Lua-Tutorial

aus dem Forumbeitrag
„Anleitungen und Tutorials“
von Goetz

PDF-Version
von stephan

```
print ("zusammen" .. "schreiben")
-- ergibt
'zusammenschreiben'
```

Lua Tutorial

Von Goetz, 16. Mai 2019 in Anleitungen und Tutorials

Vorwort

Obwohl man Lua und alle die anderen Alternativen Programmier**sprachen** nennt, heißt "Programmieren lernen" nicht, dass man eine Sprache lernen muss. Das wird leider oft so gesagt und ich finde es sehr irreführend.

Die wenigen Worte, die man für Lua oder andere Programmiersprachen benötigt, stammen alle aus der englisch Umgangssprache. Und es sind zu einem großen Teil sogar die selben Worte in allen Programmiersprachen. Für den Zweck, den Lua im 3D MBS erfüllen wird, kommt man anfangs wahrscheinlich mit einem guten Dutzend Schlüsselwörtern aus. Das ist nicht mehr, als man für einen Urlaub lernt. *("Bitte", "Danke", "Wo geht es zum Bahnhof" und "Letztes Jahr hat der Rotwein besser geschmeckt")*

Wenn Programmierer von sprachlichen Unterschieden reden, dann meinen sie z.B. folgendes:
Um eine Fallunterscheidung zu treffen, schreibe ich in Lua:

```
if Wert > 1 then
```

In der Programmiersprache Python stünde:

```
if Wert > 1:
```

Und in der Sprache Java stünde:

```
if (Wert > 1) {
```

Die Worte unterscheiden sich nicht. Sondern manche Zeichen. Mal steht ein "then" am Ende, mal ein Doppelpunkt und mal steht etwas in Klammern. Man nennt diese Regeln "Syntax". Auf diese Schreibweisen muss man achten, denn Fehler in der Syntax führen unweigerlich dazu, dass ein Programm nicht funktioniert. Ihr werdet also in diesem Tutorial die Syntax von Lua kennenlernen.

Diese Syntax kann man sich ohne Anstrengung ganz nebenbei aneignen. Anfangs schaut man sie nach und mit der Zeit bleibt sie von selbst hängen.

Man muss diese Schreibweisen also nicht lernen. Schon gar nicht pauken. Überhaupt gibt es nichts zu pauken. Vergesst das schulische Prinzip des Auswendiglernens. Es hilft euch beim Programmieren kein bisschen. Mitdenken und Verstehen sind die richtigen Mittel. Probieren und Fehler machen sind die besten Lehrmeister.

Also probiert **alles** aus. Ohne Ziel. Einfach, um Erfahrungen zu sammeln. Probiert und schaut dabei **genau** hin, was passiert. Wenn etwas falsch läuft, dann **ärgert euch nicht**! Schaut genau hin, was falsch läuft. Was passiert da? Wie hängt das, was passiert, mit dem zusammen, was in eurem Skript steht? Aus diesen Beobachtungen lernt ihr am allermeisten. Ich hasse unsere Schuldbildung dafür, dass sie einem das nicht beibringt. In der Schule sind Fehler ein Makel. Man muss sie vermeiden. Und wenn sie passieren, dann versucht man sie zu vertuschen, zu rechtfertigen oder schön zu reden. Das ist dumm, denn damit verschenkt man etwas sehr, sehr wertvolles.

Ich werde im Tutorial eine Menge kleiner, unsinniger Beispiele geben. Mit Eisenbahn haben die alle nichts zu tun. Aber sie werden euch die Mechanismen eines Computerprogramms verständlich machen. Und wenn ihr die durchschaut, dann ist es sehr leicht diese Dinge im 3D MBS anzuwenden. Also probiert diese Beispiele aus. Und lasst euch weitere, ähnliche Beispiele einfallen um zu testen, ob wirklich alles so funktioniert wie ihr denkt.

Die schnellste und einfachste Art, Lua Code auszuprobieren, findet ihr hier:
http://www.lua.org/demo.html

Das ist die offizielle Testseite der Entwickler von Lua. Dort könnt ihr Code eingeben und laufen lassen.

Wenn ihr ins Fenster `6 * 7` schreibt und dann unter dem Fenster auf den Knopf "run" klickt, dann öffnet sich

darunter ein zweites Fenster mit einer Fehlermeldung. Denn man kann in Lua nicht einfach eine Berechnung schreiben. Man muss Lua auch sagen, was es mit dem Ergebnis tun soll.

Schreibt ihr oben **print(6 * 7)** rein, dann steht nach einem Klick auf den "run" Knopf unten das Ergebnis: **42**

Probiert es aus ;-)

Funktionen - Teil 1

Mit dem **print()** aus dem Vorwort habt ihr eine Lua-**Funktion** benutzt. Das ist ein Block aus Befehlen, der einen eigenen Namen bekommt und den man komplett ausführen kann, indem man den Namen dieses Blocks schreibt. Das Besondere dabei ist, dass man beim Aufruf Daten übergeben kann, die für die Ausführung verwendet werden. Im Beispiel war das die Rechenaufgabe **6 * 7**. Die Funktion **print()** funktioniert auch mit anderen Daten richtig. Mit **17 + 4** zum Beispiel. Und mit **2 - 1** ebenso. **Schon probiert? :-)**

Beim Programmieren entwirft man Verhaltensmuster. Diese Muster bekommen einen Namen, damit man sie später anhand dieses Namens aufrufen kann. Das ist unabhängig von der verwendeten Sprache immer dasselbe Prinzip. Diese Arbeitsweise muss man durchschauen.

Die Funktion **print()** ist in Lua schon fertig definiert. Und sie kann noch viel mehr als nur das Ergebnis einer einfachen Berechnung ausgeben. Aber das soll hier noch nicht Thema sein.

Ich möchte euch gleich zu Anfang zeigen, wie ihr Funktionen selber bauen könnt. Damit fange ich diesen Kurs zwar in der Mitte an und nicht vorne. Aber ihr habt schneller die Möglichkeit, selbst zu experimentieren.

In der Programmierung werden Funktionen **definiert** und **aufgerufen**. Bei der Definition legt man fest, was getan werden soll. Der Aufruf sagt, dass es jetzt getan werden soll. Eine Funktion ist ein bisschen so wie ein Einkaufszettel. Zuhause schreibe ich auf, was gekauft werden soll und schreibe groß oben drüber ALDI. Und im Laden krame ich dann den ALDI Zettel hervor und lege all das, was da drauf steht, in den Einkaufswagen.

Um eine Funktion zu **definieren**, beginnt ihr mit dem Schlüsselwort **function**. Bitte achtet **genau** auf die Schreibweise! Hinter dem Schlüsselwort kommt der Name eurer Funktion und zuletzt zwei runde Klammern. Also so:
function Beispiel()

Die nachfolgenden Zeilen sind dann die Befehle, die beim Aufruf der Funktion ausgeführt werden sollen. Und weil es mehrere Zeilen sein können, benötigen wir zum Schluss etwas, um das Ende dieses Blocks zu kennzeichnen. Dafür benutzt man in Lua das Schlüsselwort **end**

Man könnte also sagen, dass
function Beispiel()

end
den Rahmen für die Funktion bildet. Dazwischen kommt die eigentliche Funktion. Also das, was beim Aufruf der Funktion passieren soll.

Die Klammern hinter dem Funktionsnamen sind der Platz für die Daten, welche beim Funktionsaufruf mitgegeben werden. Und weil man diese Daten beim Entwurf der Funktion noch nicht kennt, schreibt man dort bei der **Funktionsdefinition** Platzhalter hinein. Den Namen dieser Platzhalter kann man genauso selbst bestimmen wie den Funktionsnamen.

Nehmen wir für ein Beispiel an, dass Lua nicht das Quadrat einer Zahl bilden könnte. Dann würde man sich selbst eine Funktion dafür schreiben:

```
function Quadrat(Zahl)
    Ergebnis = Zahl * Zahl
end
```

Die Namen haben dabei keine Bedeutung. Man könnte ebensogut schreiben:

```
function Hinz(Kunz)
    Dingsbums = Kunz * Kunz
end
```

und die Funktion würde exakt dasselbe tun. Aber die Schlüsselworte **function** und **end** sind erforderlich und müssen **genau** so geschrieben werden. Dafür gibt es keine Alternativen.

Bei den Funktions- und Platzhalternamen gibt es für die Schreibweise ein paar strikte Regeln, die man einhalten muss:
Nur **Buchstaben, Ziffern und den Unterstrich** verwenden, sonst nichts!
Also keine Leerzeichen, keine Sonderzeichen und möglichst auch keine speziellen Buchstaben wie Umlaute oder das ß
Und der Name darf **nicht mit einer Ziffer beginnen**.
Dieser Name wäre erlaubt:
mein_Test_1

Diese Namen wären alle **nicht** erlaubt:
1.Test
Test-Funktion
Anteile in %
$Vorname
Nachname@Beispiel

Ansonsten muss man nur darauf achten, dass man den einmal gewählten Namen **exakt** beibehält.

Man muss also beim letzten Beispiel beim Funktions**aufruf** Hinz(3) schreiben, um das Quadrat von 3 ausrechnen zu lassen. Die Schreibweise hinz(3) wird zu einer Fehlermeldung führen, weil keine Funktion mit dem Namen hinz (mit klein geschriebenem h am Anfang) definiert wurde.

Funktionen - Teil 2

Die Funktion **Quadrat()** aus dem ersten Teil hat zwar das Quadrat einer Zahl ausgerechnet. Aber das Ergebnis war nirgendwo zu sehen. **Habt ihr es ausprobiert?**
Wenn man im Lua Demo Fenster folgendes eingibt

```
function Quadrat(Zahl)
    Ergebnis = Zahl * Zahl
end

Quadrat(3)
```

dann wird die Funktion in den ersten drei Zeilen definiert und in der letzten Zeile mit der Zahl 3 aufgerufen. Der Aufruf führt dazu, dass die Funktion die Zahl 3 mit sich selbst multipliziert und das Ergebnis an der Stelle des Platzhalters **Ergebnis** abspeichert. Das funktioniert alles perfekt. Man sieht nur nichts davon, weil nirgendwo steht, dass Lua das Ergebnis anzeigen soll.

Aber ihr kennt die Funktion, mit der Lua etwas anzeigt. Sie heißt **print()**
Und den Funktions**aufruf** **print()** kann man in die Funktions**definition** **Quadrat()** mit reinschreiben:

```
function Quadrat(Zahl)
    Ergebnis = Zahl * Zahl
    print(Ergebnis)
end

Quadrat(3)
```

Bitte, versucht anhand dieses Beispiels zu verstehen, was eine Funktions**definition** und was ein Funktions**aufruf** ist. Das ist essenziell wichtig für alles andere.
In den oberen vier Zeilen wird die Funktion **Quadrat()** **definiert**. In einer dieser Zeilen steht der **Aufruf** der Funktion **print()**. Dieser Aufruf wird an dieser Stelle aber noch nicht ausgeführt. Erst, wenn die Funktion **Quadrat()** in der letzten Zeile **aufgerufen** wird und somit alle Zeilen aus der Definition dieser Funktion abgearbeitet werden, wird auch die **print()** Zeile ausgeführt.

Ich versuche es mal mit dem Einkaufszettel als Vergleich:
Auf meinen Einkaufszettel hat meine Liebste mir als Letztes geschrieben: "Schau auf dein Handy!". Diese Zeile lese ich erst, wenn ich schon bei ALDI bin. Kurz vor der Kasse (vor dem **end**) . Und auf dem Handy finde ich dann eine SMS mit den Dingen, die ihr noch eingefallen sind als ich schon unterwegs war.

Der Vergleich hinkt natürlich. Aber er macht vielleicht die Reihenfolge verständlicher, in der die Dinge passieren? Sie hat den Funktions**aufruf** "aufs Handy schauen" schon zuhause auf den Zettel geschrieben. Aber ich führe ihn erst aus, wenn ich den ganzen Zettel ALDI abarbeite.

Diese Unterteilung von Code in mehrere kleine Funktionen hat eigentlich nur organisatorische Gründe. Man könnte ebenso gut alles, was im Programm passieren soll, in ellenlangem Text untereinander schreiben. Das haben wir zu Zeiten des Commodore C64 mit dem eingebauten Basic gemacht.

Ganz platt gesagt ist die Funktion in allen modernen Programmiersprachen eine bessere - wirklich **viel** bessere - Alternative zum alten goto.

Dort, wo das Schlüsselwort **function** steht, wird alles nachfolgende bis zum zugehörigen **end** übersprungen. Und dort, wo der Funktions**aufruf** (also nur der Name ohne vorangestelltes **function**) steht, springt das Programm dorthin, wo die Funktion **definiert** ist. Dann führt es alles bis zum **end** aus und kehrt zuletzt automatisch dorthin zurück, wo die Funktion aufgerufen wurde.

Wenn man dieses Prinzip beherrscht, dann kann man damit seinen Code sehr gut organisieren. Dabei ist es ratsam, die einzelnen Funktionsblöcke möglichst klein zu halten. Ich habe vor einer Weile mal den Satz gehört: "Wenn du fünf Zeilen Code hast, dann lohnt sich vielleicht schon eine Funktion." Behaltet den mal im Hinterkopf.

Funktionen - Teil 3

Für Testzwecke ist es nützlich, wenn man das Ergebnis einer Funktion ausgeben kann. Aber das will man nur in Ausnahmefällen. Meistens will man mit den berechneten Zahlen etwas anderes tun. Man möchte sie weiterverwenden.

Vielleicht benötige ich in meinem Programm an mehreren Stellen das Quadrat einer Zahl? Dann würde ich am liebsten überall dort, wo ich das benötige, einfach **Quadrat(x)** hinschreiben, wobei anstelle des **x** immer die Zahl steht, von der ich gerade das Quadrat benötige. Und dann soll Lua bitte dort, wo ich das so schreibe, einfach das Quadrat dieser Zahl einsetzen.

Also:
Irgendwo in meinem Code soll so etwas wie Geschwindigkeit = Quadrat(12) stehen und Lua soll daraus bitte Geschwindigkeit = 144 machen.

Lua kann das.Man kann in der Funktions**definition** sagen, dass Lua etwas zurückgeben soll und dann wird dieser zurückgegebene Wert dort eingesetzt, wo der Funktionsaufruf steht. Für diesen Zweck gibt es das Schlüsselwort **return**.

```
function Quadrat(Zahl)
    Ergebnis = Zahl * Zahl
    return Ergebnis
end

Geschwindigkeit = Quadrat(12)
print(Geschwindigkeit)
```

Und so geht es auch:

```
function Quadrat(Zahl)
    return Zahl * Zahl
end

print( Quadrat(12) )
```

Im Skripttext wird der Funktionsaufruf natürlich nicht durch das Ergebnis ersetzt. Das wäre nicht hilfreich.

Aber bei der Ausführung des Skripts kann das Ergebnis einer Funktion direkt dort weiterverarbeitet werden, wo die Funktion aufgerufen wurde. Und damit erspart man sich eine Menge Platzhalter um Werte von hier nach dort zu bekommen.

Das war nun in drei Teilen eine Menge Theorie zu den Funktionen. Der eine oder andere wird das erst einmal verdauen müssen. Lasst euch aber bitte nicht entmutigen, wenn ihr nicht sofort alle Zusammenhänge durchschaut. Das wird im Folgenden alles immer klarer werden, weil ihr es jetzt in vielen Übungen, die ich euch geben werde, immer wieder anwenden könnt.

Schleifen - Teil 1

Die Tochter hat gerade Multiplikationsreihen in der Schule. Ich soll ihr beim Lernen helfen. Aber das ist für mich schon sooo lange her. Außerdem konnte ich mir die Siebener-Reihe nie wirklich merken. Und so schwere Sachen wie die Elfer-Reihe hatten wir früher gar nicht in der Schule. Zum Glück habe ich einen Computer, der mir alles ausrechnen kann. Damit müsste ich doch eine Art Spickzettel hinkriegen?

Multiplikation kann ich ja schon:

```
print(1 * 1)
```

Aber das reicht nicht. Der Lehrer will das immer in ganzen Sätzen hören und nicht nur das Ergebnis. Also muss **print()** einen Text ausgeben.
1 mal 1 ist 1
2 mal 1 ist 2
und so weiter.

Deshalb muss ich zuerst lernen, wie ich **print()** dazu bringe, ein Wort als **Wort** zu erkennen. Im letzten Kapitel hatte **print()** ja zum Beispiel Quadrat(12) als **Funktionsaufruf** erkannt. Was macht man nun, wenn **print()** zum Beispiel
das Ergebnis von Quadrat(12) ist 144
schreiben soll?
Man setzt das, was als Text ausgegeben werden soll, in Anführungszeichen

```
print( "das Ergebnis von Quadrat(12) ist", Quadrat(12) )
```

Beachtet bitte, dass zwischen dem Text und dem Funktionsaufruf ein Komma steht. Dieses **Komma trennt zwei Argumente** voneinander. Diesmal bekommt **print()** mehrere Bausteine und setzt sie selbständig zu einer Textzeile zusammen. Dabei wird anstelle des Kommas ein Tabulatorsprung gesetzt. Deshalb steht die 144 bei der Ausgabe etwas weiter weg.
Versucht es mit einer einfachen Multiplikation:

```
print(1, "mal", 1, "ist", 1*1)
```

Hurra, das funktioniert! Mit mehreren Kommas und einem Gemisch aus Zahlen und Texten. Das eröffnet ganz neue Möglichkeiten. Die Abstände sind bei der Ausgabe sehr groß, aber das soll egal sein. Es geht jetzt nicht um Schönheit. Und man darf die Kommas nicht einfach weglassen, weil unterschiedliche Bausteine (Zahlen, Texte und Berechnungen) zu einem Text zusammengesetzt werden sollen. Man muss die Bausteine also von einander trennen.

Das muss jetzt in eine Funktion, die ihr diesmal mit **zwei** Argumenten aufrufen könnt. Nämlich mit den beiden Zahlen, die multipliziert werden sollen. Also benötigt ihr in der Funktionsdefinition diesmal **zwei** Platzhalter. Und die werden ebenfalls **mit einem Komma voneinander getrennt**

```
function Multiplikation(Zahl1, Zahl2)
    print(Zahl1, "mal", Zahl2, "ist", Zahl1 * Zahl2)
end
```

Sehr gut. Jetzt könnt ihr die Funktion mit jedem Zahlenpaar aufrufen

```
Multiplikation(6, 7)
```

Auch hier ist das **Komma** wieder **der Trenner zwischen den beiden Werten**. (Dezimalzahlen werden in allen Programmiersprachen mit einem Dezimal**punkt** geschrieben!)
Jetzt muss ein Weg her, um die Funktion zehnmal hintereinander aufzurufen und dabei als ersten Parameter die Zahlen 1 bis 10 zu übergeben. Damit eine komplette Siebener-Reihe ausgegeben wird.

Eine Methode ist die **for ... do Schleife**. Sie ist ähnlich wie eine Funktion aufgebaut. Aber die Schlüsselwörter sind **for, do** und - wie bei der Funktion - **end**

```
for Zahl = 1, 10, 1 do
    print(Zahl)
end
```

gibt nacheinander die Zahlen 1 bis 10 aus.

Diese Schleife beginnt damit, dass sie die erste der drei Zahlen an den Platzhalter übergibt, den ich Zahl genannt habe. Damit wird dann - wie bei einer Funktion - alles bis zum **end** durchgeführt. Aber anders als eine Funktion startet die Schleife im Anschluss gleich noch einmal. Dabei wird die dritte Zahl zu dem, was im Platzhalter steht, hinzu addiert. Dann läuft der ganze Block erneut durch. Und das immer wieder, bis der Platzhalter den zweiten Wert erreicht hat.
for Zahl = 1, 10, 1 do heißt also:
Mach alles folgende mit allen Zahlen von 1 bis 10 bei einem Zählabstand von 1.
for x = 2, 20, 2 do ginge also auch, wenn man nur gerade Zahlen verwenden will.
Und ebenso funktioniert for a = 1, 2, 0.1 do wenn man kleinere Abstände braucht.

Den gebräuchlichsten Abstand 1 muss man nicht zwingend angeben. Daher kann man den ganzen Code für die Siebener-Reihe so schreiben:

```
function Multiplikation(n1, n2)
    print(n1, "mal", n2, "ist", n1 * n2)
end

for i = 1, 10 do
    Multiplikation(i, 7)
end
```

Beachtet bitte, dass die **for … do Schleife** im Aufbau zwar wie eine Funktions**definition** aussieht, aber im Gegensatz zur Funktionsdefinition **sofort** ausgeführt wird!

Ich habe in diesem letzten Beispiel gebräuchlichere Namen für die Platzhalter verwendet. Den Buchstaben n für eine ganzzahlige Zahl. Das n steht für "number", zu Deutsch: Nummer. Und den Buchstaben i für den Wert, der in der Schleife raufgezählt wird. Das i steht in dem Fall für "increment", zu Deutsch: raufzählen.
Diese Namen werdet ihr oft in fertigen Skripten sehen, deshalb wollte ich euch damit vertraut machen. Aber sie haben keine Funktion, wie ihr an meinen ersten Beispielen sehen konntet. Sie werden einfach nur häufig verwendet. Für den Anfang ist es besser, aussagekräftige Namen zu verwenden. Dann wisst ihr morgen noch, was ihr da heute hingeschrieben habt und warum.

Die Wahrheit, die ganze Wahrheit und nichts als die Wahrheit

Programmierer stellen sich immer wieder dieselbe Frage: *"Kann das wahr sein?"*

Und Computer kennen darauf nur eine klare Antwort:
Etwas ist entweder wahr oder falsch. Im Englischen heißt es **true** bzw. **false** und das sind die Schlüsselworte, welche dafür in Lua verwendet werden.

Jeder Vergleich, den man in Lua schreibt, hat **true** oder **false** als Ergebnis.

Probiert es aus!

Schreibt in Lua Demo

```
print(2 > 1 )
```

Das bedeutet **2 ist größer als 1**.

Drückt auf "run" und im unteren Fenster wird **true** stehen

Und wenn ihr es mit

```
print(2 < 1)
```

versucht, was für **2 ist kleiner als 1** steht, dann ist das Resultat **false**.

Folgende Vergleiche gibt es in Lua:

```
a > b  -- a ist größer als b, kurz: a größer b
a >= b -- a ist größer als b oder gleich b, kurz a größer gleich b
a < b  -- a kleiner b
a <= b -- a kleiner gleich b
a == b -- a gleich b
a ~= b -- a ist nicht gleich b, kurz: a ungleich b
```

Beim letzten Vergleich steht eine Welle, Tilde genannt, vor dem = Zeichen.
Diese Welle bekommt man mit AltGr und der + Taste (im großen Tastaturteil. Nicht das Plus im Zahlenblock.)

Beim vorletzten Vergleich achtet bitte auf das **doppelte** Gleich-Zeichen. Man muss bei der Programmierung nämlich zwei Dinge klar unterscheiden:
Manchmal möchte man **prüfen**, ob etwas gleich ist. Dann benutzt man in Lua das **doppelte** = Zeichen.
Und manchmal möchte man etwas **gleichsetzen**. Dann muss man in Lua das **einfache** = Zeichen verwenden.

```
a == 5 -- prüft, ob a den Wert 5 hat

a = 5 -- setzt a auf den Wert 5
```

Auf beiden Seiten des Vergleichs können auch Rechenoperationen stehen.

Ich kann also prüfen, ob

```
a == 2 + 3
```

oder ob

```
2 * a <= b + c
```

ist. Meist geht es aber um ganz banale Fragen wie die, ob ein Zähler schon einen bestimmten Wert erreicht hat.

Schleifen - Teil 2

Es gibt eine zweite Art, Code in einer Schleife wiederholt auszuführen:
Die **while ... do Schleife**.

Hinter while erwartet Lua dabei ein **true** oder **false**. Dieses **true** oder **false** kann das Ergebnis eines Ausdrucks sein. Meistens handelt es sich dabei um das Ergebnis eines Vergleichs.

```
a = 0
while a < 5 do
  print(a)
  a = a + 1
end
print("fertig!")
```

In der ersten Zeile wird für den Platzhalter a ein Wert festgelegt. Diese Platzhalter nennt man übrigens **Variable** und es wird Zeit, dass ich den richtigen Begriff verwende. Sie heißen deshalb Variable, weil ihr Wert veränderbar ist. Er kann *variiert* werden.

Also:
Die erste Zeile setzt die Variable a auf den Wert 0

Das **while** am Anfang der zweiten Zeile heißt übersetzt so etwas wie "**solange**". Dahinter steht ein Ausdruck, der entweder wahr oder falsch sein kann. Solange a < 5 wahr ist, also der Ausdruck das Ergebnis **true** hat,

werden die nachfolgenden Zeilen bis zum **end** immer wieder ausgeführt. Hinter den Ausdruck, der geprüft werden soll, gehört noch das Schlüsselwort **do**, zu deutsch: **tu es**. Damit erkennt Lua, dass hier der gesamte Ausdruck, welcher geprüft werden soll, zu Ende ist. (Andere Programmiersprachen nutzen für diesen Zweck Klammern.)

Die dritte und vierte Zeile werden also nur ausgeführt, wenn a kleiner als 5 war.
In der dritten Zeile wird der Wert von a ausgegeben.

Und in der vierten Zeile wird zum Wert von a eine 1 hinzu addiert. Das Ergebnis wird als neuer Wert wieder in a gespeichert. Wenn a vorher den Wert 0 hatte, dann steht jetzt in a eine 1. Diesen Trick findet man in allen Programmiersprachen. Man kann eine Variable auslesen. etwas addieren (oder die Zahl auf andere Weise ändern) und das Ergebnis wieder in der **selben** Variable speichern. Das ist so praktisch, dass es in anderen Programmiersprachen sogar besondere Schreibweisen dafür gibt. Aber leider nicht in Lua. Da muss man wirklich a = a + 1 schreiben, wenn man den Wert von a um 1 erhöhen will.

Das **end** in der fünften Zeile heißt, dass der **while**-Block zu Ende ist. Damit kehrt das Programm zurück zur zweiten Zeile und prüft erneut, ob a kleiner als 5 ist. Da a jetzt den Wert 1 hat, ist es noch immer kleiner als 5. Also wird erneut in der dritten Zeile der Wert von a ausgegeben. Und in der vierten Zeile wird zum Wert von a wieder 1 hinzu addiert. Das wäre also 1 + 1. Dieser neue Wert 2 wird in a gespeichert. Dann wird wieder geprüft, ob a kleiner als 5 ist. Das passiert solange, bis a diese Bedingung nicht mehr erfüllt. Dann springt Lua ans Ende des **while**-Blocks und macht mit der sechsten Zeile weiter. In der steht, dass Lua den Text fertig! ausgeben soll.

Versucht mal **vor** dem Ausprobieren dieser Zeilen **möglichst genau** einzuschätzen, was im Ausgabefenster stehen wird.

Und wenn das tatsächliche Ergebnis nicht 100%ig mit eurer Vermutung übereinstimmt, dann zuckt nicht einfach mit den Achseln und denkt: 'Na, immerhin. War doch fast richtig.'
Sondern schaut **ganz genau** hin, **was** anders ist. Und macht euch Gedanken darüber, **warum** es anders ist. Das ist die Methode, mit der man Programmieren lernt. Alles, was man ins Skript schreibt, hat ganz eindeutige und sehr logische Konsequenzen. Ohne Ausnahme! Deshalb ist jede Abweichung im Verhalten ein guter Hinweis auf das, was man zur Zeit noch falsch versteht. Und je genauer man diese Abweichung studiert, desto besser erkennt man, wo der eigene Denkfehler sitzt.

Ihr könnt auch ausprobieren, was sich ändert wenn man die dritte und vierte Zeile vertauscht.
Macht das einen Unterschied? Wenn ja, welchen? Und warum?

Viel Spaß beim Probieren und Studieren :-)

Schleifen - Teil 3

Schleifen gehören zu den wichtigsten Hilfsmitteln in der Programmierung. Sie nehmen einem viel Arbeit ab und gerade im Zusammenspiel mit einer Software wie dem 3D-Modellbahnstudio sind das genau die Arbeiten, welche überhaupt den Einsatz eines Skripts rechtfertigen.

Nehmt zum Beispiel einen Schattenbahnhof mit zehn Gleisen. Ein eintreffender Zug soll in eins dieser Gleise einfahren. Dazu muss geprüft werden, welche Gleise noch frei sind, um dann eine Fahrstraße zu einem dieser freien Gleise zu legen.

Zunächst muss zehnmal dasselbe getan werden. Die Prüfung, ob ein Gleis besetzt ist. Und bei jedem der zehn Durchläufe muss genau ein Parameter geändert werden: Die Gleisnummer.

Für solche Zwecke ist die **for ... do** Schleife gedacht.
Zwischen die Schlüsselworte **for** und **do** schreibt man den **Namen** einer Variablen, ihren **Anfangswert**, ihren **Schlusswert** und den **Abstand**, in dem man von Anfang bis Schluss zählen möchte. Also zum Beispiel so etwas:

```
for Gleis = 1, 10, 1 do
```

Beachtet bitte das = Zeichen hinter der Variablen für die Zuweisung der Werte und die Kommas für die Trennung von Anfangswert, Schlusswert und Schrittweite.

In den folgenden Zeilen steht dann das, was mehrmals hintereinander ausgeführt werden soll und am Ende des Blocks steht, wie bei Lua üblich, das Schlüsselwort **end**.

Ich weiß noch nicht, wie man in V5 prüft, ob ein Gleis besetzt ist. Deshalb bleibe ich bei meinem bisherigen Prinzip, dass ich ersatzweise einfach etwas Text ausgebe:

```
for Gleis = 1, 10, 1 do
  print("Ich prüfe, ob Gleis "..Gleis.." besetzt ist.")
end
```

Wenn ihr diese drei Zeilen bei Lua-Demo eingebt und laufen lasst, dann steht im Ausgabefenster die Textzeile zehnmal untereinander. Dabei steht an Stelle der Variablen Gleis in jeder Zeile eine andere Nummer. Das ist die Zahl, welche bei jedem neuen Durchlauf der Schleife an die Variable Gleis übergeben wurde.

Beim Programmieren muss man **sehr genau** auf das Verhalten solcher Methoden achten. Man muss zum Beispiel sehen, dass der Anfangs- und der Schlusswert beide enthalten sind. Die Schleife wird also mit den Zahlen 1 bis 10 **inklusive** durchgeführt.

Muss man so etwas wissen? **Nein!** Muss man so etwas lernen? Auswendig können? Nein, auch nicht. Das ginge gar nicht. Es gibt zu viele Unterschiede, die man kennen müsste.

Wenn man programmiert, dann muss man ein Bewusstsein für diese Fragen entwickeln. Dann muss man kurz innehalten und denken:
"for ... do könnte sich so oder so verhalten. Es könnte inklusive oder exklusive zählen. **Ich habe keine Ahnung, wie das bei Lua ist.***"*
Und dann **probiert** man es kurz aus. Wie einfach man solche Dinge testen kann, seht ihr im obigen Beispiel.

Hier ist ein weiteres Beispiel:
Was passiert, wenn man den Wert der Variablen innerhalb der Schleife ändert? Was bedeutet das für den nächsten Durchlauf?
Keine Ahnung! Also **probiere** ich, was passiert:

```
for Gleis = 1, 10, 1 do
  Gleis = Gleis + 2
  print("Ich prüfe, ob Gleis "..Gleis.." besetzt ist.")
end
```

Werden jetzt Zahlen **übersprungen**?
Oder werden **alle** Zahlen um 2 erhöht und dann ausgegeben?

Das muss man nicht auswendig wissen. Das ist auch solange uninteressant, bis man einen Grund hat, den Wert der Variablen innerhalb der Scheife zu ändern. Aber **dann** will man es **zuverlässig** wissen. Also muss man sich jetzt bewusst machen, dass es zwei mögliche Verhaltensweisen gibt. Und man muss ausprobieren, welches von beiden das tatsächliche Verhalten ist.

Programmieren ist nichts, was man auswendig lernt und dann nach Schema abspult. Programmieren ist eine sehr kreative Beschäftigung. Ständig muss man sich etwas einfallen lassen. Und immer mit der Neugier eines Detektivs betrachten, was **genau** passiert. Deshalb kann man Programmieren auch nur lernen, indem man sich **aktiv** damit beschäftigt.

Welcher Typ bist du?

Es gibt ein paar Typen, die lernt jeder Programmierer ziemlich früh kennen. Über die müssen wir jetzt mal reden.

true und **false** sind zum Beispiel solch ein Typ.
Diesen Typ nenn man **boolean**, nach dem Mathematiker George Boole.

Daten des Typs boolean können nur zwei verschiedene Werte haben. Entweder **true** oder **false**. Man benutzt diesen Typ überall dort, wo man zwischen zwei Zuständen unterscheiden möchte: an oder aus, frei oder besetzt, auf oder zu, ja oder nein …

Alles, was im Programmcode eine Entscheidung trifft, erwartet den Typ **boolean** als Argument. Das **while** aus dem letzten Kapitel ist ein Beispiel dafür. Entweder wird die **while** Schleife wiederholt oder nicht. Je nachdem, ob das Argument wahr oder falsch, also **true** oder **false** ist.

Zahlen werden in zwei Typen unterschieden.

1. die ganzen Zahlen. Die nennt man in der Programmierung **integer**
2. die Fließkommazahlen. Die nennt man in der Programmierung **float**

Der Grund für die Unterscheidung liegt in der unterschiedlichen Genauigkeit bei Berechnungen. Das erkläre ich hier aber nicht im Detail, weil Lua diese Unterscheidung gar nicht macht. Lua kennt nur **einen** Typ für alle Zahlen und nennt diesen Typ **number**.

Die größtmögliche ganze Zahl in Lua ist **9223372036854775807**. Wenn man in einem Video bei 0 anfängt und dann bei 60 fps in jedem Bild 1 weiter zählt, dann erreicht man diese Zahl nach 4.8 Milliarden Jahren!

Rechnet es nach!

```
grosse_Zahl = 9223372036854775807
Bilder_pro_Jahr = 365 * 24 * 60 * 60 * 60

print("In einem Jahr kommen", Bilder_pro_Jahr, "Bilder zusammen")
print("Deshalb benötigt man", grosse_Zahl / Bilder_pro_Jahr, "Jahre, bis man die
größtmögliche Zahl in Lua erreicht hat.")
```

Texte sind ein weiterer Typ. Dieser Typ nennt sich **string**. Das ist das englische Wort für eine lange Aneinanderreihung von Dingen. „A string of pearls“ ist beispielsweise eine Perlenkette (und ein bekanntes Musikstück von Glen Miller.) Texte betrachtet der Programmierer also als eine Kette von Buchstaben, Zahlen und Satzzeichen.

Strings werden in einfache oder doppelte Anführungszeichen gesetzt. Beide haben genau dieselbe Funktion.

```
'Ich bin ein String'

"Ich bin auch ein String"
```

Strings müssen mit denselben Anführungszeichen anfangen und aufhören.
Das jeweils andere Anführungszeichen kann dann im String als Teil des Textes verwendet werden.

```
'Ich bin ein unvollständiger String"

"Ich bin ein 'besonderer' String"
```

Man darf nur die **simplen** Anführungszeichen verwendet, nicht die schönen für den gepflegten Schriftsatz. Das ist unproblematisch, wenn man Code direkt in einem dafür vorgesehenen Editor schreibt. Der setzt automatisch die erlaubten Anführungszeichen. Aber wenn man eine Textverarbeitung benutzt, dann kann man sich falsche Anführungszeichen einhandeln. Zum Beispiel die bei deutschen Texten gebräuchlichen Anführungszeichen unten. Die versteht der Lua-Interpreter nicht!

```
„Ich führe zu einer Fehlermeldung!“
```

Aber auch die oben stehenden Anführungszeichen sind in Textverarbeitungen oft die falschen. Nämlich die schnörkeligen, leicht schräg gestellten. Die sehen den simplen Anführungszeichen zwar sehr ähnlich. Aber sie haben einen ganz anderen Buchstabencode. **Deshalb funktionieren die nicht im Programmcode.**

Mit der Funktion **type()**, die in Lua enthalten ist, kann man den Typ eines Elements prüfen. Diese Funktion gibt einen String zurück, der den Typ des angegebenen Elements als Wort enthält. Mit **print()** kann ich diesen Typ daher auch ausgeben:

```
print(type(a))
```

Ergibt als Antwort

```
nil
```

Das ist der englische Ausdruck für: **nix**

Wenn man der Variablen a keinen Wert zugewiesen hat, dann hat sie einen Inhalt vom Typ **nil**.

Das ist ein ungewöhnliches Konzept, welches man so in anderen Programmiersprachen nicht findet. In anderen Sprachen hätte es eine Fehlermeldung gegeben die lautet, dass die Variable noch gar nicht definiert ist. In Lua ist alles, was nicht anderweitig definiert wurde, automatisch **nil**

```
a = true
print(type(a))

-- ergibt die Antwort
'boolean'

a = 1
print(type(a))

-- ergibt die Antwort
'number'
```

und

```
a = '1'
print(type(a))

-- ergibt die Antwort
'string'
```

Es gibt noch mehr Typen. Aber auf die möchte ich später eingehen.

Mir ist für den Anfang etwas anderes wichtiger:
Es gibt verschiedene Typen, weil man damit im Programm verschiedene Dinge anstellt.

numbers kann man addieren.

strings kann man aneinanderfügen.

Und **booleans** kann man logisch verknüpfen.

Das lässt sich nicht beliebig mischen. Der Ausdruck

```
"Hallo" + 123
```

ist unsinnig und führt entsprechend in Lua zu einer Fehlermeldung. Das + Zeichen ist nur für die mathematische Addition. Es wird nicht dafür benutzt, Textteile zusammenzusetzen.

Um Textteile zusammenzusetzen, benutzt man kein +, sondern zwei Punkte:

```
print("zusammen" .. "schreiben")

-- ergibt

'zusammenschreiben'
```

Weil man Typen unterschiedlich behandeln muss, war es notwendig, dass ich euch an dieser Stelle ein paar von diesen Typen vorstelle. Und die sehen nur am Anfang furchterregend aus. Eigentlich sind die alle sehr nett. Wirklich. Die tun euch nichts. Vorausgesetzt, ihr behandelt sie richtig.

Entscheidungshilfen

Ein Computerprogramm muss oft eine Entscheidung treffen.
Und für die Entscheidung muss es eine Bedingung prüfen.

Falls das Signal Halt zeigt, muss der Zug angehalten werden
Falls die Weiche auf Abzweig steht, darf das Signal nur „Fahrt mit 40“ anzeigen

Falls es ein Güterzug ist, muss er in den Güterbereich einfahren
Falls alle Gleise belegt sind, muss das Einfahrsignal auf „Halt“ bleiben

Für solche Fälle gibt es in Lua die **if … then** Konstruktion.
Zwischen das Schlüsselwort if (zu Deutsch: falls) und das Schlüsselwort **then** (zu Deutsch: dann) kommt die Bedingung
In den Zeilen darunter steht, was getan werden soll, falls die Bedingung erfüllt ist.
Und der Block wird, wie alle Blöcke in Lua, mit dem Schlüsselwort **end** abgeschlossen.

Eine Bedingung kann nur entweder erfüllt sein oder nicht. Etwas anderes als diese zwei Möglichkeiten gibt es in der Programmierung nicht.
Deshalb erwartet Lua zwischen dem **if** und dem **then** ein boolean. Also **true** oder **false**, oder etwas, dass **true** oder **false** ergibt. Zum Beispiel einen Vergleich. Die Schlüsselworte **if** und **then** bilden die Klammer um die Bedingung

```
if Test > 1 then
   print("Die Variable Test enthielt einen Wert, der größer als 1 war")
end
```

Lua kann immer nur einen **einzelnen** Vergleich durchführen, keine Kette von mehreren Vergleichen.

```
3 > Test > 1
```

ist in Lua **nicht** erlaubt.

Daher muss man oft mehrere Bedingungen logisch verknüpfen. Dafür gibt es die Schlüsselworte **and** (zu Deutsch: und) sowie **or** (zu Deutsch: oder)

```
if (Test > 1) and (Test < 3) then
    print("Der Wert von Test liegt zwischen 1 und 3")
end
```

Ich habe die beiden Vergleiche oben in Klammern gesetzt. Das wäre in Lua nicht zwingend erforderlich. Denn die Vergleichsoperatoren binden stärker als das **and**. (Es ist dasselbe Prinzip wie die vertraute Regel: Punktrechnung vor Strichrechnung).

Das **and** bedeutet, **beide** Bedingungen müssen erfüllt sein. Test muss größer als 1 **und** kleiner als 3 sein, damit man sagen kann: Test liegt zwischen 1 und 3.
Mit dem and werden also zwei Elemente, die entweder **true** oder **false** sind, miteinander verknüpft. Das Ergebnis ist ein neues, **einzelnes** **true** oder **false**. Und das Ergebnis der Verknüpfung ist nur dann **true**, wenn **beide** Elemente **true** sind. Für das if bedeutet das: Es sieht nur ein **einzelnes** **true** oder **false**. Nämlich **das Ergebnis** der Operation zwischen dem **if** und dem **then**.

Daher kann man diesen boolean Wert auch in einer Variablen zwischenspeichern, wenn man möchte:

```
Bedingung = Test > 1 and Test < 3
if Bedingung then
    print("Der Wert von Test liegt zwischen 1 und 3")
end
```

In vielen Fällen ist es sinnvoll, das Ergebnis eines Vergleichs in einer Variablen zu speichern. Dann kann man diese Variable direkt als Bedingung verwenden. Das macht den Code lesbarer. Und man hat es leichter, wenn man die Bedingungen ändern möchte. Die stehen jetzt nämlich **alleine** rechts vom = Zeichen und nicht irgendwo mitten zwischen **if** und **then**.
Wenn man die Bedingung vorab ermittelt und in einer Variablen ablegt, dann kann man sie auch in mehrere kleine Happen unterteilen:

```
Bedingung_1 = Test > 1
Bedingung_2 = Test < 3
Bedingung = Bedingung_1 and Bedingung_2
if Bedingung then
    print("Der Wert von Test liegt zwischen 1 und 3")
end
```

Im obigen Beispiel ist das natürlich vollkommen übertrieben. Aber im Ernstfall kann das helfen, den Überblick zu bewahren wenn es z.B. darum geht, ob ein Zug

1. ein Güterzug ist,
2. der an einem "Halt" zeigenden Signal ankommt
3. und nur dann "Fahrt" bekommen darf,
4. wenn das für ihn vorgesehene Gleis frei ist
5. und alle Weichen richtig gestellt wurden.

Man muss diese Dinge mit so lächerlich einfachen Beispielen üben. Nur dann hat man später im Ernstfall die nötige Routine, um alles richtig zu machen. Man macht die erste Fahrstunde aus gutem Grund auf einem leeren Parkplatz und nicht mitten im Berufsverkehr.

2b or not 2b

fragte sich ein Postbote im englischen *Stratford upon Avon* im Jahre 1584 beim Versuch, die handgeschriebene, unleserliche Adresse auf einem Brief zu entziffern. Damit inspirierte er nicht nur den jungen William Shakespeare zu seinem berühmten Monolog, sondern legte zugleich auch den Grundstein für viele Programmiersprachen.

Neben dem **and** aus dem letzten Kapitel sind **or** und **not** die anderen zwei Operationen, die man benötigt um Bedingungen zu verknüpfen.

Das englische Wort **or** entspricht dem deutschen **oder**. Die Verknüpfung zweier Elemente mit **or** bedeutet, dass **mindestens** ein Element wahr sein muss, aber auch mehr als ein Element wahr sein darf. Es ist **kein entweder ... oder**.

```
Gleis_1_frei or Gleis_2_frei
```

ist also auch dann **true**, wenn **beide** Variablen **true** sind.

Ein *entweder ... oder* (man nennt das ein exklusives oder) gibt es in Lua nicht.

Das Wort **not** entspricht dem deutschen Wort **nicht**. Mit **not** kann man ein boolean ins Gegenteil verwandeln. Es macht aus **true** ein **false** und umgekehrt.

Wenn man zum Beispiel nur dann das Gleis 3 benutzen will, wenn Gleis 1 besetzt ist, dann könnte die Prüfung so aussehen

```
if Gleis_3_frei and not Gleis_1_frei then
```

Das not bindet stärker als die Vergleichsoperatoren. Wer das Ergebnis eines Vergleichs umkehren möchte, der muss den Vergleich in Klammern setzen

```
not (a == b)
-- ist dasselbe wie
a ~= b
```

Die Bedeutung von **and**, **or** und **not** ist einfach zu verstehen. Aber daraus die logisch richtige Kombination zu bilden, fällt manchmal auch erfahrenen Programmierern schwer. Deshalb werde ich in den nächsten Tagen ein paar sehr einfache Übungen dazu vorstellen.

Für Fortgeschrittene!

Die logischen Operationen **and** und or liefern nicht immer ein boolean. Mit **and** bekommt man das zweite Element zurück, wenn beide Elemente wahr sind. Mit **or** bekommt man das erste wahre Element zurück.
Dazu muss man wissen, dass **jeder** Wert, der in einer Variable gespeichert ist, in Lua als wahr gilt. Nur **nil** und **false** gelten als falsch.
Eine Zahl ist also **true**. Und im Gegensatz zu anderen Programmiersprachen wertet Lua **auch die 0** als **true**.
Ein String ist ebenfalls **true**. Auch ein Leerstring "".

Das Ergebnis von true and 1 ist **1**, nicht true!
Das Ergebnis von 0 or 1 ist **0**
Das Ergebnis von 0 and 1 ist **1**, nicht false!
Das Ergebnis von Test or 1 ist 1, wenn Test noch keinen Wert hat bzw. noch nicht deklariert ist.
Das Ergebnis von Test or 1 ist der Inhalt von Test, wenn dieser Inhalt nicht nil oder false ist.
Das Ergebnis von "Toby" or not "Toby" ist "Toby"
Das Ergebnis von 2b or not 2b ist eine Fehlermeldung, weil kein Variablenname mit einer Zahl beginnen darf!

Wenn das Ergebnis einer Verknüpfung feststeht, dann werden die restlichen Elemente nicht mehr geprüft. Das bedeutet, dass Lua bei einem **and** aussteigt, wenn das erste Element **false** ist. Das Ergebnis der **and** Operation kann nicht mehr **true** sein, wenn ein Element **false** ist. Wird das zweite Element durch einen Funktionsaufruf gebildet, dann findet dieser Funktionsaufruf in diesem Fall nicht statt.
Bei einem **or** steigt Lua entsprechend beim ersten wahren Element aus. Ist das erste Element **true**, dann wird eine Funktion, die das zweite Element liefern soll, nicht mehr ausgeführt.

Ich glaube, dass das in vielen anderen Programmiersprachen genauso gehandhabt wird.

Wenn man das versteht, dann kann man es sich zunutze machen. Wenn man es ignoriert, dann beißt es einen irgendwann in den Hintern ;-)

An dieser Stelle würde ich gerne ein paar kleine Übungen anregen.

Denn das Konzept einer Funktion sowie die Auswertung von **true** und **false** sind für das Programmieren elementar wichtig.

Die Übungen stammen alle von **Nick Parlante**s Website Codingbat
Ich habe sie nur von Englisch nach Deutsch und von Python nach Lua übersetzt.

Die erste Aufgabe „Ausschlafen“ lautet wie folgt:
Schreibe eine Funktion, die anhand der beiden Argumente Werktag und Urlaub zurückgibt, ob man ausschlafen kann oder nicht. Man muss nur die Funktion definieren. Und der Rahmen ist schon vorgegeben.

```
function Ausschlafen(Werktag, Urlaub)

end
```

Beim Aufruf werden die Variablen Werktag und Urlaub jeweils mit true oder false befüllt. An Werktagen darf man nur im Urlaub ausschlafen. Wenn kein Werktag ist, dann darf man immer ausschlafen.

Die Aufgabe ist, zwischen die erste und letzte Zeile der Funktionsdefinition ein wenig Lua Code zu schreiben. Dieser soll Werktag und Urlaub auswerten und dann soll die Funktion je nach Ergebnis der Auswertung entweder ein true oder false zurückgeben.

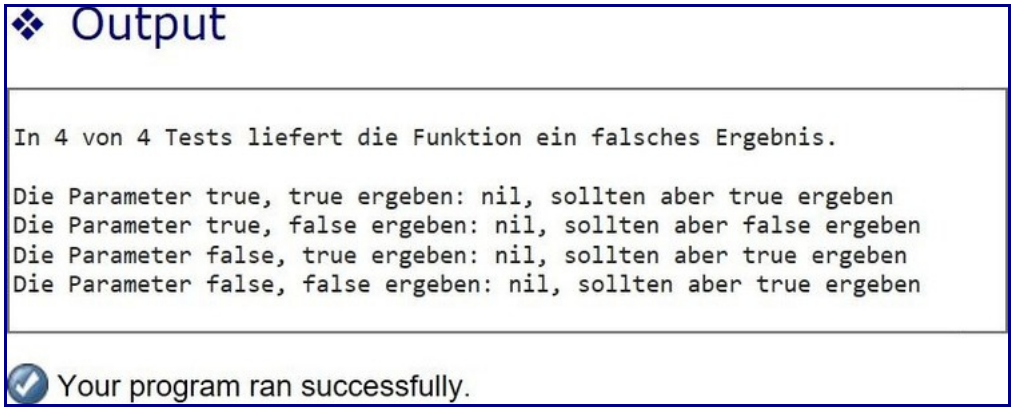
Man muss nur die Funktionsdefinition vervollständigen. Im Anhang füge ich ein kleines Testskript bei. Das kann man im oberen Teil entsprechend ergänzen und dann komplett in Lua-Demo, ZeroBrane, Notepad++ oder einem anderen Programm laufen lassen. Der untere Teil des Testskripts ist eine kleine Prüfroutine welche euch zeigt, ob eure Lösung funktioniert oder nicht. Diesen Teil dürft ihr bitte nicht verändern.

Es wäre schön, wenn ihr mir Lösungsvorschläge **nur als persönliche Nachricht** schickt und sie **nicht öffentlich** postet. So hat jeder die Chance für sich zu überlegen, wie er die Aufgabe lösen möchte.

Ausschlafen.zip

Lösungsansätze zur Aufgabe "Ausschlafen"

Wenn man mein Testskript **unverändert** in das Lua-Demo Fenster kopiert, dann bekommt man diese Antwort:



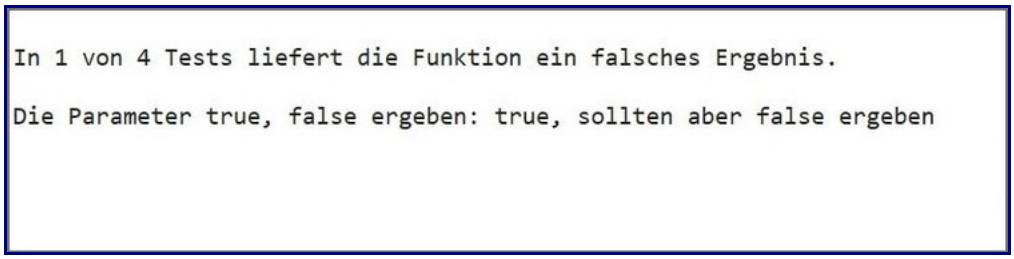
Das Programm wird fehlerfrei ausgeführt. Es enthält keine Syntaxfehler. Aber die Funktion tut noch nicht das, was sie tun soll. Weil die Funktion völlig leer ist, gibt sie auch nichts zurück. Und genau das steht in der Auflistung. Viermal ergeben die Parameter **nil**.

Sie sollten aber in drei Fällen **true** und in einem Fall **false** ergeben.

Was passiert, wenn man in die Funktion einfach **return true** schreibt?

```
function Ausschlafen(Werktag, Urlaub)
    return true
end
```

Dann ist das Ergebnis nur noch in einem von vier Fällen falsch.



Mit **return** kann man also dafür sorgen, dass die Funktion beim Aufruf etwas zurück gibt.
Jetzt muss man dafür sorgen, dass sie abhängig von den beiden Argumenten Werktag und Urlaub das richtige zurück gibt.

Wenn Urlaub wahr ist, dann muss die Funktion ein **true** zurückgeben. Denn dann kann man ausschlafen.
Das könnte so aussehen

```
function Ausschlafen(Werktag, Urlaub)
  if Urlaub then
    return true
  end
end
```

Beachtet bitte, dass man in der **if ... then** Zeile keinen Vergleich Urlaub == true benötigt. Denn dieser Vergleich liefert nur das, was in Urlaub schon drin steht. Ist Urlaub wahr, dann ist auch das Ergebnis des Vergleichs wahr. Ist Urlaub falsch, dann ist auch das Ergebnis des Vergleichs falsch. Der zusätzliche Vergleich erfordert zusätzliche Rechenzeit, bringt aber keinen Nutzen.

Wenn man diese Funktion verwendet, dann ist das Ergebnis in zwei von vier Fällen falsch.

```
In 2 von 4 Tests liefert die Funktion ein falsches Ergebnis.

Die Parameter true, false ergeben: nil, sollten aber false ergeben
Die Parameter false, false ergeben: nil, sollten aber true ergeben
```

Die Funktion gibt **nil** zurück, wenn Urlaub nicht wahr ist.
Ist Urlaub wahr, dann ist das Ergebnis der Funktion richtig.

Nun kann man den Werktag prüfen:

```
function Ausschlafen(Werktag, Urlaub)
  if Urlaub then
    return true
  end
  if Werktag then
    return false
  end
end
```

Damit ist das Ergebnis schon in drei von vier Fällen richtig.

```
In 1 von 4 Tests liefert die Funktion ein falsches Ergebnis.

Die Parameter false, false ergeben: nil, sollten aber true ergeben
```

Nur wenn **kein** Urlaub und **kein** Werktag ist, lautet das Ergebnis noch **nil**. Denn weder die erste, noch die zweite Bedingung ist erfüllt. Deshalb gibt die Funktion nichts zurück. Sie soll in diesem Fall aber **true** zurückgeben, weil man ausschlafen kann wenn kein Werktag ist.

```
function Ausschlafen(Werktag, Urlaub)
  if Urlaub then
    return true
  end
  if Werktag then
    return false
  end
  return true
end
```

In dieser Form besteht die Funktion alle 4 Tests

```
Die Funktion hat alle 4 Tests bestanden.
```

Dazu muss man wissen, dass ein **return** in einer Funktion dazu führt, dass die Funktion **komplett** verlassen wird. Alles, was nach dem **return** kommt, ignoriert Lua wenn die return Zeile ausgeführt wird.
Deshalb wird der Werktag nicht mehr geprüft, wenn Urlaub wahr ist.
Und deshalb wird die letzte Zeile nicht mehr ausgeführt, wenn Urlaub falsch, aber Werktag wahr ist.

Das bedeutet, dass bei dieser Konstruktion die Reihenfolge wichtig ist. Wenn man sie vertauscht, dann erhält man zum Teil falsche Ergebnisse.

```
function Ausschlafen(Werktag, Urlaub)
  if Werktag then
    return false
  end
  if Urlaub then
    return true
  end
  return true
end
```

Bei dieser Reihenfolge würde der Urlaub nicht mehr geprüft, wenn Werktag wahr ist.

```
In 1 von 4 Tests liefert die Funktion ein falsches Ergebnis.

Die Parameter true, true ergeben: false, sollten aber true ergeben
```

Der Wecker klingelt jetzt auch im Urlaub an Werktagen. Denn die erste Prüfung hat an Werktagen zur Folge, dass die Funktion sofort mit einem **return false** verlassen wird

Jetzt kehrt die Prüfung bei Werktag bitte einmal ins Gegenteil. Testet, ob **kein** Werktag ist.

```
function Ausschlafen(Werktag, Urlaub)
  if not Werktag then
    return true
  end
  if Urlaub then
    return true
  end
  return false
end
```

Das heißt, dass jetzt **true** zurückgegeben wird, wenn kein Werktag ist. Und wenn kein Werktag ist, dann ist es egal, ob Urlaub ist oder nicht.
Und **false** wird nur dann zurückgegeben, wenn weder die erste, noch die zweite Bedingung erfüllt ist.
Bei dieser Variante sind die beiden Prüfungen austauschbar.
Und außerdem kann man sie jetzt in einer Prüfung zusammenfassen.

```
function Ausschlafen(Werktag, Urlaub)
  if not Werktag or Urlaub then
    return true
  end
  return false
end
```

Und weil der gesamte Term **not** Werktag **or** Urlaub jetzt genau das ergibt, was wir als Antwort haben wollen, muss man auch keine **if**-Prüfung mehr durchführen. Man gibt einfach das Ergebnis der logischen Verknüpfung zurück

```
function Ausschlafen(Werktag, Urlaub)
  return not Werktag or Urlaub
end
```

Manche setzen den Term hinter dem return in Klammern, weil andere Programmiersprachen das so fordern. In Lua ist das nicht erforderlich, aber erlaubt.

Ich weiß, dass das alles abstrakt und für Einsteiger eine harte Nuss ist.

Deshalb poste ich in den nächsten Tagen weitere Übungen dieser Art. Denn diese kleinen Beispiele enthalten im Kern genau das, was man auch für die Nutzung von Lua im MBS brauchen wird. Funktionen, die anhand von Parametern eine Entscheidung treffen und dann das eine oder andere tun.

Es wird vermutlich Funktionen geben, die fertig definiert sind und nur mit geeigneten Parametern aufgerufen werden. So wie das bekannte print(). Und es wird vermutlich die Möglichkeit geben, aus der EV eine Funktion aufzurufen, die man an anderer Stelle selbst definiert. Ich weiß das nicht. Aber es ist sehr naheliegend, dass es so sein wird.

Deshalb halte ich es für sinnvoll euch anhand kleiner Beispiele das grundlegende Prinzip zu zeigen.
Und ich hoffe natürlich, dass ihr auch ein bisschen Spaß dabei habt.

Die zweite Aufgabe **„Affentheater“** ist der ersten sehr ähnlich.
Diesmal geht es um die Affen Toto und Kaya, welche uns gerne Schwierigkeiten bereiten. Aber man kann es ihnen ansehen. Wenn **beide grinsen** oder wenn **keiner von beiden** grinst, dann stecken wir in Schwierigkeiten. Grinst einer der beiden Affen, aber der andere nicht, dann haben wir nichts zu befürchten.
Der Funktionsrahmen sieht so aus

```
function Schwierigkeiten(Kaya_grinst, Toto_grinst)

end
```

Die Funktion wird wieder mit zwei Parametern aufgerufen werden, die beide entweder **true** oder **false** sein können. Und sie soll **true** zurückgeben, wenn wir in Schwierigkeiten stecken oder **false**, falls nichts zu befürchten ist.

Das Prinzip ist also dasselbe wie bei der vorherigen Aufgabe. Die möglichen Kombinationen aus **true** und **false** sollen nur diesmal zu anderen Ergebnissen führen. Deshalb muss man die beiden Parameter anders auswerten als beim ersten Mal.

Im Anhang findet ihr wieder ein Skript mit angehängter Testroutine für die Prüfung eurer Lösungsansätze.

Ich würde mich sehr freuen, wenn ihr mir eure Lösungsvorschläge per PN schickt.

Affentheater.zip

Lösungsvorschläge für das Affentheater

In dieser Aufgabe war, genau wie in der ersten, gefordert, dass die Funktion etwas zurückgeben soll. Und zwar soll sie **true** zurückgeben, wenn beide Parameter **true** sind. Wenn also beide Affen grinsen. Und die Funktion soll auch dann **true** zurückgeben, wenn beide Parameter **false** sind. Wenn also beide Affen **nicht** grinsen.

Das kann man genau so in die Funktion schreiben:

```
function Schwierigkeiten(Kaya_grinst, Toto_grinst)
    if Kaya_grinst and Toto_grinst then
        return true
    end
    if not Kaya_grinst and not Toto_grinst then
        return true
    end
    return false
end
```

Und weil die erste und die zweite Bedingung dieselbe Konsequenz haben, kann man sie auch in einer Bedingung zusammenfassen.

```
function Schwierigkeiten(Kaya_grinst, Toto_grinst)
    if (Kaya_grinst and Toto_grinst) or (not Kaya_grinst and not Toto_grinst) then
        return true
    end
    return false
end
```

Beachtet bitte, dass in der Mitte ein **or** steht und **kein** **and**. Denn es reicht ja, wenn **einer** der beiden geklammerten Ausdrücke wahr ist. Umgangssprachlich würde man in der Mitte ein "und" benutzen:
"Wenn beide Affen grinsen gib ein true zurück und wenn beide nicht grinsen, dann auch."
Aber logisch wäre ein **and** die falsche Wahl, weil es bedeutet, dass **beide** Bedingungen **zugleich** erfüllt sein müssen.

Weil jetzt der gesamte Ausdruck hinter dem if genau das ergibt, was wir als Antwort brauchen, kann man ihn auch direkt zurück geben:

```
function Schwierigkeiten(Kaya_grinst, Toto_grinst)
    return Kaya_grinst and Toto_grinst or not Kaya_grinst and not Toto_grinst
end
```

Dabei kann man sogar die Klammern weglassen, weil das **and** stärker bindet als das **or** und das **not** stärker bindet als das **and**.
Das heißt: Zuerst führt Lua die beiden **not** Operationen durch, dann die **and** Operationen und zuletzt die **or** Operation.

Diese Schreibweise ist aber furchtbar unleserlich.
Die erste Version mit den zwei einzelnen Prüfungen finde ich in diesem Punkt deutlich angenehmer. Deshalb habe ich überlegt, ob man die nicht noch etwas vereinfachen kann.

Nehmen wir den Fall, dass Kaya grinst. Dann ist das gewünschte Ergebnis **true**, wenn Toto_grinst **true** ist. Und es ist **false**, wenn Toto_grinst **false** ist. Wir können also auf die and Verknüpfung verzichten und gleich Toto_grinst zurückgeben, falls Kaya_grinst wahr ist

```
if Kaya_grinst then
    return Toto_grinst
end
```

Und wenn man auch für den Fall etwas definieren möchte, dass die Bedingung hinter dem **if** **nicht erfüllt** ist, dann gibt es in Lua dafür das Schlüsselwort **else** (zu Deutsch: "andernfalls" oder "sonst")
Und was soll andernfalls passieren? Was ist als Ergebnis gefordert, wenn Kaya **nicht** grinst?
Falls Toto_grinst jetzt **true** ist, dann ist das Ergebnis **false**. Falls aber Toto_grinst auch **false** ist und somit beide Affen **nicht** grinsen, dann ist das Ergebnis **true**. Für den Fall, dass Kaya **nicht** grinst, ist das Ergebnis also **das Gegenteil** von Toto_grinst. Es ist not Toto_grinst

```
function Schwierigkeiten(Kaya_grinst, Toto_grinst)
    if Kaya_grinst then
        return Toto_grinst
    else
        return not Toto_grinst
    end
end
```

Verblüfft?

Man kann sich aber auch ganz vom genauen Wortlaut der Aufgabe lösen und stattdessen schauen, was wirklich gefordert ist: Wenn beide Affen dasselbe tun, dann ist das Ergebnis wahr, sonst nicht.

Dann müsste es doch auch genügen, wenn man die beiden Parameter vergleicht.

```
function Schwierigkeiten(Kaya_grinst, Toto_grinst)
    return Kaya_grinst == Toto_grinst
end
```

Das funktioniert, ist wirklich gut lesbar und außerdem schön schlank.
Vielen Dank @Andy für diese sehr schöne Einsendung.

Ich hoffe, dass mit dieser zweiten Aufgabe noch deutlicher wird, was schon bei der ersten Aufgabe durchschien: Dass es nicht darum geht, für **eine** Aufgabe **eine** richtige Lösung zu finden. Und sich diese Lösung dann für die Zukunft zu merken. Das funktioniert einfach nicht.

Es geht vielmehr darum, **möglichst viele** dieser Lösungsbeispiele zu durchschauen. Zu erkennen, was da jeweils im Code passiert und **warum** das funktioniert. Je klarer euch anhand der verschiedenen Beispiele wird, wie die Mechanismen funktionieren, desto leichter wird es euch fallen Skripte zu entwerfen, die zu euren konkreten Bedürfnissen passen.

Und deshalb mag ich diese kleinen Übungen so sehr. Man kann anhand dieser Beispiele wunderbar demonstrieren, welche Denkansätze alle zum Ziel führen. Man kann die verschiedenen Lösungsvorschläge mit der eigenen Lösung vergleichen und vielleicht an der einen oder anderen Stelle denken: "Ah - der Weg ist auch nicht schlecht." Und man kann unterschiedliche Methoden sehr bequem ausprobieren.

Die dritte Aufgabe **„Doppelte Summe“** hat dasselbe Muster wie die vorherigen beiden.
Aber die beiden Parameter beim Aufruf sind diesmal keine booleans (**true** oder **false**), sondern Zahlen.

Die Funktion soll die Summe der beiden Zahlen zurückgeben. Und wenn beide Zahlen gleich sind, dann soll sie die Summe vor der Ausgabe verdoppeln.

```
function Wurf(Zahl_1, Zahl_2)

end
```

Ich habe die Funktion "Wurf" genannt, weil die Aufgabe für ein Würfelspiel geeignet wäre. Der Spieler würde per Knopfdruck zwei Zahlen würfeln und die Funktion, welche ihr hier definiert, errechnet dann das Ergebnis. Ein Pasch zählt dabei doppelt.

Im Anhang findet ihr wieder ein Skript mit angehängter Testroutine, welches ihr für die Prüfung eurer Lösungsansätze verwenden könnt.

Ich freue mich über **jeden** Lösungsvorschlag, den ihr mir als Nachricht schickt.

doppelte Summe.zip

Lösungsbeispiele zu "Doppelte Summe"

Zuerst möchte ich einen Lösungsweg zeigen, den ich häufig sehe. Er setzt den Aufgabentext "buchstäblich" um:

```
function Wurf(Zahl_1, Zahl_2)
  if Zahl_1 == Zahl_2 then
    return (Zahl_1 + Zahl_2) * 2
  else
    return Zahl_1 + Zahl_2
  end
end
```

Wenn die beiden Zahlen gleich sind, dann bilde die Summe, vedopple sie und gib das Ergebnis aus. Andernfalls gib einfach die Summe der beiden Zahlen aus.
Beachtet bitte, dass bei der Verdopplung in der dritten Zeile die Summe in Klammern stehen muss. **In Lua gilt Punktrechnung vor Strichrechnung.** Das heißt, dass ohne die Klammern nur die zweite der beiden Zahlen verdoppelt würde.

Hier ist ein Vorschlag zur Vereinfachung des obigen Beispiels:
In der zweiten Zeile wird geprüft, ob beide Zahlen gleich sind. Und die dritte Zeile wird nur dann ausgeführt, wenn diese Bedingung erfüllt war. Also ist Zahl_1 + Zahl_2 an dieser Stelle dasselbe wie Zahl_1 + Zahl_1 oder 2 * Zahl_1. Deshalb kann man die Rechenoperation in Zeile 3 vereinfachen.

```
function Wurf(Zahl_1, Zahl_2)
  if Zahl_1 == Zahl_2 then
    return 4 * Zahl_1
  else
    return Zahl_1 + Zahl_2
  end
end
```

Ein weiterer Weg wäre der, dass man zuerst eine der beiden Zahlen ändert, falls beide gleich sind. Und dann die Summe bildet.

```
function Wurf(Zahl_1, Zahl_2)
  if Zahl_1 == Zahl_2 then
    Zahl_1 = 3 * Zahl_1
  end
  return Zahl_1 + Zahl_2
end
```

Für das Ergebnis macht dieser unorthodoxe Weg keinen Unterschied. Man kommt damit genauso schnell und sicher zum Ziel wie mit den ersten beiden Wegen. Aber diese Fassung ist schwer nachvollziehbar und deshalb nicht wirklich ratsam. Trotzdem haben mir solche Gedankenexperimente geholfen, Lua und überhaupt das Programmieren besser zu verstehen. Deshalb möchte ich euch gerne dazu animieren, solche verschiedenen Lösungswege durchzuspielen. Mit diesen sehr einfachen Aufgaben, die nur wenige Zeilen Programmcode erfordern, kann man das gut machen.

Die folgende Methode ist populär und **gut lesbar**:

```
function Wurf(Zahl_1, Zahl_2)
  Ergebnis = Zahl_1 + Zahl_2
  if Zahl_1 == Zahl_2 then
    Ergebnis = 2 * Ergebnis
  end
  return Ergebnis
end
```

Diesmal wird das Ergebnis in mehreren Schritten gebildet und in einer neuen Variable namens Ergebnis zwischengespeichert. Lua kann diese Variable gleich in der Funktion bilden. Man muss sie nicht erst deklarieren, wie das vor allem bei Compiler-Sprachen erforderlich ist. Man weist der Variablen einfach einen Wert zu und kann sie anschließend verwenden.

Selbstverständlich kann diese Variable jeden beliebigen Namen haben und muss nicht zwingend Ergebnis heißen. Und genauso kann man übrigens auch die Namen der Funktions**argumente** ändern. Diese Variablen existieren ja nur innerhalb der Funktion.
Man kann das letzte Beispiel also auch so schreiben:

```
function Wurf(a, b)
  c = a + b
  if a == b then
    c = 2 * c
  end
  return c
end
```

Dem einen oder anderen ist diese Schreibweise vielleicht einleuchtender. Weil die Formeln besser zutage treten.

Nur den Namen der Funktion solltet ihr bitte beibehalten, wenn ihr mein kleines Testskript verwenden wollt. Denn das ist der Funktionsname, der weiter unten mit verschiedenen Beispielwerten aufgerufen wird.

Für Fortgeschrittene

In Lua sind Variablen grundsätzlich **global** definiert! Mit Ausnahme der Funktionsargumente, die selbstverständlich lokale Variablen sind.

Möchte man eine lokale Variable erzeugen, dann muss man dafür in Lua das Schlüsselwort **local** benutzen.

Richtig müsste mein letztes Beispiel aus dem vorherigen Posting also lauten:

```
function Wurf(a, b)
  local c = a + b
  if a == b then
    c = 2 * c
  end
  return c
end
```

Eine lokale Variable gilt nur innerhalb der Funktion und wird bei Verlassen gelöscht. Die Verwendung lokaler Variablen innerhalb von Funktionsdefinitionen

1. vermindert das Risiko von Namenskonflikten (weil der Namensbereich auf die Funktion und ihre Unterfunktionen beschränkt ist)
2. spart Speicherplatz (weil der Platz für diese Variable nach Verlassen der Funktion frei gegeben wird)
3. beschleunigt das Programm (weil diese kurzlebigen Variablen im Prozessor-nahen Speicher angelegt werden.)

Für die letzte Aufgabe gibt es noch einen weiteren, interessanten Lösungsweg.
Er macht sich das besondere Verhalten von **and** und **or** zunutze, welches ich im Kapitel **2b or not 2b** angesprochen hatte.

```
function Wurf(a, b)
  return a == b and 4 * a or a + b
end
```

Wenn a == b wahr ist, dann ist das Ergebnis der **and** Verknüpfung der zweite Term 4 * a.
Und weil dieser Term auf jeden Fall wahr ist, wird der Term hinter dem or nicht mehr geprüft, sondern das Resultat der **and** Verknüpfung ausgegeben. Nämlich 4 * a, ganz wie gewünscht.

Ist die erste Bedingung hingegen nicht wahr, dann ist das Ergebnis der **and** Verknüpfung auch nicht wahr. Und in diesem Fall wird der Term ausgegeben, der hinter dem **or** steht. Nämlich a + b.

Dieser Weg ist kein bisschen schneller als die zuvor gezeigten. Das ist **nur eine akademische Spielerei**, die aber mächtigen Spaß machen kann und zu mehr Routine führt.

Differenz21
In dieser Aufgabe hat die zu definierende Funktion nur einen Parameter vom Typ **"number"**. Ich gebe der Variable den Namen n.

Die Funktion soll den absoluten Wert der Differenz zwischen 21 und n zurückgeben. Und falls n größer als 21 ist, soll dieser Wert verdoppelt werden.

Als absoluten Wert bezeichnet man die positive Version einer Zahl.
Ein anderer Ausdruck für den absoluten Wert ist "Betrag".

Der Betrag von 5 ist 5
Der Betrag von -5 ist ebenfalls 5

Beispiele:
Für n = 1 muss die Funktion den Wert 20 ausgeben
Für n = 25 muss die Funktion den Wert 8 (das doppelte des Betrags von -4) ausgeben

```
function Differenz21(n)

end
```

Lua bringt eine vordefinierte Funktion mit, die den Betrag einer Zahl bildet. Diese Funktion heißt **math.abs()** und ist eine von mehreren Funktionen aus der "math" Bibliothek. Deshalb hat sie diesen zweiteiligen Namen mit dem Punkt dazwischen. Dazu erkläre ich in späteren Kapiteln mehr.

Ein Skript mit Prüfroutine hänge ich wieder an dieses Posting an.
Viel Spaß beim Ausprobieren und Studieren.

Differenz21.zip

Lösungsbeispiele „Differenz21“

Beginnen möchte ich wieder mit dem Lösungsweg, der die Aufgabe „buchstäblich“ umsetzt:

```
function Differenz21(n)
  Ergebnis = math.abs(21 - n)
  if n > 21 then
    Ergebnis = Ergebnis * 2
  end
  return Ergebnis
end
```

Zuerst wird die Differenz von 21 und n gebildet und der Betrag dieser Differenz in einer Variablen Ergebnis gespeichert.
Dann wird geprüft, ob n größer als 21 ist.
Falls ja, dann wird der Wert in Ergebnis verdoppelt und wieder in Ergebnis gespeichert.
Zuletzt wird der Wert von Ergebnis ausgegeben.

Es wäre sauberer, Ergebnis als eine lokale Variable anzulegen. Aber mit der globalen Variable funktioniert das Skript ebenso gut. Drum habe ich mehr Wert auf ein leicht lesbares Beispiel gelegt.

Unter anderem sollte dieses Beispiel dazu dienen, die **math.abs()** Funktion vorzustellen. Die ist praktisch und mit einem einfachen Beispiel, wie dem hier gezeigten, leicht zu verstehen. Aber für die gestellte Aufgabe kommt man auch ohne diese Funktion aus.

```
function Differenz21(n)
  Ergebnis = 21 - n
  if n > 21 then
    Ergebnis = Ergebnis * -2
  end
  return Ergebnis
end
```

Das Ergebnis von 21 - n kann ja nur dann negativ sein, wenn n größer als 21 ist. Und wenn man in diesem Fall das Ergebnis sowieso verdoppeln muss, dann kann man es auch mit -2 multiplizieren, um aus dem negativen einen positiven Wert zu machen. Damit entfällt die Notwendigkeit, den Betrag der Differenz zu bilden.

Außerdem ist eine Überlegung wert, ob man den ursprünglichen Wert von n im weiteren Verlauf behalten muss. Falls nicht, dann kann man die Variable n selbst verändern und wieder auszugeben.

```
function Differenz21(n)
  n = 21 - n
  if n < 0 then
    n = n * -2
  end
  return n
end
```

Man spart so eine zusätzliche Variable ein. Und n ist, wie alle Funktionsargumente, eine lokale Variable.

Es ist riskant, den Wert der ursprünglichen Variable zu ändern. Schnell übersieht man dabei, dass man den ursprünglichen Wert später noch benötigt. Im vorliegenden Fall für den Vergleich, ob n größer als 21 ist. Das letzte Beispiel funktioniert nur, weil ich diesen Vergleich durch einen anderen ersetzt habe. Denn wenn n größer als 21 war, dann ist das Ergebnis von 21 - n kleiner als 0.

Alternativ kann man auch erst prüfen, ob n größer als 21 ist und dann entsprechend verzweigen

```
function Differenz21(n)
  if n > 21 then
    return (n - 21) * 2
  end
  return 21 - n
end
```

Durch den Tausch der beiden Werte in der Subtraktion entfällt in der dritten Zeile das negative Vorzeichen vor dem Multiplikator.

Der Einzeiler für Fortgeschrittene kann so aussehen:

```
function Differenz21(n)
  return n > 21 and (n - 21) * 2 or 21 - n
end
```

Ich hoffe, ihr hattet Spaß an diesen Knobeleien?

In den nächsten Kapiteln will ich wieder ein paar Grundlagen zu Lua erklären.

Variablen

Bislang habe ich den Variablen kein eigenes Kapitel gewidmet.

Normalerweise fängt man ein Tutorial damit an, dass man Variablen erklärt. Aber ich weiß, dass manche hier Abneigungen verspüren, wenn dieses Thema aufkommt. Und man müsste Variablen abstrakt und ohne passenden Zusammenhang erklären, wenn man ein Tutorial mit diesem Thema beginnt. Deshalb hatte ich mich für eine andere Reihenfolge entschieden.

In den ersten Übungen habt ihr Variablen schon benutzt. Ihr habt sie kennengelernt, ohne dass ich sie im Detail erklärt habe. Und ihr habt schon den Zusammenhang gesehen, in dem man Variablen einsetzt.

Variablen sind nichts anderes als Platzhalter.
Wenn ich beim Entwurf einer Funktion bestimmte Inhalte noch nicht kenne, dann schreibe ich Platzhalter dorthin, wo später diese Inhalte eingesetzt werden sollen.

Und damit ist im Prinzip schon erklärt, was eine Variable ist. Mehr steckt nicht dahinter.

Man kann Variablen einen beliebigen Namen geben. Der darf aus einem einzelnen Buchstaben bestehen oder ein ganzes Wort sein. Aber der Name darf nur aus Buchstaben, Ziffern und dem Unterstrich bestehen. Und er darf nicht mit einer Ziffer beginnen.
(Ich hatte das in einem früheren Kapitel schon einmal geschrieben. Aber es ist so wichtig, dass ich es hier noch einmal wiederholen möchte.)

Vermeidet spezielle Buchstaben wie z.B. Umlaute oder das ß. Sicher kennt ihr von E-Mails das Problem, dass an Stelle solcher Buchstaben manchmal kryptische Zeichenfolgen auftauchen? Dasselbe könnte euch auch im Programmcode passieren und dann für Probleme sorgen.

Variablen im 3D-Modellbahn Studio

Im MBS gibt es Daten, die man als User gerne auslesen und verwenden möchte. Zum Beispiel die Stellung eines Signals. Oder die Geschwindigkeit einer Lok. Oder der Name des Fahrzeugs auf einem Gleisstück. Damit man in der EV an diese Daten kommt, haben die Speicherplätze für diese Daten eigene Namen. Das sind also Variablen, deren Name schon im MBS festgelegt wurde.

Wenn man davon absieht, dass die Namen dieser Speicherplätze schon festliegen und die Werte in diesen Speicherplätzen vom MBS stammen, unterscheiden sich diese Variablen nicht von solchen, die man selbst deklariert. Es sind Speicherplätze, die einen Namen bekommen haben, damit man sie im Programmcode oder in der EV verwenden kann.

Ein Teil der Schwierigkeiten, welche manche User mit Variablen haben, stammen möglicherweise von den Namen dieser Variablen. Man muss die richtigen Namen kennen. Und das sind vielleicht nicht die Namen, welche man selbst ausgesucht hätte. An dieser Tatsache wird auch Lua nichts ändern.

Wenn eine Person „Peter“ heißt, dann muss ich sie mit „Peter“ ansprechen. Auch, wenn ich finde, dass der Name „Klaus“ viel besser passen würde. Peter fühlt sich nur angesprochen, wenn ich ihn mit seinem Namen anrede.

Welche MBS-Variablen es in V5 geben wird, weiß ich nicht. Aber sie werden eine wichtige Rolle spielen, weil man auch mit Lua die MBS Objekte steuern will. Das geht nur, wenn ich die Namen der einzelnen Objektparameter kenne.

In Lua Skripten werden sowohl Variablen vorkommen, die vom MBS stammen als auch solche, die man selbst definiert. Und die müsst ihr unterscheiden können. Ihr müsst verstehen, wann ihr euch an vorgegebene Namen halten müsst und wann ihr eigene Namen definiert. Das ist nicht schwer. Es muss einem nur bewusst sein, dass es diesen Unterschied gibt.

Variablen für Fortgeschrittene

Lua Variablen können ohne Deklaration Daten jeden Typs speichern. Und man kann auch beim Überschreiben von Werten den Typ ändern. Ich kann beispielsweise eine Zahl mit einem String überschreiben, ohne dass Lua meckert.

Das geht deshalb, weil in den Variablen selbst gar kein Wert steht, sondern nur eine Speicheradresse. Die Variable speichert nur, wo etwas steht aber nicht, was dort steht. Und Adressen sind immer gleich lang. Egal, ob sie auf eine Zahl, einen String oder etwas anderes zeigen.

Variablen sind in Lua immer **global** definiert. Man muss ihnen bei der Deklaration das Schlüsselwort **local** voran stellen, um **lokale** Variablen zu erzeugen.

```
function Test_a()
  a = 1
  return a
end

function Test_b()
  local b = 1
  return b
end

if Test_a() == a then
  print("a ist eine globale Variable")
else
  print("a ist eine lokale Variable")
end

if Test_b() == b then
  print("b ist eine globale Variable")
else
  print("b ist eine lokale Variable")
end
```

Funktionsargumente und Schleifenzähler sind automatisch **lokale** Variablen

Lua erlaubt multiple Zuweisungen

```
a, b = 2, 3
```

Damit lassen sich auch Werte tauschen:

```
a, b = b, a
```

Wenn man mehrere Variablen zugleich deklariert, dann kann man **allen zusammen** einmal das Schlüsselwort **local** voranstellen

```
local a, b, c = 1, 2, 3
```

Tabellen - Teil 1

Variablen bieten die Möglichkeit, **einen** Wert unter einem Namen abzulegen.

Oftmals ist es aber sinnvoll, **mehrere** Werte unter einem einzelnen Namen zusammenzufassen, weil sie in irgendeiner Weise zusammengehören. Koordinaten zum Beispiel. Oder Farbwerte. Oder Personendaten. Oder alle Parameter einer Lok (Name, Geschwindigkeit, Ort …)

Für diesen Zweck bietet Lua Tabellen. Und in Lua gibt es nur Tabellen. Andere Programmiersprachen kennen außerdem Listen, Dictionaries, Tupfes und mehr. Lua fasst das alles in einem **Typ** **'table'** zusammen.

Tabellen werden mit **geschweiften** Klammern erzeugt.

```
Koordinaten = {0, 0, 0}
```

und die Elemente einer Tabelle werden durch ein **Komma** getrennt.

Die **geschweiften** Klammern findet man auf einer normalen, deutschen PC-Tastatur mit **AltGr 7** und **AltGr 0**

Die einzelnen Elemente einer Tabelle sind indiziert.
Das bedeutet, dass sie durchnummeriert sind. Und dass man sie über diese Nummer ansprechen kann.

Den Index eines Tabellenelements schreibt man bei Lua in **eckige** Klammern

```
Koordinaten[1]
```

Die **eckigen** Klammern findet man auf einer normalen, deutschen PC-Tastatur mit **AltGr 8** und **AltGr 9**

Eine Lua Tabelle darf Elemente unterschiedlichen Typs enthalten

```
Person = {"Max", "Mustermann", 182, 3.5, true}
```

Diese Tabelle enthält zwei Strings, eine Ganzzahl, eine Fließkommazahl und ein Boolean.

Das erste Element einer Tabelle hat den Index **1**

```
Person[1] == "Max"
```

Darin unterscheidet sich Lua von allen anderen Programmiersprachen!

Für den Einsteiger ist diese Nummerierung sehr angenehm, weil sie seinen Erwartungen entspricht.
Für routinierte Programmierer ist sie eine Falle, weil sie den bisherigen Gewohnheiten widerspricht.

Für die Lesbarkeit darf man beliebig viele Leerzeichen einfügen.

```
Farbe = {0, 32, 128}
```

und

```
Farbe = {  0,  32, 128}
```

sind identisch.

Ebenso darf man Zeilenumbrüche einfügen um die Lesbarkeit zu verbessern

```
Texte = {
  "Eine Lok steht auf Gleis 1",
  "Der Güterzug wurde abgefertigt",
  "Der Weg für den Schnellzug ist frei",
  "Der Schattenbahnhof ist voll!"
}
```

Das Komma als Trennzeichen zwischen den Elementen ist auch bei mehrzeiliger Schreibweise **zwingend** erforderlich.
Hinter dem letzten Element ist ein Komma optional. Man darf es schreiben, aber Lua verlangt es nicht.

Tabellen - Teil 2

Möchte man wissen, wie viele Elemente sich in einer Tabelle befinden, dann kann man diese Zahl mit einem # vor dem Namen der Tabelle bekommen:

```
Beispiel = {"Eins", "Zwei", "Drei"}
Anzahl = #Beispiel
```

Man kann in Lua eine leere Tabelle anlegen

```
Beispiel = {}
```

Und anschließend Elemente hinzufügen

```
Beispiel[1] = "Eins"
Beispiel[2] = "Zwei"
```

Bei dieser Methode werden keine Elemente verschoben. Eventuell vorhandene Elemente werden überschrieben. Wenn man den Index angibt, unter dem Elemente gespeichert werden sollen, sind Lücken in der Liste möglich

```
Beispiel[5] = "Fünf"
```

#Beispiel gibt in diesem Fall den Index vor der **ersten** Lücke zurück. Das wäre im vorliegenden Beispiel eine **2**, im folgenden Beispiel wäre es eine **5**, obwohl die Elemente 1 und 2 nicht existieren und das letzte Element den Index 7 hat.

```
Beispiel = {}
Beispiel[3] = "Drei"
Beispiel[4] = "Vier"
Beispiel[5] = "Fünf"
Beispiel[7] = "Sieben"

print(#Beispiel)
```

Man kann Tabellen wahlweise nutzen um sortierte, durchnummerierte Listen zu erstellen oder um Elemente an bestimmte Nummern wie z.B. IDs zu knüpfen. Da Lua beides im selben Typ **'table'** verwaltet, muss man selbst darauf achten, die beiden Fälle richtig zu unterscheiden.

Für Änderungen an durchnummerierten Listen gibt es unter anderem die Funktionen

table.insert(Tabelle, Index, Wert) und **table.remove(Tabelle, Index)**

Lässt man bei **table.insert()** den Index weg (**zwei** Parameter statt **drei**), hängt es den neuen Wert dort an, wo in der Liste die erste Lücke gefunden wird. Nachfolgende Elemente verschieben sich nicht! Damit fungiert es als **append**, welches Lua nicht als eigenständige Funktion bietet.

```
Beispiel = {}
Beispiel[3] = "Drei"
Beispiel[4] = "Vier"
Beispiel[5] = "Fünf"
Beispiel[9] = "Neun"

table.insert(Beispiel, "Sechs")

for i = 1, 10 do
  print(i.." - ", Beispiel[i])
end
```

Die Funktion **table.remove()** liefert den entfernten Wert als Ergebnis zurück.

```
Beispiel = {"Eins", "Zwei", "Drei", "Vier", "Fünf", "Sechs"}
for i = #Beispiel-1, 1, -1 do
  table.insert(Beispiel, table.remove(Beispiel, i))
end

for i = 1, #Beispiel do
  print(i.." - "..Beispiel[i])
end
```

Tabellen - Teil 3

Der Index eines Tabellenelements darf in Lua auch ein **String** sein. Man kann die Zellen also mit Namen versehen:

```
Farbe = {}
Farbe["Rot_Anteil"]   = 128
Farbe["Gruen_Anteil"] =  64
Farbe["Blau_Anteil"]  =   0
```

Die Anführungszeichen um den Namen sind erforderlich, damit Lua an dieser Stelle Zellennamen von Variablen unterscheiden kann.

In einer vereinfachten Schreibweise hängt man den Index nach einem Punkt an den Tabellennamen. **Bei dieser Schreibweise dürfen keine Anführungszeichen verwendet werden.**

```
Farbe = {}
Farbe.Rot_Anteil   = 128
Farbe.Gruen_Anteil =  64
Farbe.Blau_Anteil  =   0
```

Das ist gut lesbar und bequemer zu schreiben. In der Bedeutung ist es mit der ersten Schreibweise identisch.

Ebenso kann man namentliche Indexe gleich bei der Initialisierung verwenden:

```
Farbe = {["Rot_Anteil"] = 128, ["Gruen_Anteil"] = 64, ["Blau_Anteil"] = 0}
```

oder in der vereinfachten Schreibweise:

```
Farbe = {Rot_Anteil = 128, Gruen_Anteil = 64, Blau_Anteil = 0}
```

Bitte beachten: Diese vereinfachten Schreibweisen sind **ausschließlich für Strings** geeignet.
Eine **Nummer** als Zellenindex **muss in eckigen Klammern** stehen. Ein Variablename ebenfalls.

Nummeriete und namentlich bezeichnete Zellen dürfen in derselben Tabelle vorkommen:

```
Band = {"John", "Paul", "George", "Ringo", Name = "Beatles"}
print("Die "..Band.Name.." sind: "..Band[1].."", "..Band[2].."", "..Band[3].." und
"..Band[4])
```

Zur Erinnerung: Mit einem # vor dem Tabellennamen bekommt man den Index vor der ersten Lücke im durchnummerierten Teil einer Tabelle.
Im vorliegenden Beispiel liefert #Band also den Wert 4. Die Zelle "Name" wird nicht mitgezählt.

Die Funktionen **table.insert()** und **table.remove()** verlangen als Index eine **Nummer**. Sie können nicht auf namentlich indizierte Zellen angewendet werden.

Möchte man eine Zelle löschen, dann weist man ihr den Wert **nil** zu. Das funktioniert mit jedem Index. Ist der Index eine Zahl, dann werden die nachfolgenden Zellen nicht verschoben und es entsteht eine Lücke in der Nummerierung.

Tabellen - Teil 4

Eine Lua Tabelle kann Elemente vom Typ **'table'** enthalten

```
Beispiel = { {1, 2, 3} , {63, 127, 255} , {true, true, false} }
```

Das Ergebnis hat Ähnlichkeit mit mehrdimensionalen Tabellen.
Die 63 im obigen Beispiel steht an Position Beispiel[2][1], nämlich im zweiten Block an erster Stelle.

Der Vergleich mit landläufigen Tabellen ist aber irreführend. Hilfreicher ist, die Lua-Tabellen einfach als Datensammlung zu verstehen. Sie haben in mancher Hinsicht mehr Ähnlichkeit mit Objekten als mit Tabellen. Deshalb müssen Lua-Tabellen nicht dimensioniert werden. Das heißt, dass man nicht (wie in anderen Sprachen) vorher die Anzahl der Zellen festlegen muss.

```
Bands = {
  Beatles = {"John", "Paul", "George", "Ringo"},
  Stones = {"Mick", "Keith", "Brian", "Ian", "Bill", "Charlie"},
  Who = {"Pete", "Roger", "John", "Keith"}
}
print(Bands.Stones[2])
```

Die Tabelle Bands enthält 3 Elemente, deren Index jeweils ein String ist. Nämlich der Bandname. Jedes der drei Elemente ist ebenfalls eine Tabelle. Die Elemente dieser Untertabellen sind einfache Aufzählungen. Der Index jedes Elements ist also eine Nummer, beginnend mit der Zahl 1 für das erste Element.

Beachtet bitte das **Komma** hinter jeder Untertabelle. Dieses Komma ist erforderlich, um die Elemente Beatles, Stones und Who voneinander zu trennen.

Die Zeilenumbrüche und Einrückungen in der Tabelle sind optional und dienen nur der Lesbarkeit. Man könnte ebenso gut den gesamten Tabelleninhalt in eine lange Zeile schreiben. Oder umgekehrt auf noch mehr Zeilen verteilen:

```
Bands = {
  Beatles = {
    "John",
    "Paul",
    "George",
    "Ringo"
  },
  Stones = {
    "Mick",
    "Keith",
    "Brian",
    "Ian",
    "Bill",
    "Charlie"
  },
  Who = {
    "Pete",
    "Roger",
    "John",
    "Keith"
  }
}
```

Das ist reine Geschmacksache. Die Trennung der Elemente entsteht allein durch die Kommas und die geschweiften Klammern.

Aber aus Erfahrung kann ich empfehlen, Programmzeilen mit mehr als 80 Zeichen Länge zu vermeiden.

Im Übrigen werdet ihr im MBS keine großen Tabellen am Stück erstellen. Dazu besteht kein Grund. Diese Tabellen entstehen auf andere Weise. Aber es wird euch helfen, wenn ihr versteht wie Tabellen in Lua aufgebaut sind. Weil ihr später an einzelne Daten aus solchen Tabellen drankommen wollt. Weil ihr zum Beispiel von einem Zug den Namen des dritten Waggons benötigt:

```
Zuege = {
  {
    Name = "Gueterzug_1",
    Geschwindigkeit = 80,
    Typ = "Gueter",
    Lok = {
      Name = "V200",
      Typ = "Diesel",
      Kupplung_vorne = false,
      Kupplung_hinten = true
    },
    Waggons = {
      {
        Name = "G10",
        Typ = "Kasten",
        Kupplung_vorne = true,
        Kupplung_hinten = true
      },
      {
        Name = "Kbs",
        Typ = "Rungen",
        Kupplung_vorne = true,
        Kupplung_hinten = true
      },
      {
        Name = "OVP",
        Typ = "Tank",
        Kupplung_vorne = true,
        Kupplung_hinten = true
      },
      {
        Name = "G10",
        Typ = "Kasten",
        Kupplung_vorne = true,
        Kupplung_hinten = false
      },
    }
  }
}

print(Zuege[1].Waggons[3].Name)
```

Tabellen - Teil 5

Ihr habt nun gesehen, dass man in einer Tabelle **Zahlen**, **Strings**, **Booleans** und **Tabellen** speichern kann. Und das letzte Beispiel hat euch vielleicht gezeigt, warum das praktisch ist. Es geht bei Tabellen nur darum, eine Ansammlung von Daten vernünftig zu organisieren.

Das ist ungefähr so, als würde man alles, was zu einem bestimmten Waggon gehört, in eine Schachtel packen: Seinen Namen, die Beschriftungen, die Länge, den Status der beiden Kupplungen, die Position beweglicher Teile etc.

Und dann einen Zug bilden, indem man eine Lok-Schachtel und mehrere Waggon-Schachteln hintereinander in eine große Zug-Schachtel steckt.

Das hat alles **nur** organisatorische Bedeutung. @wopitir schrieb mir gestern eine PN, in der er die ganze Tabellenstruktur mit den Ordnern und Unterordnern bei Windows verglich. Und das trifft es ganz gut.

Aber Lua Tabellen können noch mehr.
Man kann in einer Tabelle auch eine Funktion speichern.

Warum auch nicht?

Man speichert in Lua ja eigentlich nur die Adressen, unter denen etwas liegt. Ob an dieser Speicheradresse eine Zahl, ein Text oder eine Funktionsdefinition zu finden ist, macht technisch also gar keinen Unterschied.

Vorweg muss ich eine alternative Schreibweise für die Definition einer Funktion vorstellen.

Diese Variante kennt ihr schon:

```
function Beispiel()
  print("Ich bin eine Beispielfunktion")
end
```

Diese Schreibweise macht dasselbe:

```
Beispiel = function()
  print("Ich bin eine Beispielfunktion")
end
```

Der einzige Unterschied ist die Reihenfolge, in der etwas passiert. Im zweiten Beispiel wird zuerst eine namenlose Funktion angelegt und dann die Adresse dieser Funktion an den Namen Beispiel übergeben. Das Resultat ist aber dasselbe wie bei der ersten Schreibweise.

Die zweite Schreibweise eignet sich, um die Adresse einer Funktion an einen Tabellenplatz zu übergeben:

```
Bands = {
  Beatles = {"John", "Paul", "George", "Ringo"},
  Stones = {"Mick", "Keith", "Brian", "Ian", "Bill", "Charlie"},
  Who = {"Pete", "Roger", "John", "Keith"},
  Anzahl = function(Bandname)
    print("Die Band "..Bandname.." hat "..#Bands[Bandname].." Mitglieder")
  end,
  Erster = function(Bandname)
    print("Das erste Bandmitglied der "..Bandname.." heißt: "..Bands[Bandname][1])
  end
}

Bands.Anzahl("Stones")
Bands.Erster("Who")
```

Beachtet bitte, dass **innerhalb der Funktion keine Kommas am Zeilenende** stehen dürfen. Denn der gesamte Funktionsblock bildet **ein** Element.

Am Ende der Funktionsdefinition muss hingegen zwingend **ein Komma stehen**, falls weitere Tabellenelemente folgen. Deshalb steht im obigen Beispiel ein Komma hinter dem end, welches die Definition von Anzahl abschließt.

Im MBS sind die Modelle Objekte. Und diese Objekte werden, wenn ich Neo richtig verstanden habe, in V5 eigene Funktionen bekommen, welche man per Lua aufrufen kann. Wie ihr oben seht, ist der Aufruf solcher Funktionen sehr einfach. Aber es hilft wenn man eine Vorstellung davon hat, wie die Daten dazu tabellarisch organisiert sind.

Tabellen - Teil 6

Für den Aufruf von Funktionen, die in einer Tabelle stehen, kennt Lua noch einen besonderen Trick. Und der macht das Anlegen von Funktionen innerhalb von Tabellen erst sinnvoll.

Man kann die Adresse der Tabelle beim Aufruf der Funktion als Argument mitgeben. Damit ist es selbst bei tief gestaffelten Tabellen ein Leichtes, direkt auf Elemente der Tabelle zuzugreifen, in der auch die Funktion steht.

Ich möchte etwas weiter ausholen, um die Zusammenhänge möglichst durchschaubar zu machen.
Als Beispiel soll noch einmal die Zugtabelle aus dem vorletzten Kapitel dienen:

```
Zuege = {
  {
    Name = "Gueterzug_1",
    Geschwindigkeit = 80,
    Typ = "Gueter",
    Lok = {
      Name = "V200",
      Typ = "Diesel",
      Kupplung_vorne = false,
      Kupplung_hinten = true
    },
    Waggons = {
      {
        Name = "G10",
        Typ = "Kasten",
        Kupplung_vorne = true,
        Kupplung_hinten = true
      },
      {
        Name = "Kbs",
        Typ = "Rungen",
        Kupplung_vorne = true,
        Kupplung_hinten = true
      },
      {
        Name = "OVP",
        Typ = "Tank",
        Kupplung_vorne = true,
        Kupplung_hinten = true
      },
      {
        Name = "G10",
        Typ = "Kasten",
        Kupplung_vorne = true,
        Kupplung_hinten = false
      },
    }
  }
}
```

Zuege enthält die Adresse der ganzen Tabelle
Zuege[1] enthält die Adresse der ersten Untertabelle (in der alle Daten des ersten Zuges stehen)
Zuege[1].Waggons enthält die Adresse der Untertabelle mit allen Waggons in der Untertabelle 1 der Tabelle Zuege.
Zuege[1].Waggons[1] enthält die Untertabelle mit den Details des ersten Waggons innerhalb der Untertabelle aller Waggons innerhalb …

Solche Adressen kann man an eine neue Variable übergeben:

```
Dings = Zuege[1].Waggons[1]
```

Das ist dann sinnvoll, wenn man anschließend diese Adresse mehrfach benötigt. Weil es die Schreibweise der Adressen innerhalb dieser Untertabelle jetzt vereinfacht

```
Dings.Name
Dings.Typ
Dings.Kupplung_vorne
```

Selbstverständlich wählt man im Ernstfall einen sinnvolleren Namen als „Dings“.

Nun stellt euch bitte vor, dass in der Untertabelle eines Waggons eine Funktion enthalten ist, die etwas mit den Daten dieses Waggons tut. Die zum Beispiel den Namen und den Typ des Waggons ausgibt (um die Geschichte

jetzt nicht ausufern zu lassen.) Dann würde man beim Aufruf der Funktion die gesamte Adresse innerhalb der Tabelle mit angeben.

```
Zuege[1].Waggons[1].Ausgabe()
```

Jetzt könnte man die Adresse außerdem als Argument mitgeben:

```
Zuege[1].Waggons[1].Ausgabe( Zuege[1].Waggons[1] )
```

Für dieses Argument legt man in der Funktionsdefinition eine Variable an. Ein gebräuchlicher Name für diese Variable ist „**self**“. Aber jeder andere Name funktioniert genauso gut.

```
Zuege[1].Waggons[1].Ausgabe = function(self)
  print("Mein Name ist "..self.Name.." und ich bin vom Typ "..self.Typ)
end
```

Das ist ungemein praktisch, weil es die Adressierung der Daten innerhalb dieser Tabelle so bequem macht. Die komplette Adresse der Untertabelle wird an die lokale Variable self übergeben. Also muss man innerhalb der Funktion nur noch **self** schreiben, wenn man Zuege[1].Waggons[1] meint

Und das ist noch nicht alles. Ich kann die identische Funktion für alle Waggons benutzen, weil ich nur die jeweils passende Adresse als Argument an **self** übergeben muss.

Und deshalb hat man in Lua die Schreibweise für den Funktionsaufruf vereinfacht. Programmierer mögen es nämlich nicht, wenn sie etwas doppelt schreiben müssen. Weil es das Risiko vergrößert, sich dabei zu verschreiben.

In Lua kann man die Adresse an einer Stelle mit einem **Doppelpunkt** anstelle des einfachen Punkts aufteilen.

```
Zuege[1].Waggons[1]:Ausgabe()
```

Dann wird automatisch alles, was vor dem Doppelpunkt steht, als erstes Argument übergeben. Werte, die man in die Klammern schreibt, sind damit die Argumente 2 bis …

Der Grund, warum ich diesen relativ schweren Stoff in einem Einsteiger-Tutorial anspreche, ist folgender:

Die einzelnen Objekte im MBS werden mit V5 **fertige** Lua Funktionen enthalten. Und der **Aufruf** dieser Funktionen wird dem folgenden Muster entsprechen:

```
Objektname:Funktionsname(Argumente)
```

Das Objekt hat eigene Daten. Und diese wird die Funktion des Objekts verwenden. Wenn die Funktion ein Signal umstellen soll, dann wird sie dafür die Achsen der Signalflügel ansprechen. Wenn man an einem Kran einen Haken anheben kann, dann muss die Funktion dafür ebenfalls die entsprechenden Parameter des Krans ansprechen und verändern. Wie das im Einzelnen passiert, ist für uns unwichtig. Die Funktion ist in V5 ja **fertig** definiert. Die Definition müssen wir nicht selbst entwerfen.

Aber der Funktions**aufruf** wird diesen Gesetzmäßigkeiten folgen. Wir schreiben möglicherweise so etwas wie:

```
Kran:Haken(0)
```

um den Haken in Nullstellung zu bringen. Wie das konkret aussieht, weiß ich auch erst wenn der Beta-Test der V5 beginnt. Mir geht es jetzt ausschließlich darum, dass ihr anhand meiner generischen Beispiele das zugrundeliegende Prinzip durchschaut. Weil dann verständlicher wird, warum was in welcher Reihenfolge im Befehl stehen muss und was es mit dem **Doppelpunkt** im Funktionsaufruf auf sich hat.

Darüber hinaus könnt ihr euch dieses Wissen natürlich auch in eurem eigenen Programmcode zunutze machen. Denn alles, was man rund um den Eisenbahnverkehr steuert, lässt sich wunderbar tabellarisch verwalten. Es lohnt sich wirklich, die Lua-Tabellen zu durchschauen und den Umgang damit zu üben. Weil man tabellarisch später auch sehr komplexe Situationen gut und überschaubar in den Griff bekommt.

Tabellen und Schleifen

Die for … do Schleife kann man auch verwenden, um ganze Tabellen auszulesen:

```
Beatles = {"John", "Paul", "George", "Ringo"}

for i = 1, 4 do
  print("Beatle Nummer "..i.." ist "..Beatles[i])
end
```

In Lua gibt es für diesen Zweck aber eine eigene, bessere Methode.
Sie heißt **in pairs()**, weil sie paarweise den **Index** und den **Inhalt** zum Index liefert:

```
Beatles = {"John", "Paul", "George", "Ringo"}

for key, value in pairs(Beatles) do
  print("Beatle Nummer "..key.." ist "..value)
end
```

Die Namen der beiden Variablen für Index und Inhalt sind natürlich frei wählbar. Gebräuchlich sind die Buchstaben **k** und **v** für key (= Schlüssel) und value (= Wert).
Deshalb habe ich im obigen Beispiel zum besseren Verständnis die Worte key und value gewählt.

Die in pairs() Methode liefert **alle** Schlüssel/Wert Paare zurück. Auch solche, bei denen der Schlüssel keine Zahl, sondern ein String ist:

```
Beatles = {John = "Gitarre", Paul = "Bass", George = "Gitarre", Ringo = "Schlagzeug"}

for k, v in pairs(Beatles) do
  print(k.." spielt "..v)
end
```

Beachtet bitte, dass die Aufzählung **nicht sortiert** ist. Sie kann zufällig mit der Reihenfolge übereinstimmen, in der die Tabelle erstellt wurde. Aber sie muss es nicht und man weiß nie, in welcher Reihenfolge die Werte ausgelesen werden. **Auch dann nicht, wenn die Indexe eine lückenlos nummerierte Aufzählung (wie im ersten Beispiel) sind.**

Aus diesem Grund gibt es eine zweite, sehr ähnliche Methode: **in ipairs()**

Der Buchstabe i im Wort **ipairs** steht für „iterable“. Damit ist gemeint, dass nur lückenlos nummerierte Elemente der Tabelle ausgelesen werden und dass die Ausgabe sortiert stattfindet.

```
Beatles = {
  "John",
  "Paul",
  "George",
  "Ringo",
  Ort = "Liverpool",
  Jahr = 1960,
  Manager = "Brian Epstein",
  Studio = "Abbey Road"
}

for k, v in ipairs(Beatles) do
  print("Schlüssel: "..k.."", Wert: "..v)
end
```

Vergleicht die Ausgabe mal mit dem , was ihr bei **in pairs()** bekommt.

Und lasst die **in pairs()** Version mehrfach laufen.
Ihr werdet sehen, dass sich Reihenfolgen ändern, obwohl ihr das Skript zwischen den einzelnen Ausführungen nicht ändert.
Der nummerierte Teil der Tabelle steht dabei für gewöhnlich vorne und ist sortiert. Aber garantiert ist auch das nicht!

Wenn eine numerisch sortierte Aufzählung benötigt wird, dann muss man **in ipairs()** verwenden.
Die Ausgabe von **in ipairs()** beginnt bei 1, endet bei der ersten Lücke in der nummerierten Liste und ignoriert namentliche Indexe.

```
Beispiel = {
  Name = "Test",
  "Eins",
  "Zwei",
  "Drei",
  "Vier",
  "Fünf",
  "Sechs",
  "Sieben",
  "Acht"
}

Beispiel[6] = nil

for k, v in ipairs(Beispiel) do
  print("Schlüssel: "..k.."", Wert: "..v)
end
```

Eine fertige Methode für eine alphabetisch sortierte Aufzählung gibt es nicht.

Video 1: Lua und der geheimnisvolle Doppelpunkt

Da ich gerade für jemanden ein kurzes Video erstellt habe, welches im Kern denselben Aspekt behandelt wie das Kapitel "Tabellen - Teil 6" in dieser Reihe hier, möchte ich das Video auch hier anbieten:

Falls das Video mit Klick auf die Grafik nicht startet, hier der Link:

Video bei Youtube ansehen https://youtu.be/k0LgqVFwQRY

Video 2: Lua Strings

Endlich kann ich euch ein weiteres Video anbieten. Die Sprachaufnahme ist leider **sehr** träge geraten. Aber besser kriege ich es derzeit nicht hin. Dafür habe ich mich bemüht den Inhalt zu strukturieren. Und weil das Video recht lang ist, habe ich bei YouTube im Kommentar Sprungmarken hinterlegt.

Das Video ist Anfänger tauglich, denn es setzt kein Vorwissen voraus. Das heißt aber nicht, dass der Stoff einfach ist. Ich bin überzeugt, dass "Anfänger" ebenso gut Schweres lernen können wie Fortgeschrittene. Es muss nur auf ihrem aktuellen Wissensstand aufbauen.

Im Video geht es um "Strings". Die werdet ihr im MBS später mal benötigen. Weil alle Namen, die ihr auslest, Strings sind. Als "String" bezeichnen Programmierer ein Stück Text. Egal, ob es sich dabei um einen Namen handelt oder um eine Nachricht.

Wer nicht alles auf Anhieb versteht, der weiß zumindest später, wo er etwas zu dem Thema findet.

Falls das Video mit Klick auf die Grafik nicht startet, hier der Link:

Video bei Youtube ansehen https://youtu.be/0A0IAyFW9Hk

Das Video reißt die verschiedenen Themen nur an. Es vermittelt ein prinzipielles Verständnis für die Materie, kein *"alles, was man wissen muss"*.
Es ist nämlich Humbug, alles auf einmal erklären zu wollen. Damit erreicht man nur, dass der Lernende am Ende überhaupt nichts mehr weiß.
Es ist viel wichtiger, den Kern zu verstehen. Dann holt man sich zu gegebener Zeit selber das, was man noch braucht.

Viel Spaß beim Lernen und Experimentieren!

Hier sind die wichtigsten Beispiele aus dem Video 2 noch einmal zum Nachlesen.
Manche habe ich um nützliche Hinweise ergänzt.

```
-- Escape Sequenzen

-- \a bell
-- \b back space
-- \f form feed
-- \n newline
-- \r carriage return
-- \t horizontal tab
-- \v vertical tab
-- \\ backslash
-- \" double quote
-- \' single quote
-- \[ left square bracket
-- \] right square bracket

-- zwei Bindestriche kennzeichnen in Lua einen Kommentar

Beispiel = "Die Version 5 des \"3D Modellbahn Studios\"\n stellt 'Lua' Befehle zur Verfügung"
print(Beispiel)



Beispiel1 = 'Die Version 5 '
Beispiel2 = 'des "3D Modellbahn Studios" '
Beispiel3 = "stellt 'Lua' Befehle zur Verfügung",

print(Beispiel1 .. Beispiel2 .. Beispiel3)
```

```lua
Beispiel = {
  'Die Version',
  'des "3D Modellbahn Studios"',
  "stellt 'Lua' Befehle zur Verfügung",
  Version = 5
}

Muster = "%s %d %s %s"
Text = Muster:format(Beispiel[1], Beispiel.Version, Beispiel[2], Beispiel[3])
print(Text)

-- Muster:format(...) ist die verkürzte Schreibweise von: string.format(Muster, ...)



Beispiel = 'des "3D Modellbahn Studios" '

Anfang, Ende = string.find(Beispiel, "3D")  -- vereinfacht: Beispiel:find("3D")
print("Die Zeichenkette '3D' im String 'Beispiel2' \nbeginnt bei Zeichen "..Anfang..
  " \nund endet bei Zeichen "..Ende)



Beispiel = 'des "3D Modellbahn Studios" '

Teiltext = string.sub(Beispiel, 9, 25)      -- vereinfacht: Beispiel:sub(9, 25)
print(Teiltext)


for i = 1, 10, 1 do
  Formel = string.format("%2d * %d = %2d", i, 3, i*3)
  print(Formel)
end

-- die 2 im %2d füllt einstellige Zahlen mit Leerzeichen auf zwei Stellen auf
-- Dadurch steht in der Ausgabe alles fein säuberlich untereinander.
```