz für den Windows Script Host installieren*

Die Sprachreferenz wird entpackt und im angegebenen Ordner gespeichert. Außerdem wird im *Programme*-Menü die neue Programmgruppe *Microsoft Windows Script* angelegt. Öffnen Sie im Startmenü das *Programme*-Menü und wählen *Microsoft Windows Script,* dann genügt ein Klick auf *Windows Script V5.6 Dkumentation,* um die Sprachreferenz zu öffnen.

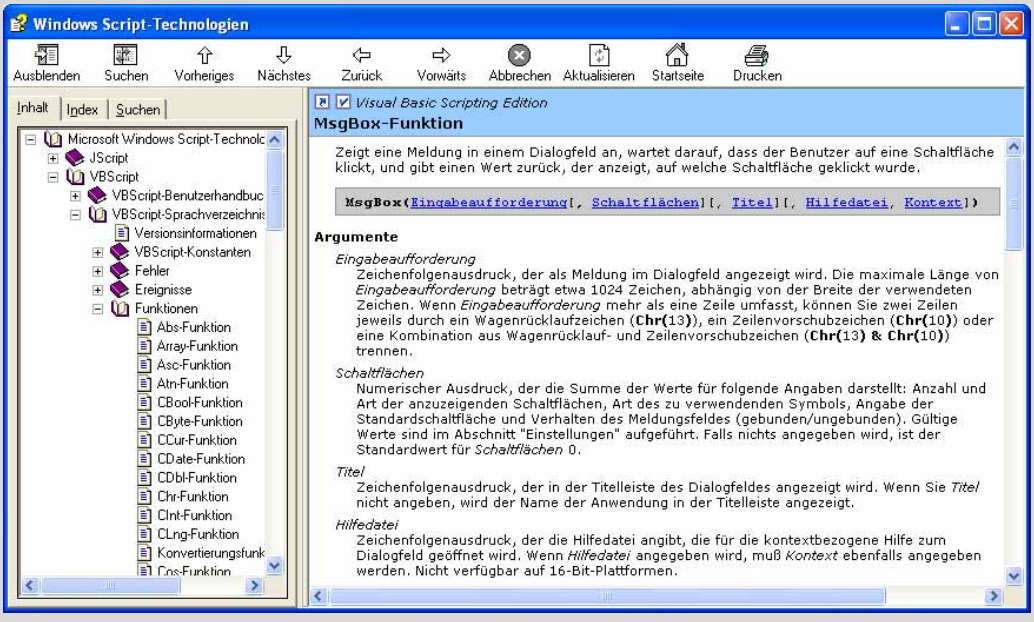


***Abbildung 2.9:*** *Die Referenz liefert Detailinformationen zu allen Befehlen und Sprachbestandteilen*

Zuerst müssen Sie wissen, welche Sprache Ihr Skript eigentlich spricht. Die Sprache heißt *VBScript* und ist ein Ableger der weit verbreiteten Sprache Visual Basic (daher »VB« in »VBScript«). Das erklärt auch, warum Ihr Text im Windows-Editor zu einem Skript wurde, als Sie die Erweiterung *.vbs* an den Dateinamen angehängt haben. Die Erweiterung *.vbs* ist reserviert für Skripts, die in der Sprache VBScript geschrieben wurden.

Jetzt, wo Sie wissen, um welche Sprache es sich handelt, brauchen Sie nur noch ein Wörterbuch für VBScript. Das brauchen Sie nicht teuer zu erstehen, denn Sie besitzen es bereits. Auf der Buch-CD finden Sie die deutschsprachige VBScript-Referenz im Ordner *WSH\Dokumentation deutsch* in der Datei *scd56de.exe*. Sie brauchen diese Datei nur noch zu installieren.

Wie nützlich die Sprachreferenz ist, erkennen Sie sofort, wenn Sie die beiden Befehle *MsgBox* und *Timer* in der Hilfe nachschlagen. Und das machen Sie am besten so:

Klicken Sie auf das Register *Index*, und geben Sie den gesuchten Begriff ein, also zum Beispiel *Timer*. Dann drücken Sie EINGABE. Einen Moment später sehen Sie alle Begriffe, die das Wort *Timer* enthalten. Doppelklicken Sie auf *Timer-Funktion* (Abbildung 2.10).

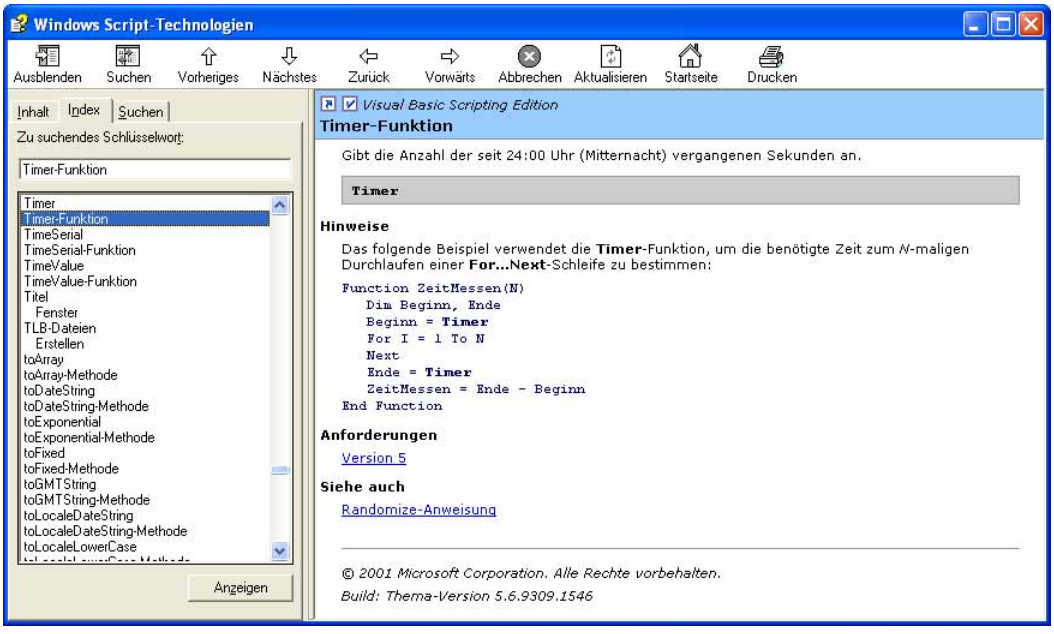


***Abbildung 2.10:*** *Die* Timer-*Funktion in der Referenz nachschlagen*

In der rechten Spalte sehen Sie jetzt alle wissenswerten Details zu *Timer*. Schon erfahren Sie, dass *Timer* die Zahl der Sekunden seit Mitternacht zurückliefert. Außerdem finden Sie ein Beispiel, das ein klein wenig so aussieht wie Ihr eigenes Skript, aber noch einige weitere Bestandteile enthält, die hier noch nicht besprochen wurden.

Schlagen Sie jetzt *MsgBox* auf gleiche Weise nach. Ein Doppelklick auf *MsgBox-Funktion* liefert einen langen Erklärungstext, und jetzt sehen Sie auch, wofür die vielen Argumente, die *MsgBox* verarbeiten kann, eigentlich da sind.

Wie man dort sieht (Abbildung 2.11), berücksichtigt *MsgBox* bis zu fünf Argumente. Bis auf das erste Argument sind alle übrigen Argumente in eckige Klammern gestellt. Eckige Klammern zei-

gen an: Diese Argumente sind optional, brauchen also nur bei Bedarf angegeben zu werden. Das erste Argument ist nicht von eckigen Klammern eingerahmt, ist also immer Pflicht.


***Abbildung 2.11:*** *Die Referenz verrät, welche Argumente ein Befehl verlangt*

Und so löst sich das Rätsel des zweiten Arguments. In Ihrem Skript war das zweite Argument immer undefiniert – Sie haben schlicht keinen Wert dafür angegeben. Ein Blick in die Dokumentation zeigt: Das zweite Argument bestimmt, welche Schaltflächen und Symbole das Dialogfeld anzeigt. Geben Sie nichts an, dann ist der Wert *0* und entspricht laut Tabelle *vbOKOnly* (Abbildung 2.12).

| Konstante | Wert | Beschreibung |
|---|---|---|
| vbOKOnly | 0 | Nur Schaltfläche **OK** anzeigen. |
| vbOKCancel | 1 | Anzeigen der Schaltflächen **OK** und **Abbrechen**. |
| vbAbortRetryIgnore | 2 | Anzeigen der Schaltflächen **Abbrechen**, **Wiederholen** und **Ignorieren**. |
| vbYesNoCancel | 3 | Anzeigen der Schaltflächen **Ja**, **Nein** und **Abbrechen**. |
| vbYesNo | 4 | Anzeigen der Schaltflächen **Ja** und **Nein**. |
| vbRetryCancel | 5 | Anzeigen der Schaltflächen **Wiederholen** und **Abbrechen**. |
| vbCritical | 16 | Anzeigen des Stopp-Symbols. |
| vbQuestion | 32 | Anzeigen des Fragezeichen-Symbols. |
| vbExclamation | 48 | Anzeigen des Symbols **Warnung**. |
| vbInformation | 64 | Anzeigen des Symbols **Information**. |
| vbDefaultButton1 | 0 | Erste Schaltfläche ist Voreinstellung. |
| vbDefaultButton2 | 256 | Zweite Schaltfläche ist Voreinstellung. |
| vbDefaultButton3 | 512 | Dritte Schaltfläche ist Voreinstellung. |
| vbDefaultButton4 | 768 | Vierte Schaltfläche ist Voreinstellung. |
| vbApplicationModal | 0 | Anwendungsgebunden. Der Benutzer muss auf das Meldungsfeld reagieren, bevor er die Arbeit mit der aktuellen Anwendung fortsetzen kann. |
| vbSystemModal | 4096 | Systemgebunden. Alle Anwendungen werden unterbrochen, bis der Benutzer auf das Meldungsfeld reagiert. |

***Abbildung 2.12:*** *Die Referenz verrät, welche Werte für das zweite* MsgBox-*Argument möglich sind*

Das stimmt: Ihre Dialogfelder zeigten immer nur eine *OK*-Schaltfläche, was für das Skript ja auch ausreichend war. Sie sehen jetzt aber, dass auch andere Werte als »nichts« oder *0* erlaubt sind. Was müssten Sie wohl als zweites Argument angeben, damit *MsgBox* Ihnen ein Dialogfeld mit *OK* und *Abbrechen* anzeigt? Richtig: entweder die Zahl *1* oder die vordefinierte Konstante *vbOKCancel*: Sie verhält sich im Grunde wie eine Variable, nur steht ihr Wert bereits von Anfang an fest und lässt sich auch nicht ändern.

```
MsgBox "Weitermachen?", vbOKCancel
MsgBox "Weitermachen?", 1
```

Die Tabelle in der Referenz zeigt, dass das zweite Argument aber nicht nur die Art der Schaltflächen festlegt, sondern zum Beispiel auch, ob und welches Symbol im Dialogfeld angezeigt wird. *vbQuestion* blendet zum Beispiel ein Fragezeichen ein. Weil auch diese Einstellung Teil des zweiten Arguments ist, addieren Sie einfach die Konstanten oder Zahlenwerte, die Sie verwenden wollen:

```
MsgBox "Weitermachen?", vbOKCancel + vbQuestion
MsgBox "Weitermachen?", 1 + 32
MsgBox "Weitermachen?", 33
```

Alle drei Varianten sind erlaubt, aber es wird sofort deutlich, dass die erste Variante am einfachsten zu lesen ist. Würden Sie Ihr Skript in einigen Tagen noch einmal hervorkramen, dann wüssten Sie sicher nicht, was der Zahlenwert *33* bedeutet. *vbOkCancel + vbQuestion* dagegen bleibt auch dann noch verständlich.

Was Sie hier erkannt haben, ist wieder eine wichtige Grundlage für alle Skripts:

- **Optionale Argumente:** Nicht alle Argumente, die ein Befehl erwartet, müssen Pflichtangaben sein. Ist ein Argument nicht Pflicht, dann nennt man es »optional«, und es braucht nicht angegeben zu werden. Optionale Argumente werden in der WSH-Referenz mit eckigen Klammern gekennzeichnet. Lässt man ein optionales Argument weg, dann gilt für dieses Argument ein Standardwert, der vom jeweiligen Befehl abhängt. Bei *MsgBox* lautet der Standardwert für das zweite Argument zum Beispiel 0.
- **Konstanten:** VBScript liefert eine ganze Reihe von fest eingebauten Variablen mit, die nicht leer sind, sondern bereits vordefinierte Werte enthalten. Weil der Inhalt dieser Variablen feststeht und auch später nicht geändert werden kann, nennt man solche Variablen auch Konstanten. Klar, denn ihr Inhalt ist nicht mehr variabel, sondern bleibt konstant. Alle eingebauten VBScript-Konstanten beginnen immer mit »vb«. *MsgBox* nutzt Konstanten zum Beispiel, damit Sie im zweiten Argument sehr einfach und leserlich festlegen können, welche Schaltflächen das Dialogfeld anzeigen soll. Oder finden Sie nicht auch, dass *vbOkCancel* aussagekräftiger ist als *1*? Sie können sich über alle eingebauten VBScript-Konstanten informieren, indem Sie in der Referenz auf das Register *Inhalt* klicken und dann in der Baumansicht zu *VBScript/VBScript-Sprachverzeichnis/VBScript-Konstanten* wechseln. Dort entdecken Sie viele weitere Konstanten, zum Beispiel die Stringkonstanten, mit denen Sie Tabulatoren oder Zeilenumbrüche in Ihre Texte einbauen.

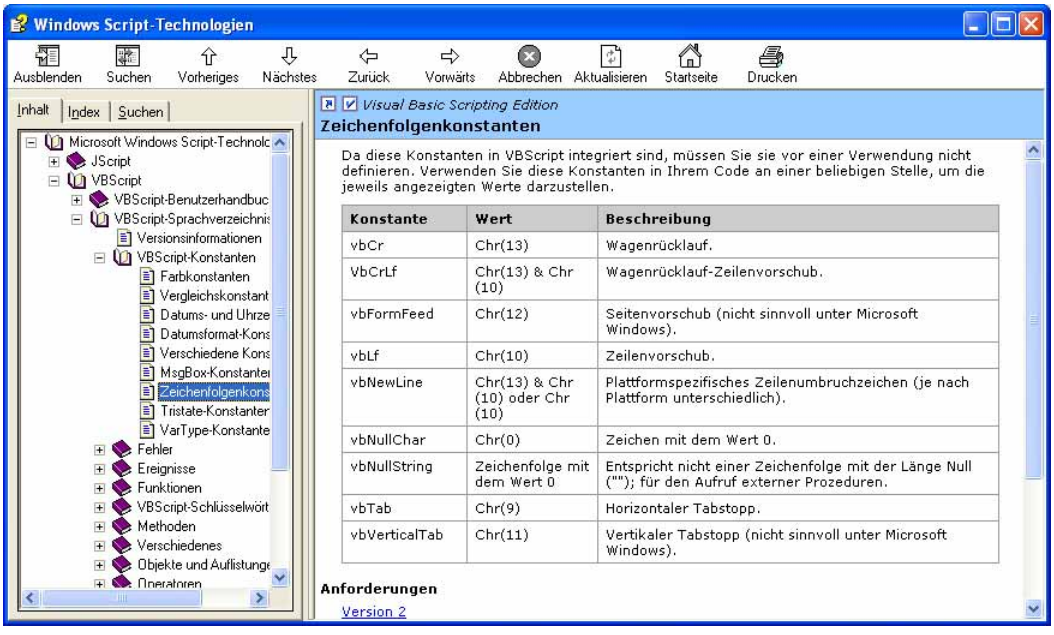


***Abbildung 2.13:*** *Alle VBScript-Konstanten in der Referenz nachschlagen*

**ACHTUNG** Die elektronische Referenz, mit der Sie gerade arbeiten, behandelt nicht nur die Sprache VBScript, sondern auch die Sprache JScript, die ebenfalls in Windows integriert ist. In diesem Buch wird nur VBScript verwendet, und Sie sollten bei Streifzügen durch die elektronische Referenz darauf achten, ebenfalls nur VBScript-Funktionen nachzuschlagen.

Würden Sie aus Versehen eine JScript-Funktion in einem VBScript-Skript verwenden, dann wären Fehler natürlich vorprogrammiert. Ob eine Referenz in der Hilfedatei für VBScript oder für JScript geschrieben ist, erkennen Sie zum Glück leicht oben rechts an der Kopfzeile.

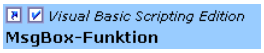


***Abbildung 2.14:*** *Achten Sie darauf, dass Sie in der Referenz auch tatsächlich eine VBScript-Funktion nachschlagen*

# Grundelemente in allen Skripts

Ihr Beispielskript mag zwar noch sehr einfach und klein sein, aber es verwendet bereits drei der vier Grundbestandteile, aus denen sich ein jedes Skript zusammensetzt:

- **Befehle:** Das Skript hat den Befehl *MsgBox* eingesetzt, um Meldungen an den Skriptbenutzer auszugeben. Außerdem hat es den Befehl *Timer* verwendet, mit dem es die aktuelle Systemzeit abfragen konnte. Diese beiden Befehle sind natürlich nur zwei von Hunderten von Befehlen, und wenn Sie anstelle einer Stoppuhr lieber Werte in die Windows-Registrierungsdatenbank schreiben oder remote einen anderen Computer herunterfahren möchten, dann geht das ebenfalls. Sie müssen nur den passenden Befehl kennen. In den folgenden Kapiteln werden Sie viele weitere Befehle kennen lernen. Auch wenn es Ihnen jetzt gerade in den Fingern juckt, sofort weitere Befehle auszuprobieren, empfehle ich Ihnen, noch ein wenig in diesem Kapitel weiterzulesen und erst mal in Ruhe die Grundlagen zu verdauen.
- **Variablen:** Das Skript hat sich zwei Werte gemerkt, nämlich die Systemzeit zum Start des Skripts und die Systemzeit beim zweiten Klick. Dazu hat es die Variablen *start* und *ende* verwendet. Variablen sind also ein wichtiger Baustoff Ihrer Skripts, und Sie dürfen so viele Variablen anlegen und verwenden, wie Sie für nötig halten. Einige der Einschränkungen, die für die Namensgebung gelten, haben Sie oben bereits gelesen. Im Grunde aber dürfen Sie Variablen so nennen, wie Sie wollen, und sollten ihnen möglichst aussagekräftige Namen geben, damit Sie auch später noch wissen, was Sie eigentlich in den einzelnen Variablen aufbewahren.
- **Operatoren:** Mit Operatoren wird gerechnet, und das Skript hat bereits zwei Operatoren verwendet. Um die gestoppte Zeit zu ermitteln, hat es mit dem Minus-Operator die Endzeit von der Startzeit abgezogen und so die zwischen den beiden Klicks verstrichene Zeit ermittelt. Und mit dem &-Operator wurde eine besondere »Berechnung« durchgeführt, nämlich verschiedene Einzelinformationen wie zum Beispiel die errechnete Zeit mit festem String (einem Literal, erinnern Sie sich?) verbunden. Das Ergebnis des &-Operators ist immer ein String, den man als Ergebnis anzeigen kann.

Fragen Sie sich gerade, welches der vierte Grundbaustein von Skripts ist? Der vierte Baustoff sind Strukturelemente wie zum Beispiel Schleifen oder Bedingungen. Die kommen allerdings in sehr einfachen Skripts nur selten vor. Schleifen und Bedingungen begegnen Ihnen etwas später in diesem Kapitel.

# Befehle genauer verstehen

Als Oberfeldwebel könnten Sie sich natürlich vor einem Rekruten aufbauen und brüllen: »Flieg! Das ist ein Befehl!«. Trotzdem würde der Rekrut nicht abheben. Er kennt Ihren Befehl »Flieg!« nicht und weiß auch nicht, wie er ihn umsetzen soll. Hätten Sie denselben Befehl einem Jagdflieger erteilt, dann wäre das schon eine ganz andere Geschichte, und er würde höchstens nachhaken: »Und wohin bitte?«.

Befehle sind also Aufforderungen an Ihr Skript, bestimmte Dinge zu tun. Bei manchen Befehlen genügt der Befehl allein, damit das Skript weiß, was Sie von ihm wollen. Bei anderen Dingen braucht der Befehl Zusatzinformationen, damit man ihn ausführen kann. Und natürlich können Sie Ihrem Skript auch Befehle erteilen, die es gar nicht kennt. Sie sind der Chef. Dann allerdings erhalten Sie vom Skript eine Fehlermeldung.

Als Chef können Sie zwei verschiedene Arten von Befehl brüllen:

- **Befehle ohne Sprecherlaubnis:** Brüllen Sie »Stillgestanden!«, dann steht Ihr Rekrut stramm, und Sie haben Ihr Ziel erreicht. Eine besondere Antwort erwarten Sie hier natürlich nicht.
- **Befehle mit Sprecherlaubnis:** Brüllen Sie dagegen »Wie spät ist es, Meier?«, dann hat dieser Befehl nur einen Sinn, wenn Rekrut Meier Ihnen eine Antwort geben darf – und eine Uhr trägt. Befehle mit Sprecherlaubnis heißen Funktionen. Sie liefern eine Antwort zurück.

Geben Sie Ihrem Skript doch einfach testweise mal einen Befehl! Dazu öffnen Sie den Windows-Editor und geben dieses Skript ein:

```
Flieg
```

Speichern Sie es als *flieg.vbs,* und führen Sie es dann aus!

Sicher haben Sie schon vermutet, dass Ihr Skript nicht fliegen kann, und so erhalten Sie relativ erwartungsgemäß den Fehler »Typen unverträglich«. Ihr Skript kennt den Befehl *Flieg* einfach nicht.


***Abbildung 2.15:*** *Unbekannte Befehle quittiert der Windows Script Host mit dem Fehler »Typen unverträglich«*

Weil Sie aber im letzten Abschnitt schon ein Skript kennen gelernt haben, kennen Sie auch schon ein paar gültige Skriptbefehle, zum Beispiel *MsgBox*. Klicken Sie Ihr Skript *befehl.vbs* deshalb mit der rechten Maustaste an und wählen Sie im Kontextmenü *Bearbeiten*. Ändern Sie dann den Befehl *Flieg* in den Befehl *MsgBox* um. Den kennt VBScript, das wissen Sie bereits. Speichern Sie das geänderte Skript mit *Datei/Speichern* ab, und führen Sie es erneut aus.

Wumm. Schon wieder ein Fehler, aber diesmal ein anderer: »Falsche Anzahl an Argumenten«.

Ihr Skript hat den *MsgBox*-Befehl also akzeptiert, aber das reicht ihm nicht. Genau wie der Jagdflieger auf Ihren »Flieg«-Befehl mit der Rückfrage »Wohin denn?« reagiert hat, kann auch der *MsgBox*-Befehl nur ausgeführt werden, wenn er weiß, welchen Text er eigentlich im Dialogfeld anzeigen soll. Diese nötigen Zusatzinformationen werden bei Befehlen »Argumente« genannt,

und man gibt sie einfach hinter dem Befehl an. Korrigieren Sie Ihr Skript also noch einmal, und schreiben Sie diesmal:

```
MsgBox "Hallo Welt!"
```

Jetzt endlich klappt die Sache: Es erscheint ein Dialogfeld mit dem Text, den Sie angegeben haben.

Sie haben also erfolgreich einen Skriptbefehl erteilt und ihm als Argument die Information mit auf den Weg gegeben, die der Befehl für seine Arbeit braucht. Na also.


***Abbildung 2.16:*** *Stimmen der Befehl und seine Argumente, dann funktioniert das Skript prima*

Haben Sie dem *MsgBox*-Befehl damit eigentlich auch Sprecherlaubnis erteilt? Die Frage erscheint komisch, denn seit wann können Dialogfelder sprechen? Gemeint ist: Kann der *MsgBox*-Befehl ein Ergebnis an Sie zurückliefern?

Nein, der Befehl kann nur sein Dialogfeld zücken, aber sobald der Anwender es per Klick auf *OK* wegklickt, führt VBScript einfach die nächste Zeile aus, kümmert sich also nicht mehr um den *MsgBox*-Befehl und etwaige Informationen, die der Befehl nach erledigter Arbeit an Sie zurückliefern möchte.

Rufen Sie deshalb *MsgBox* doch noch einmal auf, diesmal aber mit Sprecherlaubnis. Das funktioniert so:

```
antwort = MsgBox("Hallo Welt!")
```

Denken Sie einfach daran, dass VBScript-Zeilen von rechts nach links gelesen werden, und Sie verstehen schnell, was hier passiert. Rechts wird der Befehl *MsgBox* aufgerufen und wieder mit dem für ihn nötigen Argument versorgt. Links vom Befehl steht diesmal allerdings ein Gleichheitszeichen, und der Befehl kann jetzt ein Ergebnis an die Variable *antwort* zurückmelden.

Sehen können Sie das Ergebnis so natürlich noch nicht, denn es lagert unsichtbar in der Variablen *antwort*. Und noch etwas hat sich geändert: Das Argument hinter *MsgBox* steht jetzt in Klammern. Immer wenn Sie links vom Befehl ein Ergebnis abzapfen, müssen die Argumente rechts vom Befehl in Klammern stehen.

Zwei Fragen drängen sich auf:

- Wie kann man das, was in *antwort* hinterlegt wurde, nun eigentlich sichtbar machen?
- Was hat *MsgBox* da eigentlich zurückgeliefert?

Die erste Frage ist leicht beantwortet. Sie kennen ja schon den *MsgBox*-Befehl, mit dem man ein Dialogfeld öffnet. Mit *MsgBox* können Sie also auch ganz elegant den Inhalt von Variablen sichtbar machen:

```
antwort = MsgBox("Hallo Welt!")
MsgBox antwort
```

In der ersten Zeile wird also *MsgBox* mit Sprecherlaubnis aufgerufen und meldet sein Ergebnis in der Variablen *antwort* zurück. In der zweiten Zeile wird *MsgBox* ohne Sprecherlaubnis aufgerufen, und als Argument wird diesmal einfach kein fester String mit auf den Weg gegeben, sondern die Variable *antwort*. *MsgBox* zeigt hier also an, was in *antwort* gespeichert ist.


***Abbildung 2.17:*** MsgBox *liefert in diesem Fall den Wert 1 zurück. Gleich finden Sie heraus, was die Zahl bedeutet*

**Rätsel 1:** Warum steht das Argument "*Hallo Welt*" in der ersten Zeile in Klammern, aber das Argument *antwort* in der zweiten Zeile nicht?

Wenn Sie dieses Skript ausführen, sehen Sie, dass *MsgBox* offenbar den Wert *1* zurückgeliefert hat. Und wenn Sie sich jetzt zu Recht fragen, warum *MsgBox* eine *1* liefert, dann wird es Zeit, mehr über diesen Befehl zu erfahren. Oben haben Sie ja bereits gelesen, wie man VBScript-Befehle in der WSH-Referenz nachschlägt, und dies auch schon für *MsgBox* getan.

Dabei hatte sich herausgestellt, dass der *MsgBox*-Befehl mehr als ein Argument akzeptiert. Das zweite Argument legt unter anderem fest, welche Schaltflächen in dem Dialogfeld angezeigt werden. Dämmert eine Erkenntnis? Richtig: Sobald das Dialogfeld mehr als eine Schaltfläche zeigt, könnte der Rückgabewert von *MsgBox* richtig praktisch werden, denn vielleicht verrät er ja, auf welche Schaltfläche der Anwender geklickt hat.

Und richtig: Schauen Sie sich das nächste Skript an! Es liefert unterschiedliche Werte zurück, je nachdem, auf welche Schaltfläche der Anwender geklickt hat:

```
antwort = MsgBox("Wollen Sie den Rechner herunterfahren?", vbYesNoCancel)
MsgBox antwort
```

***Listing 2.2:*** *Eine Anfrage an den Benutzer auswerten*

Natürlich fährt dieses Skript (noch) keine Rechner herunter, aber es zeigt schon mal, wie Sie Ihren Skriptanwendern mit *MsgBox* Fragen stellen können und die Antwort – nämlich die angeklickte Auswahl – anschließend gemeldet bekommen.

Das soll für den Moment genügen. Sie werden etwas später, nämlich bei den Vergleichsoperatoren, den zweiten Teil dieses Skripts kennen lernen und dann erfahren, wie Sie auf unterschiedliche Reaktionen Ihrer Skriptanwender passend im Skript reagieren.

Sie haben wieder einige wesentliche Erkenntnisse gewonnen:

- **Befehle:** Befehle fordern Ihr Skript auf, bestimmte Dinge zu tun. Dabei hängt es vom Befehl ab, ob der Befehl allein für sich funktioniert oder ob er zusätzliche Informationen braucht.
- **Argumente:** Falls Ihr Befehl Zusatzinformationen braucht, um zu funktionieren, dann werden diese Zusatzinformationen »Argumente« genannt und folgen nach dem Befehl. Die einzelnen Argumente werden dabei mit Kommas voneinander getrennt, falls der Befehl mehr als ein Argument verarbeitet. Manchmal sind Argumente Pflicht. Dann funktioniert der Befehl nur, wenn Sie das Argument mit angeben. Manchmal sind Argumente auch optional. Sie können das Argument angeben, müssen es aber nicht. Geben Sie das Argument nicht an, dann nimmt der Befehl für dieses Argument selbst einen Standardwert an.
- **Funktionen:** Möchten Sie einen Ergebniswert vom Befehl lesen, dann weisen Sie das Ergebnis links vom Befehl einer Variablen zu. Wenn Sie das machen, wird Ihr Befehl »Funktion« genannt. Immer wenn Sie einen Befehl als Funktion aufrufen, gehören die Argumente rechts vom Befehl in Klammern.

# Variablen genauer verstehen

Immer, wenn sich Ihr Skript etwas merken soll – ein Zwischenergebnis zum Beispiel –, dann verwenden Sie Variablen. Variablen speichern Dinge wie Rechenergebnisse, Texte, Zahlen oder Daten unter einem bestimmten Namen, und Sie können den Inhalt der Variablen später wieder abrufen, wenn Sie ihn brauchen.

Dieses »Gedächtnis« liegt im unsichtbaren Speicher Ihres Skripts. Weil dieser Speicher automatisch gelöscht und wieder freigegeben wird, sobald Ihr Skript endet, vergisst es automatisch alle bis dahin gemerkten Variableninhalte. Variablen gelten also nur innerhalb Ihres Skripts, und auch nur so lange, wie das Skript läuft.

Was Sie in Variablen speichern, ist ganz allein Ihre Sache. Schauen Sie mal, was Sie alles in Variablen zwischenspeichern können. Dabei brauchen Sie nur daran zu denken, die Zeilen von rechts nach links zu lesen: Rechts steht, was Sie einer Variablen zuweisen möchten. Links steht der Name der Variablen, und dazwischen gehört ein Gleichheitszeichen.

### Strings zuweisen

Jeder String, den Sie wortwörtlich meinen, gehört in Anführungszeichen und wird zum Literal, weil er sich während der Skriptausführung niemals ändern wird:

```
strText = "Hallo"
```

Sie könnten den String, der nun in *strText* gespeichert ist, anschließend an einen Befehl wie *MsgBox* weiter verfüttern und so wieder sichtbar machen:

```
MsgBox strText
```

So gesehen wäre das Skript natürlich reichlich sinnlos, denn warum sollten Sie einen String zuerst in einer Variablen speichern und danach gleich wieder ausgeben? Das Skript hier ist natürlich stark verkürzt. Normalerweise würden zwischen der Speicherung in der Variablen und der späteren Ausgabe weitere Schritte im Skript passieren. Hier ein sehr einfaches Beispiel, das den *InputBox*-Befehl nutzt, um vom Anwender einen beliebigen String eingeben zu lassen:

```
name = InputBox("Hey, wie heißen Sie?")
MsgBox "Guten Tag, " & name
```

### Variablen mit dem Ergebnis von Funktionen füllen

Die Sache funktioniert verblüffend gut. Schauen Sie sich die erste Zeile unbedingt noch mal etwas genauer an:

```
name = InputBox("Hey, wie heißen Sie?")
```

Das Gleichheitszeichen fällt sofort ins Auge: Hier erfolgt eine Zuweisung an die Variable *name*. Auf der rechten Seite des Gleichheitszeichens steht normalerweise das, was der Variablen *name* zugewiesen werden soll. In diesem Fall steht dort aber nicht etwa ein fester String oder eine Zahl, sondern der *InputBox*-Befehl.

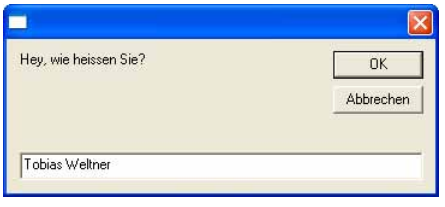


***Abbildung 2.18:*** *Der* InputBox*-Befehl stellt dem Skriptanwender eine Frage*

Wie Befehle funktionieren, haben Sie schon gelesen. Bisher hatten Sie allerdings nur den *MsgBox*-Befehl verwendet, der Dialogfelder mit Schaltflächen anzeigt. Wenn Sie vom Skriptanwender dagegen freie Eingaben erlauben wollen, verwenden Sie anstelle von *MsgBox* eben den *InputBox*-Befehl. Alles andere bleibt gleich.

Was hier genau passiert, kennen Sie schon. Erinnern Sie sich? Ihr Stoppuhr-Skript hat diese Zeile verwendet:

```
start = Timer
```

Hier wurde der Variablen *start* das zugewiesen, was der VBScript-Befehl *Timer* liefert, wenn man ihn aufruft, also die aktuelle Zeit. *Timer* war ein Beispiel für einen Befehl, der keine weiteren Argumente brauchte. Wofür auch? *Timer* soll einfach nur die Systemzeit melden und braucht dazu keine Zusatzangaben von Ihnen. Deshalb stehen hinter *Timer* auch keine weiteren Angaben.

Die neue Zeile sieht fast genauso aus:

```
name = InputBox
```

Hier wird also der Variablen *name* das zugewiesen, was der *InputBox*-Befehl zurückliefert. *InputBox* will allerdings anders als *Timer* sehr wohl mindestens ein Argument geliefert bekommen, nämlich die Frage, die es dem Anwender stellen soll, damit der weiß, was er eintippen soll:

```
name = InputBox("Hey, wie heißen Sie?")
```

**Rätsel 2:** Schlagen Sie die *InputBox*-Funktion in der Referenz nach, und versuchen Sie herauszufinden, wie man sie aufrufen muss, damit das Fragefenster nicht mit einem leeren Textfeld beginnt, sondern bereits eine Eingabe vorschlägt!

Was würde wohl passieren, wenn Sie den String, den Sie einer Variablen zuweisen wollen, nicht in Anführungszeichen setzen? Probieren Sie es aus:

```
strText = Hallo
MsgBox strText
```

Das *MsgBox*-Dialogfeld würde jetzt gar keinen Text anzeigen. Und warum? Weil Sie diesmal der Variablen *strText* nicht etwa den String »Hallo« zugewiesen haben, sondern den Inhalt der Variablen *hallo*. Und weil es diese Variable gar nicht gibt, legt VBScript notgedrungen eine neue leere Variable namens *Hallo* an und weist ihren leeren Inhalt der Variablen *strText* zu. Hier der Beweis:

```
Hallo = "Hier steht der Text!"
strText = Hallo
MsgBox strText
```

Weil Sie nun in der Variablen *hallo* einen String hinterlegt haben, wird der in der zweiten Zeile an die Variable *strText* übergeben und in der dritten Zeile in einem Dialogfeld angezeigt. Sinnvoll ist das natürlich nicht, sondern soll nur den verschlungenen Weg der Variablenzuweisungen deutlicher machen. Für Sie bedeutet das: Text, den Sie wörtlich meinen, gehört in Anführungszeichen, denn andernfalls versteht VBScript Ihren Text als Variablennamen.

Eigentlich einleuchtend. Bleibt nur noch ein Problem: Wenn Anführungszeichen einen String begrenzen, wie kann man dann Anführungszeichen innerhalb eines Strings sichtbar machen? Ganz einfach, wenn man den Trick kennt: Setzen Sie innerhalb der Anführungszeichen, die den String begrenzen, für jedes sichtbare Anführungszeichen genau zwei Anführungszeichen.

```
strText = "Hallo, hier stehen ""Anführungszeichen""!"
MsgBox strText
```


***Abbildung 2.19:*** *Anführungszeichen begrenzen nicht nur Strings, sie dürfen auch Teil des Strings sein*

### Zahlen zuweisen

Zahlen lassen sich genauso einfach in Variablen speichern wie Strings. Bei Zahlen mit Nachkommastellen müssen Sie nur darauf achten, als Dezimaltrennzeichen den Punkt anzugeben:

```
dblFaktor = 4.78
```

VBScript kann sogar hexadezimale und oktale Zahlen verstehen, wenn Sie das entsprechende Kennzeichen davor schreiben:

```
lngHex = &hFFF6253AC
lngOct = &o633775103
```

**Rätsel 3:** Eben haben Sie schon den *InputBox*-Befehl kennen gelernt. Können Sie ein Skript schreiben, das mit *InputBox* einen Zahlenwert erfragt und dann den eingegebenen Betrag in Dollar umrechnet?

### Datumsangaben zuweisen

Selbst Datumsangaben können Sie Variablen zuweisen. Im einfachsten Fall speichern Sie das Datum einfach als normalen Text. Das könnte zum Beispiel so aussehen:

```
datum = "18.3.68"
```

Oder Sie schreiben:

```
datum = "18. März 1968"
```

Obwohl das Datum als String gespeichert wurde und vielleicht sogar ausgeschriebene Monatsnamen enthält, können Sie damit Berechnungen durchführen, denn VBScript ist schlau genug, den String automatisch in ein Datum umzuwandeln. Möchten Sie zum Beispiel wissen, auf welchen Wochentag das Datum fällt, dann könnten Sie den VBScript-Befehl *FormatDateTime* einsetzen. Der errechnet zum Datum den Wochentag und zeigt den Monat als Klartext an:

```
datum = "18.3.1968"
MsgBox datum
MsgBox FormatDateTime(datum, vbLongDate)
```


***Abbildung 2.20:*** *Finden Sie heraus, auf welchen Wochentag ein Datum fällt – zum Beispiel Ihr Geburtstag*

**Rätsel 4:** Schlagen Sie doch mal den Befehl *FormatDateTime* in der Referenz nach. Welche Aufgabe hat *vbLongDate*? Was passiert, wenn Sie stattdessen *vbShortDate* verwenden? Welche *Format…*-Befehle kennt VBScript sonst noch?

Was Sie sonst noch mit Datumsangaben machen können und wie Sie zum Beispiel Zeitdifferenzen ausrechnen, erfahren Sie in einem Moment, wenn Sie die Datums-Rechenfunktionen von VBScript kennen lernen.

### Ja/Nein-Werte zuweisen (Boolesche Werte)

Manchmal sollen Variablen nur Auskunft darüber geben, ob etwas stimmt oder nicht. Es genügt also, wenn die Variable nur zwei Zustände speichert, die »Ja« oder »Nein« beziehungsweise »Wahr« oder »Falsch« entsprechen. Solche Variablentypen werden »Boolean« genannt.

Anstelle der deutschen Begriffe »wahr« und »falsch« sollten Sie immer mit den englischen vordefinierten Begriffen *True* und *False* arbeiten. Dahinter stecken die beiden Zahlenwerte -1 (für *True*) und 0 (für *False*). Technisch gesehen sind boolesche Werte die kleinstmöglichen Variablen, nämlich Bits.

```
boolFirstRun = True
```

Geben Sie den Inhalt einer booleschen Variablen aus, dann kann das Ergebnis wahlweise eine Zahl (-1 oder 0) oder ein Wort (*True* oder *False*, *Wahr* oder *Falsch*) sein. Das Wort erscheint, wenn die boolesche Variable zuvor in einen String konvertiert wurde, ist also die Textrepräsentation eines booleschen Wertes. Das passiert automatisch, wenn Sie eine boolesche Variable mit *MsgBox* ausgeben.

Ja/Nein-Werte sind im Moment noch nicht besonders interessant. Später, wenn Sie die Vergleichsoperatoren und die Bedingungen kennen lernen, begegnen Ihnen boolesche Werte allerdings wieder – und dann sind sie auch spannend. Zumindest so spannend, wie boolesche Werte eben sein können.

# Operatoren und Rechenfunktionen

In Agententhrillern vermischen sich harmlose rote und gelbe Flüssigkeiten meist zu fulminanten Supersprengstoffen, und Ähnliches passiert auch mit Ihren harmlosen Variablen, wenn Sie sie in Kontakt mit Operatoren bringen. Jeder für sich ist machtlos, aber zusammen lösen beide erstaunliche Probleme.

Operatoren verknüpfen jeweils die Werte, die rechts und links von ihnen stehen, und das einfachste Beispiel für Operatoren sind die Rechenoperatoren für die Grundrechenarten: +, -, * und /. Daneben gibt es außerdem noch Vergleichsoperatoren und logische Operatoren, die auf Bitebene arbeiten.

Zusätzlich liefert VBScript zahlreiche Funktionen aus dem Bereich der Mathematik, Daten und Zeiten sowie für die Stringmanipulation mit, die Sie im nächsten Abschnitt kennen lernen. Aber der Reihe nach:

## Rechenoperatoren

Tabelle 2.1 listet alle Rechenoperatoren auf, die Ihr Skript nutzen kann. Damit kann Ihr Skript schon einmal all die Berechnungen durchführen, für die man normalerweise einen einfachen Taschenrechner zückt.

**Rätsel 5:** Haben Sie eine Idee, warum VBScript für die Division den Operator »/« verwendet und nicht etwa den Doppelpunkt »:«?

Ein Skript könnte zum Beispiel auf folgende Weise rechnen und das Ergebnis anzeigen:

```
ergebnis = (12 + 4) * 6 / 12
MsgBox ergebnis
```

***Listing 2.3:*** *Eine einfache Berechnung durchführen*

Dabei gilt wie üblich Punkt- vor Strichrechnung, aber Sie dürfen runde Klammern einsetzen, um diese Berechungsreihenfolge zu modifizieren.

| Operator | Beschreibung |
|---|---|
| + | Addiert Zahlen, fügt Strings aneinander |
| - | Subtrahiert Zahlen |
| * | Multipliziert Zahlen |
| / | Dividiert Zahlen |
| ^ | Potenziert Zahlen |
| & | Verknüpft Elemente zu einem String |
| \ | Dividiert Zahlen und gibt nur den ganzzahligen Wert zurück |
| *Mod* | Dividiert Zahlen und liefert den Rest zurück (nicht die Nachkommastellen) |

***Tabelle 2.1:*** *Rechenoperatoren*


***Abbildung 2.21:*** *Skripts können rechnen und das Ergebnis mit dem* MsgBox-*Befehl anzeigen*

Allerdings sind Skripts nicht dazu da, um Taschenrechner wegzurationalisieren, und Allerweltsberechnungen wie Listing 2.3 erledigen Sie besser mit dem in Windows eingebauten Taschenrechner *CALC.EXE*. Skripts sind eher Spezialisten, wenn es darum geht, immer dieselben Handgriffe zu automatisieren.

Wenn Sie also immer wieder zum Taschenrechner greifen, um dieselben Berechnungen durchzuführen, dann ist das ein Fall für Skripts.

Das folgende Skript berechnet zum Beispiel den Mehrwertsteueranteil und den Nettopreis, wenn Sie einen Endbetrag eingeben – wesentlich bequemer als mit jedem Taschenrechner, vor allem wenn Sie das häufiger müssen:

```
betrag = InputBox("Geben Sie den Betrag ein!")   ' Betrag erfragen

' Formel: netto * mwst = brutto; netto = brutto / mwst

mwst = 1.16      ' 16% MwSt.
netto = betrag / mwst

' Ergebnis als String zusammenstellen:
ergebnis = "netto:" & vbTab & vbTab & FormatCurrency(netto) & vbNewLine
ergebnis = ergebnis & "+16% MwSt:" & vbTab & FormatCurrency(betrag - netto) & vbNewLine
ergebnis = ergebnis & "Gesamt:" & vbTab & vbTab & FormatCurrency(betrag)

' Ergebnis ausgeben
MsgBox ergebnis
```

***Listing 2.4:*** *Brutto- und Nettopreise ausrechnen*

**Das Skript Schritt für Schritt analysieren**

Schauen Sie sich auch dieses Skript Zeile für Zeile an, denn es gibt viel Neues zu entdecken:

```
' Betrag erfragen
```

Diese Zeile ist eine Kommentarzeile. VBScript ignoriert alles, was hinter dem einfachen Anführungszeichen steht. So können Sie längere Skripts mit eigenen Anmerkungen kommentieren, und wenn Sie das Skript ein paar Tage später noch einmal hervorkramen, helfen Ihnen diese Hinweise, das Skript viel schneller zu verstehen. Sie erreichen das einfache Anführungszeichen auf deutschen Tastaturen über UMSCHALT+#.

Erst in der zweiten Zeile wird VBScript aktiv. Diese Zeile kennen Sie schon:

```
betrag = InputBox("Geben Sie den Betrag ein!")
```

Das Gleichheitszeichen zeigt an, dass hier eine Zuweisung an die Variable *betrag* erfolgt. Der Variablen wird aber kein fester Zahlenwert zugewiesen, sondern das, was die *InputBox*-Funktion vom Skriptanwender erfragt. Anschließend wird in *betrag* das gespeichert, was der Anwender eingegeben hat.


***Abbildung 2.22:*** *Dieses Skript erfragt einen beliebigen Preis …*

Aus dieser Eingabe versucht das Skript nun, den Nettobetrag und die enthaltene Mehrwertsteuer auszurechnen. Wie das passiert, ist weniger ein Skript- als eher ein mathematisches Problem. Ausgehend von der Formel *brutto = netto * mwst* wird die Formel so umgeformt, dass *netto* zum Ergebnis wird: *netto = brutto / mwst*. Als Mehrwertsteuer wird in der Variablen *mwst* der Faktor *1.16* angegeben, denn mit diesem Faktor errechnet sich aus dem Nettowert der Bruttowert.

```
mwst = 1.16      ' 16% MwSt.
netto = betrag / mwst
```

Sie sehen gleich eine ganze Reihe von Details: Es kann manchmal ein wenig Gehirnschmalz erfordern, um die richtige Formel zu basteln, die das ausrechnet, was Sie tatsächlich errechnen wollen. Tut Ihr Skript aber erst einmal das, was Sie wollen, dann brauchen Sie anders als beim Taschenrechner künftig keinen Gedanken mehr daran zu verschwenden.

In der Variablen *netto* speichert das Skript das Rechenergebnis, nämlich den Betrag ohne Mehrwertsteueranteil. Der wird zum Schluss noch in einen Ergebnistext umgewandelt. Auch hier lohnt sich ein genauer Blick:

```
ergebnis = "netto:" & vbTab & vbTab & FormatCurrency(netto) & vbNewLine
ergebnis = ergebnis & "+16% MwSt:" & vbTab & FormatCurrency(betrag - netto) & vbNewLine
ergebnis = ergebnis & "Gesamt:" & vbTab & vbTab & FormatCurrency(betrag)
```

Die erste Zeile sieht noch ganz ähnlich aus wie alle übrigen Variablenzuweisungen, die Sie inzwischen gesehen haben. Der Variablen *ergebnis* wird ein aus mehreren Teilen zusammengesetzter String zugewiesen. Erinnern Sie sich? Der &-Operator kittet die einzelnen Teile zusammen.

Einige dieser Teile sind allerdings neu. Was bedeutet *vbTab*? Ganz so neu wie *vbTab* auf den ersten Blick erscheint, ist es gar nicht. Sie haben oben bereits die eingebauten VBScript-Konstanten kennen gelernt. Der *MsgBox*-Befehl konnte zum Beispiel mit Hilfe der Konstante *vbOKCan-*

*cel* eine *OK*- und eine *Abbrechen*-Schaltfläche sichtbar machen, und in Wirklichkeit steckte hinter der Konstanten *vbOkCancel* eine Konstante mit dem Wert *4*. *vbTab* ist auch eine vordefinierte Konstante und beginnt wie alle vordefinierten Konstanten ebenfalls mit »vb«. In *vbTab* ist ein Tabulatorzeichen vorgespeichert. Sie können damit also Tabulatoren in Strings einbauen, und genau das passiert hier auch. Tabulatoren helfen dabei, Strings ordentlich in Spalten auszurichten, damit alles schön bündig ist.

Jetzt ahnen Sie bereits, was *vbNewLine* ist (und tut). Richtig: *vbNewLine* ist ebenfalls eine vordefinierte Konstante und enthält einen Zeilenumbruch. Nach *vbNewLine* beginnt also immer eine neue Textzeile. Alternativ können Sie dafür auch die Konstante *vbCrLf* (für *Carriage Return* und *Line Feed*) verwenden, die denselben Effekt hat.

Und was macht *FormatCurrency*? Auch dieser Befehl ist nicht ganz so neu, wie er erscheint. Sie haben bereits den Befehl *FormatDateTime* kennen gelernt, der eine Datumsangabe formatiert. VBScript enthält eine ganze Reihe solcher *Format…*-Befehle, und *FormatCurrency* ist dafür zuständig, einen Zahlenwert als Geldbetrag zu formatieren.


***Abbildung 2.23:*** *… und berechnet dann Nettopreis und Mehrwertsteueranteil*

Dazu füttert man *FormatCurrency* rechts als Argument den Betrag ein und erhält links das formatierte Ergebnis, einschließlich eines schicken Euro-Symbols. Tatsächlich ist *FormatCurrency* also genau wie *InputBox* eine Funktion, denn *FormatCurrency* liefert wie *InputBox* ein Ergebnis zurück. Das ist auch der Grund, warum das Argument – der Betrag, den *FormatCurrency* umwandeln soll – in runde Klammern gestellt ist. In diesem Fall wird das Ergebnis, das *FormatCurrency* zurückgibt, lediglich nicht sofort in einer Variablen gespeichert, sondern zuerst vom &-Operator mit den übrigen Teilen zu einem String zusammengefasst, der dann in *ergebnis* gespeichert wird.

Bleibt noch eine Frage: Warum beginnen die übrigen beiden Zeilen auf diese Weise:

```
ergebnis = ergebnis & ...
```

Die Frage können Sie sich ganz schnell selbst beantworten, wenn Sie die Zeile wieder formal analysieren. Auch hier findet eine Variablenzuweisung statt, und zwar links an die Variable *ergebnis*. Die Variable *ergebnis* speichert das, was rechts vom Gleichheitszeichen steht. Dort allerdings steht unter anderem ebenfalls *ergebnis*. Jedem Mathematiker würden hier die Haare zu Berge stehen, denn streng mathematisch sind die Werte rechts und links vom Gleichheitszeichen natürlich nicht gleich.

Tatsächlich ist das Gleichheitszeichen aber ein Zuweisungszeichen, und man muss den Vorgang im zeitlichen Verlauf sehen: Das, was rechts vom Gleichheitszeichen steht, wird vom Gleichheitszeichen dem zugewiesen, was links vom Gleichheitszeichen steht. Die neue Variable *ergebnis* soll also den Inhalt der alten Variablen *ergebnis* plus zusätzliche Informationen enthalten. Oder anders gesagt: Es werden weitere Informationen an die Variable *ergebnis* angehängt.

Das Ergebnis sehen Sie, wenn Sie das Skript aufrufen und einen Betrag eingeben. Als Ergebnis werden drei Textzeilen ausgegeben, nämlich der Nettobetrag, die Mehrwertsteuer und der Gesamtbetrag.

**Rätsel 6:** Was passiert, wenn Sie in Ihrem Skript keinen Betrag eingeben, sondern Text – oder gar nichts? Können Sie sich das Verhalten erklären?

### Exotische Rechenoperatoren und mathematische Funktionen

Die meisten Menschen kommen mit den vier Grundrechenarten prima zurecht, und das gilt auch für Leute, die Skripts erstellen. Wenn Sie allerdings kniffligere mathematische Probleme lösen müssen, dann gibt es natürlich auch dafür die passenden Hilfsmittel. Dazu zählen einerseits ein paar exotischere Operatoren und andererseits eine ganze Armada an Rechenfunktionen. Schauen Sie sich beide doch mal näher an:

Die Operatoren aus Tabelle 2.1 umfassen nicht nur die Grundrechenarten. Möchten Sie Potenzen ausrechnen, dann verwenden Sie das »Dach« (Zirkumflex). Sie finden dieses Zeichen über der Tabulatortaste, und wenn Sie diese Taste drücken, erscheint zunächst gar kein Zeichen. Sie müssen erst ein weiteres Zeichen eingeben, damit das Sonderzeichen sichtbar wird.

In der EDV sind vor allen Dingen die Potenzen zur Basis 2 wichtig. Möchten Sie zum Beispiel ausrechnen, welche Zahl den gesetzten Bits 0, 4 und 5 entspricht, dann geht das so:

```
MsgBox 2^0 + 2^4 + 2^5
```

Genauso einfach können Sie sich ausrechnen lassen, wie viel Bytes in einem Kilobyte oder in einem Megabyte enthalten sind. Dabei räumen Sie gleich mit dem Missverständnis auf, ein Megabyte entspräche 1.000.000 Byte – eine G. Jauch'sche Fangfrage weniger.

```
MsgBox "Kilobyte: " & 2^10

MsgBox "Megabyte: " & 2^10^2
MsgBox "Megabyte: " & 2^20

MsgBox "Gigabyte: " & 2^10^2^2
MsgBox "Gigabyte: " & 2^10^4
MsgBox "Gigabyte: " & 2^40

MsgBox "Terabyte: " & 2^10^2^2^2
MsgBox "Terabyte: " & 2^10^8
MsgBox "Terabyte: " & 2^80
```

***Listing 2.5:*** *Potenzen bilden*

Sonderlinge sind auch die Operatoren \ und *Mod*. Auf den ersten Blick scheinen sie nicht besonders nützlich zu sein:

```
MsgBox 45 / 7
MsgBox 45 \ 7
MsgBox 45 MOD 7
MsgBox 45 - (45 MOD 7)
```

***Listing 2.6:*** *Divisionsoperatoren im Einsatz*

Während also / zwei Zahlen teilt, liefert \ nur den Teil des Ergebnisses zurück, der vor dem Komma steht. *Mod* liefert den Rest der Division zurück. Die letzte Zeile des Skripts errechnet also den nächstniedrigeren Wert, der ganzzahlig durch 7 teilbar ist.


***Abbildung 2.24:*** *Aus den Tagen einer Dienstreise berechnet dieses Skript …*

Wofür könnte man so etwas gebrauchen? Stellen Sie sich vor, Sie wissen nur, wie viel Tage eine Dienstreise gedauert hat, und möchten nun ausrechnen, wie viel Wochen und wie viel Tage die Reise genau gedauert hat, vielleicht weil Sie das Glück haben, für jede geleistete Woche in Timbuktu Süd eine satte Prämie zu kassieren. Das folgende Skript erledigt diese Aufgabe und berechnet aus einer Anzahl von Tagen die vollständigen Wochen, die diese Tage ausmachen:

```
intTageGesamt = InputBox("Geben Sie die Anzahl Tage ein!")
intWochen = intTageGesamt \ 7
intTage = intTageGesamt Mod 7
ergebnis = intTageGesamt & " Tage entsprechen" & vbNewLine
ergebnis = ergebnis & intWochen & " Wochen und " & intTage & " Tagen"
MsgBox ergebnis
```

***Listing 2.7:*** *Tage und Wochen aus einer Zeitangabe ermitteln*


***Abbildung 2.25:*** *… die Anzahl der Wochen und die restlichen Tage*

**Rätsel 7:** Trauen Sie sich zu, Listing 2.7 so zu erweitern, dass der Skriptanwender ein Datum eingeben kann und das Skript dann mit *DateDiff* die Anzahl Tage zurückliefert, die seither vergangen sind? Wenn Sie sich unsicher sind, schauen Sie sich noch mal Listing 2.15 an. Mit den Operatoren \ und *Mod* können Sie dann angeben, wie viel Wochen und wie viel Tagen dies entspricht.

Reichen Ihnen diese Operatoren nicht, dann schauen Sie sich die mathematischen Sonderfunktionen in Tabelle 2.2 an, die sich besonders gut für die Automation von Mathematikhausaufgaben eignen.

**TIPP** Für Schulmathematiker ein kleiner Hinweis: Zur Umrechnung von Grad ins Bogenmaß muss die Gradangabe mit *pi/180* multipliziert werden. Zur Umrechnung von Bogenmaß in Grad muss das Bogenmaß mit *180/pi* multipliziert werden.

### Pi ausrechnen

Vielleicht brauchen Sie in Ihren Berechnungen den Wert *pi*. Der ist zwar nirgends vordefiniert, kann aber leicht ausgerechnet werden:

```
pi = 4 * Atn(1)
MsgBox pi
```

***Listing 2.8:*** *Pi ausrechnen*

| Funktion | Beschreibung |
|---|---|
| *Abs* | Der Absolutwert einer Zahl ist der positive Betrag der Zahl. Sowohl *Abs*(-1) als auch *Abs*(1) geben z.B. den Wert 1 zurück. |
| *Atn* | Die *Atn*-Funktion berechnet aus dem Verhältnis von zwei Seiten eines rechtwinkligen Dreiecks (Zahl) den zugehörigen Winkel im Bogenmaß. Der Quotient ist das Längenverhältnis von Gegenkathete zu Ankathete. Der Wertebereich des Ergebnisses reicht von -pi/2 bis pi/2 (Bogenmaß). |
| *Cos* | Die *Cos*-Funktion berechnet das Verhältnis von zwei Seiten eines rechtwinkligen Dreiecks aus einem Winkel. Das Verhältnis ergibt sich aus der Länge der Ankathete dividiert durch die Länge der Hypotenuse. Das Ergebnis liegt im Bereich von -1 bis 1. |
| *Exp* | Gibt *e* (die Basis des natürlichen Logarithmus) potenziert mit einer Zahl zurück. |
| *Hex* | Gibt einen String mit der Hexadezimaldarstellung einer Zahl zurück. |
| *Log* | Der natürliche Logarithmus ist der Logarithmus zur Basis *e*. Die Konstante *e* entspricht ungefähr dem Wert 2,718282. Sie können den Logarithmus zur Basis *n* für jede Zahl *x* berechnen, indem Sie den natürlichen Logarithmus von *x* wie folgt durch den natürlichen Logarithmus von *n* dividieren: Logn(x) = *Log*(x) / *Log*(n) |
| *Oct* | Gibt einen String mit der Oktaldarstellung einer Zahl zurück. |
| *Round* | Gibt eine auf die angegebene Anzahl Dezimalstellen gerundete Zahl zurück. |
| *Sgn* | Gibt einen ganzzahligen Wert zurück, der das Vorzeichen einer Zahl darstellt (1=größer als null, 0=null, -1=kleiner als null). |
| *Sin* | Die Funktion *Sin* berechnet aus einem Winkel den Quotienten von zwei Seiten eines rechtwinkligen Dreiecks. Der Quotient ergibt sich aus dem Längenverhältnis von Gegenkathete zu Hypotenuse. Das Ergebnis liegt im Bereich von -1 bis 1. |
| *Sqr* | Gibt die Quadratwurzel einer Zahl zurück. |
| *Tan* | *Tan* berechnet aus einem Winkel den Quotienten von zwei Seiten eines rechtwinkligen Dreiecks. Der Quotient ist das Längenverhältnis von Gegenkathete zu Ankathete. |

***Tabelle 2.2:*** *Mathematische Funktionen*

### Potenz zur Basis 2 ermitteln

Eben haben Sie gesehen, dass *2^10* den Wert *1024* ergibt, also ein Kilobyte. Auch der umgekehrte Weg ist möglich, wenn Sie *Log* verwenden:

```
MsgBox Log(1024) / Log(2)
```

### Hexadezimale und oktale Zahlen

Hexadezimale Zahlen sind Zahlen zur Basis 16 und spielen im Bereich der Computer und des Internets eine große Rolle. Oktale Zahlen sind Zahlen zur Basis 8 und eher ungebräuchlich.

Möchten Sie aus einer »normalen« dezimalen Zahl eine hexadezimale Zahl machen, dann hilft Ihnen die *Hex*-Funktion:

```
dezimal = InputBox("Geben Sie eine dezimale Zahl ein!")
MsgBox "Hexadezimale Form: " & Hex(dezimal)
```

***Listing 2.9:*** *Dezimale Zahl in hexadezimale Stringdarstellung umwandeln*

Umgekehrt ist es leider etwas schwieriger. Zwar kann VBScript auch hexadezimale Zahlen in dezimale Zahlen umwandeln, aber nur wenn Sie die hexadezimale Zahl mit dem Präfix &h direkt in Ihr Skript schreiben:

```
zahl = &hFF
MsgBox zahl
```

Möchten Sie dagegen, dass Ihr Skript beliebige hexadezimale Zahlen in dezimale Zahlen umwandelt, zum Beispiel die Eingaben eines Skriptanwenders, dann hilft nur ein Trick:

```
zahl = InputBox("Geben Sie eine hexadezimale Zahl ein!",,"FF")
dezimal = Eval("&h" & zahl)
MsgBox dezimal
```

***Listing 2.10:*** *Hexadezimale Stringdarstellung in dezimalen Wert umwandeln*

Das Skript nutzt die Funktion *Eval*. *Eval* kann man einen mathematischen Ausdruck als Argument übergeben, der dann von *Eval* so ausgewertet wird, als stünde er so in Ihrem Skript. Das Skript nimmt also die hexadezimale Zahl entgegen, die der Skriptanwender eingibt, und fügt davor ein »&h« an. Danach übergibt es diese Eingabe an *Eval* und erhält die dezimale Zahl zurück.

Allerdings funktioniert die Sache nur so lange, wie der Anwender auch tatsächlich eine gültige hexadezimale Zahl eingibt und wie diese hexadezimale Zahl in einem gültigen Zahlenbereich bleibt. Zu große hexadezimale Zahlen führen zu einem Fehler. Erlaubt sind bis zu acht hexadezimale Stellen.

**Mondphase berechnen**

Mit den trigonometrischen Funktionen kann man nicht nur langweilige Sinuskurven ausrechnen. Auch die Astronomie und Sternenbahnen gehorchen diesen mathematischen Prinzipien, und deshalb genügt ein Rückgriff auf das magische Wissen von Kopernikus und seiner Kollegen, um per Skript zum Beispiel die aktuelle Mondphase auszurechnen.

So könnten Sie auch herausfinden, ob bei der Mitternachtsparty an Ihrem nächsten Geburtstag auch der Mond vorbeischaut oder nicht.

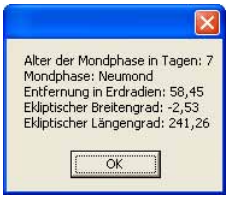


***Abbildung 2.26:*** *Ob Vollmond oder nicht ermitteln Sie für beliebige Tage künftig selbst*

Die Aufgabe erledigt das nächste Skript und zeigt, dass auch anspruchsvollere Mathematik per Skript möglich ist, selbst wenn die dabei verwendeten Formeln für uns Normalsterbliche wohl immer ein Buch mit sieben Siegeln bleiben werden.

**Rätsel 8:** Zwar müssen Sie dieses Skript noch nicht verstehen, aber die formalen Elemente darin sollten Sie bereits erkennen können, auch wenn noch nicht alle Elemente besprochen sind. Was sind *Int*, *intJahr* und *intJahrNeu* (rein formal)? Können Sie sich vorstellen, wofür der Befehl *Array* da ist? Sie dürfen in der WSH-Referenz nachschlagen und sollten das auch gleich mal tun! Haben Sie eine Idee, wofür die *If*-Anweisung da ist, die gleich mehrmals im Skript vorkommt?

```
' Die Mondphasenberechnung beruht auf Formeln von Bradley E.
' Schaefer und einem Skript von Paul R. Sadowski, MVP MSN Client.

' Zur Mondphasenberechnung ist es nötig, das Verhältnis zwischen
' Erde und Mond unabhängig von jahreszeitlichen Schwankungen zu
' betrachten. Das Skript kann deshalb nicht die für uns normalerweise
' üblichen Sonnen- und Mondtage verwenden, die durch die Drehung
' der Himmelskörper umeinander definiert sind und sich von Tag zu
' Tag leicht unterscheiden.
```

```
' Stattdessen werden die Himmelskörper aus Sicht der Fixsterne
' betrachtet. Solche Berechnungen nennt man im englischen Sprachgebrauch
' "sidereal" (siderisch), und eine siderische Erddrehung dauert unabhängig
' von Jahreszeit und Stellung der Himmelskörper zueinander immer
' 23 Stunden, 56 Minuten und 4,1 Sekunden.

' Auch die Drehung des Mondes kann "astronomisch" berechnet werden,
' indem man den Schnittpunkt des Mondorbits mit dem Erdorbit betrachtet
' und ausrechnet, wie lange es für den Mond dauert, um ein und denselben
' Schnittpunkt wieder zu erreichen. Dabei handelt es sich um einen
' Zeitraum von 27,212220 Tagen, wiederum also einen konstanten Wert,
' der engl. "draconic phase" genannt wird.

phase = Array ( "Neumond", "zunehmend, gebogen", "erstes Viertel", _
   "zunehmend, gewölbt", "Vollmond", "abnehmend gewölbt", _
   "letztes Viertel", "abnehmend gebogen" )

datum = InputBox("Geben Sie das Datum an!",,Date)

' Datum zerlegen
intTag = Day(datum)
intMonat = Month(datum)
intJahr = Year(datum)

' Wichtige vordefinierte Werte berechnen
P2= 8 * Atn(1)
intJahrNeu = intJahr - Int((12 - intMonat) / 10)
intMonatNeu = intMonat + 9

If intMonatNeu >= 12 Then
   intMonatNeu = intMonatNeu - 12
End If

K1 = Int(365.25 * (intJahrNeu + 4712))
K2 = Int(30.6 * intMonatNeu + .5)
K3 = Int(Int((intJahrNeu / 100) + 49) * .75) - 38

' J = Julianisches Datum um 12h UT am gewünschten Tag
J = K1 + K2 + intTag + 59
If J > 2299160 Then
   ' Anpassung Gregorianischer Kalender
    J = J - K3
End If

' Synodische Phase berechnen
' [illumination (synodic) phase]
V = (J - 2451550.1) / 29.530588853
V = V - Int(V)

If V < 0 Then
    V = V + 1
End If

IP = V

' Mondalter in Tagen
AG = IP * 29.53

' In Bogenmaß konvertieren
IP = IP * P2

' Distanz berechnen
V = (J - 2451562.2) / 27.55454988
V = V - Int(V)
```

```
If V < 0 Then
    V = V + 1
End If
DP=V
DP = DP * P2: ' In Bogenmaß konvertieren
DI = 60.4 - 3.3 * Cos(DP) - .6 * Cos(2 * IP - DP) - .5 * Cos(2 * IP)
' Breitengrad von 'draconic phase'
' [latitude from nodal (draconic) phase]
V = (J - 2451565.2) / 27.212220817
V = V- Int(V)
If V < 0 Then
 V = V + 1
End If
NP = V
' In Bogenmaß konvertieren
NP = NP * P2
LA = 5.1 * Sin(NP)
' Längengrad der siderischen Bewegung berechnen
' [longitude from sidereal motion]
V =(J - 2451555.8) / 27.321582241
V = V - Int(V)
If V < 0 Then
    V = V + 1
End If
RP = V
LO = 360 * RP + 6.3 * Sin(DP) + 1.3 * Sin(2 * IP - DP) + .7 * Sin(2 * IP)
If AG = 0 OR AG = 29 Then
  aktuellePhase = 0
ElseIf AG >= 1 AND AG <= 6 Then
  aktuellePhase = 1
ElseIf AG = 7 Then
  aktuellePhase = 2
ElseIf AG >= 8 AND AG <= 13 Then
  aktuellePhase = 3
ElseIf AG = 14 Then
  aktuellePhase = 4
ElseIf AG >= 15 AND AG <= 21 Then
  aktuellePhase = 5
ElseIf AG = 22 Then
  aktuellePhase = 6
ElseIf AG >= 23 AND AG <= 28 Then
  aktuellePhase = 7
End If
ergebnis = "Alter der Mondphase in Tagen: " & Int(AG) & vbNewLine
ergebnis = ergebnis &  "Mondphase: " & phase(aktuellePhase) & vbNewLine
ergebnis = ergebnis &  "Entfernung in Erdradien: " & FormatNumber(DI,2) & vbNewLine
ergebnis = ergebnis &  "Ekliptischer Breitengrad: " & FormatNumber(LA,2) & vbNewLine
ergebnis = ergebnis &  "Ekliptischer Längengrad: " & FormatNumber(LO,2)
MsgBox ergebnis
```

***Listing 2.11:*** *Mondphase berechnen*

Falls Ihnen die bisherigen Mathematikfunktionen aus der Tabelle 2.2 noch nicht ausreichen, dann werfen Sie einen Blick in Tabelle 2.3! Viele aus der Schulmathematik bekannte Funktionen sind zwar nicht direkt in VBScript enthalten, können aber ganz einfach aus den Grundfunktionen hergeleitet werden:

| Funktion | Äquivalent |
|---|---|
| Sekans | *1 / Cos(X)* |
| Kosekans | *1 / Sin(X)* |
| Kotangens | *1 / Tan(X)* |
| Arkussinus | *Atn(X / Sqr(-X * X + 1))* |
| Arkuskosinus | *Atn(-X / Sqr(-X * X + 1)) + 2 * Atn(1)* |
| Arkussekans | *Atn(X / Sqr(X * X - 1)) +Sgn((X) - 1) *(2 * Atn(1))* |
| Arkuskosekans | *Atn(X / Sqr(X * X - 1)) + (Sgn(X) - 1) * (2 * Atn(1))* |
| Arkuskotangens | *Atn(X) + 2 * Atn(1)* |
| Sinus Hyperbol. | *(Exp(X) - Exp(-X)) / 2* |
| Kosinus Hyperbol. | *(Exp(X) + Exp(-X)) / 2* |
| Tangens Hyperbol. | *(Exp(X) - Exp(-X)) / (Exp(X) +Exp(-X))* |
| Sekans Hyperbol. | *2 / (Exp(X) + Exp(-X))* |
| Kosekans Hyperbol. | *2 /(Exp(X) - Exp(-X))* |
| Kotangens Hyperbol. | *(Exp(X) + Exp(-X)) / (Exp(X) - Exp(-X))* |
| Arkussinus Hyperbol. | *Log(X + Sqr(X * X + 1))* |
| Arkuskosinus Hyperbol. | *Log(X + Sqr(X * X - 1))* |
| Arkustangens Hyperbol. | *Log((1 + X)/ (1 - X)) / 2* |
| Arkussekans Hyperbol. | *Log((Sqr(-X * X + 1) + 1) / X)* |
| Arkuskosekans Hyperbol. | *Log((Sgn(X) * Sqr(X * X + 1) + 1) / X)* |
| Arkuskotangens Hyperbol. | *Log((X + 1) / (X - 1)) / 2* |
| Logarithmus zur Basis N | *Log(X) / Log(N)* |

***Tabelle 2.3:*** *Abgeleitete mathematische Funktionen*

### Mit Strings »rechnen«

Auch mit Strings kann man »rechnen«, und wenn Sie die trigonometrischen Formeln von eben nicht wirklich vom Hocker gerissen haben, weil Sie als Computer-Praktiker eher selten mit Sinuskurven und Mondphasen zu tun haben, dann sind die String-Rechenfunktionen für Sie bestimmt sehr viel wichtiger.

**Rätsel 9:** Was kommt wohl als Ergebnis bei der folgenden Berechnung heraus?

```
text1 = "Hallo"
text2 = "Welt"
ergebnis = text1 + text2
MsgBox ergebnis
```

***Listing 2.12:*** *Textbausteine addieren*

Und was passiert Ihrer Meinung nach, wenn man den »+«-Operator durch ein Minuszeichen ersetzt?

Ganz offenbar kann VBScript Strings mit dem Plus-Operator zusammensetzen. "*Hallo*" + "*Welt*" ergibt den String "*HalloWelt*". Allerdings enden die Pseudo-String-Berechnungen an dieser Stelle

bereits, denn wenn Sie versuchen würden, String und Zahlen zu mischen, gäbe es Ärger: "*Hallo*" + *2* führt zum Fehler »Typen unverträglich«, weil man eben nicht die Zahl *2* zum String "*Hallo*" hinzuzählen kann.

Und auch "*Hallo*" - "*Welt*" ist unsinnig und führt zu keinem Ergebnis. Der Einsatz von Operatoren ist also bei Strings nur ganz selten sinnvoll, und der einzige wirklich nützliche Operator für Strings ist der &-Operator, der die beiden Werte rechts und links davon zuerst in Strings umwandelt und dann zusammenfügt. "*Hallo*" & *2* würde also im Gegensatz zu "*Hallo*" + *2* funktionieren und als Ergebnis "*Hallo2*" liefern.

Trotzdem kann man mit Strings sehr sinnvolle Berechnungen anstellen, braucht dafür aber die speziellen String-Rechenfunktionen aus Tabelle 2.4.

| Funktion | Beschreibung |
|---|---|
| *Instr* | Ermittelt die Position, ab der ein Suchbegriff in einem String vorkommt |
| *InstrRev* | Wie *Instr*, sucht jedoch von hinten |
| *LCase* | Verwandelt den String in Kleinbuchstaben |
| *Left* | Liefert Zeichen vom linken Rand eines Strings |
| *LTrim* | Entfernt alle Leerzeichen am linken Rand des Strings |
| *Mid* | Liefert Zeichen aus der Mitte eines Strings oder von der Mitte an bis zum Ende |
| *Replace* | Ersetzt Textstellen durch Ersatztext |
| *Right* | Liefert Zeichen vom rechten Rand eines Strings |
| *RTrim* | Entfernt alle Leerzeichen am rechten Rand des Strings |
| *Split* | Wandelt einen String in ein Feld um |
| *Trim* | Entfernt alle Leerzeichen an beiden Enden eines Strings |
| *UCase* | Verwandelt den String in Großbuchstaben |

***Tabelle 2.4:*** *Wichtige Stringmanipulationsfunktionen*

### Informationen aus Strings extrahieren

Häufig haben Skripts die Aufgabe, einen bestimmten Teil aus einem String zu extrahieren. Vielleicht enthält Ihr Skript den Pfadnamen einer Datei, und Sie wollen ermitteln, wie die Dateierweiterung dieser Datei heißt.

```
dateiname = "c:\windows\system32\ansi.sys"
erweiterung = Right(dateiname, 3)
MsgBox erweiterung
```

***Listing 2.13:*** *Dateierweiterung ermitteln*

Wie Sie sehen, kann man mit den Funktionen *Right* und *Left* eine beliebige Anzahl von Zeichen jeweils rechts oder links vom String extrahieren.


***Abbildung 2.27:*** *Den Typ einer Datei bestimmen*

**Rätsel 10:** Können Sie in Tabelle 2.4 eine Funktion entdecken, mit der man Text aus der Mitte eines Strings herauslesen kann? Wie muss man vorgehen, wenn man aus dem Text "*Hallo Welt*" alle Zeichen ab Position 6 lesen will?

Allerdings hat dieser Ansatz einen kleinen Haken, denn was geschieht, wenn die Dateierweiterung der Datei ausnahmsweise einmal kürzer oder länger ist als drei Zeichen?

Die nächste Lösung macht es besser, denn sie verlässt sich nicht auf eine feste Länge für die Erweiterung, sondern benutzt ein Erkennungszeichen:

Die Erweiterung einer Datei beginnt immer hinter dem letzten Punkt im Dateinamen. Deshalb bestimmt das nächste Skript zuerst mit *InstrRev* die Position des letzten Punktes im String. Ist das Ergebnis *0,* dann wurde gar kein Punkt gefunden – bei normalen Dateinamen sollte das nie passieren.

Mit *Mid* kann das Skript dann die Erweiterung aus dem String extrahieren. Dazu will *Mid* wissen, ab welcher Position es lesen soll. Das Skript gibt dafür die Position des gefundenen Punkts plus 1 an, denn die Erweiterung beginnt ja ein Zeichen nach dem Punkt.

```
dateiname = "c:\windows\system32\ansi.html"
position = InStrRev(dateiname, ".")
Erweiterung = Mid(dateiname, position + 1)
MsgBox Erweiterung
```

***Listing 2.14:*** *Einen Suchbegriff in einem String suchen und von dort an alle Zeichen ausgeben*

### Mit Datumswerten rechnen

Oben haben Sie bereits gesehen, dass Sie Variablen alle denkbaren Inhalte zuweisen können: Zahlen, Texte, aber auch zum Beispiel Datumswerte. Und Sie haben gerade erlebt, dass VBScript für jeden Datentyp ein eigenes Sortiment an Rechenfunktionen mitbringt. Auch für den Umgang mit Datumswerten steht Ihnen eine kleine Armada von Hilfsfunktionen zur Verfügung, die in Tabelle 2.5 aufgeführt sind.

Mit dem VBScript-Befehl *DateDiff* können Sie zum Beispiel ausrechnen, wie viele Tage zwischen zwei beliebigen Daten liegen. Das aktuelle heutige Datum liefert Ihnen die VBScript-Funktion *Date*.

```
datum = "18.3.1968"
differenz = DateDiff("d", datum, Date)
MsgBox differenz & " Tage liegen zwischen " & datum & " und " & Date
```

***Listing 2.15:*** *Anzahl Tage zwischen zwei Daten berechnen*

Damit die Berechnung einwandfrei funktioniert, wandelt VBScript Ihr Datum, das Sie in der Variablen *datum* als String gespeichert haben, vollautomatisch in einen Datumswert um, denn nur mit echten Datumswerten kann VBScript Zeitdifferenzen und Wochentage berechnen.


***Abbildung 2.28:*** *Rechnen Sie aus, wie viele Tage seit Ihrem Geburtstag vergangen sind*

*DateDiff* eignet sich natürlich nicht nur dazu, um auszurechnen, wie viel Tage Ihr Cousin alt ist. Man kann damit auch ausrechnen, seit wie viel Tagen ein Benutzer sein Kennwort nicht geändert hat, vorausgesetzt, Sie wissen, wann das zum letzten Mal geschah. Die dafür nötigen Befehle lernen Sie in einem späteren Kapitel kennen.

Nehmen Sie die einfachen Beispiele in diesem Kapitel also nicht auf die leichte Schulter, Sie werden viel von dem, was Sie hier lesen und lernen, auch später noch in Ihren Profiskripts einsetzen.

**Rätsel 11:** Schlagen Sie den Befehl *DateDiff* in der Referenz nach. Wie müssen Sie das Skript ändern, damit die Zeitdifferenz nicht in Tagen, sondern in Minuten oder in Monaten angegeben wird? Warum muss das erste Argument von *DateDiff* in Anführungszeichen gesetzt werden? Was passiert, wenn Sie es nicht in Anführungszeichen setzen? Und warum kann es sein, dass *DateDiff* negative Zahlen zurückmeldet?

Wollen Sie das Datum lieber gleich selbst als echtes Datum in einer Variablen speichern, dann können Sie das folgende Format verwenden:

```
datum = #18/3/1968#
differenz = DateDiff("d", datum, Date)
MsgBox differenz & " Tage liegen zwischen " & datum & " und " & Date
```

***Listing 2.16:*** *Datum direkt angeben*

Schauen Sie sich das Ergebnis im Dialogfeld genauer an, dann sehen Sie den Unterschied zwischen beiden Varianten: Im letzten Skript wurde das Datum als echtes Datum gespeichert, und deshalb meldet *MsgBox* den Monat als 03, also zweistellig.


***Abbildung 2.29:*** *Diesmal zeigt das Dialogfeld den Monat mit einer zweistelliger Zahl*

Im vorherigen Skript war das Datum als reiner String gespeichert und erschien deshalb in der *MsgBox* genau so, wie Sie das Datum angegeben hatten.

Ob Sie Datumsangaben nun als String oder mit den #-Zeichen angeben, richtig eindeutig ist Ihre Datumsangabe nicht. Welche der drei Zahlen nämlich den Tag, den Monat und das Jahr angibt, ist von Sprache zu Sprache verschieden, und wenn Ihr Skript auf einem US-System läuft, werden Monat und Tag möglicherweise vertauscht. Noch schlimmer: Die #-Variante möchte eigentlich das Datum als Monat/Tag/Jahr erhalten. Ist die Monatsangabe aber größer als 12 und deshalb ungültig, dann wechselt VBScript die Reihenfolge automatisch in Tag/Monat/Jahr. Diese freundliche Art des Mitdenkens geht an dieser Stelle eindeutig zu weit und stiftet mehr Verwirrung als Nutzen.

Deshalb sollten Sie Datumsangaben besser ganz eindeutig festlegen und dazu den VBScript-Befehl *DateSerial* einsetzen. Der verlangt unabhängig von der Windows-Sprachversion Ihre Angaben immer in der Reihenfolge Jahr, Monat, Tag.

```
datum = DateSerial(1968,3,18)
differenz = DateDiff("d", datum, Date)
MsgBox differenz & " Tage liegen zwischen " & datum & " und " & Date
```

***Listing 2.17:*** *Datum eindeutig und zuverlässig ohne Rücksicht auf nationale Standards festlegen*

| Funktion | Beschreibung |
|---|---|
| *CDate* | Konvertiert einen Wert in ein Datum, sofern möglich |
| *Date* | Liefert das aktuelle Datum |
| *DateAdd* | Addiert zu einem Datum ein beliebiges Zeitintervall hinzu |
| *DateDiff* | Berechnet den Zeitunterschied zwischen zwei Daten |
| *DatePart* | Extrahiert einen bestimmten Teil eines Datums, zum Beispiel das Jahr oder den Monat |
| *DateSerial* | Generiert ein Datum aus Tag, Monat und Jahr |
| *DateValue* | Wandelt einen Wert in ein Datum, sofern möglich |
| *Day* | Liefert den Tag aus einem Datum zurück |
| *FormatDateTime* | Stellt eine Datumsangabe in verschiedenen Formaten dar |
| *Hour* | Liefert die Stunde aus einer Zeitangabe zurück |
| *isDate* | True, wenn der angegebene Wert als Datum interpretierbar ist |
| *Minute* | Liefert die Minute aus einer Zeitangabe zurück |
| *Month* | Liefert den Monat aus einem Datum zurück |
| *MonthName* | Liefert den ausgeschriebenen Monatsnamen zurück |
| *Now* | Liefert das aktuelle Datum und die aktuelle Zeit zurück |
| *Second* | Liefert die Sekunde einer Zeitangabe zurück |
| *Time* | Liefert die aktuelle Zeit |
| *Timer* | Gibt die Sekunden hochauflösend seit Mitternacht zurück |
| *TimeSerial* | Erstellt eine Zeitinformation aus Stunde, Minute und Sekunde |
| *TimeValue* | Wandelt einen Wert in eine Zeitinformation um |
| *Weekday* | Liefert den Wochentag aus einem Datum zurück |
| *WeekdayName* | Liefert den Klartextnamen des Wochentags |
| *Year* | Liefert das Jahr aus einer Datumsangabe zurück |

***Tabelle 2.5:*** *Zeitfunktionen*

## Vergleichsoperatoren und Bedingungen

Die Rechenoperatoren und mathematischen Funktionen, die Sie gerade kennen gelernt haben, rechnen die unterschiedlichsten Dinge für Sie aus, und je nach Berechnung und Ausgangswerten erhalten Sie die unterschiedlichsten Ergebnisse zurück.

Vergleichsoperatoren funktionieren anders. Sie liefern als Ergebnis immer nur entweder *True* oder *False*. Das ist verständlich, wenn man sich näher ansieht, was Vergleichsoperatoren eigentlich leisten. Sie beantworten nämlich stets nur eine Frage, die Sie stellen, mit »stimmt« oder »stimmt nicht«.

Schauen Sie sich zuerst die Riege der Vergleichsoperatoren an, die VBScript Ihnen zur Verfügung stellt.

| Operator | Beschreibung |
|---|---|
| = | Gleichheit |
| < | Kleiner als |
| > | Größer als |
| <> | Ungleich |
| <= | Kleiner gleich |
| >= | Größer gleich |

***Tabelle 2.6:*** *Vergleichsoperatoren*

Im einfachsten Fall prüft also ein Vergleichsoperator, ob ein Wert einem anderen Vergleichswert entspricht:

```
MsgBox 1 = 1
MsgBox 1 = 2
```

Auch wenn sich der tiefere Sinn dieser beiden Zeilen noch nicht sofort zeigt, sehen Sie zumindest schon einmal Vergleichsoperatoren bei der Arbeit. In der ersten Zeile gibt *MsgBox* das Ergebnis des Vergleichs *1 = 1* aus. Weil 1 gleich 1 ist, lautet das Ergebnis *True* oder *Wahr.* In der zweiten Zeile wird *1 = 2* verglichen. Erwartungsgemäß ist das Ergebnis diesmal *False* oder *Falsch.*

Bevor Sie Vergleichsoperatoren in sinnvolleren Skripts erleben, lassen Sie uns die erste Zeile noch einmal näher ansehen:

```
MsgBox 1 = 1
```

Hier steht ein Gleichheitszeichen, und bisher hatten Gleichheitszeichen immer Zuweisungen an eine Variable zur Folge. Hier ist das Gleichheitszeichen aber plötzlich keine Zuweisung mehr, sondern ein Vergleichsoperator. Wieso?

Immer wenn der Begriff links vom Gleichheitszeichen der erste Begriff in einer Zeile ist, handelt es sich um eine Variablenzuweisung. Die folgende Zeile würde also den Wert *12* an die Variable *test* zuweisen:

```
test = 12
```

Ist der Begriff links vom Gleichheitszeichen nicht der erste in der Zeile, dann handelt es sich um einen Vergleich. In der folgenden Zeile wird also verglichen, ob die Variable *test* den Wert *12* enthält oder nicht, und das Ergebnis von *MsgBox* ausgegeben:

```
MsgBox test = 12
```

**Rätsel 12:** Schauen Sie sich die folgenden Zeilen an und versuchen Sie zu erklären, was hier passiert. Was bedeuten die Gleichheitszeichen hier? Welches Gleichheitszeichen hat welche Bedeutung, und was erwarten Sie als Ergebnis in *ok*?

```
alter = 12
ok = alter = 18
MsgBox ok
```

Vergleichsoperatoren allein sind äußerst langweilig und nutzlos. Zusammen mit einem Verbündeten werden Vergleichsoperatoren allerdings zu einem der mächtigsten und wichtigsten Elemente Ihrer Skripts. Bei diesem Verbündeten handelt es sich um die Gruppe der Bedingungen.

### Bedingungen und Vergleichsoperatoren gehören zusammen

Bedingungen bilden die Grundlage für die Intelligenz der Skripts. Dank Bedingungen können Ihre Skripts mal so und mal so reagieren, je nachdem, wie die Ausgangssituation – oder genauer: die ausgewertete Bedingung – aussieht. Eine Bedingung nimmt also einen Vergleich und schaut, ob der Vergleich stimmt oder nicht – ob der Vergleich also *True* oder *False* zurückliefert. Je nach Ergebnis führt die Bedingung dann unterschiedliche Skriptanweisungen aus.

Am besten schauen Sie sich das an einem Beispiel an:

```
kennwort = InputBox("Ihr Kennwort?")
If kennwort <> "strenggeheim" Then
    MsgBox "Du kommst hier nicht rein!", vbCritical
    WScript.Quit
End If
MsgBox "Drin."
```

***Listing 2.18:*** *Eine sehr simple Kennwortabfrage als Beispiel für Vergleiche*

In der ersten Zeile wird zuerst mit der inzwischen bekannten *InputBox*-Funktion ein Kennwort vom Skriptanwender abgefragt und in der Variablen *kennwort* gespeichert.

```
kennwort = InputBox("Ihr Kennwort?")
```

Gleich danach wird es spannend, denn in der zweiten Zeile folgt die Bedingung. Die Bedingung prüft, ob das eingegebene Wort dem vereinbarten Kennwort »strenggeheim« entspricht. Als Vergleichsoperator kommt hier »<>« zum Einsatz, was laut Tabelle 2.6 dem Vergleich »ungleich« entspricht. Wenn also der Inhalt der Variablen *kennwort* ungleich dem vereinbarten Kennwort ist, dann ergibt der Vergleich *False,* sonst *True.*

Das Ergebnis dieses Vergleichs wird an die Bedingung weitergegeben und ausgewertet. Die Bedingung selbst besteht aus diesem einfachen Gerüst:

```
If (Vergleich) Then
...
End If
```

An die Stelle von *(Vergleich)* gehört der Vergleich, der geprüft werden soll. Ergibt der Vergleich *True,* dann wird der Skriptteil darunter ausgeführt, sonst nicht. *End If* beendet die Bedingung. Alles, was anschließend folgt, wird wieder in jedem Fall und unabhängig vom Vergleich ausgeführt.

In der dritten Zeile wird eine Meldung ausgegeben, wenn das Kennwort nicht stimmt:

```
    MsgBox "Du kommst hier nicht rein!", vbCritical
```

Diese Zeile ist ein wenig eingerückt. Das ist zwar nicht Pflicht, aber eine gute Angewohnheit. Alle Zeilen nach *If* sollten bis zur *End If*-Anweisung eingerückt werden, damit man den Anfang und das Ende der Bedingung besser erkennen kann.

**Rätsel 13:** In dieser Zeile wird dem *MsgBox*-Befehl als zweites Argument *vbCritical* übergeben. Wissen Sie, was *vbCritical* ist? Können Sie sich vorstellen, was es bewirkt? Schlagen Sie den *MsgBox*-Befehl in der WSH-Referenz nach, wenn Sie sich nicht sicher sind!

Nachdem das Skript bei einem falschen Kennwort seine Warnmeldung ausgegeben hat, folgt in der vierten Zeile noch eine neue Anweisung:

```
    WScript.Quit
```

Dieser Befehl sieht etwas sonderbar aus, weil er in der Mitte einen Punkt hat. Sie werden etwas später näher untersuchen, was es mit solchen Anweisungen auf sich hat. Für den Augenblick genügt es völlig zu wissen, dass diese Anweisung das Skript abbricht. Wenn Sie neugierig sind, dann schlagen Sie den Befehl *Quit* einfach mal in der WSH-Referenz nach.

**Rätsel 14:** Können Sie sich vorstellen, warum das Skript ein falsches Kennwort bemängelt, wenn Sie »Strenggeheim« statt »strenggeheim« eingeben?

Die *If…End If*-Bedingung kommt nicht nur mit einem einzigen Vergleich zurecht, sondern kann beinahe beliebig erweitert werden. Erinnern Sie sich noch an Ihr *MsgBox*-Skript Listing 2.2? Mit Ihrem neuen Wissen können Sie zwar immer noch keine Computer herunterfahren, aber das Ergebnis der *MsgBox* nun wenigstens schon mal auswerten:

```
antwort = MsgBox("Wollen Sie den Rechner herunterfahren?", vbYesNoCancel)
if antwort = vbYes then
   MsgBox "Ok, fahre die Kiste jetzt herunter!"
   MsgBox "Nur geblufft... Na, geschwitzt?"
ElseIf antwort = vbNo Then
   MsgBox "Dann eben nicht..."
ElseIf antwort = vbCancel Then
   MsgBox "Nicht sehr entscheidungsfreudig, scheint mir..."
Else
   MsgBox "Huch? Diese Antwort kenne ich gar nicht!"
End If
```

***Listing 2.19:*** *Ergebnis einer MsgBox detailliert auswerten*

**Rätsel 15:** Können Sie erklären, was *vbYes, vbNo* und *vbCancel* sind? Warum stehen diese Begriffe nicht in Anführungszeichen? Wann wird der Skriptcode hinter *Else* ausgeführt?

Die *If…End If*-Bedingung, die Sie gerade eingesetzt haben, ist nicht die einzige Art, wie Skripts Vergleiche auswerten können. Sie werden später noch mehr dazu lesen, aber für den Moment reicht das, was *If…End If* für Sie leistet, völlig aus.

## Logische Operatoren

Vergleichsoperatoren sind wichtig für einzelne konkrete Vergleiche. Ist das Alter größer oder gleich 18? Ist der freie Speicherplatz kleiner 100.000 Byte? Entspricht das eingegebene Kennwort der Vorgabe?

Logische Operatoren gehen eine Stufe weiter und verknüpfen die Ergebnisse mehrerer Vergleiche zu einem einzigen Resultat. So kann Ihre Bedingung auf sehr komplexe Situationen reagieren, denn die Ergebnisse der einzelnen Vergleiche werden durch die logischen Operatoren wieder auf *True* oder *False* reduziert.

Die Entscheidungen, die sich im Kopf eines Türstehers abspielen, könnten zum Beispiel auf diese Weise formuliert werden (ohne dieser Berufsgruppe eine simplifizierte Denkweise unterstellen zu wollen):

```
If (alter >= 18 And geschlecht = "m") Or besterfreund = True Or geschlecht = "w" Then
     MsgBox "Herzlich Willkommen!"
End If
```

***Listing 2.20:*** *Einfache »Türsteher«-Regel verknüpft Vergleiche mit logischen Operatoren*

| Operator | Beschreibung |
|---|---|
| *Not* | Dreht *True* und *False* um |
| *And* | *True*, wenn beide Vergleiche *True* sind (alle Vergleiche müssen stimmen) |
| *Or* | *True*, wenn einer der beiden Vergleiche *True* ist (mindestens ein Vergleich muss stimmen) |
| *Xor* | Entweder/Oder: *True*, wenn genau einer der beiden Vergleiche *True* ist |

***Tabelle 2.7:*** *Logische Operatoren*

Erinnern Sie sich noch an Ihr einfaches Kennwort-Skript? Dort kam diese Bedingung zum Einsatz:

```
If kennwort <> "strenggeheim" Then
```

Ein Blick in Tabelle 2.7 erklärt, warum Sie auch ebenso gut die Bedingung auf diese Weise hätten formulieren können:

```
If Not kennwort = "strenggeheim" Then
```

Der Vergleich prüft hier genau das Gegenteil, nämlich ob der Inhalt von *kennwort* mit dem vereinbarten Kennwort identisch ist. Der logische Operator *Not* dreht dann das Ergebnis nachträglich um, sodass letzten Endes ein *True* geliefert wird, wenn das Kennwort nicht stimmt.

### Bitte ein Bit: logische Operatoren auf Bitebene

Letzten Endes bestehen alle Daten und Zahlen aus einzelnen Bits, die entweder 1 oder 0 enthalten können. Viele Anwender und nicht wenige Programmierer versuchen diese binäre Welt zu ignorieren, weil sie nicht so einfach zu verstehen ist, und meistens gelingt diese Verdrängung auch. Für die allermeisten Anwendungen und Aufgaben braucht man wirklich nichts über Bits und Bitoperationen zu wissen, und wenn Sie gerade keine Lust darauf haben, dann überspringen Sie diesen Teil einfach.

Manchmal aber braucht man Bits aber doch. Und zwar meistens dann, wenn man mit Bitfeldern zu tun bekommt, bei denen jedes einzelne Bit eine bestimmte Funktion hat. Dabei kann man sich die einzelnen Bits wie separate Schalter vorstellen, und alle Bits zusammen funktionieren wie ein Schaltpult. Das findet sich zum Beispiel bei Dateien und den so genannten Dateiattributen (Archiv, Versteckt, System, etc.), über die später noch mehr berichtet wird.

Wenn Sie mit solchen Bitfeldern zu tun haben, dann brauchen Sie logische Operatoren, um herauszufinden, ob ein bestimmtes Bit gesetzt ist oder nicht. Logische Operatoren können nämlich nicht nur zwei Bits vergleichen, wie das bei Vergleichen auf *True* oder *False* der Fall ist, sondern auch mehrere Bits in einem Aufwasch.

Eben haben Sie gesehen, dass der Operator *And* den Wert *True* ergibt, wenn die beiden Ausgangswerte *True* sind:

```
1 And 1 = 1
1 And 0 = 0
0 And 1 = 0
0 And 0 = 0
```

Stellen Sie sich vor, die Ausgangswerte sind keine einfachen *True*- oder *False*-Werte, sondern Zahlen. Eine Zahl besteht aus vielen Bits. Auch hier funktioniert *And,* und zwar so:

```
11010 And 00011 = 00010
```

Wieder kommt nur das Bit durch, das in beiden Ausgangswerten gesetzt war. Und wofür könnte man das gebrauchen?

**Ist ein Bit gesetzt?**

Möchten Sie zum Beispiel prüfen, ob genau das 4. Bit in einem Parameter gesetzt ist, dann tun Sie das so:

```
flag = 12346
If (flag And 2^4) = 2^4 Then
   MsgBox "Bit gesetzt!"
Else
   MsgBox "Bit nicht gesetzt!"
End If
```

***Listing 2.21:*** *Überprüfung auf gesetztes Bit*

Mit logischen Operatoren können Sie umgekehrt auch einzelne Bits setzen oder löschen, ohne die übrigen Bits zu beeinflussen. Wollen Sie zum Beispiel das fünfte Bit in einer Zahl setzen, dann wählen Sie den Operator *Or.* Bei einfachen *True*- und *False*-Werten funktioniert dieser Operator so:

```
1 Or 1 = 1
1 Or 0 = 1
0 Or 1 = 1
0 Or 0 = 0
```

Es braucht also nur das Bit eines der beiden Ausgangswerte gesetzt zu sein, damit es auch im Ergebnis gesetzt ist. Ideal also, um zu einer Zahl ein bestimmtes Bit zu addieren:

```
11010 Or 00011 = 11011
```

**Selbst ein Bit setzen**

Wollen Sie per Skript das fünfte Bit einer Zahl setzen, dann machen Sie das so:

```
flag = 22222
MsgBox flag
flag = flag Or 2^5
MsgBox flag
```

***Listing 2.22:*** *Bitwert zu einer Zahl addieren*

**Ein bestimmtes Bit löschen**

Und wie könnte man Bits in Zahlen löschen? Dazu verwenden Sie den Operator *And.* Der löscht alle Bits, die nicht in beiden Ausgangswerten vorkommen. Das folgende Skript löscht also alle Bits bis auf das Bit 3:

```
flag = 22222
MsgBox flag
flag = flag And 2^3
MsgBox flag
```

***Listing 2.23:*** *Alle Bits bis auf das angegebene löschen*

Sie wollen aber eigentlich den umgekehrten Fall, nämlich alle Bits erhalten und nur Bit 3 löschen. Um das zu erreichen, drehen Sie den zweiten Ausgangswert für *And* einfach um, verknüpfen also den Inhalt von *flag* nicht mit *2^3*, sondern mit *Not 2^3*. Not setzt alle Bits außer Bit 3. Und weil bei *And* nur die Bits überleben, die in beiden Ausgangswerten vorkommen, fliegt Bit 3 heraus:

```
flag = 22222
MsgBox flag
flag = flag And Not 2^3
MsgBox flag
```

***Listing 2.24:*** *Ein bestimmtes Bit löschen*

**Zusammengefasst**

Die einfache Faustregel lautet also: *Or* setzt ein Bit, während *And Not* ein Bit löscht. Bits werden dabei über ihren Zahlenwert angesprochen:

# Zufallszahlen

Zufallszahlen machen Spaß. Mit Zufallszahlen kann man elektronische Würfel, Lottozahlengeneratoren und Promilletester bauen. Nur: Wie kommt ein Skript an Zufallszahlen heran?

Auch für Zufallszahlen gibt es eine passende Funktion, die *Rnd* heißt. Sie sehen also: Skripts werden immer vielseitiger, je mehr Befehle und Funktionen Sie kennen lernen. Die Befehle und Funktionen kann man sich fast wie die Zaubersprüche in Harry Potters Zauberbüchern vorstellen. Je mehr Sie kennen, desto mächtiger werden Sie – oder jedenfalls Ihre Skripts.

### Ein elektronischer Würfel

*Rnd* liefert bei jedem Aufruf eine Zufallszahl, die größer oder gleich 0, aber kleiner als 1 ist. Damit Sie eine Zufallszahl aus einem ganz bestimmten Bereich bekommen, ist eine kleine Formel nötig. Das nächste Skript zeigt, wie man damit zufällige Zahlen zwischen 1 und 6 würfeln kann:

```
Untergrenze = 1
Obergrenze = 6
zahl = Int((Obergrenze - Untergrenze + 1) * Rnd + Untergrenze)
MsgBox zahl
zahl = Int((Obergrenze - Untergrenze + 1) * Rnd + Untergrenze)
MsgBox zahl
zahl = Int((Obergrenze - Untergrenze + 1) * Rnd + Untergrenze)
MsgBox zahl
```

***Listing 2.25:*** *Ein einfacher elektronischer Würfel*

Da *Rnd* stets Zufallszahlen zwischen *0* und <1 liefert, aber man meist Zufallszahlen aus einem ganz anderen Zahlenbereich braucht, muss das Skript die gelieferte Zufallszahl zuerst in den gewünschten Zahlenbereich transformieren.

Die Anzahl der Zufallszahlen wird durch *Obergrenze - Untergrenze + 1* bestimmt. Bei einem Würfel liegt die Untergrenze bei *1*, die Obergrenze bei *6*, und *Obergrenze - Untergrenze + 1* ergibt 6 benötigte Zufallszahlen. Multipliziert man nun das Ergebnis von *Rnd* mit der Anzahl der benötigten Zufallszahlen, also in diesem Fall *6*, dann ist das Ergebnis nicht mehr eine Zufallszahl zwischen *0* und *0,999999*, sondern zwischen *0* und *0,999999 * 6*, also zwischen *0* und *5,999999*. Jetzt wird noch die Untergrenze hinzugezählt. Sie erhalten nun Zufallszahlen zwischen *1* und *6,999999*.

*Int* schneidet die Nachkommastellen ab, sodass vor dem Komma die gewünschten Zufallszahlen zwischen *1* und *6* übrig bleiben.

**HINWEIS** Können Sie das Skript so umformulieren, dass es nicht für Sie würfelt, sondern Lottozahlen ausspuckt? Wie muss das Skript aussehen, damit es für Spiele funktioniert, die zwei Würfel verwenden?

Eins ist allerdings komisch, und das werden Sie feststellen, wenn Sie das Skript ein paar Mal aufrufen: Es liefert Ihnen nämlich bei jedem Aufruf dieselbe Folge von Zufallszahlen.

Der Grund: natürlich stammen die Zufallszahlen nicht aus dem Weltall, sondern sind Ergebnis einer mathematischen Formel. Diese Formel beginnt immer mit demselben Startwert, und wie jede gut erzogene mathematische Formel liefert sie bei gleichem Ausgangswert auch immer die gleichen Ergebnisse. Die Zufallszahlen sind also gar nicht wirklich zufällig.


***Abbildung 2.30:*** *Ihr elektronischer Würfel würfelt für Sie zufällige Zahlen zwischen 1 und 6*

Deshalb brauchen Sie noch einen zweiten Befehl, der an den Anfang Ihres Skripts gehört und *Randomize* heißt. Der Befehl wirbelt den Startwert des Zufallszahlengenerators kräftig durcheinander, indem er die interne Windows-Systemzeit als neuen Ausgangswert setzt, sodass Ihr Skript nun endlich wirkliche Zufallszahlen liefert. Es sei denn, sie starten es in einem Paralleluniversum zur selben Zeit.

### Promilletester stoppt Reaktionszeit

Mit *Rnd* und *Randomize* kann man also skriptgesteuert Zufallszahlen erzeugen. Außer zum Würfeln oder als Lottoassistent lassen sich damit auch zufällige Verzögerungen in Skripts einbauen – ideal für einen Promilletester.


***Abbildung 2.31:*** *Das Skript stellt fest, wie schnell Sie klicken*

Dazu wird die Stoppuhr aus Listing 2.1, mit der Sie dieses Kapitel begonnen haben, ein klitzekleines bisschen umgebaut:

```
' Zufällige Verzögerung zwischen 500 und 5000 Millisekunden
' (0,5 - 5 Sekunden)
Untergrenze = 500
Obergrenze = 5000

zahl = Int((Obergrenze - Untergrenze + 1) * Rnd + Untergrenze)

' Eine zufällige Zeit verzögern
WScript.Sleep zahl

' Zeitpunkt merken
start = Timer

' Reaktionstester anzeigen
MsgBox "Klicken Sie so schnell wie möglich auf OK!"

ende = Timer
reaktionszeit = ende - start
```

```
MsgBox "Ihre Reaktionszeit: " & reaktionszeit & " Sekunden."
If reaktionszeit < 0.5 Then
   MsgBox "Super!",,"Schnell"
ElseIf reaktionszeit < 1 Then
   MsgBox "Naja, versuchen wir's lieber später noch mal...!",,"Mittel"
Else
   MsgBox "Verwendet Ihr Hirn noch Kupferkabel? Tsis...",,"LANGSAM!!!"
End If
```

***Listing 2.26:*** *Reaktionszeit-Tester*

Die wesentliche Änderung findet sich am Skriptanfang: Hier wird eine Zufallszahl zwischen 500 und 5000 generiert, die bestimmt, wie lange das Skript warten soll, bevor es Sie zum Klicken auffordert. Danach pausiert das Skript mit *WScript.Sleep* zwischen 0,5 und 5 Sekunden (500 bis 5000 Millisekunden, das Ergebnis des Zufallsgenerators). *WScript.Sleep* ist ähnlich wie der früher schon einmal aufgetauchte *WScript.Quit*-Befehl ein besonderer Befehl, der nicht von VBScript stammt, sondern direkt vom Windows Script Host, der das Skript ausführt. Im Augenblick genügt es völlig zu wissen, dass *WScript.Sleep* das Skript für die angegebene Zahl von Millisekunden anhält, und dass eine Sekunde aus genau 1000 Millisekunden besteht.

## Schleifen: Skriptteile mehrfach ausführen

Langsam wird es Zeit, Schleifen kennen zu lernen, denn Schleifen machen Ihre bisherigen Skripts noch vielseitiger.

Die offizielle Job-Beschreibung einer Schleife liest sich so: »wiederholt einen bestimmten Skriptbereich so oft wie angegeben«. Weil das langweilig klingt, schauen Sie lieber mal, was man mit Schleifen in der Praxis wirklich macht:

- **Schleifen mit fester Anzahl:** Wollen Sie genau sieben Zufallszahlen für den Lottoschein auswerfen, dann brauchen Sie eine Schleife, die genau sieben Mal läuft. Diese Art der Schleife wird *For…Next*-Schleife genannt.
- **Schleifen mit variabler Anzahl:** Wissen Sie selbst nicht so genau, wie oft eine Schleife laufen soll, wenn Sie das Skript entwickeln, dann braucht Ihnen das nicht peinlich zu sein. Häufig richtet es sich nach den Laufzeitbedingungen des Skripts, wie oft ein Skriptteil wiederholt werden soll, und da Sie kein Wahrsager sind, können Sie bei der Skripterstellung nicht sagen, wie oft ein Skriptanwender zum Beispiel mit einem elektronischen Würfel würfeln will, bevor er keine Lust mehr hat oder das Essen auf dem Tisch steht. Hier brauchen Sie eine *Do…Loop*-Schleife.

### Eine feste Anzahl von Wiederholungen

Wie Sie genau eine Zufallszahl ziehen, haben Sie bereits gelesen. Falls Sie zu faul sind, Ihren Lottoschein selbst auszufüllen, könnten Sie natürlich ein Skript damit beauftragen, die Lottozahlen für Sie zu ziehen. Allerdings brauchen Sie sechs davon. Nicht nur eine. Es sei denn, Sie füllen einen Systemschein aus.

Ihr Skript könnte mit Ihrem bisherigen Wissen die Arbeit durchaus erledigen. Sie würden einfach die Skriptzeile, die die Zufallszahl erzeugt und ausgibt, sechs Mal hintereinander schreiben. Das wäre allerdings nicht nur viel Schreibarbeit, es würde auch hässlich aussehen. Spätestens wenn Ihr Skript irgendwann nicht nur sechs, sondern vielleicht 500 gleichartige Dinge tun soll, stößt diese Variante auch an die Grenzen der Praktikabilität.


***Abbildung 2.32:*** *Automatischer Lottozahlengenerator*

Das nächste Skript macht es eleganter und nutzt eine *For…Next*-Schleife, um so viele Zufallszahlen zu ziehen, wie Sie brauchen.

```
Randomize
Untergrenze = 1
Obergrenze = 49
For x = 1 To 6
   zahl = Int((Obergrenze - Untergrenze + 1) * Rnd + Untergrenze)
   liste = liste & zahl & vbTab
Next
MsgBox liste,,"Ihre Lottozahlen"
```

***Listing 2.27:*** *Einfacher Lottozahlengenerator*

**Rätsel 16:** Wo müssen Sie Ihr Skript ändern, damit es nicht sechs Zufallszahlen liefert, sondern wie in Abbildung 2.32 sieben Zahlen?

Allerdings hat das Skript noch einen kleinen Schönheitsfehler: Doppelt vorkommende Zufallszahlen werden in dieser Version noch nicht erkannt und abgefangen. In einem späteren Kapitel werden Sie die so genannten Dictionarys kennen lernen, mit denen man Doppelgänger finden und eliminieren kann.

### Eine variable Anzahl von Wiederholungen

Nicht immer ist Ihnen am grünen Tisch bei der Programmierung eines Skripts klar, wie oft eine Schleife laufen muss, bis die Arbeit erledigt ist. Manchmal hängt die Anzahl der Schleifendurchläufe von Umständen ab, die erst dann ersichtlich sind, wenn das Skript im Alltag verwendet wird.

#### Ein elektronischer Würfel

Das nächste Skript ist so ein Fall und baut wieder auf einem schon bekannten Skript auf, nämlich auf Listing 2.25, dem einfachen elektronischen Würfel. Weil ein Würfelwurf meist nicht ausreicht, um eine anständige Runde Mensch-ärgere-Dich-nicht zu Ende zu spielen, liefert das nächste Skript so viele Würfelwürfe, wie Sie wollen. Und weil Sie bei der Programmierung des Skripts nicht wissen, wie oft der Würfel fliegen muss, bis das Spiel (oder die Spiellust) zu Ende ist, wird diesmal nicht die *For…Next*-Schleife eingesetzt, sondern die *Do…Loop*-Schleife.

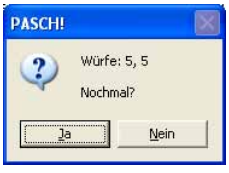


***Abbildung 2.33:*** *Ein elektronischer Doppelwürfel mit Pasch-Automatik*

Damit Sie mit dem Würfel jede Art von Würfelspiel spielen können, handelt es sich außerdem um einen Doppelwürfel, der gleich ein Augenpaar liefert und sogar eine eingebaute Paschautomatik enthält: Würfeln beide Würfel dieselbe Zahl, wird ein besonderer Hinweis angezeigt.

```
Randomize
Untergrenze = 1
Obergrenze = 6
Do
   zahl1 = Int((Obergrenze - Untergrenze + 1) * Rnd + Untergrenze)
   zahl2 = Int((Obergrenze - Untergrenze + 1) * Rnd + Untergrenze)
   ' Pasch-Automatik
   If zahl1 = zahl2 Then
      titel = "PASCH!"
   Else
      titel = ""
   End If
   text = "Würfe: " & zahl1 & ", " & zahl2 & vbNewLine & vbNewLine & "Nochmal?"
   antwort = MsgBox(text, vbYesNo + vbQuestion, titel)
Loop Until antwort = vbNo
```

***Listing 2.28:*** *Doppelwürfel*

Der Skriptteil innerhalb der *Do…Loop*-Schleife wird dabei unendlich oft wiederholt. Wobei das etwas gelogen ist. Die Schleife endet, wenn die Abbruchbedingung eintritt, und diese Abbruchbedingung steht hinter *Loop*. Sie heißt in diesem Fall *Until antwort = vbNo*.

Die Schleife würfelt also unermüdlich so lange neue Zahlen, bis die Variable *antwort* den Inhalt *vbNo* enthält. Und wie kommt die Variable *antwort* zu diesem Inhalt?

Innerhalb der Schleife wird das aktuelle Würfelergebnis von einer *MsgBox* angezeigt. Die *MsgBox* fragt außerdem nach, ob der Anwender noch mal würfeln möchte. Dazu zeigt die *MsgBox* die beiden Schaltflächen *vbYes* und *vbNo* an. Das Ergebnis, also die Kennziffer der tatsächlich angeklickten Schaltfläche, wird in der Variablen *antwort* gespeichert.

Ahnen Sie, wie der Schleifenabbruch funktioniert? Genau: Hat der Spieler keine Lust mehr, dann klickt er auf die *Nein*-Schaltfläche, und in *antwort* wird *vbNo* gespeichert. Danach erreicht das Skript die *Loop*-Anweisung, die die Abbruchbedingung überprüft. Weil jetzt der erwartete Wert *vbNo* in *antwort* steht, ist die Schleife zufrieden und bricht ab. Das Skript wird nun nach der Schleife fortgeführt.

Solche Endlosschleifen sind enorm praktisch, weil das Abbruchkriterium erst zur Laufzeit des Skripts ausgewertet wird.

Allerdings sind *Do…Loop*-Schleifen auch ein bisschen gefährlich: Stimmt Ihr Abbruchkriterium nicht oder wird es nie erfüllt, dann läuft das Skript ewig. Sie müssen es dann im Windows Task Manager von Hand abbrechen (dort finden Sie es in der Taskliste unter dem Namen *WScript.exe*), oder Sie melden sich ab und wieder an.

### Anwender-Eingaben überprüfen

Schauen Sie sich zum Beispiel einmal an, wie Sie per Skript so lange nach einem Datum fragen, bis der Anwender tatsächlich ein Datum eingegeben hat:

```
frage = "Bitte geben Sie ein Datum ein!"
Do
   datum = InputBox(frage)
   If IsEmpty(datum) Then
      MsgBox "Sie wollen abbrechen? In Ordnung!"
```

```
      WScript.Quit
   End If
   frage = """" & datum & """ ist kein Datum. Bitte geben Sie ein Datum ein!"
Loop Until IsDate(datum)
MsgBox FormatDateTime(datum, vbLongDate)
```

***Listing 2.29:*** *Ein Datum mit Überprüfung erfragen*

Es lohnt sich, dieses Skript wieder Schritt für Schritt zu besprechen. Beginnen Sie wieder mit der ersten Zeile:

```
frage = "Bitte geben Sie ein Datum ein!"
```

Hier wird in der Variablen *frage* die Frage festgelegt, die gleich in der Schleife dem Anwender gestellt wird. Sie werden gleich sehen, warum die Frage in einer eigenen Variablen untergebracht ist. Danach beginnt die *Do…Loop*-Schleife, und das Skript fragt den Anwender nach einem Datum. Die Eingabe, die der Anwender macht, wird in der zweiten Variablen *datum* gespeichert.

```
datum = InputBox(frage)
```


***Abbildung 2.34:*** *Das Skript prüft die Eingabe des Anwenders und akzeptiert nur ein Datum*

Nun folgt eine Bedingung. Sie erinnern sich? Bedingungen führen den Skriptteil dahinter nur aus, wenn die Bedingung stimmt. Die Zeile lautet:

```
If IsEmpty(datum) Then
```

Der eigentliche Vergleich, den die Bedingung auswertet, lautet: *IsEmpty(datum)*. Wie bei jedem Vergleich muss das Ergebnis entweder *True* oder *False* sein. Wenn Sie die *IsEmpty*-Funktion in der Referenz nachschlagen, dann werden Sie feststellen, dass *IsEmpty* tatsächlich entweder *True* oder *False* zurückliefert. Anstelle von Vergleichsoperatoren können Sie in Bedingungen also auch Funktionen einsetzen, wenn diese *True* oder *False* zurückliefern. Der Bedingung selbst ist es nämlich egal, wer ihr das Ergebnis liefert, solange es *True* oder *False* ist.


***Abbildung 2.35:*** *Das Skript reagiert auf die* Abbrechen*-Schaltfläche und bricht nach Bestätigung ab*

*IsEmpty* prüft, ob die angegebene Variable leer ist. Und zwar richtig leer, nämlich undefiniert. Gibt der Skriptanwender in der *InputBox* einfach nichts an, indem er auf *OK* klickt, dann ist die Variable *datum* nicht leer, sondern enthält einen String der Länge 0. Leer ist *datum* nur, wenn *InputBox* gar nichts in der Variablen speichert. Das passiert nur in einem einzigen Fall: wenn der Anwender auf die *Abbrechen*-Schaltfläche der *InputBox* klickt.

Die Bedingung ist also nur dann erfüllt, wenn jemand auf *Abbrechen* klickt. In diesem Fall bestätigt das Skript den gewünschten Abbruch und beendet sich mit *WScript.Quit* selbst.

Hat der Anwender nicht auf *Abbrechen* geklickt, dann wird nun diese Zeile ausgeführt:

```
frage = """" & datum & """ ist kein Datum. Bitte geben Sie ein Datum ein!"
```

Hier wird die Variable *frage* neu zugewiesen. Huch! Das klingt sonderbar, denn das Skript hat die Frage dem Anwender doch schon gestellt, als *InputBox* aufgerufen wurde. Tatsächlich könnte es aber sein, dass der Anwender gar kein Datum eingegeben hat und deshalb die Schleife gleich wiederholt wird. Damit der Anwender dann weiß, warum ihn schon wieder eine *InputBox* heimsucht und was er falsch gemacht hat, ändert das Skript die Frage – nur für den Fall.

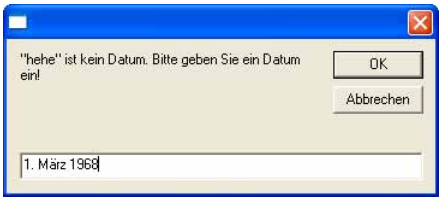


***Abbildung 2.36:*** *Bei einer falschen Eingabe ändert sich die Frage, und es erscheint eine neue Abfrage*

Die Schleife endet mit der *Loop*-Anweisung und der Abbruchbedingung:

```
Loop Until IsDate(datum)
```

Hier entscheidet sich also, ob die Schleife noch einmal läuft oder nicht. Die Abbruchbedingung selbst heißt *IsDate(datum)*. Schlagen Sie *IsDate* in der Referenz nach, dann werden Sie feststellen, dass *IsDate* fast genauso funktioniert wie *IsEmpty,* nur prüft *IsDate* diesmal, ob die angegebene Variable ein Datum ist. Falls nicht, gibt *IsDate* den Wert *False* zurück. Damit ist die Abbruchbedingung nicht gegeben, und das Skript würde den Anwender mit der neuen Frage erneut um eine Eingabe bitten.

| Funktion | Beschreibung |
|---|---|
| *isArray* | *True*, wenn die Variable ein Feld enthält |
| *isDate* | *True*, wenn der Variableninhalt als Datum interpretierbar ist |
| *isEmpty* | *True*, wenn die Variable keinen Wert enthält |
| *isNull* | *True*, wenn die Variable einen Nullwert enthält |
| *isObject* | *True*, wenn die Variable einen Objektzeiger enthält |
| *isNumeric* | *True*, wenn der Variableninhalt als Zahl interpretierbar ist |

***Tabelle 2.8:*** *Funktionen zum Erkennen eines bestimmten Datentyps*

Gibt der Anwender endlich ein Datum ein, dann belohnt ihn das Skript mit der Information, auf welchen Wochentag dieses Datum fällt. Die Magie dahinter – der *FormatDateTime*-Befehl – ist für Sie allerdings bereits ein alter Hut.


***Abbildung 2.37:*** *Den Wochentag des eingegebenen Datums bestimmen*

**Rätsel 17:** Schlagen Sie in der Referenz auch die übrigen *is*…-Funktionen nach. Fällt Ihnen eine *is*…-Funktion ins Auge, mit der Sie überprüfen können, ob der Anwender eine Zahl eingegeben hat? Formulieren Sie das Skript doch einmal so um, dass es den Anwender so lange nervt, bis er eine Zahl eingegeben hat!

### Speed-Calculator

Schleifen können noch viel mehr. Dabei kommt es hauptsächlich auf Ihre Fantasie an. Schauen Sie sich doch mal das nächste Skript an, einen Speed-Calculator. Wenn Sie schnell größere Zahlenkolonnen zusammenzählen müssen, dann lassen die sich hier ganz besonders bequem eingeben. Die Schleife bricht erst ab, wenn Sie entweder eine leere Eingabe machen (EINGABE drücken, ohne Text eingegeben zu haben) oder eine *0* eingeben.

```
zahl = 0
Do
   text = historie & vbNewLine & "_____________" & vbNewLine & "= " & zahl
   eingabe = InputBox(text & vbNewLine & vbNewLine & "Hinzuzählen:","Speed-Calc", "0")
   If Not IsNumeric(eingabe) Then Exit Do
   If eingabe = "" Or eingabe = "0" Then Exit Do
   historie = historie & vbNewLine & "+ " & eingabe
   zahl = zahl + eingabe
Loop
```

***Listing 2.30:*** *Speed-Calc-Rechner für das schnelle Addieren von Zahlenwerten*

Haben Sie übrigens den wesentlichsten Unterschied zum letzten Skript bemerkt? Dieses Skript verwendet eine *Do…Loop*-Schleife ohne Abbruchbedingung. Jedenfalls scheint das so auf den ersten Blick, denn hinter *Loop* steht nichts.

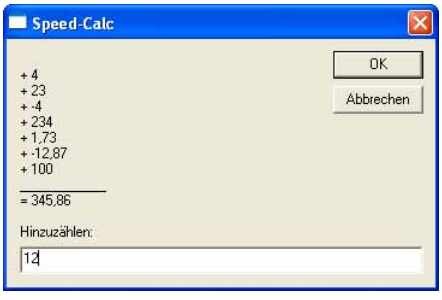


***Abbildung 2.38:*** *Per Skript ganz schnell ein paar Zahlen addieren*

Skripts können Schleifen über den *Exit*-Befehl auch mittendrin abbrechen, und dieses Skript tut das genau auf diese Weise.

# Zusammenfassung

Auf den vorangegangenen Seiten haben Sie erste Erfahrung mit Skripts sammeln können und dabei die wesentlichen Zutaten kennen gelernt, aus denen jedes Skript besteht:

- Variablen
- Operatoren
- Befehle
- Strukturelemente

Lassen Sie uns zum Abschluss noch einmal die wichtigsten Regeln für diese Elemente rekapitulieren.

## Befehle

Befehle lösen bestimmte Aufgaben und können zum Beispiel ein Dialogfeld anzeigen (*MsgBox*), vom Anwender eine Antwort erfragen (*InputBox*) oder die aktuelle Zeit liefern (*Timer, Date* oder *Now*). Die meisten Befehle benötigen dazu von Ihnen zusätzliche Informationen, damit sie wissen, was Sie von ihnen erwarten. Diese zusätzlichen Informationen heißen Argumente und werden durch ein Leerzeichen getrennt hinter den Befehl geschrieben.

Liefert ein Befehl einen Ergebniswert zurück, dann nennt man diesen Befehl auch »Funktion«. Damit man das Ergebnis einer Funktion lesen kann, übergibt man der Funktion die Argumente in Klammern und erhält auf der linken Seite der Funktion das Ergebnis zurück. Das kann man dann entweder einer Variablen zuweisen oder direkt ausgeben, zum Beispiel mit dem *MsgBox*-Befehl.

- **Befehle können Funktionen oder Prozeduren sein:** Befehle sind alle Kommandos, die Sie an VBScript richten können. Schaut man sich die Befehle genauer an, dann sieht man zwei Untergruppen: Prozeduren und Funktionen. Prozeduren führen lediglich eine Aufgabe aus, ohne einen Rückgabewert zu liefern. Funktionen liefern zusätzlich ein Ergebnis zurück. VBScript enthält zum Beispiel eine ganze Reihe von mathematischen Funktionen, die in Tabelle 2.2 aufgeführt sind und die die einfachen Rechenoperatoren ergänzen. Aber auch die *InputBox,* die vom Anwender eine Eingabe erfragt, ist eine Funktion. Funktionen müssen also keineswegs immer mathematische Ergebnisse zurückliefern.
- **Argumente und Klammern:** Möchten Sie einen Befehl aufrufen, der kein Ergebnis zurückliefert, dann geben Sie die für den Befehl nötigen Zusatzinformationen, die Argumente, durch ein Leerzeichen getrennt hinter dem Befehl an. Rufen Sie dagegen einen Befehl auf, der ein Ergebnis zurückliefert (eine Funktion also), dann gehören die Argumente in Klammern. Ob ein Befehl wie eine Funktion oder wie eine Prozedur funktioniert, bestimmen Sie also selbst durch die Art des Aufrufs. Natürlich würde es Ihnen wenig bringen, den Rückgabewert einer Prozedur zu erfragen, denn weil Prozeduren keinen liefern, wäre das Ergebnis stets unbrauchbar. Umgekehrt können Sie bei Funktionen aber durchaus auf die Verwertung des Rückgabewertes verzichten und die Funktionen also wie Prozeduren aufrufen. Diese Praxis wird bei *MsgBox* sehr häufig eingesetzt, wenn damit nur Texte ausgegeben werden sollen.
- **Befehle nachschlagen:** Sie können alle in diesem Kapitel besprochenen Befehle in der WSH-Referenz nachschlagen und erfahren dann zu jedem Befehl, welche Argumente er erwartet, ob Argumente optional sind und was der Befehl eigentlich ganz genau leistet.

Wollen Sie sich die Befehle noch einmal näher ansehen, dann schlagen Sie auf Seite 21 nach.

## Variablen

Variablen speichern Zwischenergebnisse, damit spätere Skriptzeilen auf diese Zwischenergebnisse zugreifen und sie weiterverarbeiten können. Für Variablen gelten die folgenden Regeln:

- **Variablennamen:** Dürfen bis zu 255 Zeichen lang sein und aus den Zeichen »A« bis »Z« sowie allen Zahlen bestehen, solange die Zahlen nicht am Anfang stehen. Nicht erlaubt sind Zahlen am Sonderzeichen wie zum Beispiel die deutschen Umlaute und alle von VBScript belegten Namen, zum Beispiel *MsgBox* oder *vbYesNo*.
- **Variableninhalte:** Variablen können ganz beliebige Inhalte speichern, Zahlen ebenso wie Daten oder Texte. Sie brauchen sich um den Datentyp nicht zu kümmern. Damit werden die Variablen zu universellen Datenspeichern. Allerdings bleibt der Inhalt der Variablen nur so

lange erhalten, wie Ihr Skript läuft. Sobald es endet, werden automatisch auch alle Variablen gelöscht. Um einer Variablen Inhalt zuzuweisen, setzen Sie das Gleichheitszeichen ein.

Wollen Sie sich die Grundlagen der Variablen noch einmal näher ansehen, dann schlagen Sie auf Seite 24 nach.

## Operatoren

Operatoren verknüpfen die Werte rechts und links vom Operator. Es gibt drei verschiedene Operatorarten:

- **Rechenoperatoren:** Sie führen Berechnungen durch. Das einfachste Beispiel ist der »+«-Operator, der zwei Werte zusammenzählt. Eine vollständige Aufstellung aller Rechenoperatoren liefert Tabelle 2.1.
- **Vergleichsoperatoren:** Sie vergleichen die Ausdrücke rechts und links vom Operator und liefern *True* zurück, wenn der Vergleich stimmt, sonst *False*. Vergleichsoperatoren brauchen Sie, wenn Sie in Ihrem Skript eine Bedingung verwenden wollen. Tabelle 2.6 listet alle Vergleichsoperatoren auf, die es gibt.
- **Logische Operatoren:** Sie fassen das Ergebnis von mehreren Vergleichen zusammen und liefern entweder *True* oder *False* zurück. Logische Operatoren brauchen Sie, wenn Sie in Bedingungen nicht nur eine, sondern mehrere Vergleiche auswerten möchten. Außerdem können logische Operatoren prüfen, ob bestimmte Bits in einer Zahl gesetzt sind oder nicht. Eine Liste der logischen Operatoren finden Sie in Tabelle 2.7.

Wollen Sie sich die Grundlagen der Operatoren noch einmal näher ansehen, dann schlagen Sie ab Seite 27 nach.

## Strukturelemente

Normalerweise führt VBScript Ihr Skript streng zeilenweise von oben nach unten aus. Wurde die letzte Zeile des Skripts bearbeitet, dann endet es automatisch.

Strukturelemente sorgen dafür, dass bestimmte Teile Ihres Skripts mehrmals (Schleifen) oder nur unter bestimmten Voraussetzungen (Bedingungen) ausgeführt werden.

- **Schleifen:** Sie wiederholen den in der Schleife eingeschlossenen Skriptteil so oft, wie Sie wollen. Dabei stehen Ihnen zwei verschiedene Schleifenarten zur Verfügung. Die *For…Next*-Schleife wiederholt den dazwischen liegenden Skriptteil so oft, wie Sie angeben. Die *Do…Loop*-Schleife wiederholt den dazwischen liegenden Skriptteil unendlich lange, und Ihr Skript muss selbst entscheiden, wann es Zeit ist, die Schleife zu verlassen. Wollen Sie sich Schleifen noch einmal ansehen, dann schlagen Sie auf Seite 49 nach.
- **Bedingungen:** In diesem Kapitel haben Sie die *If…End If*-Bedingung kennen gelernt. Sie wertet einen Vergleich aus, und nur wenn der Vergleich stimmt, wird der eingeschlossene Skriptteil ausgeführt, andernfalls übersprungen. Möchten Sie sich die Grundlagen der Vergleichsoperatoren und Bedingungen noch einmal ansehen, dann schauen Sie auf Seite 42 nach.

## Allgemeine Regeln

VBScript ist eine einfache Programmiersprache, die versucht, möglichst viele Fehlerquellen von vornherein auszuschließen. Deshalb unterscheidet sie nicht zwischen Groß- und Kleinbuchstaben. Es gibt auch kein spezielles Zeilenendezeichen. Eine Skriptzeile endet genau dann, wenn Sie EINGABE drücken. Nur wenn eine Zeile mit einem Leerzeichen und einem Unterstrich endet, wird sie in der nächsten Zeile fortgesetzt.

Auch Kommentare sind erlaubt und dienen dazu, Ihr Skript möglichst gut zu erklären, damit man auch nächste Woche noch versteht, was Sie da programmiert haben. Kommentare werden mit einem einfachen Anführungszeichen eingeleitet, und VBScript ignoriert in dieser Zeile alles, was nach dem einfachen Anführungszeichen folgt.

Kann Ihr Skript nicht ausgeführt werden, weil es einen Fehler enthält, dann erscheint eine Fehlermeldung und meldet Zeile und Spalte, in denen der Fehler in Erscheinung getreten ist. Außerdem erhalten Sie eine kurze Fehlerbeschreibung. Sie können Ihr Skript dann per Rechtsklick und *Bearbeiten* zurück in den Editor laden, um den Fehler zu verbessern. Vergessen Sie aber nicht, das korrigierte Skript anschließend mit *Datei/Speichern* wieder abzuspeichern, bevor Sie es noch einmal ausprobieren.

# Testen Sie sich selbst

Mit Ihren neuen Kenntnissen sollten Sie die folgenden Aufgaben lösen können. Falls Sie bei einigen Aufgaben noch Schwierigkeiten haben, schauen Sie sich die in der Aufgabe genannten Seiten noch einmal näher an. Die Lösungen zu den Aufgaben finden Sie im Verzeichnis *Lösungen* auf der Buch-CD.

## Aufgabe 1

Tatendurstig versucht ein Skriptanwender, per Skript eine Meldung auf den Bildschirm zu zaubern, und setzt dafür den *MsgBox*-Befehl ein. Das Dialogfeld soll *Ja/Nein*-Schaltflächen und ein Fragezeichen-Symbol anzeigen. Die dafür nötigen Konstanten hat sich der Skriptanwender aus der Sprachreferenz herausgesucht, aber trotzdem meldet das Skript: »Beim Aufrufen einer Routine dürfen keine Klammern verwendet werden«.

```
MsgBox("Geht es Ihnen gut?", vbYesNo + vbQuestion)
```

***Listing 2.31:*** *Diese Zeile enthält einen sehr verbreiteten Einsteiger-Fehler. Nur welchen?*

Wo liegt der Fehler?

## Aufgabe 2

Voller Begeisterung versucht ein Skriptanwender, seine Reisekasse in Urlaubsdevisen umzurechnen. Der Umrechnungsfaktor für Euro nach Dollar beträgt bei den aktuellen Wechselkursen 0,96. Schnell ist das entsprechende Skript geschrieben und erfragt die Urlaubskasse sogar elegant mit einer *InputBox*. Bevor das Skript allerdings auch nur Piep macht, erscheint eine Fehlermeldung und gibt für die zweite Zeile aus: »Anweisungsende erwartet«. Welchen Fehler hat der Skriptautor gemacht?

```
euro = InputBox("Euro-Betrag eingeben!")
faktor = 0,96
dollar = euro * faktor
MsgBox euro & " Euronen entsprechen " & dollar & " Dollars"
```

***Listing 2.32:*** *Fehlerhafter Euro-Dollar-Rechner*

## Aufgabe 3

Ein Skriptanwender möchte ein Dialogfeld mit einer Anweisung für den Skriptanwender ausgeben und macht das so:

```
MsgBox "Bitte drücken Sie die "OK"-Schaltfläche!"
```

***Listing 2.33:*** *Wieso kann diese Zeile nicht ausgeführt werden?*

Allerdings erscheint nicht das erwünschte Dialogfeld, sondern eine Fehlermeldung: *Anweisungsende erwartet*. Wieso?

## Aufgabe 4

Ein Skriptanwender möchte ein wenig mit Kennwortabfragen experimentieren und verwendet das folgende Skript:

```
kennwort = InputBox("Geben Sie das Kennwort ein!")
If kennwort = "geheim"
   MsgBox "Richtig!"
Else
   MsgBox "Falsch"
End If
```

***Listing 2.34:*** *Eine Bedingung auswerten. Hier fehlt allerdings noch etwas*

Leider funktioniert das Skript so nicht, denn es erscheint sofort ein Fehler und meldet »Then erwartet«. Wie lässt sich das Problem beheben?