ABSOLUTE BEGINNER'S TUTORIAL

Eine Anleitung für Anfänger
zum Programmieren
mit BlitzMax

Noch unvollständige Alpha-Version:
9 von ca. 15 geplanten Kapitel fertiggestellt

Verbesserungsvorschläge sowie Tippfehler oder sachliche Kritik usw. bitte im entsprechenden Thread melden.

Inhaltsverzeichnis:

Weiterhin geplante Kapitel (unter Vorbehalt):

Kapitel 10 Funktionen
(Einführung, Argumente, Local - Global, Rueckgabewerte)

Kapitel 11 Mathematisches: Funktionen (Max, Min, ^, Mod, Abs, Sqr, Abstände (Pythagoras und Quadratabstand), Polarkoordinaten, ATan2, Hexzahlen, ... und anderes

Kapitel 12 Datei-Management:
Dateien laden, lesen, speichern, Daten schreiben, System-Dialoge u.a.

Kapitel 13 Grafik Teil 3: Bilder, animierte Bilder, Alpha (und SetBlend) Rotation, Scale, Orientation, Handle, Pixmaps, Timer u.a.

Kapitel 14 Interaktion Teil 2: Mausrad, Maus-Speed, Joystick, GetChar() Anwendungsbeispiel Drag & Drop, Anwendungsbeispiel Input()

Kapitel 15 Variablen Teil 3: True, False, Byte, Long, Double, (Zahlen-bereich, Speicherplatz), (Problematik kleiner und großer Zahlen), Kurzschreibweisen, Slices, eigene Types, Listen

VORWORT

Ob bei der Arbeit oder zum Vergnügen - Computer sind ein fester Bestandteil unseres alltäglichen Lebens geworden. Ob Schreibprogramme, Internetbrowser, Bildbearbeitungsprogramme oder Spiele, wir nutzen ganz selbstverständlich die faszinierenden Möglichkeiten, welche uns die verschiedenen Programme bieten.
Allerdings sind wir auch immer abhängig von dem, was sie uns bieten. Und Hand auf's Herz: Wer hat noch nicht fluchen müssen, dass ein Programm abstürzt, oder nicht so will, wie man selbst möchte oder auch einfach nur wieder unnötig Geld kostet?!
Wie wäre es also, wenn man sich einmal selbst an's Werk machen und ein eigenes Programm entwickeln würde? Dann könnte man sich endlich seine eigenen Wünsche erfüllen. Und wenn etwas schief geht, braucht man nicht auf andere zu schimpfen, sondern darf sich selbst verfluchen.
Die Erwartungen sollten anfangs allerdings nicht zu hoch sein. Es braucht ein bisschen Geduld und Ausdauer, um zu lernen, wie man programmiert, also dem Computer sagt, was er genau machen soll und was nicht.
Bekanntlich schwirren in den Eingeweiden des Computers letztlich nur Einsen und Nullen herum - stellvertretend für winzig kleine Schalter, die entweder an oder aus sind. Eigentlich müsste man also nur lernen, wie man riesige Ketten von Nullen und Einsen aneinanderreiht, die verschiedene Konsequenzen zur Folge haben.
Doch keine Sorge, so schlimm wird es nicht werden, denn es gibt ja die sogenannten "Hochsprachen", in denen keine Einsen und Nullen mehr auftauchen, sondern ganz vernünftige Worte bzw. Befehle, unter denen man sich etwas vorstellen kann.
"BlitzMax" ist eine solche Hochsprache.
Sie ist nicht so bekannt wie andere Programmiersprachen, wie etwa C++ oder Java, bietet aber beinahe die gleiche Vielseitigkeit. Außerdem eignet sie sich deshalb für Anfänger, weil sie ein Ableger der besonders einfachen "Basic"-Sprachfamilie ist, mit der man auch ohne viel Wissen ziemlich schnell loslegen kann.
Die vorliegende Anleitung ist keinesfalls eine erschöpfende Darstellung aller Befehle und Möglichkeiten von BlitzMax, sondern dient lediglich als erste Einführung für absolute Anfänger, also solche, die weder mit BlitzMax im Speziellen, noch mit dem eigenen Programmieren im Allgemeinen Erfahrungen haben. Vorausgesetzt wird daher in etwa nur der Wissensstand eines Schülers der 7.Klassenstufe.
Alle aufgeführten Code-Beispiele sind frei verwendbar, können ausgewählt, kopiert, im BlitzMaxIDE-Editor wieder eingefügt und dort mit dem Schalter der kleinen startenden Rakete  ausgeführt werden.
Zu jedem Kapitel gibt es kleine Übungsaufgaben, an denen das erlernte Wissen angewandt werden soll. Das selbstständige Erarbeiten dieser Übungen sollte nicht vernachlässigt werden. Zu jeder Übung gibt es im Anhang am Ende des Tutorials ein Lösungsbeispiel.
Nachdem alle Kapitel und Aufgaben in diesem Anfänger-Tutorial durchgearbeitet wurden, wird es immer noch eine Menge weiterer Funktionen und Befehle von BlitzMax zu entdecken geben. Diesbezüglich wird es sich dann lohnen, das deutsche BlitzForum zu durchstöbern und insbesondere seine Suchfunktion zu benutzen.
Außerdem findet man dort die Aufgaben der "Beginner's Practise Series", die eine erste lohnenswerte Herausforderung für angehende Programmierer darstellen und verschiedene relativ bescheidene Problemstellungen inklusive mehrerer Lösungsvarianten zur Diskussion stellen.

Ich wünsche viel Erfolg beim Lesen, Lernen und Programmieren

BlitzMoritz

Kapitel 1

Variablen Teil 1

1.1 Einführung

Bestimmt hast du in der Schule bereits mathematische Gleichungen wie etwa $x^2 + 1 = 5$ gesehen. Das x wird Variable oder auch Platzhalter genannt. Man kann das x durch Zahlen ersetzen und erhält dann entweder wahre oder falsche Aussagen:

Setze x = 0	$0^2 + 1 = 5$	falsch
Setze x = 1	$1^2 + 1 = 5$	falsch
Setze x = 2	$2^2 + 1 = 5$	richtig
Setze x = 3	$3^2 + 1 = 5$	falsch
Setze x = -2	$(-2)^2 + 1 = 5$	richtig

Selbstredend sucht man in der Mathematik meist jene Zahlen, die wahre Aussagen ergeben. Trotzdem bleibt festzustellen, dass man für x prinzipiell jede Zahl einsetzen kann, die man möchte. Der Platzhalter bleibt variabel.
Beim Programmieren muss man Gottlob nicht nur solche starren Gleichungen wie die obige lösen, sondern kann Variablen gewissermaßen mit echtem Leben füllen, indem man sie ständig variiert. Der Computer reserviert dabei für jede Variable einen bestimmten begrenzten Platz, den man mit etwas füllen kann, was man später wieder herausholt. Dies ist sehr praktisch, wenn man Dinge kontrollieren oder beobachten möchten, die sich über einen gewissen Zeitraum verändern.

1.2 Variablen verändern

Nehmen wir beispielsweise einmal an, du wolltest deine gesparten Euros von derzeit 85 € in einer Variable speichern. Dann notierst du einfach `x = 85` und der Computer reserviert dafür einen kleinen Bereich, dessen Inhalt du nun nach Belieben verändern kannst.
Würde dir deine Oma beim nächsten Geburtstag 50 € schenken, dann aktualisiert du dein Guthaben einfach auf `x = 135`.
Wie in der Einführung erwähnt, kannst du jederzeit überprüfen, was gerade in deiner Variablen drinsteckt, zum Beispiel mit dem Befehl `Print`, der dir beim Ausführen deines Codes das Gewünschte in das weiße Konsolenfenster der BlitzMax-IDE ausgibt.
`Print x` wird dir die Zahl 135 ausgeben, also den aktuellen Wert, nicht die alten 85 €.
Zugegeben - besonders überraschend ist das nicht. Schließlich haben wir ja vorher selbst den neuen Wert ausgerechnet und das Ergebnis 135 direkt der Variablen übergeben. Angenehmer wäre es, wenn uns der Computer die Rechenarbeit abnehmen würde.
Erhieltest du also etwa von einer anderen Oma noch 77 € dazu, dann können wir diesen Fortschritt so notieren: `x = x + 77`. Der Computer rechnet nun zu den alten 135 € die dazukommenden 77 € dazu und die Ausgabe durch `Print x` ergibt 212 €, ohne dass wir selbst etwas rechnen mussten, ähnlich wie bei einem Taschenrechner.
Beachte dabei jedoch unbedingt, dass `x = x + 77` keine mathematische Gleichung darstellt!
Es ist eine Zuweisung, bei der die Reihenfolge wichtig ist: Das x, das rechts vom Gleichheitszeichen steht, ist gewissermaßen das alte x mit dem vorherigen Wert 135. Dagegen steht links vom Gleichheitszeichen das neue x mit dem aktualisierten Wert 212.
Andere Programmiersprachen unterscheiden deshalb zwischen der Schreibweise für eine Gleichung und der für eine Wert-Zuweisung. Basic-verwandte Sprachen wie BlitzMax machen es sich dagegen leicht und verwenden für Beides ein einfaches Gleichheitszeichen `=`.

Natürlich können auch alle übrigen Rechenoperationen auf analoge Weise notiert werden, also z.B. das Subtrahieren. Manchmal willst du dir ja auch etwas von deinem Geld leisten, z.B. ein paar spannende Bücher für insgesamt 64 €. Nachdem du also `x = x - 64` eingibst, wirst du mit einem anschließenden `Print x` statt der 212 € nur noch 148 € erhalten.

Will man sich nun nicht nur ab und zu etwas schenken lassen, sondern seine Ersparnisse durch regelmäßige Arbeit und festes Gehalt anwachsen lassen, so bietet sich an, für dieses Gehalt eine neue Variable einzuführen, denn schließlich kann sich ja auch die Höhe dieses Gehalts ändern. Nehmen wir also einmal an, du verdienst zunächst monatlich 40 € dazu: `y = 40`. Das monatliche Anwachsen deines Vermögens ist dann durch `x = x + y` umsetzbar.
Die Kontrolle mit `Print x` ergibt 188 €. Es können also auch mehrere Variablen bzw. ihre Werte miteinander verrechnet werden. Versuche doch einmal, die nun folgenden Codezeilen zu interpretieren und auf die letzte Ausgabe zu schließen:

```
x = x + y
x = x + y
y = y + 10
x = x + y
Print x
```

Ganz klar: Nach dem du noch einmal zwei Monatsgehälter dazu verdient hast, ist dein Gehalt um 10 € auf 50 € gestiegen und dieses neue Monatsgehalt ist noch einmal dazugekommen, dein aktuelles Vermögen beträgt nun 318 €.

1.3 Variablen benennen

Wir haben eben bereits zwei Variablen benutzt. In den meisten Programmen werden jedoch noch viel mehr Variablen gebraucht. Wenn man diese nun x, y, z, t, v, w usw. nennen würde, dann kannst du dir denken, dass man da bald den Überblick verlieren würde, welche Variable was bedeutet. Einen guten Überblick zu bewahren ist aber sehr wichtig.
Glücklicher Weise dürfen wir beim Programmieren fantasievoll sein und den Variablen längere Namen geben, die helfen, sie richtig zu deuten.
Lasst uns also für x einen besseren Namen erfinden, etwa `MeineEuros = 318` und für das Monatseinkommen `MeinGehalt = 50`. Der Zuwachs der Ersparnisse ließe sich dann pro Monat zweifellos viel einleuchtender ausdrücken durch `MeineEuros = MeineEuros + MeinGehalt`.
Wichtig ist, dass der Variablenname zusammenhängend ist: Eine Trennung zu *Meine Euros* ist nicht erlaubt, ansonsten deutet dies der Compiler als zwei verschiedene Variablen namens *Meine* und *Euros* und bemängelt deren fehlende Verknüpfung.
Außerdem ist auf drei weitere Dinge zu achten:

1.) Der Variablenname darf keinerlei Sonderzeichen enthalten, insbesondere nicht die deutschen Umlaute und das ß. Daher wäre der Name `MeinVermögen = 318` nicht erlaubt, weil er den Umlaut ö enthält.
2.) Dem Variablennamen darf keine Zahl vorangestellt sein: `50Cent` wäre verboten. Innerhalb des Namens oder danach (`Take5`) ist eine Zahl jedoch erlaubt.
3.) Die Wahl des Namens darf nicht mit einem BlitzMax-Befehl kollidieren, z.B. darf man keine Variable *Print* nennen, weil dieser Name schon durch die Ausgabe `Print` belegt ist, wie man an der automatisch gelben Einfärbung erkennen kann.

Übrigens unterscheidet der Compiler von BlitzMax bei Variablennamen Großbuchstaben nicht von Kleinbuchstaben. Wenn du statt `MeineEuros` an anderer Stelle `meineeuros` oder gar `mEiNeeURos` schreiben würdest, würde damit stets die gleiche Variable identifiziert werden. Im Sinne der eigenen Orientierung sollte man aber stets die gleiche Schreibweise beibehalten.
Einige Programmierer bevorzugen englische Variablenbezeichnungen, weil diese sich gut in die übliche englische Sprache der Befehle integrieren und Nichtdeutschen besser verständlich ist.

1.4 Übungsaufgabe

Angenommen, dein Lieblingssong dauert 237 Sekunden und du hörst ihn dir fünfmal hintereinander an. Sofort danach hörst du dir dreimal einen anderen Song an, der 192 Sekunden dauert.
Schreibe einen Code, der diese Situation mit Hilfe von Variablen darstellt und dir am Ende die Gesamtdauer ausgibt, also die Anzahl an Sekunden, die du insgesamt Musik gehört hast.
Natürlich könntest du das auch direkt mit dem Taschenrechner oder einer einzigen Zeile ausrechnen, doch es geht darum, den ersten Umgang mit Variablen zu üben.

Kapitel 2

Fallunterscheidungen

2.1 Einführung

Computerprogramme sollen vielseitig und flexibel sein. Wenn immer nur exakt das Gleiche ablaufen würde, wären sie langweilig und ziemlich sinnlos.
Wenn man also selbst ein Programm entwickelt, sollte man davon ausgehen, dass Verschiedenes passieren kann: Situationen ändern sich, der Benutzer macht verschiedene Eingaben, der Spieler reagiert 'mal so oder so, nicht zuletzt spielt auch der Zufall eine Rolle.
Als Programmierer kommt man daher nicht umhin, sich auf mehrere Fälle vorzubereiten, die eintreten könnten.
Nehmen wir einmal an, du möchtest den Benutzer fragen, ob es ihm gut geht, dann könntest du schreiben `Proceed("Geht es dir gut?")`. Der Befehl `Proceed()` öffnet nämlich eine kleines Dialogfenster, in dem deine innerhalb zweier Anführungszeichen gesetzte Frage erscheint, und die der Benutzer durch entsprechende Schalter bejahen oder verneinen kann.
Nun soll dein Programm natürlich auf die Antwort reagieren können, z.B. bei der Antwort Nein dein Bedauern ausdrücken durch `Notify("Das ist aber schade.")`. Der Befehl `Notify()` öffnet ähnlich Proceed ein kleines Dialogfenster, allerdings ohne Ja- und Nein-Schalter. Vergiss auch hier nicht, deinen Antworttext wie ein Zitat in Anführungsstriche zu setzen.
Du siehst, du musst mehrere Fälle unterscheiden: ***Wenn*** er das tut, ***dann*** reagiere ich so, und ***wenn*** er etwas anderes tut, ***dann*** reagiere ich anders.

> Randbemerkung für Linux-Nutzer: Damit die obigen Dialogfenster tatsächlich erscheinen, muss unter Linux möglicher Weise noch folgende Zeile ganz zu Beginn des Codes hinzugefügt werden: `Import maxgui.drivers`

2.2 If ... Then

Die englische Übersetzung `If ... Then` von ***wenn ... dann*** bildet eine der möglichen Fallunterscheidungen von BlitzMax.
Woher sollen wir aber wissen, welchen Schalter der Benutzer gedrückt hat? Ganz einfach: Der Befehl `Proceed()` gibt als Antwort eine 1 für "Ja" zurück, eine 0 für "Nein" und eine -1 für "Abbrechen". Lasst uns also diese Antwort in einer Variablen speichern:

```
Antwort = Proceed("Geht es dir gut?")
```

Wenn nun auf den Schalter "Ja" gedrückt wurde, wird die Variable `Antwort = 1` gesetzt.
Wir können also abfragen, ob die Antwort = 1 ist, dann soll man dies gutheißen, also etwa

```
If Antwort = 1 Then Notify("Prima, mir auch!")
```

Andernfalls, bei Antwort = 0, sollte eine andere Reaktion erfolgen:

```
If Antwort = 0 Then Notify("Das ist aber schade!")
```

Voila! Dein Programm kommuniziert bereits mit dem Benutzer, wenngleich auch ziemlich bescheiden. Das Abbrechen, also die Anwort = -1, wurde z.B. noch nicht behandelt.
Unterscheidet man mehrere Fälle oder folgen daraus umfangreichere Konsequenzen, so sollte man die Fallunterscheidung in einem einheitlichen Block zusammenfassen, wobei das "andernfalls" mit dem englischen `ElseIf` übersetzt wird:

```
Antwort = Proceed("Geht es dir gut?")   '<=  hier wird die Antwort-Variable gefüllt
If Antwort = 1 Then                     '<=  falls "Ja" gedrückt wird, dann ...
        Notify("Prima, mir auch!")      '<=  folgt in einer separaten Zeile die Konsequenz
ElseIf Antwort = 0 Then                 '<=  andernfalls "Nein" gedrückt wird, dann ...
        Notify("Das ist aber schade!")  '<=  folgt eine andere Konsequenz
ElseIf Antwort = -1 Then                '<=  andernfalls "Abbrechen" gedrückt wird, dann
        Notify("Und Tschüss!")          '<=  folgt die dritte und letzte Konsequenz.
End If                                  '<=  Mit diesem abschließenden End If wird die
                                        '    Fallunterscheidung beendet.
```

Beachte bitte, dass man zwecks Leserlichkeit und Übersichtlichkeit des Codes immer auf eine klare Struktur achten sollte. So kannst du beispielsweise oben erkennen, dass jene drei Zeilen, welche die reagierenden Konsequenzen beinhalten, eingerückt sind, da sie nur zum Tragen kommen, wenn der jeweilige Fall eintritt. Dadurch kann man schnell und auf einen Blick die jeweiligen Fälle von den Folgerungen unterscheiden. Zum Einrücken verwendet man am Besten die Tabulatortaste.
Will man wenige spezielle Fälle von allen übrigen möglichen Fällen unterscheiden, verwendet man zuletzt ein einfaches Else, das man mit "***ansonsten*** ..." übersetzen könnte. Das folgende verkürzte Beispiel fasst die Fälle "Nein" und "Abbrechen" zusammen:

```
Antwort = Proceed("Geht es dir gut?")
If Antwort = 1 Then                    '<=   spezieller Fall "Ja"
        Notify("Prima, mir auch!")
Else                                   '<=   ansonsten alle übrigen Fälle
        Notify("Schade!")                    also Antwort = 0  oder  Antwort = -1
End If
```

Vergiss am Ende nicht das End If, weil der Compiler sonst denkt, du würdest noch weitere Fälle unterscheiden wollen. Dann wird er sich darüber beschweren, dass ein *End If* fehlt.

2.3 Select ... Case

Es gibt noch eine andere Möglichkeit der Fallunterscheidung von BlitzMax. Der Vollständigkeit halber sei sie hier ergänzt, die Unterschiede sind gering: Sie wirkt etwas kürzer, ist aber auch weniger vielseitig als die Fallunterscheidung *If ... Then*, weil sie sich ausschließlich auf eine einzige Variable bezieht:

```
Antwort = Proceed("Geht es dir gut?")
Select Antwort                         '<= Fallunterscheidung auf eine Variable beschränkt
Case 1                                 '<= entspricht dem If ... Then
        Notify("Prima, mir auch!")
Case 0                                 '<= entspricht dem ElseIf ... Then
        Notify("Das ist aber schade!")
Default                                '<= entspricht dem Else
        Notify("Und Tschüss!")
End Select                             '<= entspricht dem End If
```

2.4 Übungsaufgabe

Schreibe einen Code, in dem du den Benutzer fragst, ob er mit dir ausgehen möchte. Falls ja, schlägst du ihm einen Kinobesuch vor. Wenn er das nicht möchte, geht ihr gemeinsam essen.

Kapitel 3

Schleifen

3.1 Einführung

Kehren wir noch einmal zu deinem Vermögen `MeineEuros = 318` und deinem monatlichen Verdienst `MeinGehalt = 50` aus Kapitel 1 zurück. Angenommen, dein Monatsgehalt würde sich ab sofort jeden Monat um 2 € erhöhen. Wir wollen nun einen Code schreiben, der uns ausrechnet, nach wieviel Monaten du der Besitzer von 1000 € bist.
Folgende Schritte würde die monatliche Zunahme deines Geldes darstellen:

```
MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
```

Diesen Schritt müssten wir jetzt stetig wiederholen ...

```
MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
```

usw...

Aber das reicht ja noch nicht. Zusätzlich müssten wir nach jedem Monat prüfen, ob die Marke von 1000 € bereits erreicht wurde (die Konsolenausgabe `Print` wird uns dann benachrichtigen) und das Programm dann mit dem einfachen `End` -Befehl beenden.

```
MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
If MeineEuros = 1000 Then
        Print "Geschafft!"
        End
End If

MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
If MeineEuros = 1000 Then
        Print "Geschafft!"
        End
End If

MeinGehalt = MeinGehalt + 2
MeineEuros = MeineEuros + MeinGehalt
If MeineEuros = 1000 Then
        Print "Geschafft!"
        End
End If
```

usw...

Du wirst zugeben, dass dieser Schreibaufwand nicht zufriedenstellend ist. Darüber hinaus wissen wir ja noch nicht einmal, wie häufig wir diese Schritte wiederholen müssen, um endlich zum Ziel zu kommen. Da diese und ähnliche Situationen jedoch beim Programmieren oft vorkommen, gibt es in jeder Programmiersprache ein paar Befehle, die ein (beliebig) häufiges Wiederholen von Schritten automatisieren. Man nennt sie "Schleifen":

3.2 Repeat ... Until ...

Die englischen Bezeichnungen `Repeat` und `Until` sprechen für sich:
Wiederhole etwas solange ... **bis** dieser oder jener Fall eintritt. Du merkst: Das Ende dieser Schleife ist im Grunde auch eine Fallunterscheidung mit eben jener Konsequenz, die ständige Wiederholung abzubrechen. In unserem Beispiel würde es ausreichen, den monatlichen Geldzuwachs als eingerückten Block ein einziges Mal in eine solche Repeat-Schleife zu notieren und als Abbruchbedingung das Erreichen der gewünschten 1000 € anzugeben:

```
MeineEuros = 318
MeinGehalt = 50
Repeat
        MeinGehalt = MeinGehalt + 2
        MeineEuros = MeineEuros + MeinGehalt
Until MeineEuros = 1000
Print "Geschafft!"
```

Du merkst: Schleifen sind ziemlich praktisch und sparen Zeit und Schreibarbeit.
Doch Halt! Etwas haben wir vergessen: Wir wollten doch wissen, wie viel Monate verstreichen müssen. Für diese Aufgabe könnten wir eine dritte Variable benutzen, z.B. `Monate = 0` und sie jeden Schleifendurchgang mitzählen lassen: `Monate = Monate + 1`. Wir ergänzen also zu

```
MeineEuros = 318
MeinGehalt = 50
Monate = 0
Repeat
        MeinGehalt = MeinGehalt + 2
        MeineEuros = MeineEuros + MeinGehalt
        Monate = Monate + 1
Until MeineEuros = 1000
Print "Geschafft nach so viel Monaten:"
Print Monate
```

und erhalten die notwendige Anzahl der Monate als zusätzliche Nachrichtenzeile in der Konsole.

3.3 Achtung: Endlosschleifen!

Es soll nicht versäumt werden, neben den großen Vorteilen von Schleifen auch gleich auf ihre Tücken einzugehen. Denn wenn man nicht aufpasst, passieren unerwünschte Dinge, die nicht mehr aufzuhalten sind: Gemeint sind "Endlosschleifen", mit der man solche Schleifen bezeichnet, deren Abbruchbedingung nie eintritt und die darum nie aufhören, sich zu wiederholen. Das Ärgerliche ist nämlich, dass in diesem unangenehmen Fall das Programm mitunter auf keinerlei Eingaben mehr reagiert und mit normalen "sanften" Mitteln nicht zu stoppen ist.
Die Ursachen für Endlosschleifen sind vielfältig und stecken im Detail - Beispiel gefällig?
Vielleicht hast du es noch gar nicht wahrgenommen, aber ausgerechnet unsere eben erstellte Repeat-Until-Schleife ist nur mit sehr viel Glück einer Endlosschleife entkommen.
Hätte nämlich dein Vermögen nicht `MeineEuros = 318` betragen, sondern `MeineEuros = 317` oder `MeineEuros = 319`, dann wäre es schon passiert, denn das Problematische der Schleife ist ein typischer Anfängerfehler: Wir haben die Abbruchbedingung `Until MeineEuros = 1000` zu eng eingeschränkt bzw. auf eine einzige Zahl bezogen. Wenn aber die anwachsende Variable `MeineEuros` diese eine Zahl 1000 zufällig gar nicht exakt trifft, sondern knapp daran vorbeiläuft, wird die Schleife logischer Weise nie abgebrochen.
Man sollte sich daher von Anfang an angewöhnen, die Abbruchbedingung statt auf eine einzige Zahl auf einen größeren Zahlenbereich zu erweitern. In unserem Fall sollte die Schleife auch dann aufhören, wenn du geringfügig mehr als 1000 € verdient hast.
Soll bei einer Fallunterscheidung egal sein, ob der eine oder der andere Fall eintritt, so kann man diese beiden Fälle durch das englische Wort für *oder* = `Or` verknüpfen:
`Until MeineEuros = 1000 Or MeineEuros > 1000`

Damit hätten wir unsere vorige Schleife gegen das Risiko einer Endlosschleife abgesichert. Lasst uns nun zu Demonstrationszwecken dein Startkapital auf `MeineEuros = 319` ändern und mit der regelmäßigen Konsolenausgabe `Print MeineEuros` Monat für Monat verfolgen, wie dein Vermögen anwächst und wie es haarscharf an den 1000 € vorbeischrammt:

```
MeineEuros = 319
MeinGehalt = 50
Monate = 0
Repeat
        MeinGehalt = MeinGehalt + 2
        MeineEuros = MeineEuros + MeinGehalt
        Print MeineEuros
        Monate = Monate + 1
Until MeineEuros = 1000 Or MeineEuros > 1000
Print "Geschafft nach so viel Monaten:"
Print Monate
```

Übrigens dürften wir `=` und `>` zu einem `>=` zusammenfassen und bräuchten dann keine `Or`-Verknüpfung zweier Fälle. Die Abbruchbedingung `Until MeineEuros >= 1000` würde das Gleiche machen.

3.4 While ... Wend

Als Alternative zur Repeat-Until-Schleife gibt es die While-Wend-Schleife. Sie ist gewissermaßen umgekehrt konzipiert: Statt "***Wiederhole*** *etwas,* ***bis*** *dieser oder jener Fall eintritt*" gilt für die While-Wend-Schleife das Motto: "***Solange*** *dies oder jenes gilt,* ***wiederhole*** *etwas*". Das klingt ganz ähnlich, hat aber doch einen entscheidenden Unterschied in der Reihenfolge: Die Repeat-Until-Schleife startet sofort die Wiederholung und wartet dann periodisch auf einen Abbruch, die While-Wend-Schleife dagegen prüft erst einmal, ob sie die Schleife überhaupt starten soll. Diese geprüfte Gültigkeit `While MeineEuros < 1000` zu Beginn der Schleife muss daher das genaue Gegenteil sein von `Until MeineEuros >= 1000` am Ende der Repeat-Until-Schleife. Unser obiges Beispiel sähe als While-Wend-Schleife so aus:

```
MeineEuros = 319
MeinGehalt = 50
Monate = 0
While MeineEuros < 1000
        MeinGehalt = MeinGehalt + 2
        MeineEuros = MeineEuros + MeinGehalt
        Print MeineEuros
        Monate = Monate + 1
Wend
Print "Geschafft nach so viel Monaten:"
Print Monate
```

While-Wend-Schleifen bieten sich immer dann an, wenn man sich noch nicht sicher ist, ob die Schleife überhaupt ein einziges Mal durchlaufen werden soll: Hättest du etwa zu Anfang unseres Beispiels statt `MeineEuros = 319` bereits `MeineEuros = 1000` besessen, dann wäre die vorherige Repeat-Schleife trotzdem gestartet und hätte dich darüber informiert, dass du noch einen Monat warten müsstest, bis du 1000 € besitzt - offensichtlich eine Fehlinformation, denn du hättest ja bereits jetzt 1000 € besessen. Mit der While-Wend-Schleife wäre dieser Fehler nicht passiert.

3.5 For ... Next

Die **For-Next**-Schleife ist die dritte Schleifenvariante, die sich von den anderen Beiden etwas absetzt: Bei ihr wird von vorneherein eine "Laufvariable" benötigt, im Folgenden kurz `i` genannt, die in Einer-Schritten bis zu einer bestimmten, vorher festgelegten Obergrenze automatisch weiter zählt, hier ein Beispiel, das die natürlichen Zahlen von 1 bis 10 ausgibt:

```
For i = 1 To 10
      Print i
Next
```

Beachte bitte, dass du die Laufvariable nicht selbst weiter zählen musst (wie bei den anderen Schleifen) und auch gar nicht weiter zählen darfst. Dies wird nämlich durch den Befehl `Next` automatisch sichergestellt. Würdest du trotzdem ein `i = i + 1` einflechten, würde das anschließende `Next` ein zusätzliches Mal weiter zählen, also quasi eine Zahl überspringen:

```
For i = 1 To 10
      Print i
       i = i + 1
Next
```

... listet nur die ungeraden Zahlen auf. Grundsätzlich sollte man es bei **For-Next**-Schleifen eher vermeiden, die Laufvariable zusätzlich zu manipulieren, weil dies eine weitere Gefahr für unbeabsichtigte Endlosschleifen bietet.
Was ist aber, wenn man tatsächlich einmal nur die ungeraden Zahlen auflisten möchte, also doch ein Fortschreiten in Zweierschritten benötigt? Für diesen Zweck bietet die For-Next-Schleife die zusätzliche Option `Step` am Ende der ersten Schleifenzeile an, mit der man die Schrittweite wie gewünscht verändern kann:

```
For i = 1 To 10 Step 2
      Print i
Next
```

Jetzt wird die Laufvariable i automatisch in Zweierschritten weiter gezählt.
Dabei macht es im Übrigen nichts aus, dass die Obergrenze 10 nicht direkt getroffen wird, die **For-Next**-Schleife bricht die Schleife ab, sobald die Laufvariable gleich <u>oder</u> größer gleich der Obergrenze ist, also bei `i >= 10`
Fügt man hinter der Schleife noch die zusätzliche Ausgabe `Print i` an,

```
For i = 1 To 10 Step 2
      Print i
Next
Print i
```

so offenbart sich, dass die Laufvariable i <u>nach</u> der Schleife tatsächlich den Wert 11 besitzt.

Es gibt manchmal Situationen, in denen man möchte, dass die **For-Next**-Schleife schon einen Schritt **<u>vor</u>** der Obergrenze aufhört. Dann ersetzt man einfach das `To` durch ein `Until`. Die folgende Schleife zählt daher nur von 1 bis 9, obwohl als Obergrenze 10 festgelegt wurde:

```
For i = 1 Until 10
      Print i
Next
```

Der Beginn und die Obergrenze der Schleife kann variabel festgelegt werden. Das folgende Beispiel listet die Zahlen der Dreierreihe auf:

```
Beginn = 3
Obergrenze = 30
For i = Beginn To Obergrenze Step 3
        Print i
Next
```

Lässt sich bei der **Repeat-Until**- und der **While-Wend**-Schleife auch die Schrittweite variabel handhaben, so ist dies bei der **For-Next**-Schleife nicht der Fall! Hinter der Option `Step` muss immer ein konstanter Wert stehen, eine Variable akzeptiert der Compiler nicht, wie folgendes Gegenbeispiel bei der Ausführung zeigen wird:

```
Schritt = 2
For i = 1 To 10 Step Schritt
        Print i
Next
```

... funktioniert nicht.

3.6 Exit

Alle drei Schleifenvarianten haben eine Art "Notausgang" gemein, mit dem man die Schleife zu jeder Zeit sofort verlassen kann, der Befehl lautet `Exit`. Die folgende Schleife wird deswegen trotz der Obergrenze 10 bereits bei 7 verlassen:

```
For i = 1 To 10
        Print i
        If i = 7 Then Exit
Next
```

Die Existenz des `Exit`-Befehls erlaubt es sogar, bewusst scheinbare Endlosschleifen zu verwenden. Beendet man die Repeat-Schleife nämlich nicht mit `Until` und einer Abbruchbedingung, sondern mit `Forever`, so würde sie "für immer" wiederholt werden und kein Ende finden. Einzig der Befehl `Exit` verschafft dann ein Ausbrechen aus der Nonstop-Schleife:

```
i = 1
Repeat
        Print i
        If i = 7 Then Exit
        i = i + 1
Forever
```

3.7 Übungsaufgabe

Schreibe ein Programm, dass alle Zahlen des kleinen Einmaleins ausgibt. Als Multiplikationszeichen wird übrigens das Sternchen * benutzt.

Kapitel 4

Grafik Teil 1

4.1 Einführung

Bisher waren wir, was die Grafik anging, ja noch recht bescheiden. Außer einer öden Konsolenausgabe und ein paar Mini-Standart-Dialogen haben wir noch nicht einmal ein eigenes Fenster erstellt. Dies soll sich nun mit dem Befehl `Graphics` ändern, allerdings möchte dieser Befehl aus naheliegenden Gründen noch wissen, wie groß das gewünschte Fenster bzw. welche Breite und Höhe es haben soll. Wie du sicherlich weißt, werden Bildschirminhalte in "Pixeln" gemessen. Pixel sind ziemlich kleine Punkte, daher brauchen wir schon eine ganze Menge von ihnen. Lasst uns also ein Fenster erstellen, das 400 Pixel breit und 300 Pixel hoch ist und notieren dies, durch ein Komma getrennt, hinter den Graphics-Befehl: `Graphics 400, 300`
Wenn wir dies nun ausführen, wird das Ergebnis erst einmal enttäuschen: Vielleicht sehen wir irgend etwas aufblitzen, aber dann ist es schon wieder verschwunden. Damit das Programm nach getaner Arbeit nicht sofort wieder aufhört, müssen wir es quasi "anhalten", um das Fenster wenigstens in Ruhe betrachten zu können. Dieses Anhalten könnte man durch den Befehl `WaitKey` erreichen, der nichts anderes macht, als auf (irgendeine) Taste deiner Computertastatur zu warten, bis es weiterläuft (bzw. aufhört). Nebenbei könnten wir mit der Anweisung `AppTitle = "Mein erstes Fenster"` (vor der Erstellung des Fensters) der Titelleiste eine Überschrift verleihen und blicken beim Ausführen voller Stolz auf das Ergebnis:

```
AppTitle = "Mein erstes Fenster"
Graphics 400, 300
WaitKey
```

4.2 Zeichnen

Das kleine Fenster liegt nun wie ein Zeichenblatt vor uns. Wir können nun den Computer anweisen, etwas darauf zu zeichnen. Wie du gleich merken wirst, malt er dabei "Weiss auf Schwarz" statt "Schwarz auf Weiss", beispielsweise mit dem Befehl `Plot`, der ein einziges Pixel malt. Lasst uns das Pixel genau in der Mitte des Fensters weiß malen. Dazu notieren wir die halbierte Breite und die halbierte Höhe des Fensters als waagerechte und senkrechte Koordinate des Punktes: `Plot 200, 150`
Stell' dir einmal vor, der Computer sitzt wie eine Person dir gegenüber und malt etwas auf "sein Zeichenblatt" (das man übrigens "Buffer" nennt). Damit nicht nur er, sondern auch du sehen kannst, was er gemalt hat, muss er sein Zeichenblatt noch umdrehen, damit es dir zugewandt ist. So etwas Ähnliches wie dieses Umdrehen muss auch tatsächlich passieren (ansonsten siehst du nichts) und zwar mit dem anschließenden Befehl `Flip`:

```
Graphics 400, 300
Plot 200, 150
Flip
WaitKey
```

So ein einziges Pixel ist ja nun wirklich sehr klein! Wir wollen mehr sehen und tauschen ihn aus durch jenen Befehl, der ein Rechteck zeichnet: `DrawRect`. Ein Rechteck braucht nicht nur zwei Koordinaten für seine Position, sondern noch zwei weitere Angaben für die Breite und die Höhe des Rechtecks. Mit `DrawRect 200, 150, 80, 60` zeichnet der Computer ein 80 Pixel breites und 60 Pixel hohes Rechteck. Nun lass es laufen und schau dir das Ergebnis einmal ganz genau an:

```
Graphics 400, 300
DrawRect 200, 150, 80, 60
Flip
WaitKey
```

Es sollte dir auffallen, dass die Lage des Rechtsecks nicht dem entspricht, was du vielleicht erwartet hattest: Wollten wir nicht ein Rechteck in die Mitte des Fensters zeichnen? Danach sieht es aber gar nicht aus: Dort, wo vorhin das einzelne weiße Pixel war, liegt jetzt die linke, obere Ecke des Rechtecks. Tatsächlich musst du dich daran gewöhnen, dass standartmäßig alle Zeichenbefehle - nicht nur der vom Rechteck - stets die linke, obere Ecke als Verankerungsposition verwenden. Mehr noch: Das Koordinatensystem des gesamten Fensters ist darauf ausgerichtet, dass der "Ursprung", also der Punkt mit den Koordinaten `0, 0`, links oben liegt und nicht links unten wie in der deutschen Schulmathematik. Um dich selbst zu überzeugen, kannst du `DrawRect 0, 0, 80, 60` oder andere Positionskoordinaten ausprobieren.
Aufgrund dieser anderen Orientierung von Links-Oben aus kommt es der Vorstellung beim Programmieren entgegen, hinsichtlich der Maße eines Zeichenobjektes eher von der "Tiefe" als von der "Höhe" zu reden.
Kehren wir zu unserem Wunsch zurück, das Rechteck nun doch genau in die Mitte zu platzieren. Wir müssen eben seine linke obere Ecke aus der Mitte herausbewegen, und zwar um die Hälfte seiner Breite nach links (also 200 - 40) und um die Hälfte seiner Höhe nach oben (also 150 - 30), damit die Mitte des Rechtecks mit der Mitte des Fenster übereinstimmt:

```
Graphics 400, 300
DrawRect 160, 120, 80, 60
Flip
WaitKey
```

Statt eines Rechtsecks kann man mit `DrawOval 160, 120, 80, 60` auch ein Oval bzw. eine Ellipse zeichnen. Soll die Ellipse ein Kreis sein (oder das Rechteck ein Quadrat), müssen Breite und Höhe identisch sein.
Zuletzt sei noch der Zeichenbefehl für Strecken, also "geraden Linien", genannt: `DrawLine`. Zur Verankerung benötigt werden hierbei zwei Punkte, bzw. deren Koordinaten. Im folgenden Beispiel werden die gegenüberliegenden Ecken des Fensters durch zwei Strecken miteinander verbunden:

```
Graphics 400, 300
DrawLine 0, 0, 400, 300
DrawLine 400, 0, 0, 300
Flip
WaitKey
```

Wenn man will, kann man die Linien mit dem vorausgehenden Befehl `SetLineWidth` und einer Zahl größer 1 beliebig dicker machen:

```
Graphics 400, 300
SetLineWidth 3
DrawLine 0, 0, 400, 300
DrawLine 400, 0, 0, 300
Flip
WaitKey
```

4.3 Übungsaufgabe

Schreibe ein Programm, das ein Gesicht genau in die Mitte deines Fensters zeichnet: Male dabei die beiden Augen und den Mund mit Ellipsen, die Nase mit einem ausgefüllten Rechteck und die Augenbrauen mit zwei Linien. Die Stimmungslage des Gesichtes sollte zornig sein.

Kapitel 5

Zufall Teil 1

5.1 Einführung

Nicht selten soll ein Computerprogramm einen Gegner, einen Partner oder ganz allgemein ein Gegenüber simulieren, der einen freien Willen zu haben scheint und 'mal so oder so entscheidet. Da dies ja offensichtlich nicht so ist, müssen wir das unkalkulierbare Verhalten vortäuschen, und zwar mit dem Befehl `Rand(7)` können wir beispielsweise den Computer anweisen, eine zufällige natürliche Zahl mit der Obergrenze 7 auszugeben. Lasst uns das ein paar mal tun:

```
For i = 1 To 10
	Print Rand(7)
Next
```

So weit, so gut. Wenn du diesen Code allerdings mehrmals hintereinander ausführst, wirst du merken, dass er immer wieder dieselbe Zahlenfolge ausspuckt. So zufällig scheinen die Zahlen also doch nicht zu sein. Möchte man diesen Wiederholungseffekt vermeiden, muss man den zufällig ermittelten Zahlen noch den Befehl `SeedRnd(MilliSecs())` voranstellen, der erst so richtig den Zufallsgenerator des Computers aktiviert.
Einige Tests sollten dir zeigen, dass sich die Zahlenfolge nun nicht wiederholt:

```
SeedRnd(MilliSecs())
For i = 1 To 10
	Print Rand(7)
Next
```

Man kann optional auch eine Untergrenze festlegen, die nicht unterschritten wird und schreibt sie einfach also Argument links vor die Obergrenze, durch ein Komma getrennt: `Rand(4, 9)`
Das folgende Beispiel gibt Zufallszahlen zwischen 4 und 9 aus:

```
SeedRnd(MilliSecs())
For i = 1 To 10
	Print Rand(4, 9)
Next
```

Die Untergrenze muss nicht größer Null sein: `Rand(-5, 8)` liefert Zahlen zwischen -5 und 8.
Manchmal möchte man gar keine ganzen Zahlen haben, sondern Dezimalbrüche, also Zahlen mit einem Komma (das beim Programmieren allerdings als Punkt erscheint bzw. notiert werden muss). Für diesen Zweck braucht man `Rand` nur geringfügig in `Rnd` zu ändern und erhält im Folgenden eben Kommazahlen zwischen -2.5 und 3.5:

```
SeedRnd(MilliSecs())
For i = 1 To 10
	Print Rnd(-2.5, 3.5)
Next
```

5.2 Übungsaufgabe

Schreibe einen Code, der simuliert, dass ein Würfel 10.000 Mal geworfen wird. Zähle dabei, wie häufig der Würfel eine gerade Augenzahl zeigt und gib die Summe in der Konsole aus.

Kapitel 6

Kommentare

Hier in dieser Anleitung wird jeder kleine Schritt und jeder Codeausschnitt beschrieben und erklärt. Solche Kommentare kann und sollte man auch direkt im eigenen Code unterbringen. Sie helfen einerseits anderen, den Code besser zu verstehen und geben andererseits einem selbst eine gute Erinnerungsstütze, falls man an einem größeren Projekt arbeitet, oder nach längerer Unterbrechung versucht, sich wieder im Code zu orientieren.
Allerdings muss man diese Kommentare besonders kenntlich machen, damit der Compiler sie nicht als Befehle missdeutet, sondern ignoriert. Dazu gibt es zwei Möglichkeiten:
Sobald in einer Zeile ein Apostroph ' erscheint (die hochgestellte Taste #), wird alles Nachfolgende als Kommentar gewertet:

```
Graphics 500, 500                'hier kommentiere ich, dass ich ein quadratisches Fenster will
DrawOval 200, 200, 100, 100      'und in die Mitte einen Kreis zeichnen möchte
```

Falls mein einen größeren separaten Block kommentierenden Text einfügen möchte, z.B. am Anfang, dann kann man diesen Block durch die Befehle `Rem` und `End Rem` einrahmen bzw. kennzeichnen. Alles darin Liegende wird der Compiler nicht beachten:

```
Graphics 400, 300
Rem
Dieses Programm soll etwas ungeheuer Aufregendes machen:
Es soll einen Punkt in die linke obere Ecke zaubern - ist das nicht toll?
Jetzt geht's gleich los - ich kann es kaum erwarten:
End Rem
Plot 0, 0
```

Nicht-Compiler-relevante Kommentare erkennt man im Code an der einheitlichen mattgrau-blauen Einfärbung (die man übrigens, wie alle anderen Einfärbungen auch, in den IDE-Options individuell einstellen kann).

Kapitel 7

Grafik Teil 2

7.1 Farben

Im Grafik-Teil 1 war alles, was wir auf das schwarze "Zeichenblatt" zeichnen ließen, weiß, was auf die Dauer ziemlich eintönig und langweilig ist. Selbstverständlich ist der Computer in der Lage, auch farbige Objekte zu zeichnen. Der Befehl dafür lautet `SetColor` und erwartet gleich drei Argumente bzw. drei Zahlen zwischen 0 und 255. Der erste ist der Rotanteil, der zweite der Grünanteil und der dritte der Blauanteil einer Farbe. Dahinter steckt das RGB-Farben-Verständnis: Mit den drei Grundfarben Rot, Grün und Blau kann man alle anderen Farben mischen. Türkis beispielsweise wird mit "vollem" Grün (255) und "vollem" Blau (255), aber ohne Rot (0) gemischt. Also würde der Befehl `SetColor 0, 255, 255` die Farbgebung auf Türkis setzen und alles danach Gezeichnete wäre türkis.
Natürlich sind auch alle Werte zwischen 0 und 255 möglich: Je kleiner der Wert, desto stärker wird der Farbton abgedunkelt. Dies wollen wir an folgendem Beispiel demonstrieren, der einen Farbverlauf von Schwarz bis Rot darstellen soll: Man braucht einfach nur 256 schmale Streifen, also etwa 1 Pixel breite Rechtecke, nebeneinander zu malen, wobei der Rotanteil jeweils um eins ansteigt. Selbstverständlich bietet sich dafür eine Schleife an:

```
Graphics 256, 256
For Farbanteil = 0 To 255
        SetColor Farbanteil , 0, 0
        DrawRect Farbanteil, 0, 1, 256
Next
Flip
WaitKey
```

Man könnte gleichzeitig einen anderen Farbton, z.B. Grün, in umgekehrter Richtung von 255 auf 0 absinken lassen `GegenFarbanteil = 255 - Farbanteil` und bekäme einen Farbverlauf von Grün nach Rot:

```
Graphics 256, 256
For Farbanteil = 0 To 255
        GegenFarbanteil = 255 - Farbanteil
        SetColor Farbanteil , GegenFarbanteil, 0
        DrawRect Farbanteil, 0, 1, 256
Next
Flip
WaitKey
```

Natürlich kannst du nun ganz nach Belieben experimentieren, welche Mischung welche Farbe ergibt. Als helfende Orientierung möge dir folgende kleine Tabelle dienen:

Rotanteil:	Grünanteil:	Blauanteil:	Farbe:
255	0	0	
255	127	0	
255	255	0	
0	255	0	
0	255	255	
0	0	255	
255	0	255	
0	0	0	
127	127	127	
255	255	255	(weiß)

7.2 Übungsaufgabe

Erstelle ein 512 Pixel breites Fenster mit einem doppelten Farbverlauf von Links nach Rechts, bei dem zuerst Blau nach Orange und ab der Fenstermitte Orange nach Violett wechselt.

7.3 Übungsaufgabe

Lies dir einmal den folgenden Code durch, stelle dir das grafische Ergebnis vor und errate, nach was dieses Fenster beim Ausführen wohl aussieht:

```
Graphics 120, 340
SetColor  255, 0, 0
DrawOval 10, 10, 100, 100
SetColor  255, 255, 0
DrawOval 10, 120, 100, 100
SetColor  0, 255, 0
DrawOval 10, 230, 100, 100
Flip
WaitKey
```

Verändere bzw. erweitere den Code so, dass die Ampel zuerst auf "Rot" steht, dann (nach irgendeinem Tastendruck deinerseits) auf "Gelb-Rot" umschlägt und zum Schluss "Grün" zeigt. Dunkle die gerade nicht leuchtenden Lampen entsprechend ab.

7.4 Bunte Zufallskompositionen

Lasst uns nun unser Wissen über Zeichenobjekte, Farbgebung und Zufallszahlen kombinieren: Wir wollen 1000 Kreise malen. Alle sollen zufällig innerhalb des Grafikfensters verteilt sein und einen zufälligen Durchmesser zwischen 10 und 30 Pixeln besitzen. Zusätzlich sollen sie Grün oder Rot sein.
Diesmal wollen wir uns ein etwas größeres Fenster leisten: `Graphics 800, 600`
Für den Kreisdurchmesser nehmen wir am Besten eine eigene Variable, die zufällig mit einer Zahl zwischen 10 und 30 gefüllt wird: `Durchmesser = Rand(10, 30)`
Nun soll der Kreis innerhalb des Fensters liegen, also nicht über den Rand hinaus. Da die Zeichenposition des Kreises bekanntlich Links-Oben liegt, dürfen die waagerechte und die senkrechte Position minimal bei 0 liegen. Maximal hängt sie jedoch vom Durchmesser ab: Man muss ihn von den jeweiligen Fenstermaßen subtrahieren: Die waagerechte Position darf bei maximal 800 - Durchmesser, die senkrechte bei maximal 600 - Durchmesser liegen. Dafür nehmen wir wieder zwei Variablen x und y und füllen sie entsprechend zufällig:

`x = Rand(0, 800 - Durchmesser)` und `y = Rand(0, 600 - Durchmesser)`

Unsere Kreise können dann mittels der definierten Variablen gezeichnet werden:

`DrawOval x, y, Durchmesser, Durchmesser`

Vorher wollen wir jedoch noch zufällig Rot oder Grün auswählen. Das könnten wir mit folgender simplen Fallunterscheidung machen

```
Zufallsfarbe = Rand(0,1)
If Zufallsfarbe = 0 Then
        SetColor 255, 0, 0    'Rot
Else
        SetColor 0, 255, 0    'Grün
End If
```

Nun lasst uns das Ganze noch in einer Schleife wiederholen, denn wir wollten ja 1000 solcher Kreise malen, und fertig ist unser Zufallsbild:

```
SeedRnd(MilliSecs())
Graphics 800, 600
For i = 1 To 1000
	Durchmesser = Rand(10, 30)
	x = Rand(0, 800 - Durchmesser)
	y = Rand(0, 600 - Durchmesser)
	Zufallsfarbe = Rand(0,1)
	If Zufallsfarbe = 0 Then
		SetColor 255, 0, 0    'Rot
	Else
		SetColor 0, 255, 0    'Grün
	End If
	DrawOval x, y, Durchmesser, Durchmesser
Next
Flip
WaitKey
```

Schon nicht ganz schlecht. Aber irgendwie sind die Farben noch zu eintönig. Vielleicht könnte man nicht nur reines Rot und reines Grün, sondern noch alle möglichen Zwischenwerte und Mischungen von Rot- und Grünanteilen verwenden. Dies lässt sich dadurch erreichen, dass man statt der Farb-Fallunterscheidung den Zufall direkt in die Farbanteile integriert:
`SetColor Rand(0, 255), Rand(0, 255), 0`
Als weiteren Vorschlag sei genannt, sich mehrere Versionen der Zufallskompositionen anzuschauen, d.h. unsere obigen 1000 Kreiszeichnungen in eine weitere Schleife zu packen, mit der man bei beliebigem Tastendruck jeweils noch einmal 1000 Kreise zeichnen lässt. Damit die neuen Kreise nicht die alten übermalen (bzw. die alten noch sichtbar sind), müssen wir vorher jedesmal das Fenster "sauberwischen", und zwar mit dem Befehl `Cls`, einer Abkürzung für "Clear Screen", der englischen Übersetzung von "Reinige den Bildschirm". Geschlossen wird das Programm bzw. die Nonstop-Schleife durch einen Klick auf den roten Stop-Knopf der BlitzMax-IDE:

```
SeedRnd(MilliSecs())
Graphics 800, 600
Repeat
	Cls
	For i = 1 To 1000
		Durchmesser = Rand(10, 30)
		x = Rand(0, 800 - Durchmesser)
		y = Rand(0, 600 - Durchmesser)
		SetColor Rand(0, 255), Rand(0, 255), 0
		DrawOval x, y, Durchmesser, Durchmesser
	Next
	Flip
	WaitKey
Forever
```

Eigentlich wischt `Cls` das Fenster gar nicht sauber, sondern übermalt es einfach komplett mit Schwarz. Man kann allerdings auch eine andere Farbe als Schwarz einstellen und zwar mit dem Befehl `SetClsColor` und entsprechenden Rot-, Grün- und Blauwerten.
Schreibt man beispielsweise im obigen Code `SetClsColor 0, 255, 255` direkt hinter den Graphics-befehl, so malt danach `Cls` den Hintergrund stetig Türkis.

7.5 Übungsaufgabe

Zeichne auf einem dunkelblauen Hintergrund ein feuriges "Strahlenbündel" von 1000 Linien, die alle von der Mitte des Fensters ausgehen und deren "Ende" ansonsten zufällig, aber innerhalb des Fensters liegt. Die Farbe der Linien soll Rot bis Gelb in verschiedenen (zufälligen) Helligkeitsstufen sein, grünliche Farben sollen ausdrücklich nicht vorkommen. Statt auf eine Tastatureingabe zu warten, sollen die "Strahlenbündel" ständig neu gezeichnet werden, was einen gewissen Flimmereffekt gibt. Beendet wird das Programm wie oben mit dem roten Stop-Schalter der BlitzMax-IDE.

7.6 Wörter zeichnen

Mit `Print` kennst du bereits die Möglichkeit, Text in der Konsole ausgeben zu lassen. Dies wird meist nur zu Kontrollzwecken beim Programmieren benutzt. Mit `DrawText` kannst du Wörter aber auch in dein Grafikfenster zeichnen lassen, wobei im Gegensatz zu Print viele Formatierungen möglich sind, allen voran die Position, die hinter dem Befehl `DrawText` und der anschließenden Textzeile als zweites und drittes Argument angegeben werden muss, ansonsten ist die Behandlung genauso wie bei den anderen Zeichenobjekten. Beachte wieder das Einrahmen der Textzeile durch zwei Anführungsstriche:

```
Graphics 800, 600
SetColor 255, 127, 0
DrawText "Link Oben", 0, 0
Flip
WaitKey
```

Du siehst: Damit werden die Wörter "Links Oben" orange in die linke obere Ecke gezeichnet. Im Gegensatz zu Zeichenobjekten wie den Rechtecken oder Ovalen, bei denen man ja die Breite und Tiefe mit angeben muss, weiß man dies bei einer variablen Textzeile noch nicht. Man kann sie sich jedoch durch folgende Befehle errechnen lassen: `TextWidth("Links Oben")` liefert die Breite und `TextHeight("Links Oben")` die Tiefe der darzustellenden Textzeile. Diese Informationen braucht man beispielsweise, um die Textzeile rechtsbündig zu zeichen, also etwa, um auch die Worte "Rechts Oben" waagerecht entsprechend zu positionieren, nämlich mit dem entsprechenden Abstand zum rechten Rand des 800 Pixel breiten Fensters:

```
Graphics 800, 600
SetColor 255, 127, 0
DrawText "Links Oben", 0, 0
DrawText "Rechts Oben", 800 - TextWidth("Rechts Oben"), 0
Flip
WaitKey
```

7.7 Übungsaufgabe

Erweitere obigen Code durch das Zeichnen der drei zusätzlichen Textzeilen "Links Unten" , "Rechts Unten" und "In der Mitte" an den entsprechend sinnvollen Stellen.

7.8 Animationen

Bisher blieb alles, was wir gezeichnet haben, starr auf dem Bildschirm kleben. Nun wollen wir den Dingen ein bisschen Leben einhauchen, indem sie sich bewegen, d.h. ihre Position verändern. Und immer wenn wir etwas Veränderliches kontrollieren, brauchen wir geeignete Variablen, nicht nur für die Position, also etwa `x` (waagerecht) und `y` (senkrecht), sondern auch für die Bewegungsschritte, also `xSchritt` (waagerechte Bewegung) und `ySchritt` (senkrechte Bewegung).
Lasst uns einen 100 Pixel großen Kreis in die Mitte des 800 x 600 großen Fensters platzieren: `x = 800/2 - 100/2` und `y = 600/2 - 100/2` und ihn mit ein paar zufälligen Pixeln irgendwohin bewegen. Damit er allerdings nicht nur nach Rechts und Unten wandert, brauchen wir auch die Möglichkeit negativer Schrittweiten, mit denen der Kreis nach Links oder nach Oben wandern könnte, also etwa `xSchritt = Rand(-4, 4)` und `ySchritt = Rand(-4, 4)`.
Damit sich überhaupt etwas tut, also die Position verändert und der Kreis jedesmal neu gezeichnet wird, müssen wir den Fortschritt `x = x + xSchritt` und `y = y + ySchritt` und das Zeichnen in eine unablässige Wiederholungs-Schleife packen. Damit die Bewegung selbstständig ohne unser Zutun verläuft, dürfen wir nicht mit `WaitKey` warten. Andererseits darf nicht das `Cls` zum Wegwischen des alten Kreises vergessen werden. Beendet wird das Programm dann wieder mit dem roten Stop-Schalter der BlitzMax-IDE:

```
SeedRnd(MilliSecs())
Graphics 800, 600
x = 350
y = 250
xSchritt = Rand(-4, 4)
ySchritt = Rand(-4, 4)
Repeat
        Cls
        DrawOval x, y, 100, 100
        x = x + xSchritt
        y = y + ySchritt
        Flip
Forever
```

Schon ganz hübsch, aber zweierlei sollte uns noch stören. Erstens kann es vorkommen, dass zufällig beide Schrittweiten gleich Null sind und sich der Kreis gar nicht bewegt, sondern stehenbleibt. Dies lässt sich beispielsweise dadurch vermeiden, dass wir die Zuweisung der zufälligen Schrittweite (so häufig wie nötig) wiederholen, bis entweder der eine oder der andere Schritt ungleich Null ist:

```
Repeat
        xSchritt = Rand(-4, 4)
        ySchritt = Rand(-4, 4)
Until xSchritt <> 0 Or ySchritt <> 0
```

Zweitens ist unser Kreis ja recht bald aus unserem Blickfeld verschwunden, weil er das Fenster verlässt. Es wäre aber doch schön, wenn er wie ein Ball von den Rändern des Fensters abprallen würde und (je nachdem, um welchen Rand es sich handelt) entsprechend die Richtung ändert, d.h. die Schritte umkehrt. Wir brauchen also eine Fallunterscheidung: Falls der Kreis (mit einem negativen xSchritt) den linken Fensterrand überschreitet `If x < 0 Then` , soll er wieder nach Rechts fliegen, also mit `xSchritt = -xSchritt` eine Umkehrung des Vorzeichens erfolgen. Genauso natürlich umgekehrt, wobei dabei noch die Breite des Kreises beachtet werden muss, denn die Zeichenposition ist ja die gedachte linke obere "Ecke". Da in beiden Fällen die Richtung wechselt bzw. das Vorzeichen des xSchritts umgekehrt wird, können wir sie zusammenfassen zu `If x < 0 Or x > 700 Then xSchritt = -xSchritt`. Für die Kollision des Kreises mit dem oberen oder unteren Fensterrand sieht die Sache ganz ähnlich aus
`If y < 0 Or y > 500 Then ySchritt = -ySchritt` und wir erhalten insgesamt folgenden Code:

```
SeedRnd(MilliSecs())
Graphics 800, 600
x = 350
y = 250
Repeat
	xSchritt = Rand(-4, 4)
	ySchritt = Rand(-4, 4)
Until xSchritt <> 0 Or ySchritt <> 0
Repeat
	Cls
	DrawOval x, y, 100, 100
	x = x + xSchritt
	y = y + ySchritt
	If x < 0 Or x > 700 Then xSchritt = -xSchritt
	If y < 0 Or y > 500 Then ySchritt = -ySchritt
	Flip
Forever
```

7.9 Übungsaufgabe

Schreib einen Code, der auf die Mitte des unteren Fensterrands ein rotes Rechteck zeichnet, das einen Schornstein symbolisieren soll, aus dem ein kreisrundes, graues "Wölkchen" langsam und gleichmäßig nach oben schwebt. Ist es verschwunden, soll ein neues Wölkchen aufsteigen usw.

7.10 Übungsaufgabe

Simuliere in deinem Fenster einen Live-Ticker, der die folgende Infozeile wiederholt:
"- Deutsche Nationalmanschaft gewinnt Weltmeisterschaft 2222 -"

Kapitel 8

Interaktion Teil 1

8.1 Maus-Position

Fast alle Computerspiele kann man mit der Maus spielen. Dabei ist zunächst einmal wichtig, wo sie sich befindet und hinbewegt wird. Die waagerechte Mausposition kann mit `MouseX()` und die senkrechte Mausposition mit `MouseY()` erfragt werden. Am Besten speichert man dies in zwei Variablen. Wir wollen uns diese Informationen einmal mittels `DrawText` etwas rechts neben dem Mauszeiger auf dem Bildschirm anzeigen lassen, so können wir noch einmal direkt das Koordinatensystem des Fensters demonstrieren.
Gleichzeitig wollen wir anstelle des Abwürgens über den Stop-Schalter der BlitzMax-IDE mit dem Befehl `AppTerminate()` eine sauberere Methode kennenlernen, die Schleife abzubrechen, das Programm zu beenden und das Fenster zu schließen. `AppTerminate()` tritt in Kraft, wenn du mit der Maus auf den Kreuz-Schalter auf der oberen Fensterleiste klickst:

```
Graphics 800, 600
Repeat
	Cls
	MausX = MouseX()
	MausY = MouseY()
	DrawText MausX, MausX + 20, MausY - 20
	DrawText MausY, MausX + 20, MausY
	Flip
Until AppTerminate()
```

Von nun an wollen wir stets `AppTerminate()` als Abbruchbedingung verwenden.

8.2 Übungsaufgabe

Programmiere ein kleines "Fangenspiel", bei dem man mit dem Mauszeiger ein Quadrat, dessen Größe zufällig und variabel sein soll, berühren muss, so dass dieses dann zufällig an eine andere Stelle im Fenster springt und eine andere Größe erhält.
Erinnere dich bezüglich der Kontrolle des Berührens daran, dass das Quadrat mit der linken oberen Ecke verankert ist.

8.3 Maus-Tasten

Jede Maus hat mindestens zwei Tasten. Die erste bzw. "Haupt"-Taste ist die linke Maustaste (wenn sie nicht in den Systemeinstellungen umgestellt wurde). Durch `If MouseDown(1)` prüft man, ob sie herunter gedrückt wird. Das Betätigen der zweiten, rechten Maustaste wird entsprechend durch `If MouseDown(2)` ermittelt. Wenn du nachfolgenden Code ausführst, wirst du merken, dass sich die Farbe der Fensterhälften ändert, je nachdem, welche Maustaste du herunter drückst:

```
Graphics 400, 300
Repeat
	Cls
	DrawRect 0, 0, 200, 300
	If MouseDown(1) Then SetColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If MouseDown(2) Then SetClsColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	Flip
Until AppTerminate()
```

Dabei wirst du statt eines einfachen einen mehrfachen Farbwechsel in Form eines Flimmerns beobachten haben und zwar je nachdem, wie lange du die jeweilige Maustaste gedrückt gehalten hast. Mehr noch: Es scheint fast unmöglich zu sein, eine Maustaste so kurz anzutippen, dass wirklich nur ein einziger Farbwechsel passiert. Sei dir also bewusst, dass du allein mit dem Befehl `MouseDown` so gut wie nie einen einzigen Moment erfassen wirst, sondern stets einen längeren Zustand.
Als Gegensatz dazu gibt es `MouseHit(1)` und `MouseHit(2)`. Diese zählen nur den einmaligen "Klick" und nicht das Niedergedrückt-Halten der jeweiligen Maustaste. Tauscht man dies im obigen Beispielcode aus, wirst du von nun an kein Flimmern mehr erleben, teste selbst:

```
Graphics 400, 300
Repeat
	Cls
	DrawRect 0, 0, 200, 300
	If MouseHit(1) Then SetColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If MouseHit(2) Then SetClsColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	Flip
Until AppTerminate()
```

Allerdings kann man `MouseHit` nur "einmal pro Klick" verwenden, denn das Klick-Ereignis wird durch die Abfrage egalisiert. Würde man im Code noch ein weiteres Mal ein `If MouseHit(1)` in der Schleife hinzufügen, so würde dieser Fall nie eintreten, denn dieses MoustHit(1) könnte nur ein erneutes Klicken registrieren - und so schnell kann niemand hintereinander klicken, was folgender Anti-Code beweist: Der Infotext "Rechte Maustaste" wird nie erscheinen:

```
Graphics 400, 300
Repeat
	Cls
	DrawRect 0, 0, 200, 300
	If MouseHit(1) Then SetColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If MouseHit(2) Then SetClsColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If MouseHit(2) Then DrawText "Rechte Maustaste", 250, 100
	Flip
Until AppTerminate()
```

Möchte man sich aber die Möglichkeit vorbehalten, das Klickverhalten auch mehrfach in einer Schleife abfragen zu können, so muss man die Eingabe in einer Variablen speichern, etwa so: `RechtsKlick = MouseHit(2)`. Erst mittels dieser Korrektur wäre der folgendermaßen korrigierte Code (zumindest) tauglich:

```
Graphics 400, 300
Repeat
	Cls
	DrawRect 0, 0, 200, 300
	Linksklick = MouseHit(1)
	RechtsKlick = MouseHit(2)
	If Linksklick Then SetColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If RechtsKlick Then SetClsColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If RechtsKlick Then DrawText "Rechte Maustaste", 250, 100
	Flip
Until AppTerminate()
```

Sinnvoll ist er aber immer noch nicht. Am Besten sollte der InfoText so lang zu sehen sein, wie die Maustaste gedrückt gehalten wird. Für die Textanzeige verwendet man daher eher `MouseDown` und für den Farbwechsel die Variable für `MouseHit`:

```
Graphics 400, 300
Repeat
	Cls
	DrawRect 0, 0, 200, 300
	Linksklick = MouseHit(1)
	RechtsKlick = MouseHit(2)
	If Linksklick Then SetColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If RechtsKlick Then SetClsColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	If MouseDown(2) Then DrawText "Rechte Maustaste", 250, 100
	Flip
Until AppTerminate()
```

Möchte man einmal das Gegenteil prüfen, also ob etwa die linke Maustaste **nicht** gedrückt wird, so können wir diese Verneinung durch `Not` ausdrücken, also: `If Not MouseDown(1)`.
Zum Abschluss sei noch angemerkt, dass die besprochenen Mauseingabe-Prüfungen nur innerhalb eines selbst erstellten und gerade aktiven Grafikfensters anwendbar sind.

8.4 Übungsaufgabe

Baue das "Fangen-Spiel" aus der Übungsaufgabe 8.2 dergestalt um, dass ein Kreis erst dann als gefangen gilt, wenn man (neu) darauf geklickt hat. In diesem Fall soll eine kleine Meldung so lang erscheinen, wie man den Kreis "festhält". Und erst wenn man die Maustaste wieder loslässt, soll der Kreis dann davon springen bzw. woanders auftauchen und das Spiel weitergehen.

8.5 Tastatur-Eingaben

Für das Betätigen der Computertastatur gilt im Prinzip die gleiche Unterscheidung. Dafür gibt es nämlich ebenso `KeyDown()` und `KeyHit()`. Allerdings gibt es bekanntermaßen viel mehr Tasten bzw. Nummern, die man als Argument in die Klammern schreiben kann, um zu registrieren, ob eine bestimmte Taste betätigt wird. Gottlob müssen wir diese Nummern nicht auswendig lernen, denn BlitzMax stellt uns den Service sogenannter Konstanten zur Verfügung, in denen die Nummern gespeichert sind. Diese Konstanten haben (englische) Namen, die man sich leichter merken kann. Mit `KeyDown(KEY_A)` oder `KeyDown(KEY_B)` könnten wir beispielsweise prüfen, ob die Tasten A oder B gedrückt werden. Dann ist auch kein Geheimnis, was z.B. `KEY_C` und `KEY_D` bedeuten.
Ausnahmen bilden spezielle Funktionstasten, von denen einige der wichtigsten und ihre Bezeichnungen im Folgenden aufgelistet werden:

Leertaste	=	KEY_SPACE
Enter-Taste	=	KEY_RETURN
Escape-Taste	=	KEY_ESCAPE
Linke Control-Taste	=	KEY_LCONTROL
Rechte Control-Taste	=	KEY_RCONTROL
Linke Alt-Taste	=	KEY_LALT
Rechte Alt-Taste	=	KEY_RALT
Pfeiltaste nach Links	=	KEY_LEFT
Pfeiltaste nach Rechts	=	KEY_RIGHT
Pfeiltaste nach Oben	=	KEY_UP
Pfeiltaste nach Unten	=	KEY_DOWN

Weitere findest du (gut versteckt) in der Hilfe der BlitzMax-IDE, wenn du dich in der rechten Tabulator durch *Home - Index - KeyDown* klickst und auf den Link zu den *key codes* stößt.

Es hat sich zum guten Ton entwickelt, als weitere Abbruchbedingung der Hauptschleife bzw. des Programms neben AppTerminate() auch noch das Drücken der Escape-Taste hinzuzufügen, auch wir wollen uns ab jetzt daran gewöhnen und notieren als Schleifenabbruch:

```
Until AppTerminate() Or KeyDown(KEY_ESCAPE)
```

8.6 Eingabespeicher löschen

Auch außerhalb der eigentlichen MouseHit und KeyHit -Abfragen werden die entsprechenden Aktionen registriert und gespeichert. Wenn sicherstellen willst, dass nicht an anderer Stelle gemachte Eingaben unerwünschte Konsequenzen nach sich ziehen, dann lösche mit den Befehlen FlushMouse() und FlushKeys() alles, was bis dato eingegeben wurde.

8.7 Übungsaufgabe

Schreibe einen Code, mit dem man einen Kreis innerhalb eines Grafikfensters mit den Pfeiltasten Rechts, Links, Oben und Unten steuern kann. Außerdem soll man mit der Leertaste die Farbe des Kreises (zufällig) ändern können.

Kapitel 9

Variablen Teil 2

9.1 Dezimalbrüche

Wir haben bisher den bequemen Service einer Basic-Programmiersprache genossen, bei Bedarf einfach einen Name für eine Variablen zu erfinden und sofort loszulegen.
Nun wird es aber Zeit, sich etwas ernsthafter mit der Einführung neuer Variablen zu beschäftigen, denn die bislang so einfache Handhabung birgt empfindliche Risiken.
Die Kürze und Drastik einiger Codebeispiele dieses Kapitels mag nicht darüber hinwegtäuschen, dass solche oder ähnliche Fehler in längeren Codes mitunter schwer aufzuspüren sind. Nehmen wir einmal an, wir wollten zwei Zahlen wie folgt dividieren:

```
ErsteZahl = 3
ZweiteZahl = 6
Quotient = ErsteZahl / ZweiteZahl
Print Quotient
```

Zu unserer Überraschung kommt aber nicht das heraus, was wir erwarten, nämlich den Dezimalbruch 0.5 (Das Komma wird immer als Punkt notiert). Dies liegt daran, dass dazu ein anderer Variablentyp notwendig ist und dass bei nicht weiter spezifierter Einführung einer neuen Variablen standartmäßig stets von ganzen Zahlen, also z.B. 3 oder -4 usw. ausgegangen wird. Verrechnet der Computer zwei ganze Zahlen, dann sucht er das Ergebnis ebenfalls im ganzzahligen Bereich, d.h. die Nachkommastellen werden einfach abgeschnitten und statt 0.5 kommt eben nur 0 heraus.
Unterschiedliche Variablentypen haben verschiedene Namen. Die angedeuteten ganzen Zahlen heißen "Integer", kurz `Int` und die Dezimalbrüche bzw. "Fließkommazahlen" `Float`. Um bei der Einführung einer neuen Variable festzulegen, zu welchem Typ sie gehört, genügt es, direkt dahinter einen Doppelpunkt und die entsprechende Typenbezeichnung zu notieren:
`Quotient:Float` Nun weiß der Compiler Bescheid, dass die Variable mit dem Namen "Quotient" eine Kommazahl sein soll. Ganz am Ziel sind wir aber immer noch nicht, denn der obige Code würde mit `Quotient:Float = ErsteZahl / ZweiteZahl` immer noch keine 0.5 ausgeben. Dies liegt darin begründet, dass bei dieser Zuweisung zunächst einmal die rechte Seite des Gleichheitszeichens ausgerechnet wird, bevor ihr Ergebnis nach Links weitergereicht wird. Rechts stehen aber nach wie vor zwei ganze Zahlen, deren Ergebnis ganzzahlig bleibt. Man muss also noch mindestens eine der rechten Variablen als Kommazahl einführen, dann endlich kommt das Gewünschte heraus:

```
ErsteZahl:Float = 3
ZweiteZahl:Int = 6
Quotient:Float = ErsteZahl / ZweiteZahl
Print Quotient
```

An den zusätzlichen, mathematisch unsinnigen rechten Nullen der Ausgabe 0.500000000 kannst du erkennen, dass für den Float-Variablentyp ein fester Speicherplatz reserviert und auch stets ausgefüllt wird. Dies wird uns später noch beschäftigen.
Übrigens: Auch wenn wir ganz auf Variablen verzichteten, würde `Print 3 / 6` nur 0 ausgeben, weil die Zahlenschreibweise ohne Punkt als ganzzahlig gelesen wird. Erst wenn mindestens eine der Zahlen einen Punkt besitzt, also als Fließkommazahl notiert wird, wird auch entsprechend gerechnet: `Print 3 / 6.0` ergibt 0.5 genauso wie `Print 3.0 / 6`.

Das bislang als "Rechenfehler" wahrgenommene ganzzahlige Ergebnis der Verrechnung zweier Ganzzahlen kann als Chance genutzt werden, um zu prüfen, ob eine Rechnung aufgeht oder nicht. Nehmen wir einmal an, der Computer soll selbstständig herausfinden, ob 123456 durch 7 teilbar ist oder nicht. Dazu könnten wir das Ergebnis bewusst in einer Integer-Variablen speichern `GanzZahl:Int = 123456 / 7` und anschließend prüfen, ob das Multiplizieren mit 7 tatsächlich wieder 123456 ergibt, denn hätte die Ganzzahl notwendige Nachkommastellen abgeschnitten, käme man beim Produkt nicht wieder auf 123456:

```
If GanzZahl * 7 = 123456 Then Print "123456 ist durch 7 teilbar."
```

9.2 Die Tücken von Float-Variablen

Wie eben behandelt, können wir durch die Verwendung von Float-Variablen bzw. Kommazahlen genauere Ergebnisse ausrechnen. Der Begriff "genauer" ist jedoch überraschender Weise fehl am Platz, denn Float-Variablen sind eigentlich "fehlerhafter" als die ganzzahligen Integer-Variablen. Um dies zu verstehen, sei daran erinnert dass ein Computer intern ja nur mit Nullen und Einsen - also mit Binärzahlen bzw. Zweierzahlen - rechnet. Jede Ganzzahl kann man ohne Verluste in eine Binärzahl übersetzen und umgekehrt: Die Zahl 23 entspricht beispielsweise der Binärzahl 10111. Mit Kommazahlen ist dies jedoch nicht so ohne weiteres möglich. Eine Float-Variable besitzt gemäß ihrem reservierten Speicherplatz eine bestimmte Anzahl von Stellen und versucht, diese "irgendwie" binär zu speichern. Manchmal geht es gut:

```
Kommazahl:Float = 2.0
Print Kommazahl
```

manchmal aber passt es nicht so ganz und wir erhalten in den hinteren Nachkommastellen Abweichungen bzw. Ungenauigkeit, teste einfach folgenden Code:

```
Kommazahl:Float = 0.2
Print Kommazahl
```

An diese möglichen Ungenauigkeiten sollte man stets denken, wenn man Float-Variablen verwendet, denn die Abweichungen können sich verstärken und - beispielsweise in einer Schleife - unvorhergesehene Probleme bereiten. Führe zur Demonstration folgenden Code aus:

```
Summe:Float = 0
Print "In Fünftelschritten bis (mindestens) Zehn ergibt den Endwert:"
Repeat
        Summe = Summe + 0.2
Until Summe >= 10
Print Summe
Print "In Fünftelschritten zurück bis (mindestens) Null ergibt den Endwert:"
Repeat
        Summe = Summe - 0.2
Until Summe <= 0
Print Summe
```

Hätte man hier als Abbruchbedingung der Repeat-Until-Schleifen keine Ungleichungen, sondern Gleichungen verwendet, hätten sich unweigerlich zwei Endlosschleifen ergeben, ohne dass dies vom rein mathematischen Gesichtspunkt her verständlich gewesen wäre.
Zum vorläufigen Abschluss sei noch darauf hingewiesen, dass sehr kleine Dezimalbrüche, also solche, die erst nach vielen Nachkommastellen von Null verschiedene Ziffern haben, ähnlich wie beim Taschenrechner in einer speziellen Kurzschreibweise dargestellt werden, Beispiel:

```
Millionstel:Float = 1.0 / 1000000
Print Millionstel
```

Hier erhält man 9.99999997e-07, was eigentlich $9.99999997 * 10^{-7}$ bedeutet. Wenn du noch keine Potenzen mit negativen Exponenten kennst, genügt es zu wissen, dass die -7 hinter dem Buchstaben e bedeutet, dass die erste 9 von Links eigentlich erst in der siebten Nachkommastelle auftaucht. Eigentlich lautet die Zahl nämlich 0.000000999999997. Dabei sehen wir wieder die Abweichung von der Millionstel, die hier drei Billiardstel beträgt.

9.3 Übungsaufgabe

Schreibe einen Code, der ermittelt, wie viel ganze Zufallszahlen zwischen 1 und 999 gebildet werden müssen, damit ihr Durchschnitt um höchstens ein Tausendstel vom statistischen Mittelwert 500 abweicht. Die gesuchte notwendige Anzahl ist übrigens keine feste Größe, sondern ebenfalls stark dem Zufall unterworfen.

9.4 String-Variablen

Nachdem wir mit `Int` und `Float` bereits ganz gut versorgt sind, was Zahlen angeht, wollen wir uns nun mit dem dritten wichtigen Variablentyp beschäftigen: `String`. In String-Variablen kann man Texte speichern, z.B.:

```
Nachricht:String = "Mir geht es ganz gut."
```

Wie bei allen Variablen könnte man dann danach den Inhalt ändern und beispielsweise mit Print in der Konsole ausgeben lassen:

```
Nachricht = "Neee, mir geht's eigentlich miserabel."
Print Nachricht
```

Das Addieren von Strings bzw. Stringvariablen hat natürlich eine andere Bedeutung als jene bei den Zahlenvariablen, denn die Texte werden einfach aneinander gekettet, wie folgender Code belegt:

```
Nachricht:String = "Ich bin voll gut d'rauf!"
Nachricht = Nachricht + " Wie geht's dir?"
Nachricht = "Hallo! " + Nachricht
Print Nachricht
```

Übrigens kann man auch Zahlen bzw. Int- oder Float-Variablen an solche Strings ketten, sie werden dadurch insgesamt ebenfalls zu einem String umgewandelt:

```
Alter:Int = 15
Info:String = "Ich bin " + Alter + " Jahre alt, und du?"
Print Info
```

Andererseits muss man aufpassen, wenn man bei diesem Einbinden von Zahlen in einen String noch etwas ausrechnen, also tatsächlich numerisch addieren will. Teste folgenden Code

```
Info:String = "Eins plus Zwei gibt " + 1 + 2
Print Info
```

und du siehst den Unsinn. In diesem Fall muss die numerische Rechnung noch in eine Klammer gefasst werden, damit sie zuerst erledigt wird, bevor sie in den String eingebunden wird:

```
Info:String = "Eins plus Zwei gibt " + (1 + 2)
Print Info
```

Das Angenehme ist, dass man Strings genauso wie Zahlen abgleichen bzw. überprüfen kann, ob sie übereinstimmen:

```
Info:String = "Eins plus Zwei gibt " + (1 + 2)
If Info = "Eins plus Zwei gibt 3" Then Print "Die Info stimmt!"
```

Für Strings gibt es eine Menge zusätzlicher Funktionen, mit denen Detail-Informationen analysiert oder manipuliert werden können. Einige von ihnen seien hier aufgelistet:

`Len(String)` ermittelt die Länge des Strings, also die Anzahl ihrer Buchstaben bzw. Zeichen, beispielsweise gibt `Len("Er ist meistens zu Hause")` die Länge 24 zurück.

`Left(String, Laenge)` gibt den linken Teil des Strings mit der geforderten Laenge zurück. beispielsweise gibt `Left("Er ist meistens zu Hause", 6)` den Teilstring "Er ist" zurück.

`Right(String, Laenge)` gibt den rechten Teil des Strings mit der geforderten Laenge zurück. beispielsweise gibt `Right("Er ist meistens zu Hause", 8)` den Teilstring "zu Hause" zurück.

`Mid(String, Stelle, Laenge)` gibt jenen Teil des Strings zurück, der an der gewünschten Stelle beginnt und die erforderliche Laenge hat, wobei das erste Zeichen von Links die Stelle 1 hat. Mit `Mid("Er ist meistens zu Hause", 8, 8)` erhält man den Teilstring "meistens".

`Instr(String, Teilstring, Start)` sucht den String ab der Stelle "Start" durch, ob der geforderte Teilstring enthalten ist. Wenn ja, gibt er die beginnende Stelle des Teilstrings zurück, ansonsten 0, dazu drei Beispiele:

`Instr("Er ist meistens zu Hause", "ist", 1)` ergibt 4, da ab dem 4.Zeichen ein "ist" beginnt.
`Instr("Er ist meistens zu Hause", "ist", 5)` ergibt 10, da dort ein zweites "ist" beginnt.
`Instr("Er ist meistens zu Hause", "ist", 11)` ergibt 0, da danach kein weiteres "ist" existiert.

`Replace(GesamtString, Teilstring, StringErsatz)` gibt einen String zurück, bei dem innerhalb des GesamtStrings der zu ersetzende Teilstring (falls vorhanden) durch den gewünschten StringErsatz ersetzt wird. `Replace("Er ist meistens zu Hause", "ist meistens", "war selten")` würde den String "Er war selten zu Hause" zurückgeben.

`Upper(String)` verwandelt den kompletten String in Großbuchstaben:
`Upper("Er ist meistens zu Hause")` gibt den String "ER IST MEISTENS ZU HAUSE" zurück.

`Lower(String)` verwandelt den kompletten String in Kleinbuchstaben:
`Lower("Er ist meistens zu Hause")` gibt den String "er ist meistens zu hause" zurück.

9.5 Übungsaufgabe

Schreibe einen Code, der einen beliebigen Satz, z.B. *"Sechs Hessen vergessen in Essen ihr Essen mit Messern zu fressen"* in seine einzelnen Worte zerlegt bzw. diese mit Print auflistet.

9.6 Local und Global

Neben der Notwendigkeit, den gewünschten Variablentyp anzugeben, müssen wir uns noch mit einem weiteren Aspekt bei der Einführung neuer Variablen beschäftigen, nämlich mit jenem, "wo die Variable gelten soll bzw. gebraucht wird". Dies klingt für dich wahrscheinlich erst einmal recht merkwürdig, hat jedoch einen vernünftigen Grund: Wie du weißt, braucht jede Variable ihren (Speicher-)Platz. Platz zu sparen ist jedoch ein wichtiges Gebot beim Programmieren: Wir wollen davon nur dort etwas vereinnahmen, wo wir es auch tatsächlich brauchen, ansonsten aber den Platz wieder frei machen. In unseren bisherigen winzigen Code-Beispielen spielt dies natürlich noch nicht wirklich eine Rolle. Das Prinzip sollten wir jedoch begreifen und übernehmen:
Setzen wir vor der Einführung einer neuen Variablen ein `Global` davor, so gilt die Variable (nach ihrer Einführung) "global", also überall ohne Einschränkungen im gesamten Code des Programms.
Setzt man jedoch ein `Local` davor, so gilt sie nur "lokal" bzw. nur in jenem Bereich, wo sie definiert wurde. Wenn der Bereich verlassen wird, wird auch der Speicherplatz wieder frei. Was unter einem solchen "Bereich" zu verstehen ist, wird im Kapitel 10 noch erklärt werden.
Vorerst sollten wir uns damit begnügen und uns ab jetzt angewöhnen, alle Variabeln entweder `Local` oder `Global` einzuführen. Vorerst reicht zumeist ein `Local`.
Übrigens braucht man der Variablen dann nicht wie bisher gleich einen Wert zuzuweisen, sondern schreibt ganz einfach `Local MeineNeueVariable:Int` und ist fertig.
Man darf hinter einem beginnenden Local oder Global - durch ein Komma getrennt - sogar mehrere Variablen hintereinander in einer Zeile einführen:
`Local a:Int, b:Int, c:Int, f:Float, s:String`
Selbstverständlich gilt dann das Local bzw. Global für alle Variablen dieser Zeile

9.7 Strict und SuperStrict

Kommen wir nun zu etwas ganz Wichtigem, das einem das Programmierleben zu Hölle machen kann, wenn man es nicht beachtet.
Lies dir folgenden kleinen Code in Ruhe durch: Was wird dort `Print Summe` wohl ausgeben?
Kopiere den Code so wie er ist, füge ihn im BlitzMaxIDE-Editor wieder ein und führe ihn aus:

```
Local ErsteZahl:Int, ZweiteZahl:Int, Summe:Int
ErsteZahl = 1
ZweiteZahl = 2
Summe = ErsteZahl + ZweiteZahl
Print Summe
```

Merkwürdiger Weise kommt blanker Unsinn heraus! Trotzdem wird es dir (je nach Schriftart) kaum bis unmöglich sein, hier irgend einen Fehler zu finden. Es ist wahrlich zum Verzweifeln!
Nun stell' dir das Ganze in einem seitenlangen Code vor ...

Ich will nun das Geheimnis lüften, das uns solch Kopfzerbrechen bereitet: Es ist ein Tippfehler, allerdings - das will ich zugeben - ein ganz gemeiner: Der letzte Buchstabe in der 4.Codezeile `Summe = ErsteZahl + ZweiteZahl` ist kein kleines L, sondern ein großes i. Je nach Schriftart sieht dieses große i dem kleinen L täuschend ähnlich, so dass der Fehler überhaupt nicht auffällt. Doch was macht an dieser Stelle der Compiler? Er diagnostiziert mit der falsch geschriebenen `ZweiteZahl` eine neue Variable, die natürlich erst einmal den Wert 0 besitzt. Daher kommt bei der Rechnung 1 + 0 tatsächlich 1 heraus.
Auch wenn das Beispiel vielleicht ein wenig konstruiert wirkt: Tippfehler machen wir alle. Und wenn der Code nicht nur aus fünf Zeilen besteht, sondern aus vielen Seiten, dann findet man ihn genauso wenig. Wie sollten daher einsehen, dass der bequeme Service einer Basic-Programmiersprache, mittendrin nach Belieben neue Variablen einzuführen, auch ein Fluch bedeuten kann, und zwar dann, wenn vertippte Variablennamen fälschlich als neue Variablen diagnostiziert werden.
Aus diesem Grund bietet uns BlitzMax den Befehl `Strict` ganz am Anfang unseres Codes an, mit dem wir freiwillig auf den geschilderten Basic-Service verzichten und uns den Zwang auferlegen, jede Variable, die wir nutzen wollen, zunächst ordentlich mit Local oder Global zu deklarieren. Stellen wir dem obigen Anti-Codebeispiel in der ersten Zeile ein `Strict` voran,

```
Strict
Local ErsteZahl:Int, ZweiteZahl:Int, Summe:Int
ErsteZahl = 1
ZweiteZahl = 2
Summe = ErsteZahl + ZweiteZahl
Print Summe
```

dann wirst du merken, dass der Code gar nicht ausführbar ist: Der Compiler beschwert sich, dass er die falsch geschriebene Variable `ZweiteZahl` gar nicht kennt. Damit zeigt er uns genau jene Stelle, an der etwas nicht stimmt, was ein außerordentlich hilfreicher Hinweis ist.
Verwendet man statt `Strict` die Variante `SuperStrict`, dann wird die Variablendeklaration noch zwingender, den SuperStrict verlangt, dass wir bei jeder eingeführten Variable festlegen müssen, zu welchem Typ sie gehören soll - für uns eigentlich schon eine Selbstverständlichkeit. Wenn wir nicht gerade winzige Codebeispiele erstellen, wollen wir uns daher ab jetzt die Verbindlichkeit von `SuperStrict` aneignen.
Anfangs mag einem dies ein wenig nerven, beispielsweise wenn sogar simple Laufvariablen in einfachen Schleifen erst wie folgt definiert werden wollen:

```
SuperStrict
For Local i:Int = 1 To 10
        Print i
Next
```

aber man wird sich rasch an `SuperStrict` gewöhnt haben und es schließlich gar nicht mehr missen wollen. Abschließend sei noch einmal zusammengefasst:

`SuperStrict` verlangt z.B. `Local Variable:Int`

`Strict` verlangt z.B. `Local Variable`

Nichts verlangt nichts. Auch mitten in Rechnungen können neue Variablen auftauchen.

Alle folgenden Codebeispiele bauen auf `SuperStrict` auf und sind auch nur mit `SuperStrict` ausführbar. Allein aus Platzgründen wird der Befehl aber nicht ständig aufgeführt. Wenn du die Code-Beispiele kopierst und in den BlitzMaxIDE-Editor einfügst, sollte dort stets `SuperStrict` in der ersten Zeile vorangestellt werden.

9.8 Feldvariablen (Arrays)

Nicht selten kommt es vor, dass man mehrere Werte - z.B. zehn Zufallszahlen zwischen 1 und 100 - nicht nur einmalig erzeugen, sondern im weiteren Verlauf noch einzeln nutzen und verwenden will, beispielsweise um sie der Größe nach zu sortieren.
Folgendermaßen anzufangen:

```
Local Zahl:Int
For Local i:Int = 1 To 10
	Zahl = Rand(1, 100)
Next
```

wäre dann unsinnig, weil die Variablen Zahl ja jedesmal neu gefüllt wird und dabei die vorherige Information bzw. Zufallszahl verloren geht.
Nun könnten wir ja statt dessen zehn Variablen mit den Namen Zahl1, Zahl2, Zahl3 usw. einführen, doch dies erscheint arg umständlich. Außerdem könnten wir dann die Schleife nicht mehr nutzen und das Sortieren geriete im übrigen zu einem ziemlichen Chaos.
Für diesen Zweck bieten daher alle Programmiersprachen sogenannte "Feldvariablen" an, die man im Englischen "Array" nennt. Wie etwa Spargel schön eingereiht in der gleichen Reihe eines Feldes stecken, genauso stecken nun mehrere Werte in einer Variablen gleichen Namens. Einzeln angesprochen werden sie über einen sogenannten Index, der in eckigen Klammern hinter dem Variablennamen steht. Für Anfänger gewöhnungsbedürftig, aber unbedingt zu beachten ist die Eigenart, dass der Index bzw. die Nummerierung nicht mit 1, sondern mit 0 beginnt und daher bei einem zehn-Einträge-großen Array der letzte Eintrag nicht den Index 10, sondern den Index 9 besitzt.

Eine Feldvariable bzw. Array mit 10 Einträgen wird so eingeführt:

```
Local Zahl:Int[10]
```

Danach könnte man z.B. den dritten Eintrag (der den Index 2 hat!) mit der Zahl 37 füllen:

```
Zahl[2] = 37
```

oder den ersten Eintrag (der den Index 0 hat!) mit der Zahl 48:

```
Zahl[0] = 48
```

oder den zehnten, letzten Eintrag (der den Index 9 hat!) mit der Zahl 56:

```
Zahl[9] = 56
```

Per Print ließen sich die Werte dann wieder einzeln abrufen:

```
Print Zahl[2]
```
ergäbe 37

```
Print Zahl[0]
```
ergäbe 48

```
Print Zahl[9]
```
ergäbe 56

```
Print Zahl[4]
```
ergäbe 0, denn dort wurde noch nichts hinein gefüllt.

Würde man Indizes verwenden, welche außerhalb der eingeführten Größe des Arrays liegen, in unserem Beispiel also etwa Zahl[777] oder Zahl[-1] oder Zahl[10], ergäbe dies ein Laufzeitfehler und das Programm würde abstürzen.
Das Angenehme an Feldvariablen bzw. Arrays ist, dass man sie sinnvoll in Schleifen einbinden kann: Der folgende Code fängt scheinbar genauso an, wie jener zu Beginn des Kapitels. Der entscheidende Unterschied liegt jedoch darin, dass sämtliche Zufallszahlen auch nach ihrer Erzeugung noch einzeln über den entsprechenden Index abrufbar sind, z.B. mit Print:

```
Local Zahl:Int[10]
For Local i:Int = 0 To 9
	Zahl[i] = Rand(1, 100)
Next
For Local i:Int = 0 To 9
	Print Zahl[i]
Next
```

Die Anzahl der Einträge, also die Größe eines Arrays, die man auch "Dimension" nennt, kann auch durch eine Variable definiert werden:

```
Local Wieviel:Int= 23
Local Zahl:Int[Wieviel]
For Local i:Int = 0 Until Wieviel
	Zahl[i] = Rand(1, 100)
Next
For Local i:Int = 0 Until Wieviel
	Print Zahl[i]
Next
```

'Beachte hier das Until statt des To: Da der "Wieviel"-te
'Eintrag gar nicht existiert, muss man vorher aufhören!

'(hier ebenso)

Allerdings ist die Größe des Arrays nicht dynamisch an diese Variable gekoppelt: Würde man ihren Wert ändern, hätte das keinen Einfluss auf die Größe des Arrays, die normalerweise nach ihrer Einführung konstant bleibt. Wie man die Größe eines Arrays trotzdem noch nachträglich ändern kann, wird in einem späteren Kapitel behandelt.

BlitzMax bietet bei Arrays noch weitere Annehmlichkeiten an, die einem ohne eigenes Zutun Arbeit abnehmen und zusätzliche Eigenschaften verraten:
Normalerweise kennt man ja die Größe eines Arrays. Sie könnte aber auch einmal zufällig oder aus einem anderen Grund unbekannt sein. Der englische Begriff "length" – hinter der Array-Variablen durch einen Punkt getrennt – verrät einem diese Eigenschaft:

```
Local Zahl:Int[Rand(8,13)]
Print Zahl.length
```

Vorhin sprachen wir an, die Werte eines Arrays sortieren zu wollen. Abgesehen davon, dass dies nicht so ganz einfach sein würde, wie es klingt, verwenden wir hierfür lieber die Methode sort(), welche man - wieder durch einen Punkt getrennt - hinter das Array notiert:

```
Local Zahl:Int[Rand(8,13)]
For Local i:Int = 0 Until Zahl.length
	Zahl[i] = Rand(1, 100)
Next
Print Zahl.length + " unsortierte Zufallszahlen:"
For Local i:Int = 0 Until Zahl.length
	Print Zahl[i]
Next
Print "Nun der Größe nach sortieren:"
Zahl.sort()
For Local i:Int = 0 Until Zahl.length
	Print Zahl[i]
Next
```

Bekanntlich erlaubt BlitzMax bei normalen Variablen, gleichzeitig mit der Deklaration nach einem Gleichheitszeichen sofort einen Wert zu übergeben, z.B.: `Local Zahl:Int = 66`
Bei (eindimensionalen) Arrays ist so etwas Ähnliches auch möglich: Dazu lässt man die eckige Klammer hinter der Typendeklaration einfach leer, schreibt also dort keine Größe hin. Diese ergibt sich dann automatisch durch die Anzahl der Einträge in der Aufzählung der nachfolgenden eckigen Klammer:

```
Local Zahl:Int[] = [ 66, 77, 88, 99 ]
Print "Anzahl der Einträge: " + Zahl.length
For Local Index:Int = 0 Until Zahl.length
	Print "Zahl[" + Index + "] = " + Zahl[Index]
Next
```

9.9 Übungsaufgabe

Ein Float-Array soll eine zufällige Dimension zwischen 500 und 1000 besitzen.
Fülle alle Einträge mit zufälligen Dezimalbrüchen zwischen 0 und 1 aus. Lass dann das Float-Array sortieren und gib mit Print die kleinste und die größte der Zufallszahlen an.

9.10 Übungsaufgabe

Bilde zwei Integer-Arrays mit je 5 Einträgen und fülle sie mit Zufallszahlen zwischen 1 und 10.
Vergleiche nun beide Arrays, indem du untersuchst, ob und welche gemeinsamen Werte sie haben und ermittle die dazugehörigen Indizes.

9.11 Mehrdimensionale Arrays

Zur Veranschaulichung von Feldvariablen habe ich im Kapitel 9.8 das Bild eines Spargelfelds herangezogen, bei dem die Spargel hintereinander in einer Reihe im Boden stecken.
Nun hat solch ein Spargelfeld natürlich mehrere Reihen. Ebenso ist es auch bei Arrays möglich, nicht nur eine Dimension, sondern mehrere zu besitzen: Innerhalb der eckigen Klammern werden sie durch ein Komma getrennt, z.B. im folgenden zweidimensionalen Array:
`Local Feld:String[4,3]`
Achtung: Diese zwei durch ein Komma getrennte Zahlen innerhalb einer eckigen Klammer definieren tatsächlich **zwei** verschiedene Dimension und haben damit eine ganz andere Bedeutung, als die fast gleich aussehende Übergabe von zwei Werten an ein **ein**dimensionales Array, welches so aussieht:
`Local Feld:String[] = [4,3]`
Denn um das obige zweidimensionale Array komplett durchzugehen, braucht man zwei ineinander verschachtelte Schleifen: Für jede der 4 Einträge der ersten Dimension gibt es 3 Einträge der zweiten Dimension bzw. umgekehrt. In der Summe sind das 4 * 3 = 12 Einträge. Vielleicht kann man sich dies am Besten grafisch veranschaulichen. Malen wir also ein Feld aus 4 * 3 Spargeln:

```
Local Feld:String[4,3]
For Local a:Int = 0 To 3
	For Local b:Int =  0 To 2
		Feld[a,b] = "Spargel[" + a  + "," + b  + "]"
	Next
Next

Graphics 700, 450
For Local x:Int = 0 To 3
	For Local y:Int =  0 To 2
		DrawText Feld[x,y], 50 + 150 * x, 100 + 100 * y
	Next
Next
Flip
WaitKey
```

Auch noch beliebig mehr Dimensionen sind denkbar, z.B. `Local Array:Float[77, 8, 999]`
Wieviel Einträge hätte dieses dreidimensionale Array? Genau: 77 * 8 * 999 = 615384, was schon eine ganze Menge ist und entsprechend viel Speicherplatz verbraucht.

9.12 Übungsaufgabe

Schreibe einen Code, der in einem 600 Pixel breiten und 400 Pixel tiefen Grafikfenster ein regelmäßiges Karomuster aus Quadraten zeichnet, welche alle die Seitenlänge 50 Pixel haben und nur in Grautönen gefärbt sind. Wenn man dann mit der Maus auf eines der Quadrate klickt, soll es sich dann beliebig bunt färben.

Lösungen der Übungsaufgaben

Übungsaufgabe 1.4:

```
Songdauer = 237
Wiederholungen = 5
Gesamtdauer = Wiederholungen * Songdauer
Songdauer = 192
Wiederholungen = 3
Gesamtdauer = Gesamtdauer + Wiederholungen * Songdauer
Print Gesamtdauer
```

Übungsaufgabe 2.4:

```
Antwort = Proceed("Willst du mit mir ausgehen?")
If Antwort = 1 Then
        Antwort = Proceed("Prima! Wollen wir ins Kino gehen?")
        If Antwort = 1 Then
                Notify("Toll! Ich bestelle gleich zwei Karten!")
        ElseIf Antwort = 0 Then
                Notify("Na, dann lasst uns lieber essen gehen.")
        Else
                Notify("Nanu, warum kneifst du auf einmal? Schade!")
        End If
Else
       Notify("Schade! Naja, vielleicht ein andermal ...")
End If
```

Übungsaufgabe 3.7:

```
For Zahl = 1 To 10
        Print "Nächste Reihe:"
        For Faktor = 1 To 10
                Produkt = Zahl * Faktor
                Print Produkt
         Next
Next
```

Übungsaufgabe 4.3:

```
Graphics 400, 300

DrawOval 140, 80, 40, 25          'Linkes Auge
DrawOval 220, 80, 40, 25          'Rechtes Auge

SetLineWidth 4
DrawLine 140, 50, 190, 80         'Linke Augenbraue
DrawLine 260, 50, 210, 80         'Rechte Augenbraue

DrawRect 190, 120, 20, 40         'Nase

DrawOval 160, 180, 80, 20         'Mund

Flip
WaitKey
```

Übungsaufgabe 5.2:

```
SeedRnd(MilliSecs())        'Zufallsgenerator aktivieren
Anzahl_Gerade = 0           'Variable zum Mitzählen der geraden Augen
For Gesamtanzahl = 1 To 10000   'die geforderte Anzahl an Wurf-Wiederholungen
        Wuerfel = Rand(6)   'Wuerfelwurf
        If Wuerfel = 2 Or Wuerfel = 4 Or Wuerfel = 6 Then       'falls gerade Augen
                Anzahl_Gerade  = Anzahl_Gerade + 1              '...mitzählen!
        End If
Next
Print(Anzahl_Gerade)        'Ausgabe der Anzahl der gewürfelten geraden Augen
```

Übungsaufgabe 7.2:

```
Graphics 512, 256
For x = 0 To 255
        Rotanteil = x
        Gruenanteil = x/2
        Blauanteil = 255 - x
        SetColor Rotanteil , Gruenanteil,  Blauanteil
        DrawRect x, 0, 1, 256
Next
For x = 0 To 255
        Rotanteil = 255
        Gruenanteil = 127 - x/2
        Blauanteil = x
        SetColor Rotanteil , Gruenanteil,  Blauanteil
        DrawRect 256 + x, 0, 1, 256
Next
Flip
WaitKey
```

Übungsaufgabe 7.3:

```
Graphics 120, 340

SetColor  255, 0, 0                 'Volles Rot
DrawOval 10, 10, 100, 100
SetColor  127, 127, 0               'Halbdunkles Gelb
DrawOval 10, 120, 100, 100
SetColor  0, 127, 0                 'Halbdunkles Grün
DrawOval 10, 230, 100, 100
Flip
WaitKey

SetColor  255, 255, 0               'Volles Gelb
DrawOval 10, 120, 100, 100
Flip
WaitKey

SetColor  127, 0, 0                 'Halbdunkles Rot
DrawOval 10, 10, 100, 100
SetColor  127, 127, 0               'Halbdunkles Gelb
DrawOval 10, 120, 100, 100
SetColor  0, 255, 0                 'Volles Grün
DrawOval 10, 230, 100, 100
Flip
WaitKey
```

Übungsaufgabe 7.5:

```
SeedRnd(MilliSecs())
Graphics 800, 600
SetClsColor 0, 0, 127
Repeat
	Cls
	For i = 1 To 1000
		Rotanteil = Rand(0, 255)
		Gruenanteil = Rand(0, Rotanteil)
		SetColor Rotanteil, Gruenanteil, 0
		DrawLine 400, 300, Rand(0, 800), Rand(0, 600)
	Next
	Flip
Forever
```

Übungsaufgabe 7.7:

```
Graphics 800, 600
SetColor 255, 127, 0
DrawText "Links Oben", 0, 0
DrawText "Rechts Oben", 800 - TextWidth("Rechts Oben"), 0
DrawText "Links Unten", 0, 600 - TextHeight("Links Unten")
DrawText "Rechts Unten", 800 - TextWidth("Rechts Unten"), 600 - TextHeight("Rechts Unten")
DrawText "In der Mitte", 400 - TextWidth("In der Mitte")/2, 300 - TextHeight("In der Mitte")/2
Flip
WaitKey
```

Übungsaufgabe 7.9:

```
Graphics 800, 600
y = 500
Repeat
	Cls
	SetColor 127, 127, 127
	DrawOval 350, y, 100, 100
	y = y - 1
	If y < -100 Then y =  500
	SetColor 127, 0, 0
	DrawRect 340, 400, 120, 200
	Flip
Forever
```

Übungsaufgabe 7.10:

```
Graphics 800, 600
Textbreite = TextWidth("- Deutsche Nationalmanschaft gewinnt Weltmeisterschaft 2222 - ")
x = 800
Repeat
	SetColor 255, 255, 255
	DrawRect 0, 545, 800, 25
	SetColor 255, 0, 0
	DrawText "- Deutsche Nationalmanschaft gewinnt Weltmeisterschaft 2222 - ", x,  550
	x = x - 2
	If x < - Textbreite Then x = 800
	Flip
Forever
```

Übungsaufgabe 8.2:

```
SeedRnd(MilliSecs())
AppTitle = "Fang' mich!"
Graphics 800, 600
Quadratseite = Rand(10, 100)
QuadratX = Rand(0, 800 - Quadratseite)
QuadratY = Rand(0, 600 - Quadratseite)
Repeat
	Cls
	MausX = MouseX()
	MausY = MouseY()
	If MausX >= QuadratX And MausX <= QuadratX + Quadratseite Then
		If MausY >= QuadratY And MausY <= QuadratY + Quadratseite
			Quadratseite = Rand(10, 100)
			QuadratX = Rand(0, 800 - Quadratseite)
			QuadratY = Rand(0, 600 - Quadratseite)
		End If
	End If
	DrawRect QuadratX, QuadratY, Quadratseite, Quadratseite
	Flip
Until AppTerminate()
```

Übungsaufgabe 8.4:

```
SeedRnd(MilliSecs())
AppTitle = "Fang' mich! Version 2 mit Mausklick"
Graphics 800, 600
Quadratseite = Rand(10, 100)
QuadratX = Rand(0, 800 - Quadratseite)
QuadratY = Rand(0, 600 - Quadratseite)
Gefangen = 0

Repeat
	Cls
	MausX = MouseX()
	MausY = MouseY()
	MausKlick = MouseHit(1)
	If MausX >= QuadratX And MausX <= QuadratX + Quadratseite Then
		If MausY >= QuadratY And MausY <= QuadratY + Quadratseite
			If Gefangen = 0 And MausKlick Then Gefangen = 1
		End If
	End If

	If Gefangen = 1 Then
		DrawText "Gefangen!", 380, 10
		If Not MouseDown(1) Then
			Quadratseite = Rand(10, 100)
			QuadratX = Rand(0, 800 - Quadratseite)
			QuadratY = Rand(0, 600 - Quadratseite)
			Gefangen = 0
		End If
	End If

	DrawRect QuadratX, QuadratY, Quadratseite, Quadratseite
	Flip
Until AppTerminate()
```

Übungsaufgabe 8.7:

```
Graphics 800, 600
Durchmesser = 100
KreisX = 400 - Durchmesser/2
KreisY = 300 - Durchmesser/2
Repeat
	Cls
	If KeyDown(KEY_LEFT) And KreisX > 0 Then KreisX = KreisX - 2
	If KeyDown(KEY_RIGHT) And KreisX < 800 - Durchmesser Then KreisX = KreisX + 2
	If KeyDown(KEY_UP) And KreisY > 0 Then KreisY = KreisY - 2
	If KeyDown(KEY_DOWN) And KreisY < 600 - Durchmesser Then KreisY = KreisY + 2
	Leertaste = KeyHit(KEY_SPACE)
	If Leertaste Then SetColor Rand(0, 255), Rand(0, 255), Rand(0, 255)
	DrawOval KreisX, KreisY, Durchmesser, Durchmesser
	Flip
Until AppTerminate()
```

Übungsaufgabe 9.3:

```
SeedRnd MilliSecs()
Anzahl:Int = 0
Summe:Float = 0
Durchschnitt:Float = 0
Abweichung:Float = 0
Repeat
	ZufallsZahl:Int = Rand(1, 999)
	Summe = Summe + ZufallsZahl
	Anzahl = Anzahl + 1
	Durchschnitt = Summe / Anzahl
	Abweichung = Durchschnitt - 500
Until Abweichung < 0.001 And Abweichung > -0.001
Print "Anzahl der Zufallszahlen:"
Print Anzahl
Print "Durchschnitt:"
Print Durchschnitt
```

Übungsaufgabe 9.5:

```
Satz:String = "Sechs Hessen vergessen in Essen ihr Essen mit Messern zu fressen"
Wortanfang:Int = 1
Leerzeichen:Int = Instr(Satz, " ", Wortanfang)
While Leerzeichen > 0
	Wort:String = Mid(Satz, Wortanfang, Leerzeichen - Wortanfang)
	Print Wort
	Wortanfang = Leerzeichen + 1
	Leerzeichen = Instr(Satz, " ", Wortanfang)
Wend
LetztesWort:String = Right(Satz, Len(Satz) + 1 -  Wortanfang)
Print LetztesWort
```

Übungsaufgabe 9.9:

```
SuperStrict
Local Kommazahl:Float[Rand(500, 1000)]
For Local i:Int = 0 Until Kommazahl.length
	Kommazahl[i] = Rnd(0, 1)
Next
Kommazahl.sort()
Print "Kleinste: " + Kommazahl[0]
Print "Größte: " + Kommazahl[Kommazahl.length-1]
```

Übungsaufgabe 9.10:

```
SuperStrict
Local ArrayA:Int[5], ArrayB:Int[5]
For Local i:Int = 0 To 5
	ArrayA[i] = Rand(1, 10)
	ArrayB[i] = Rand(1, 10)
Next
For Local a:Int = 0 To 4
	For Local b:Int =  0 To 4
		If ArrayA[a] = ArrayB[b] Then
			Print "ArrayA[" + a + "] = ArrayB[" + b + "] = " + ArrayA[a]
		End If
	Next
Next
```

Übungsaufgabe 9.12:

```
SuperStrict

Graphics 600, 400

Local Quadratseite:Int = 50
Local Spalten:Int = 600 / Quadratseite
Local Reihen:Int = 400 / Quadratseite

Local Rot:Int[Spalten, Reihen], Gruen:Int[Spalten, Reihen], Blau:Int[Spalten, Reihen]
For Local i:Int = 0 Until Spalten
   For Local j:Int = 0 Until Reihen
      Rot[i,j] = Rand(80, 160)
      Gruen[i,j] = Rot[i,j]
      Blau[i,j] = Rot[i,j]
   Next
Next

Repeat

   Local Maus:Int[] = [MouseHit(1), MouseX(), MouseY()]

   For Local x:Int = 0 Until Spalten
      For Local y:Int = 0 Until Reihen
         If Maus[0] Then
            If Maus[1] >= x*Quadratseite And Maus[1] < (x+1)*Quadratseite Then
               If Maus[2] >= y*Quadratseite And Maus[2] < (y+1)*Quadratseite Then
                  Rot[x,y] = Rand(0, 255)
                  Gruen[x,y] = Rand(0, 255)
                  Blau[x,y] = Rand(0, 255)
               End If
            End If
         End If
         SetColor Rot[x,y], Gruen[x,y], Blau[x,y]
         DrawRect x*Quadratseite, y*Quadratseite, Quadratseite, Quadratseite
      Next
   Next

   Flip
Until KeyDown(KEY_ESCAPE) Or AppTerminate()
```