Programmieren lernen mit Go

Eine praktische Einführung

Bastian Isensee

1. Ausgabe - 14.12.2015

Bastian Isensee, Stadtkoppel 45, 21337 Lüneburg
E-Mail: bastian-isensee@gmx.de

Inhaltsverzeichnis

1 Warum programmieren lernen?

Zugegeben, wenn du dieses Buch in der Hand hältst, hast du wahrscheinlich schon eine Antwort auf diese Frage gefunden. Es könnte aber auch sein, dass du dir gar nicht so sicher bist, warum es dich eigentlich interessiert. Vielleicht bist du dir auch noch gar nicht sicher, ob du dieses Buch überhaupt lesen sollst. Wie auch immer, es gibt eine ganze Reihe guter Gründe sich für das Thema zu interessieren. Ein paar dieser Gründe möchte ich hier aufzählen.

- Computer sind überall. Ob Auto, Waschmaschine, Telefon, Wohnung oder Medizin: Überall sind heute kleine Computer enthalten. Sie sind allgegenwärtig, was ein guter Grund dafür sein kann, etwas über deren Funktion zu lernen. Diese Funktionen werden zum Großteil durch Software bestimmt.

- Software macht das Leben leichter. Auch wenn man sich über den Nutzen einiger Anwendungen streiten kann, so steht doch eines fest: Ohne Software wäre unser modernes Leben in Wohlstand nicht möglich. Man denke dabei nur an die moderne Produktion von Gütern. Die Kommunikation zwischen Menschen ist problemlos in Echtzeit möglich und kostet fast nichts. Ampelsteuerungen regeln unser hohes Verkehrsaufkommen vollautomatisch. Wir lassen uns durch moderne Spiele oder „smarte" Geräte unterhalten.

- Im Gegensatz zu vielen anderen Dingen, kann man ein Programm nicht anfassen. Dennoch belebt es Geräte zum Leben. Dieser Umstand übt, zumindest auf mich, eine große Faszination aus. Hinzu kommt, dass Programme „weich" sind. Man kann sie relativ leicht ändern, im Gegensatz zu realen Gegenständen. Aus einer Gabel kann ich nicht so einfach einen Löffel machen. Auf Software übertragen kann das funktionieren. (Obwohl das nicht so leicht ist, wie es sich anhört.)

- Software ermöglicht Kreativität. Obwohl Software-Entwicklung ein sehr technisches Thema ist, lässt sich hier Kreativität einbringen. In der Regel gibt es mehrere Wege zum Ziel und kreative Köpfe sind gefragt einen möglichst optimalen Weg zu finden. Auch kann ich mit einer Idee im Kopf einfach mal anfangen etwas zu programmieren, ohne dass dabei zwingend etwas produktiv Nutzbares herauskommen muss. Das nennt sich Prototyping und dient der Erprobung.

Dies sind nur ein paar Gründe und Beispiele, warum es für dich interessant sein kann Programmieren zu lernen. Wenn dich das nicht interessiert, kannst du das Buch gerne weglegen. Denn Programmieren lernen erfordert vor allem den Willen zum Lernen und

Anwenden. Auch ein bisschen Frustrationstoleranz ist von Vorteil, denn nicht immer klappt alles gleich so, wie man es sich vorgestellt hat. Wenn du aber Interesse mitbringst und am Ball bleibst, kann ich dich nur ermuntern weiterzulesen. Wenn du dir unsicher bist ob das was für dich ist, lies einfach weiter und finde es heraus.

2 Vom Quellcode zum lauffähigen Programm

Software wird in einer (oder mehreren) Programmiersprachen verfasst. Das Ergebnis ist ein Programm, das in Textform vorliegt, dem sogenannten Quellcode. Ein Mensch mit entsprechenden Kenntnissen kann den Quellcode lesen, verstehen und anpassen. Programmiersprachen sind Werkzeuge. Sie sind so gestaltet, dass ein Programmierer möglichst leicht eine bestimmte Anforderung in Software „gießen" kann. Anforderungen bestimmen dabei was dein Programm können soll. Sie sind die eigentliche Motivation, ein konkretes Programm zu schreiben. Wie ein Werkzeug aus dem Handwerk, so unterscheiden sich auch Programmiersprachen. Sie haben unterschiedliche Stärken und Schwächen. Nicht jedes Werkzeug ist für jede Aufgabe gleich gut geeignet. Nehmen wir z.B. eine Bohrmaschine. Sie ist dazu ausgelegt Löcher zu bohren. Eventuell könnte man damit auch einen Nagel in ein Stück Holz schlagen. Dafür ist die Bohrmaschine jedoch nicht gut geeignet und deshalb wäre ein Hammer die klügere Wahl. Nachfolgend sind einige Beispielkriterien genannt, in denen sich Programmiersprachen unterscheiden:

- Verwendungszweck: Es gibt Programmiersprachen, die als „Allzweck-Programmiersprache" angesehen werden und andere Sprachen, die einem konkreten Anwendungsgebiet zugeordnet sind. Allzweck-Sprachen sind theoretisch für alle möglichen Anwendungsszenarien einsetzbar. Das bedeutet jedoch nicht, dass diese Sprachen unter sich für jede Aufgabe gleich gut geeignet sind. Eine Sprache mit konkretem Zweck ist beispielsweise SQL. Sie ist dafür ausgelegt Informationen aus Datenbanken möglichst einfach abzufragen und zu verwalten. Sie bietet jedoch keine Unterstützung zur Steuerung von Programmabläufen. Go, Java und C# hingegen sehen sich als Allzweck-Sprachen und sind für ein breites Einsatzgebiet einsetzbar.

- Syntax: Die Syntax beschreibt das Aussehen einer Sprache. Ähnlich einer richtigen Sprache, besitzen auch Programmiersprachen eine Grammatik und kennen eine Reihe von (Schlüssel-)Wörtern. Der wesentliche Unterschied besteht jedoch darin, dass Programmiersprachen wesentlich präziser sind als unsere oft vage Sprache. Kurzum, das Erscheinungsbild einer Sprache wird durch ihre Syntax bestimmt. Hier trifft Zweckmäßigkeit oft auf persönlichen Geschmack. Bereits die Syntax entscheidet bei einigen Entwicklern darüber, welche Sprachen sie „ästhetischer" finden als andere.

- Paradigmen: Programmiersprachen lassen sich verschiedenen Programmierparadigmen zuordnen. Ein Paradigma beschreibt, nach welchem

Konzept programmiert wird. Es hat starken Einfluss darauf, wie man ein Programm formuliert und darüber nachdenkt. Bekannte Paradigmen sind z.B. die imperative, strukturierte, objektorientierte und funktionale Programmierung. Eine jeweilige Erklärung würde an dieser Stelle zu weit führen. Wichtig zu wissen ist jedoch, dass viele (oder alle) Konzepte zum Ziel führen können. Dennoch kann sich der Weg dorthin massiv unterscheiden und mit mehr oder weniger vielen Vor- und Nachteilen gepflastert sein.

- Typisierung: Dies ist ein weiteres Merkmal, in dem sich Sprachen unterscheiden können. Mit „Typisierung" bezeichnet man die Art und Weise wie eine Sprache mit Datentypen umgeht. Auf den Begriff „Datentyp" werden wir später noch näher eingehen. Man kann Typprüfungen beispielsweise nach „statisch" und „dynamisch" unterscheiden. Hiermit kann die Gültigkeit von konkreten Werten vor bzw. während der Ausführung eines Programms geprüft werden. Beispielsweise gibt es Datentypen, die ausschließlich ganze Zahlen abbilden. Der Versuch einen Wert dieses Typs in Verbindung mit einem Wort zu verwenden, wird dann als Fehler erkannt.

Auch hier habe ich wieder nur einige wesentliche Unterscheidungskriterien zwischen Programmiersprachen aufgezählt. Man kann Sprachen beliebig detailliert vergleichen, wenn man das will.

Nun möchte ich zu einem anderen Grundlagenthema kommen: Dem Weg vom Quelltext zum laufenden Programm. Damit ein Computer die Absichten eines Programmierers verstehen und ausführen kann, benötigt er etwas anderes, als den Quelltext: den Maschinencode. Maschinencode ist Binärcode, besteht also aus Nullen und Einsen, deren Format dem Computer eindeutige Instruktionen liefert. Programmiersprachen sind dafür ausgelegt sind, dem Programmierer die Arbeit zu ermöglichen. Sie sind daher eher auf Programmierer, als auf Maschinen ausgerichtet. Die Sprache der Maschinen hingegen, ist für einen Programmierer nicht lesbar. Auch wenn es einige Spezialisten gibt, die in der Lage sind solchen Code zu lesen und zu verändern, so ist die Handhabbarkeit praktisch nicht vorhanden. Stell dir vor, du schreibst einen Text in einer kodierten Geheimschrift. In der Geheimschrift kannst du den Text nicht vernünftig lesen und anpassen. Wenn du jedoch den Code kennst, so kannst du ihn zumindest entziffern, in normaler Form anpassen und anschließend wieder verschlüsseln. So ähnlich verhält es sich mit Programmen. Was gebraucht wird, ist also eine Übersetzung zwischen Programmierer und Maschine. Diese Übersetzung geschieht Software-gestützt. Sie ist daher vollautomatisch und geschieht häufig im Hintergrund. Fehler sind dabei extrem selten, da diese Basis-Software in der Regel sehr ausgereift ist. Man nennt diese Basisprogramme „Compiler". Ein Compiler „kompiliert" (übersetzt) ein Programm von einer Quellsprache (z.B. Go) in eine Zielsprache (Maschinencode). Ein typischer Ablauf während der Entwicklungsarbeit ist der folgende:

1. Schreiben des Programms. Dies geschieht in einem Editor oder einer sogenannten IDE (Integrated Development Environment). Dabei handelt es

sich um eine Art Textverarbeitungsprogramm, speziell für die Erstellung von Computerprogrammen.

2. Übergabe des Quellcodes an den Compiler. Der Compiler selbst ist ein bewährtes Programm. Es übersetzt das Programm von einer Quellsprache in eine Zielsprache. Typischerweise ist das Programm in der Zielsprache durch einen Computer ausführbar.

3. Ausführen des Programms. Jetzt können Tests stattfinden. So kann man prüfen, ob sich das Programm auch korrekt verhält.

Diese drei Schritte sind ein grundlegender Arbeitsfluss, nach dem entwickelt werden kann. Die Schritte werden jedoch nicht nur einmal abgearbeitet, sondern immer wieder in kleinen Durchläufen. Wie auch im realen Leben, ist es empfehlenswert kleine Schritte zu machen. Man entwickelt das Programm ein kleines Stück weiter (hier ein neuer Text, da ein neuer Schalter). Dann lässt man das Programm übersetzen. Anschließend prüft man die Funktionalität. Verhält sich das Programm nicht wie erwartet, es könnte z.B. abstürzen, geht man zurück zu Schritt 1 und korrigiert mögliche Fehler. Verhält sich das Programm wie erwartet, kann man ebenfalls wieder beim ersten Schritt beginnen, diesmal jedoch mit der Entwicklung einer weiteren Funktion.

Wie ein menschlicher Übersetzer, erwartet auch der Compiler, dass die Quellsprache für ihn verständlich ist. Er ist dabei jedoch viel kleinlicher, als ein Mensch. Es wird erwartet, dass die Regeln der Sprache (Syntax und Grammatik) eingehalten wurden. Du brauchst jedoch keine Angst zu haben: Der Compiler sagt dir ziemlich genau an welcher Stelle er mit deinem Code nicht zufrieden ist, wenn du dich nicht an die Regeln hältst. Am Anfang wirken die Fehlermeldung ein bisschen kryptisch. Doch ein Entwickler mit etwas Erfahrung weiß schnell, wo er z.B. einen Tippfehler gemacht hat. Findet er keinen formalen Fehler, so wird er dein Programm anstandslos übersetzen. Das heißt jedoch noch nicht, dass dein Programm so funktioniert, wie du es erwartest. Das kann der Compiler nämlich in den meisten Fällen gar nicht feststellen. Dein Computer kann z.B. nicht wissen, ob ein Schalter blau oder grün sein soll. Dafür wird Software getestet. Je ausführlicher man dies tut, um so sicherer kann man sich bezüglich der korrekten Funktion sein.

Neben dem Übersetzen durch den Compiler gibt es noch einen weiteren Weg, wie ein Programm zur Ausführung gebracht werden kann. Dies geschieht ebenfalls durch ein Stück Software und wird „Interpreter" genannt. Ein Interpreter „interpretiert" dein Programm in der Quellsprache und gibt dem Computer „live" verständliche Anweisungen. Du kannst ihn dir in etwa wie einen Dolmetscher vorstellen. Eine Quellsprache wird (nahezu) unmittelbar in eine Zielsprache übersetzt und interpretiert. Im Gegensatz dazu, kann man einen Compiler mit einem Übersetzer für Bücher vergleichen. Er übersetzt erst das komplette Buch, bevor dieses verkauft und konsumiert wird. Wenn ein Interpreter verwendet wird, ergeben sich typischerweise folgende Arbeitsschritte:

1. Schreiben des Programms.

Kriterium	Compiler	Interpreter
Wartezeit bis zur lauffähigen Version	wenige Sekunden bis einige Stunden	keine
Abhängigkeit zur Plattform	kompiliert wird immer für eine konkrete Plattform (z.B. Linux, 64 Bit)	direkt ausführbar auf allen Maschinen, für die es einen Interpreter gibt
Geschwindigkeit der Ausführung	tendenziell schneller als interpretierter Code	tendenziell langsamer als kompilierter Code
Erkennung von Fehlern	einige Fehler werden bereits vor dem Ausführen erkannt (bei statischer Typprüfung)	Fehler werden erst zur Laufzeit entdeckt

Tabelle 2.1: Gegenüberstellung von Compiler und Interpreter

2. Ausführen des Programms durch Übergabe an den Interpreter. Jetzt können sofort Tests stattfinden.

Vielleicht fragst du dich jetzt, warum man denn überhaupt einen Compiler benutzen sollte. Der Weg über einen Interpreter erscheint doch viel kürzer. Tatsächlich ist es so, dass Interpreter oftmals schneller zu einem greifbaren Ergebnis führen. Das liegt wie gesagt daran, dass der Compiler zunächst das komplette Programm übersetzt. Das kann, je nach Sprache und Größe des Programms, auch einige Minuten bis Stunden dauern. Die Wahrheit ist, dass sowohl kompilierte, als auch interpretierte Sprachen unterschiedliche Vor- und Nachteile haben. Du siehst: Was verwendet wird hängt von der Sprache ab und ist damit ein weiteres Unterscheidungsmerkmal. Wir wollen nun kurz auf die Vor- und Nachteile der beiden Varianten eingehen:

Man sieht bereits anhand dieses kurzen Vergleichs, dass eine Pauschalaussage keinen Sinn macht. Wie auch bei der Wahl der Programmiersprache, gibt es Vor- und Nachteile, die man kennen sollte. Weiterhin hin gibt es Mischformen aus den Konzepten Compiler und Interpreter. Besonders die heute populären, relativ jungen Sprachen, wie Java und C# verwenden eine Mischform. Man erzielt dadurch einen Kompromiss, der die Vorteile aus beiden Welten vereinen soll:

- Übersetzung durch den Compiler in eine Zwischensprache („Bytecode"). Dies ist eine Art Vorarbeit für den Interpreter.
- Interpretation des übersetzten Zwischencodes durch die Laufzeitumgebung (virtuelle Maschine oder Just-In-Time-Compiler).

Auch Mischformen können also reizvoll sein. Jedoch sind sie, wie gesagt, ein Kompromiss. Wenn man also Hardware-nahe Systemprogrammierung betreibt wird man eher auf Sprachen wie C oder C++ greifen, da man auf eine maximale Ausführungsgeschwindigkeit angewiesen ist. Möchte man viel Prototyping betreiben und z.B. eine Webanwendung entwickeln, ist man interpretierten Sprachen, wie Python oder Ruby eher zugeneigt. Auch die Hybridform könnte sich hier anbieten, so wie auch für übliche Desktop-Anwendungen.

3 Entwicklungsumgebung und erstes Programm

Nachdem wir einige Grundlagen geklärt haben, wollen wir zügig zum praktischen Teil übergehen. Dazu werden wir ein kleines, aber sehr populäres Programm schreiben: Hello world! Dabei handelt es sich um ein simples Programm, dass die Begrüßung "Hello world!" auf dem Bildschirm ausgibt. Häufig ist dies das erste Programm, dass man in einer neu zu lernenden Sprache ausprobiert.

Bevor wir loslegen können, musst du nur noch wissen, wie du ein Programm erstellst und ausführst. Normalerweise verwendet man dafür Editoren oder Integrierte Entwicklungsumgebungen. Der Einstieg in Go kann jedoch viel einfacher sein. Du benötigst lediglich einen Internetzugang und einen normalen Webbrowser. Navigiere bitte zu https://play.golang.org/ indem du die URL in die Adresszeile deines Browsers eingibst. Alternativ kannst du mit einer Suchmaschine nach "The Go Playground" suchen. Es sollte eine Website ähnlich der folgenden erscheinen:

Dies ist der Go Playground. Hierbei handelt es sich um eine interaktive Website. Sie dient dazu, schnell und unkompliziert kleine Programme schreiben zu können. Das bietet sich vor allem für Beispiele, zum Lernen oder zum Austausch von Code-Schnipseln an. Der Vorteil liegt auf der Hand. Man muss sich nicht mit Installationen und Konfigurationen aufhalten. Außerdem spielt es keine Rolle, an welchem Rechner man gerade sitzt, solange ein Internetzugang besteht. Daher ist diese Anwendung sehr gut für uns geeignet. Wenn du, z.B. nach dem Durcharbeiten dieses Buchs, größere Programme entwickeln möchtest, findest du im Internet genügend Anleitungen zur Installation. Wenn du (zumindest ein bisschen) Englisch kannst, bietet sich als Einstieg die offizielle Seite an: https://golang.org/doc/install Als Entwicklungsumgebung kann ich dir die freie Software „LiteIDE" empfehlen (https://github.com/visualfc/liteide). Ich nutze sie selbst und bin damit sehr zufrieden. Sie bietet eine sehr gute Mischung aus Komfortfunktionen und Leichtigkeit. Viele IDEs neigen nämlich trotz guter Features dazu, schwergewichtig und somit Ressourcen-hungrig sowie Überladen zu sein. Das ist bei LiteIDE nicht der Fall. Alternativ kannst du auch einen Editor deiner Wahl verwenden. Einige Entwickler ziehen simple Editoren vor, weil sie schlanker sind als IDEs. Das hat jedoch oftmals den Nachteil, dass man auf Features wie Compiler-Integration oder Auto-Vervollständigung für die Syntax verzichten muss. Womit du deine Programme schreibst, bleibt letztendlich dir überlassen. Die meisten Sprachen, darunter auch Go, machen dir hier keine Vorgaben. Ich empfehle dir zunächst den einfachen Einstieg über den Go Playground. Dadurch kannst du dich vorerst auf das Wesentliche konzentrieren, nämlich das Erlernen der Grundlagen.

Lass uns nun zusammen ein Hello-world-Programm schreiben. Wahrscheinlich zeigt

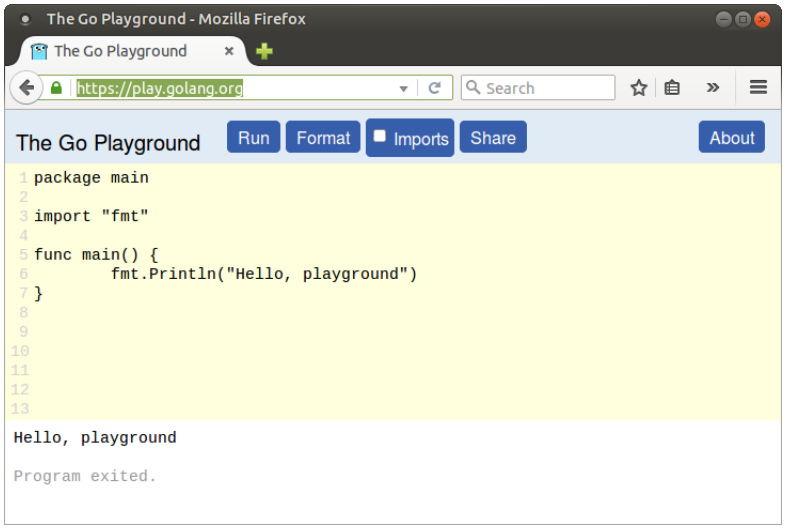

Abbildung 3.1: Der Go Playground

das Eingabefenster bei dir bereits so ein Programm. Wir wollen jedoch von vorn beginnen, sodass ich dich bitten möchte, den Inhalt des Eingabefensters zu löschen. Es handelt sich dabei um das gelb hinterlegte Fenster.

Nun gibst du in dasselbe Fenster folgendes Programm ein:

Listing 3.1: Erstes Programm: Hello world!

```
1 package main
2
3 import "fmt"
4
5 func main() {
6   // Werte ausgeben.
7   fmt.Println("Hello World!")
8 }
```

Im Anschluss klicke bitte auf den blauen Button “Run” im oberen Bereich. Nach kurzer Zeit erhältst du unter dem gelben Eingabefenster die Programmausgabe. Wird hier „Hello World!” ausgegeben, dann hast du gerade dein erstes eigenes Programm ausgeführt. Wenn hier etwas anderes erscheint, hast du vermutlich einen Fehler beim Abtippen gemacht. Bitte sieh es dir nochmal genau an. Wenn du genau liest, wirst du auch erkennen, wo der Compiler einen Fehler sieht. Dazu ein Beispiel. Wenn ich die öffnende geschweifte Klammer in Zeile 5 vergesse, bekomme ich folgende Ausgabe:

Was ist passiert? Der Compiler kann mein Programm nicht übersetzen, da ich mich nicht an die Regeln gehalten habe. Eine Klammer die aufgeht, muss auch wieder zugehen und zwar an einer gültigen Stelle. Sobald ich die Klammer wieder hinzufüge und auf

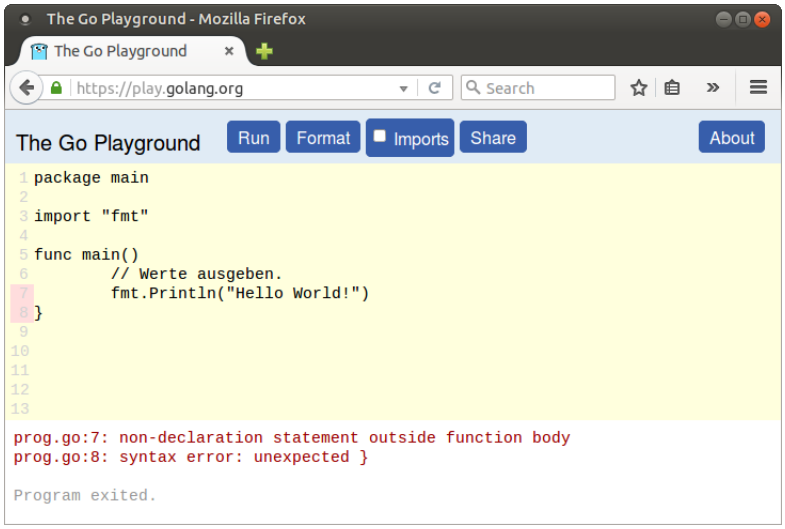

Abbildung 3.2: Ein klassischer Compiler-Fehler

„Run“ klicke, wird mein Programm wieder anstandslos übersetzt und ausgeführt.

Kommen wir nun zu einer kurzen Analyse deines Programms. Softwareentwicklung funktioniert nämlich nicht dadurch, nur irgendetwas abzutippen. Versteh mich bitte nicht falsch. Es ist prinzipiell nicht falsch, den Code anderer Entwickler zu lesen und gegebenenfalls wiederzuverwenden. Wichtig ist jedoch, dass man es sich zur Aufgabe macht, das Programm zu verstehen. Wenn man selbst programmiert, sollte man ohnehin ein klares Bild im Kopf haben. Es ist wie mit der menschlichen Sprache und dem Rechnen. Man kann zwar ein paar Dinge stumpf auswendig lernen und aufschreiben, wenn man jedoch nicht weiß, was man da eigentlich tut, geht das Ganze schnell nach hinten los. Daher gilt auch in der Softwareentwicklung, dass Verstehen und Anwenden wichtiger sind als stumpfes auswendig lernen. Einen richtigen Entwickler beeindruckt es auch nicht, wenn jemand alle Funktionen einer Programmbibliothek auswendig gelernt hat. Das ist nicht wichtig. Wichtig ist, dass man die grundlegenden Konzepte der Programmierung und der verwendeten Werkzeuge (z.B. Sprache) kennt und anwenden kann. Details kann man bei Bedarf immer nachschlagen. Aber nun zur Analyse unseres kleinen Programms. Gehen wir es zeilenweise durch:

- Zeile 1: Hier wird angegeben, zu welchem „package” deine Quellcodedatei gehört. Packages sind eine Organisationsform von Quellcode-Dateien. Sie sind wie Schubladen zur Sortierung. „main” ist der Name des Standard-packages. Die Angabe des packages muss in jeder Quellcode-Datei genau einmal am Anfang aufgeführt sein. Zum Lernen der Grundlagen benötigen wir zunächst nur eine Datei und auch nur ein package. Du musst daher lediglich sicherstellen, dass stets package main angegeben ist.

- Zeile 3: Nach der package-Klausel finden Imports statt. Imports geben an, welche Funktionen du aus anderen Programmen, genauer Packages, in dieser

Quellcode-Datei wiederverwendet werden sollen. Es ist normal, dass man Basisfunktionalitäten, wie die Ausgabe von Werten, nicht selbst schreibt, sondern einfach „importiert" und verwendet. Bei häufig genutzten Basisfunktionen existieren Standardfunktionen, die bereits mit der Sprache installiert werden. Wir wollen das Standard-package „fmt" verwenden (Kurzform für format). Es bietet zahlreiche Funktionen zur Ausgabe und Formatierung von Werten an.

- Zeile 5: Hier beginnt die Definition der Funktion „main". Eine Funktion kapselt eine Reihe von Anweisungen, die einem gemeinsamen Zweck dienen. Unsere Funktion main dient der Ausgabe einer Begrüßung auf dem Bildschirm. Funktionsdefinitionen werden mit dem Schlüsselwort „func", gefolgt von einem beliebigen Namen, eingeleitet. Der Name „main" stellt jedoch eine Ausnahme dar. Er kennzeichnet den Einstiegspunkt des Programms. Da wir irgendwo einsteigen müssen und nur diese eine Funktion benötigen, ist dies unsere main-Funktion. Es darf maximal eine davon in jedem Programm geben. Andernfalls meldet der Compiler einen Fehler. Wenn man nur eine Bibliothek entwickelt, das heißt eine Sammlung von Funktionen zur Wiederverwendung, muss man keine main-Funktion schreiben. Das Programm selbst ist dann nicht lauffähig, sondern dient als Hilfe für andere Programme. Hinter dem Namen folgt ein Paar runder Klammern. Ihre Bedeutung werden wir an einem späteren Punkt klären. Wichtig ist im Moment nur, dass sie vorhanden sein müssen. Es schließt sich eine öffnende geschweifte Klammer an. Sie leitet den sogenannten Funktionskörper ein. Im Gegensatz zum Funktionskopf, dessen Bestandteile ich gerade erläutert habe, beschreibt der Funktionskörper wie eine Funktion arbeitet.

- Zeile 6: Wir befinden uns nun im Funktionskörper der main-Funktion. Hier steht jedoch zunächst keine Anweisung, sondern ein Kommentar. Einzeilige Kommentare werden mit einem doppelten Slash eingeleitet (//). Dahinter kann man ein Kommentar setzen. Wofür sind Kommentare gut? Sie dienen als eine Art „Notiz" sowohl für den Entwickler, als auch für andere Personen, die den Code einmal lesen könnten. Es kann leicht passieren, dass man bereits nach wenigen Wochen nicht mehr genau weiß, welchen Zweck man mit einem Stück Code erzielen wollte. Ein kurzer, aussagekräftiger Kommentar kann hier helfen. Es ist jedoch wichtig zu wissen, dass man mit Kommentaren relativ sparsam umgehen sollte. Zu viel erklärende Kommentare sind ein Symptom von zwei möglichen Ursachen: Die erste Möglichkeit ist, dass zu viel wiederholt wird. Es macht keinen Sinn Erklärungen für etwas zu liefern, dass der Code selbst sehr klar ausdrückt. Die Kommentare werden zum Rauschen, dass beim Lesen stört und auch noch aktuell gehalten werden muss, wenn sich der Code dazu ändert. Die zweite Möglichkeit ist, dass der Code selbst schlecht geschrieben und ausdrucksschwach ist. Das fehlende Verständnis dann mit Kommentaren zu kompensieren ist eine schlechte Idee. Besser wäre es, ausdrucksstärkeren Code zu schreiben, z.B. indem man treffendere Namen verwendet. Abschließend sei gesagt, dass Kommentare lediglich dem besseren Verständnis für Menschen dienen. Der Compiler und damit auch der Computer

ignorieren Kommentare einfach.

- Zeile 7: Hier steht unsere erste und einzige Anweisung. Es handelt sich um den Aufruf einer Funktion aus dem zuvor importierten Standard-package „fmt". Unter Angabe des package-Namens, gefolgt von einem Punkt, geben wir den gewünschten Funktionsnamen an. Die Funktion heißt „Println", ein kurzer Name für „print line". In runden Klammer geben wir an, was wir ausgeben lassen wollen. Hier ist es die Zeichenkette "Hello World!". Zeichenketten, auch Strings genannt, sind ein Datentyp. Dazu später mehr. Wichtig ist, dass Zeichenketten in doppelte Anführungszeichen eingehüllt werden. Was wir hier an die Funktion übergeben, nämlich unsere Zeichenkette, nennt sich „Parameter". Damit offenbart sich nun auch der Zweck der runden Klammern hinter unserer main-Funktion. Parameter sind Werte, die man in eine Funktion „hineingeben", oder wie wir später noch sehen kann auch „herausgeben", kann. Es sind Daten, auf denen gearbeitet wird. Wie du vielleicht allgemein von Computer-Programmen weißt, benötigt sie immer Daten, auf denen sie arbeiten. Ohne Daten, nützen die besten Funktionen nichts, denn womit sollen die Programme arbeiten? Ein Programm ohne Daten wäre wie ein Hammer ohne Nägel. Es ist außerdem wichtig zwischen Funktionsdefinitionen und -aufrufen zu unterscheiden. Wenn ich eine Funktion definiere, wird dies wie in Zeile 5 mit dem Schlüsselwort func eingeleitet und nach dem Funktionskopf wird der -körper definiert. Ein Funktionsaufruf fällt einfacher aus. Man gibt die gewünschte Funktion mittels package und Name an und übergibt Parameterwerte in runden Klammern, fertig.

- Zeile 8: An dieser Stelle wird unsere main-Funktionsdefinition durch eine schließende geschweifte Klammer abgeschlossen. Klammern findet man in der Syntax sehr vieler Programmiersprachen wieder. Insbesondere markieren sie Beginn und Ende vieler Konstrukte. Wo ein Anfang ist, da muss auch ein Ende sein. Nur wenn wir dem Compiler mitteilen, wo der Funktionskörper endet, wird er diesen korrekt einlesen und übersetzen können.

Wie wir gesehen haben, ist es gar nicht schwer, ein minimales Programm zu schreiben. In Go benötigst du lediglich eine Quellcode-Datei mit package-Klausel, main-Funktion und gegebenenfalls import-Statements. Das sind jedoch nur sehr wenige Sprachkonstrukte. Im weiteren Verlauf des Buchs werden wir zusätzliche Konstrukte kennenlernen und unsere Kenntnisse schrittweise ausbauen. Dabei ist es auch von Bedeutung zu wissen, dass sich viele Konzepte in anderen Sprachen wiederfinden. Eine Funktion beispielsweise ist ein zentrales Konzept zur Kapselung von Anweisungen. Du wirst Funktionen in (fast) allen Programmiersprachen wiederfinden. Sie unterscheiden sich nur in den Details. Der Zweck bleibt gleich.

4 Die Programmiersprache Go

Bevor wir uns weiter mit dem Erlernen der Programmierung beschäftigen, möchte ich kurz auf die Programmiersprache meiner Wahl eingehen, nämlich Go. Ziel ist, dass du einen kurzen Überblick zur Sprache bekommst. Schließlich sollte man ein bisschen über die Werkzeuge wissen, die man verwendet. Andernfalls kann es passieren, dass sie einem auf die Füße fallen oder dass man sie für etwas hält, was sie nicht sind.

- Go ist eine Open-Source Programmiersprache. Das bedeutet, sie ist frei einsetzbar, ohne Lizenzgebühren. Weiterhin kann sich theoretisch jeder an der Weiterentwicklung beteiligen. Praktisch wird die Entwicklung natürlich von einigen kompetenten Leuten gesteuert, sodass kein Chaos entsteht. Auf GitHub kann man den Quellcode einsehen und auch ohne öffentliche Beteiligung, für die eigenen Zwecke herunterladen und anpassen (https://github.com/golang/go). Go selbst ist (mittlerweile) in Go geschrieben. Dass ein Werkzeug zur Weiterentwicklung des Werkzeugs selbst verwendet wird, ist in der Softwareentwicklung durchaus üblich und eine faszinierende Vorstellung.

- Go wurde im Wesentlichen von drei Personen entwickelt, die in der Software-Szene recht bekannt sind. Es handelt sich dabei um Ken Thompson (Unix), Robert Pike (Unix, UTF-8) sowie Robert Griesemer (Java HotSpot VM). Die drei Personen entwickelten die Sprache bei Google. Das Ganze begann zunächst als eine Art Hobby-Projekt und entwickelte sich im Laufe der Zeit zu einer vielversprechenden Alternative zu anderen Sprachen. Go ist noch recht jung und erschien 2009 zum ersten mal in einer Art Vorabversion. 2012 erschien die erste stabile Version. Während ich dieses Buch schreibe, ist Version 1.5 das aktuelle stabile Release.

- Motivation für die Entwicklung von Go war die Unzufriedenheit mit bestehenden, populären Sprachen, wie z.B. C++ und Java. Bei diesen Sprachen störte die Entwickler vor allem deren hohe sprachliche Komplexität (zu viele Sonderregeln, Möglichkeiten und Fallstricke). Weiterhin sahen sie die benötigten Compile-Zeiten für große Programme, wie sie z.B. bei Google vorkommen, als viel zu lang an. Es kommt nicht selten vor, dass man etliche Minuten oder sogar Stunden auf die Übersetzung warten muss. Ein weiterer Kritikpunkt an den bestehenden Sprachen war die schlechte Skalierbarkeit. Damit ist erstens die gleichzeitige Bearbeitung von Programmen durch mehrere Entwickler gemeint. Üblicherweise tritt man sich häufig gegenseitig auf die Füße, wenn man mit einem großen Team am gleichen Programm arbeitet. Zweitens bezeichnet Skalierbarkeit die gute Nutzung moderner Hardware. Multicore-CPUs sind heute bereits in Einsteigergeräten enthalten und langweilen sich meist im normalen Betrieb. Es

gilt immer noch als große Herausforderung die hohe Rechenleistung heutiger Computer in tatsächlich abrufbare (Software-)Leistung umzuwandeln. Das liegt auch an den verfügbaren Werkzeugen (Sprachen), die den Entwickler bei Themen wie nebenläufiger Programmierung besser unterstützen sollten. Diese Lücke will Go schließen.

- Einfachheit ist zentraler Bestandteil der Philosophie von Go. Der Sprachumfang wird bewusst kompakt gehalten. Minimalismus hat hier entscheidende Vorteile. Auch wenn Entwickler dazu neigen Komplexität zu mögen, wird der Wert von Einfachheit immer wieder unterschätzt. Die steigende Größe von wachsenden Programmen, das unterschiedliche Wissen in Entwicklerteams und die Komplexität, welche durch Kombinatorik entsteht tragen bereits ihren Teil zur Komplexität bei. Man könnte auf die Idee kommen, dass ein einfacher Sprachumfang die Möglichkeiten des Entwicklers einschränkt. Dies ist jedoch nur sehr bedingt der Fall. Es ist viel häufiger zutreffend, dass „Sonderlocken" für Fälle eingebaut werden, die kaum praktischen Nutzen haben. Die Folge ist eine überladene Sprache. Wie ein extrem kompliziert zu bedienendes Programm auch, sind diese dann deutlich schwieriger zu handhaben und zu erlernen.

- Geringe Compile-Zeiten sind ein Merkmal, welches Go sehr deutlich von bestehenden Sprachen abhebt. Obwohl wir es hier mit einer kompilierten Sprache zu tun haben, die keine Interpreter oder Mischformen verwendet, sind die Übersetzungsvorgänge rasend schnell. Von Beginn an hat man großen Wert auf die Compile-Geschwindigkeit gelegt, sodass selbst sehr große Programme in wenigen Sekunden übersetzt werden. Das gilt übrigens für handelsübliche, durchschnittliche Computer.

Das soll ein grober Überblick zur Sprache Go gewesen sein. Wir fahren nun mit dem Erlernen des Programmierens fort.

5 Datentypen und Werte

Wir werden uns nun mit den wichtigsten Datentypen auseinandersetzen. Ein Datentyp beschreibt die Art der Information, die in einem gegebenen Kontext behandelt wird. Ganze Zahlen haben beispielsweise einen anderen Datentyp als Gleitkommazahlen. Zeichenketten, die wir bereits kurz angesprochen haben, sind ein weiterer Datentyp. Warum sind Datentypen nun wichtig? Welchem Zweck dienen sie? Die wesentlichen Gründe lauten wie folgt:

- Datentypen ermöglichen ein besseres Verständnis von Programmen. Insbesondere durch die Angabe von Datentypen kann die Lesbarkeit verbessert werden. Stell dir vor, du stehst vor einem Haufen Aktenordner und fragst dich, was du vor langer Zeit wo abgelegt hast. Wenn die Ordner nicht beschriftet und der Inhalt nicht sortiert ist, wirst du es schwer haben, dich zurechtzufinden. Ist jedoch leicht erkennbar, was sich worin befindet, machst du es dir wesentlich leichter. Ähnlich verhält es sich mit Datentypen.

- Datentypen ermöglichen Programmfehler frühzeitig zu erkennen. Ohne Datentypen kann man einer Operation nicht ansehen, ob sie erlaubt ist, oder nicht. Nehmen wir zum Beispiel zwei Zahlen. Du kannst problemlos Berechnungen mit ihnen durchführen. Wenn du jedoch eine Zeichenkette in der Hand hast, kannst du sie z.B. nicht mit einer Zahl verrechnen. Durch Datentypen lassen sich diese Art von Fehlern frühzeitig erkennen. Bei statischer Typprüfung, wie sie z.B. auch in Go anzutreffen ist, wird der Fehler bereits durch den Compiler aufgedeckt.

- Datentypen strukturieren Speicherbereiche. Der (Haupt)-Speicher eines Computers besteht aus „durchnummerierten" Speicherzellen. Unterschiedliche Datentypen können unterschiedliche Werte aufnehmen und benötigen daher unterschiedlich viel Speicher. Darüber hinaus gibt es Datentypen, deren Größe von vornherein feststeht und Typen, die dynamisch wachsen können.

Du siehst also, dass das Wissen über Datentypen wichtig und notwendig ist. Wir erstellen nun ein neues Programm mit einigen Beispielausgaben, die Werte verschiedener Datentypen verwenden. Bitte öffne wieder den Go Playground wie bereits erklärt und übernimm das nachfolgende Programm.

Wenn du das Programm ausführst, indem du auf den Button "Run" gehst erhältst du die unten abgebildete Ausgabe. Sollte der Compiler Fehler melden kümmere dich bitte zunächst um deren Beseitigung. Wenn du genau liest, wirst du nützliche Hinweise erkennen, wie z.B. die Zeilennummer. Ein weiterer Abgleich mit dem abgedruckten Programm kann ebenfalls helfen den Fehler zu finden.

Listing 5.1: Grundlegende Datentypen

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // string (Zeichenketten)
7    fmt.Println("Ein String")
8    fmt.Println("Ein" + " zusammengesetzter " + "String")
9
10   // int (Ganze Zahlen)
11   fmt.Println("1+2 =", 1+2)
12   fmt.Println("42/4 =", 42/4)
13
14   // float32 (Dezimalzahlen)
15   fmt.Println("42.0/4.0 =", 42.0/4.0)
16
17   // bool (Wahrheitswerte)
18   fmt.Println(true)
19   fmt.Println(false)
20   fmt.Println("true AND false:", true && false)
21   fmt.Println("true OR false:", true || false)
22   fmt.Println("NOT true:", !true)
23
24   // Auch Vergleiche liefern Wahrheitswerte
25   fmt.Println("2 > 3:", 2 > 3)
26   fmt.Println("2 == 3:", 2 == 3)
27   fmt.Println("2 < 3:", 2 < 3)
28
29   // Zweck der Typprüfung: Typfehler vermeiden
30   // fmt.Println("Nicht gültig:", 42/"Ein String")
31 }
```

Wir wollen nun Schritt für Schritt die wesentlichen Datentypen und deren Beispielausgabe durchgehen. Jeder Stichpunkt widmet sich einem Abschnitt in der main-Funktion (jeweils getrennt durch eine Leerzeile).

- String (Zeichenkette): Strings sind Zeichenketten und können daher Buchstaben, Wörter, Sätze und beliebig lange Texte abbilden. Sie werden in doppelte Anführungszeichen eingeschlossen um sie als String kenntlich zu machen. Mit Hilfe des Verkettungsoperators (+) können aus einzelnen Strings neue Strings gebildet werden. Dies geschieht durch „Aneinanderreihung" und ist eine übliche Operation für Zeichenketten. Durch die Kennzeichnung von Strings als solche, kann der Compiler z.B. zwischen einer Verkettung und der Addition von Zahlen unterscheiden.
- Integer (kurz int, ganze Zahlen): Integer dienen der Abbildung von ganzen Zahlen. Man kann mit ihnen rechnen, indem man die entsprechenden Operatoren angibt,

Listing 5.2: Ausgabe des Programms zu den grundlegenden Datentypen

```
1  Ein String
2  Ein zusammengesetzter String
3  1+2 = 3
4  42/4 = 10
5  42.0/4.0 = 10.5
6  true
7  false
8  true AND false: false
9  true OR false: true
10 NOT true: false
11 2 > 3: false
12 2 == 3: false
13 2 < 3: true
```

die auch aus der Mathematik bekannt sind. Die Division wird mit dem Slash-Operator (/) gekennzeichnet. Es ist wichtig zu wissen, dass Integer immer nur ganze Zahlen abbilden. Das heißt konsequenterweise, dass Reste bei der Division wegfallen. Aus der Sicht von ganzzahligen Operationen ergibt 42 durch vier also 10. Es wird immer dann mit ganzen Zahlen gerechnet, wenn kein Dezimalpunkt angegeben ist und keine weitere Typangabe (z.B. Variable, dazu später mehr) dem widerspricht. Zur Notation der Println-Statements sei gesagt, dass hier zwei Parameter verwendet werden. Dabei werden alle Parameter ausgegeben und durch ein Leerzeichen getrennt. Somit können wir eine vorangestellte Ausgabe (die Operation) und das eigentliche Ergebnis separieren und komfortabel ausgeben.

- Floating Point (kurz float, Gleitkommazahlen): Hierbei handelt es sich um Dezimalzahlen. Somit können auch Zahlen mit Nachkommastellen abgebildet werden, wenn man mit diesen rechnen muss. Man kann sich nun die Frage stellen, wozu man überhaupt Integer benötigt, wenn ich doch lieber gleich immer den genaueren Typ, float wählen kann. Dazu sei gesagt, dass Gleitkommazahlen immer nur eine begrenzte Genauigkeit besitzen. Anders als z.B. ein Bruch oder eine Periode in der Mathematik sind Gleitkommazahlen in Computern stets mit einer gewissen Ungenauigkeit versehen. Dies liegt in der endlichen Größe von Speicher- und Rechnerarchitekturen begründet. Auch die Abbildung im Binärformat spielt dabei eine Rolle. Die Dezimalzahl 0,1 z.B. ergibt im Binärsystem eine sich periodisch wiederholende Ziffernfolge. Das hat zur Folge, dass Vergleichsoperationen, die auf exakten Annahmen beruhen unerwartet scheitern können. Hinzu kommt, dass Gleitkommaoperationen teuerer für die CPU sind (im Sinne der Ausführungsgeschwindigkeit). Wichtig für dich ist vor allem: Es gibt Gleitkommazahlen-Typen, mit denen man mit begrenzter Genauigkeit rechnen kann. Wenn jedoch exakte Ergebnisse nötig sind, sollte man sehr genau hinsehen. Braucht man keine Nachkommastellen, sollte man stets Integer verwenden. Floats

Operator	Wert 1	Wert 2	Ergebnis
AND	true	true	true
AND	true	false	false
AND	false	true	false
AND	false	false	false
OR	true	true	true
OR	true	false	true
OR	false	true	true
OR	false	false	false
NOT	true	-	false
NOT	false	-	true

Tabelle 5.1: Boolesche Operationen

werden mit einem Dezimalpunkt angegeben. Die Punkt-Notation entstammt dem Englischen, wie in allen Programmiersprachen und sollte nicht mit einem Komma-Zeichen verwechselt werden. Go unterscheidet weiterhin float32 und float64. Die Zahl gibt die Genauigkeit in Bit an. Wenn eine sehr hohe Genauigkeit wichtiger ist, als Speicherbedarf und Ausführungsgeschwindigkeit kann man daher zu float64 greifen.

- Boolesche Werte (kurz bool oder boolean, Wahrheitswerte): Boolesche Werte bilden einen binären Zustand ab (0 oder 1, false oder true). Du kannst dir analog dazu einen einfachen Kippschalter vorstellen. Ein boolescher Wert kann genau zwei verschieden Zustände einnehmen: true oder false (an oder aus). Beide Werte kann man entweder direkt eingeben oder auch als Ergebnis, z.B. von Vergleichen oder Operationen erhalten. Wozu das gut ist, werden wir in einem späteren Kapitel erfahren. Zur Einstimmung sei aber schon mal gesagt, dass hiermit vor allem der Kontrollfluss in Programmen gesteuert wird. Anhand von Wahrheitswerten, die Bedingungen abbilden, werden Entscheidungen getroffen. Wir wollen noch kurz einen Überblick zu den wichtigsten Operationen gewinnen, die man mit booleschen Operationen abbilden kann:
  - AND (UND): Alle Werte müssen true sein, damit das Ergebnis ebenfalls true ist. Sobald einer der mit AND verknüpften Werte false ist, ergibt das gesamte Ergebnis false. Du kannst es dir ganz einfach anhand eines Merksatzes vorstellen: Sind alle Türen in einem Haus geöffnet? Sobald eine Tür geschlossen ist, ist das Ergebnis false (falsch). Sind alle offen, gilt true. Im Programmcode wird AND durch && abgebildet, wobei links und rechts vom Operator jeweils ein Wert steht.
  - OR (ODER): Mindestens ein Wert muss true sein, damit das Ergebnis ebenfalls true ist. Nur wenn alle Werte false sind, ist auch dass Ergebnis false. Die Analogie hierzu wäre: Ist mindestens eine Tür geöffnet? Im

Programmcode wird AND durch || abgebildet, wobei links und rechts vom Operator jeweils ein Wert steht.

- NOT (NICHT): NOT kehrt den gegebenen Wert um. Das heißt aus dem Eingabewert wird genau der entgegengesetzte Werte. Eine geöffnete Tür wird geschlossen. Eine geschlossene Tür wird geöffnet. Im Programmcode wird NOT durch ! gefolgt vom zu negierenden Wert, abgebildet.

- Wie du am Ende unseres Quellcodes siehst, habe ich einige Vergleichsoperationen eingebaut, die als Ergebnis boolesche Werte liefern. Mittels größer und kleiner als (> und <) können normale Zahlenvergleiche angestellt werden. Auf Gleichheit wird mittels == geprüft (wichtig: zwei Gleichheitszeichen). Dies funktioniert z.B. auch bei Zeichenketten. Zwei Strings sind gleich, wenn sie exakt gleich aussehen. Weiterhin gibt es größer gleich (>=) sowie kleiner gleich (<=). Probiere es doch einfach aus!

- Die letzte Zeile im Funktionskörper ist auskommentiert, da sie einen Zweck der Verwendung von Typen demonstrieren soll, nämlich die Typprüfung. Nimm einfach mal die Slashes am Beginn der Zeile weg und versuche das Programm auszuführen. Der Compiler wird sich beschweren, dass die Werte 42 und “ein String” nicht für eine Division herangezogen werden können. Das Abfangen dieses Fehlers ist nur mit Hilfe von Datentypen möglich.

Wir haben uns nun die wichtigsten Basisdatentypen angesehen und ihren Zweck besprochen. Dazu sei gesagt, dass es noch weitere Datentypen gibt, auf die wir in den folgenden Kapiteln noch näher eingehen werden. Darüber hinaus kann man eigene Datentypen definieren, wovon Entwickler regen Gebrauch machen. Auch dieses Thema werden wir noch einmal gesondert behandeln.

6 Variablen

Wir kommen nun zu einem Thema, dass nicht weniger wichtig ist als Datentypen und genau genommen sogar eng damit verwandt: Variablen. Variablen dienen der Zwischen-Speicherung und der Adressierung von konkreten Werten. Wir haben bisher immer nur konkrete Werte „an Ort und Stelle" (inline) einprogrammiert und uns ihre Typen angesehen. In der echten Programmierung benötigt man jedoch eine höhere Flexibilität. Wenn ich in ein Programm z.B. Namen eingeben, und diese dann speichern, verarbeiten oder ausgeben möchte, so muss ich sie irgendwie (zwischen)speichern. Variablen bieten genau dies. Du kannst dir eine Variable wie eine Art Schublade vorstellen, in die genau ein konkreter Wert eines bestimmten Datentyps gelegt werden kann. Für das Beispiel eines Namens stellen wir uns also eine Schublade vor, die mit „name" beschriftet ist. Wir beschriften sie, damit wir sofort wissen, was wir darin aufbewahren und den Überblick behalten. Neben dem Bezeichner „name", kleben wir ein Schild mit dem zugehörigen Datentyp drauf: string. Das tun wir, weil es sich bei Go um eine Sprache mit statischer Typprüfung handelt und wir somit in den Genuss von Typprüfungen durch den Compiler kommen. Wir erinnern uns: Wenn ich nun versuchen würde, mit dem Wert aus der Schublade vom Typ string, Rechenoperationen durchzuführen, kann der Compiler dieses Versehen erkennen und melden. Bitte öffne, wenn nötig, wieder den Go Playground. Lösche die gelb hinterlegte Eingabe und gib folgendes Programm ein:

Wir gehen den abgebildeten Code wieder Schritt für Schritt durch. Ein Stichpunkt entspricht einem Abschnitt in der main-Funktion.

- Unsere main-Funktion beginnt damit, eine neue Variable bekannt zu machen. Dies nennt man Deklaration (Verb: deklarieren). Eingeleitet wird dies durch das Schlüsselwort „var" gefolgt vom Namen der Variablen und ihrem Typ. Es wurde somit eine neue Variable unter dem angegebenen Namen (hier name) und Typ (string) bekannt gemacht. Bevor du eine Variable verwenden kannst, um Werte abzulegen oder auszulesen, muss sie stets bekannt gemacht werden. Der Name, auch Bezeichner genannt, muss eindeutig sein. Wenn du versuchst eine weitere Variable mit dem gleichen Namen anzulegen, wird der Compiler einen Fehler melden. Er muss stets genau zuordnen können „welche Schublade" du meinst. Variablen innerhalb von Funktionen nennt man lokale Variablen. In Go werden diese per Konvention klein geschrieben. Solche Formatierungskonventionen dienen dazu, einen möglichst gleichförmigen, leicht zu lesenden Stil innerhalb einer Sprache zu erzielen. Es ist z.B. ebenfalls Konvention, dass wir geschachtelte Anweisungen mit einem Tabulator einrücken. Falls du dich also bisher gewundert hast, warum die Anweisungen in der main-Funktion immer um einen Tabulator eingerückt sind, so sei gesagt, dass dies streng genommen für den Compiler nicht wichtig, für

Entwickler aber von sehr hoher Bedeutung ist. Durch diese optische Strukturierung lässt sich viel leichter erkennen, welche Code-Blöcke wohin gehören. Einrückung ist dabei eine gängige Konvention in praktisch allen Programmiersprachen. Für Go gibt es sogar eine automatische Formatierung. Du kannst sie über den Button „Format" im Go Playground ausprobieren. Welchen konkreten Wert hat nun eine Variable, die frisch deklariert wurde? Die Antwort lautet keinen oder in Go den sogenannten „Zero-Value". Unsere Schublade ist leer und somit ist auch der Wert leer. Eine Besonderheit in Go ist der Zero-Value, da dieser einen leeren Wert darstellen kann. In vielen anderen Sprachen ist der Wert einfach nicht vorhanden oder nicht definiert. Der Unterschied dabei ist, dass ich mit einem nicht vorhandenen oder undefinierten Wert nichts tun kann. Ein Zero-Value jedoch ist ein wohldefinierter Wert, mit dem ich durchaus etwas tun kann. Das println-Statement gibt uns für name nämlich den leeren String aus (""). Der leere String ist eine Zeichenkette ohne Zeichen. Wozu das gut sein soll? Nun, du könntest z.B. Strings verketten ohne dir immer Gedanken darüber machen zu müssen ob bereits ein Wert in den Variablen steht oder nicht. Wenn dies nicht der Fall ist, kannst du trotzdem verketten! Ein Integer in Go (int) hat den Zero-Value 0 und kann z.B. sofort zum Hochzählen verwendet werden. Eine boolesche Variable (bool) hat den Wert false. Somit kann man von Anfang an sinnvolle Operationen ausführen, da ein wohldefinierter Anfangswert existiert.

- Wir weisen unserer existierenden Variable nun einen (neuen) Wert zu. Dies geschieht mit Hilfe des Zuweisungsoperators (einfaches =). Jetzt siehst du auch, warum ich in einem früheren Kapitel den Vergleich von Werten mit doppeltem Gleichheitszeichen hervorgehoben habe. Ein einfaches Gleichheitszeichen kennzeichnet eine Zuweisung, keinen Vergleich. Eine Zuweisung läuft wie folgt ab: Ein Wert auf der rechten Seite wird einer Variablen auf der linken Seite zugewiesen. Das heißt konkret, nach Ausführung dieser Zeile besitzt die Variable einen neuen Wert, nämlich "Bob Müller". Die Zuweisung greift also schreibend auf die Variable links des Gleichheitszeichens zu. Der alte Wert geht verloren. Wir müssten ihn zuvor woanders zwischenspeichern, wenn wir ihn aufbewahren wollten. Auf der rechten Seite einer Zuweisung muss kein direkter Wert stehen. Es kann sich dabei auch um eine andere Variable handeln, denn diese repräsentiert ja einen Wert. Dieser wird dann ebenfalls der Variablen auf der linken Seite zugewiesen, während die Variable auf der rechten Seite unverändert bleibt. Zur besseren Übersicht ist nachfolgend eine kleine Tabelle mit Beispielzuweisungen aufgeführt.

- Es ist jederzeit möglich einer existierenden Variablen einen neuen Wert zuzuweisen. Wie wir im abgebildeten Code sehen, kann ich name auch anschließend den Wert "Bob Schulz" zuweisen. Wenn ich die Variable der println-Funktion übergebe, wird der entsprechende Wert ausgegeben. Variablen sind also benannte Platzhalter für konkrete Werte.

- Sehr häufig möchte man eine neu deklarierte Variable sofort mit einem Anfangswert

Zuweisung	Wert von name1 nach Zuweisung	Hinweis
name1 = “Michael”	“Michael”	alter Wert wird überschrieben
name1 = “”	“”	leerer String, Zero-Value für Typ string
name1 = name2	Wert aus name2 (z.B. “Susi”)	der Name einer Variablen ist stets ein Platzhalter für einen Wert
name1 = name1 + “ Schulz”	“Michael Schulz”	rechte Seite wird zuerst ausgewertet, daher kann man auch den alten Wert verwenden

Tabelle 6.1: Beispielzuweisungen und deren Ergebnisse

belegen, der nicht dem Standard-Wert (Zero-Value) entspricht. Dies nennt sich Deklaration mit Initialisierung, da hier ein Initialer Wert zugewiesen wird. Das kann einfach umgesetzt werden, indem hinter der Deklaration eine Zuweisung angeschlossen wird. Ich kann sofort einen direkten Wert oder den enthaltenen Wert einer anderen Variablen zuweisen. Das spart Schreibarbeit (Deklaration und anschließende Zuweisung) ohne an Lesbarkeit zu verlieren.

- Einige Programmiersprachen, darunter auch Go, bieten eine verkürzte Form für das Deklarieren und Initialisieren von Variablen an. Der Compiler ist nämlich intelligent genug, den Typ anhand der initialen Zuweisung zu erkennen. Auch das Schlüsselwort var kann man im Falle einer Initialisierung weglassen. Stattdessen wird eine „shorthand” Deklaration und Initialisierung in Form von Name := Wert geschrieben. Was passiert nun, wenn der Compiler wohnort := “Hamburg” sieht? Er sieht, dass du eine neue Variable anlegen und initialisieren möchtest, da du den Operator := verwendest. Den Typ ermittelt er, indem er auf die rechte Seite nachsieht. “Hamburg” ist ein string, da er in doppelten Anführungszeichen angegeben ist. Schon weiß der Compiler, dass dieser Ausdruck gleichbedeutend ist mit: var wohnort string = “Hamburg”. Ziemlich cool, oder? Man kann diese Notation ohne Probleme verwenden, wenn man weiß, was dahintersteht. Es ist kürzer zu schreiben, aber trotzdem verständlich. Man kann übrigens auch eine weniger stark verkürzte Form anwenden: var wohnort = “Hamburg”. Hierbei wird nur der Typ von der rechten Seite abgeleitet. Dass es sich um eine Variablendeklaration handelt, wird bereits durch Angabe von „var” klar. Du kannst problemlos die kürzeste Form wählen, wenn du eine Deklaration mit Initialisierung vornehmen möchtest. Wir wollen jedoch kurz abtesten, ob du den kompletten Mechanismus dahinter verstanden hast. Dazu schreibe bitte für jede der 4 kurzen Variablendeklarationen und -Initialisierungen die ausführliche Form auf. Teste sie am besten auch im Playground, um die Gültigkeit zu prüfen. Bedenke, dass

Kurzform	Ausführliche Form (Lösung)
wohnort := "Lüneburg"	var wohnortLang string = "Lüneburg"
hausnummer := 45	var hausnummerLang int = 45
pi := 3.14	var piLang float32 = 3.14
fünfGrößerVier := 5 > 4	var fünfGrößerVierLang bool = 5 > 4

Tabelle 6.2: Lösung zur Aufgabe „Ausführliche und verkürzte Variablendeklaration und -Initialisierung”

Variablennamen eindeutig sein müssen innerhalb des umschließenden Blocks und du daher eventuell neue Namen verwenden musst. Die Lösung findest du in der unteren Tabelle. Bitte sieh jedoch erst nach, wenn du eine eigene Lösung ausprobiert hast. Bitte stelle sicher, dass du die Lösung verstehst und anwenden kannst. Diese Grundlagen sind enorm wichtig für das weitere Lernen. Wenn du noch Verständnisprobleme hast, lies das Kapitel nochmal in Ruhe durch oder verwende weitere Informationsquellen. Auch das Kapitel zum Thema Datentypen kann nochmal hilfreich sein.

Listing 6.1: Variablen deklarieren und initialisieren

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // Variable deklarieren (Name und Typ bekannt machen)
7    // Wert ist nicht gesetzt (Zero-Value)
8    var name string
9    fmt.Println(name)
10
11   // Variable einen Wert zuweisen
12   name = "Bob Müller"
13   fmt.Println(name)
14
15   // Variable einen neuen Wert zuweisen
16   name = "Bob Schulz"
17   fmt.Println(name)
18
19   // Variable deklarieren
20   // und einen initialen Wert zuweisen (initialisieren)
21   var alter int = 42
22   fmt.Println("Alter ist:", alter)
23
24   // Kurzform: Variable deklarieren und initialisieren
25   // Typ automatisch vom Wert ableiten
26   wohnort := "Hamburg"
27   hausnummer := 45
28   pi := 3.14
29   fünfGrößerVier := 5 > 4
30
31   fmt.Println(wohnort, hausnummer, pi, fünfGrößerVier)
32
33   // Übung: Welche Datentypen haben die oberen 4 Variablen?
34   // Wie sieht die ausf. Deklaration und Initialisierung aus?
35 }
```

7 Fallunterscheidungen (if-else)

Wir beschäftigen uns nun zum ersten Mal mit einem Konstrukt, das zur Ablaufsteuerung dient. Sicher kannst du es dir schon denken: Echte Programme sind dynamisch und verhalten sich, je nach Situation, unterschiedlich. Eines der wichtigsten Konstrukte sind daher bedingte Anweisungen. Je nachdem, ob ein Ausdruck den booleschen Wert true (wahr) oder false (falsch) ergibt, wird ein Block mit Anweisungen ausgeführt oder übersprungen. Wir betrachten dazu das nachfolgende Programm. Bitte gib es jetzt oder nach der Analyse auch im Go Playground ein und probiere es aus:

Lass uns nun einen näheren Blick auf das Programm werfen. Ein erläuternder Stichpunkt beschreibt jeweils den Code zwischen zwei kommentierten Absätzen:

- Wie bereits kurz angesprochen, hängt die Ausführung einer Verzweigung von dem booleschen Ergebnis ihrer Bedingung ab. Eine Verzweigung wird mit dem Schlüsselwort “if” (wenn) eingeleitet. Dahinter folgt, nach einem Leerzeichen, ein Ausdruck, der zu einem booleschen Wert (d.h. true oder false) evaluieren muss. Das heißt nichts anderes, als dass es sich dabei tatsächlich um eine Bedingung handeln muss. Umgangssprachlich kann man fragen: Kann der Ausdruck mit wahr oder falsch beantwortet werden? Im einfachsten Fall kann man direkt true (wahr) oder false (falsch) hineinschreiben, wie hier gezeigt. Dies macht jedoch wenig Sinn, da der zugehörige Code dann entweder immer (für true) oder niemals (für false) ausgeführt wird. Es soll nur der Demonstration dienen und erfüllt die Anforderungen an einen booleschen Ausdruck. Ein Gegenbeispiel ist z.B. der Ausdruck 3.14 oder „zweiundvierzig“. Diese Ausdrücke ergeben keinen Wahrheitswert, sie stehen für sich (anders als z.B. ein Vergleich). Die Bedingung wird wiederum mit einem Leerzeichen abgeschlossen und es folgt eine öffnende geschweifte Klammer um den Anweisungsblock einzuleiten. Wie wir bereits bei der main-Funktion gelernt haben, wird ein Block aus Anweisungen in geschweifte Klammern eingeschlossen, um ihn als solchen zu Kennzeichnen. So geschieht es auch bei bedingten Anweisungssequenzen. Um zu zeigen, wo eine Reihe bedingter Anweisungen beginnt und wo sie endet, muss man die Klammern setzen. Andernfalls beklagt sich der Compiler.

- Des Weiteren sehen wir ein Beispiel für einen Vergleich: $3 > 2$ liefert einen Wahrheitswert. Das Ergebnis ist true, also wird der Zweig ausgeführt.

- Im nächsten Abschnitt sehen wir ein if-Statement mit angeschlossenem else-Zweig. Ein else Zweig ist eine alternative Verzweigung. Sie wird immer genau dann ausgeführt, wenn der vorherige Zweig (oder die vorherigen Zweige) nicht ausgeführt wurden. Du kannst dir ein if gefolgt von else umgangssprachlich als „wenn, dann,

sonst ...” Vorstellen. Das else nimmt dabei die Rolle des Wortes „sonst” ein. Drei mal drei ergibt nicht zehn, also wird der else-Zweig ausgeführt.

- Im letzten Abschnitt sehen wir eine neue Erweiterung von if-else-Verzweigungen. Durch die Kombination von if und else, kann man beliebig komplexe Fallunterscheidungen bauen. Ausgesprochen gestaltet sich der gezeigte Code wie folgt: Wir deklarieren und initialisieren eine neue Variable „name” vom Type string mit dem Wert „Klaus”. Nun findet eine Fallunterscheidung statt: Wenn die Variable „name” den string Wert “Bob” beinhaltet, gib die Begrüßung für Bob aus. Andernfalls, wenn die Variable „name” den string Wert “Klaus” enthält, gib die Begrüßung für Klaus aus. Andernfalls (ohne wenn d.h. wenn keine der vorherigen Bedingungen zutrifft) gib eine allgemeingültige Begrüßung aus.

Mit diesem relativ einfachen Konstrukt lassen sich bereits sehr viele Programmabläufe abbilden. Zur Demonstration habe ich Ausdrücke als Bedingungen verwendet, deren Ergebnis sofort ersichtlich ist. Üblicherweise erfolgt die Auswertung aber dynamisch zur Laufzeit, das heißt man weiß nicht sofort, was ausgeführt wird. Stell dir vor, wir lesen eine Eingabe vom Nutzer oder machen eine Ausführung von der Tageszeit abhängig. Mit solchen Bedingungen wird ein Programm erst richtig dynamisch. Zur Übung und um praktisch dazuzulernen, finde bitte folgende Dinge durch ausprobieren heraus:

1. Kann ich weitere else-if Statements in den zuletzt gezeigten Abschnitt einbauen?
2. Was passiert, wenn ich nur if und else if-Zweige schreibe, jedoch keinen else Zweig?
3. Was passiert, wenn ich einfach nur if-Statments aneinanderreihe? Was ist der Unterschied zu einer langen if-else-if-else-Kaskade?
4. Was passiert, wenn ich ein else-Statment vor ein else-if Statement schreibe?

Bitte bearbeite die Aufgaben erst selbstständig und sieh dir dann die Lösungen unten an. Nur so lernst du wirklich dazu.

Lösungen:

1. Das ist kein Problem und durchaus üblich. Die Verzweigung kann beliebig lang werden. Lediglich die Lesbarkeit verschlechtert sich mit zunehmender Länge.
2. Das ist erlaubt und durchaus üblich. Es führt dazu, dass nur spezifische Bedingungen geprüft werden. Es gibt dann keine „allgemeine” Alternative. Es wird genau ein oder kein Zweig durchlaufen.
3. Das ist erlaubt und durchaus üblich. Es führt dazu, dass die Zweige unabhängig voneinander geprüft und ausgeführt werden. So können auch alle if-Statements durchlaufen werden (oder keiner oder eine Teilmenge).
4. Das ist nicht erlaubt und der Compiler meldet einen Fehler. Da der Ablauf hier dem Lesefluss (von oben nach unten) entspricht, kann logischerweise eine (letzte)

„allgemeine Alternative" (else) nicht vor einer spezifischen Abfrage (else if oder if) geschehen. Das würde alle folgenden Zweige obsolet machen, da sie nie durchlaufen werden, denn das else greift jede übrige Bedingung auf.

Listing 7.1: Fallunterscheidungen mit if / else

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // Ausdrücke, die Wahrheitswerte liefern,
7    // können für bedingte Anweisungen verwendet werden.
8    // Dazu zählen Wahrheitswerte (Typ bool).
9    if true {
10     fmt.Println("Wird immer ausgeführt.")
11   }
12   if false {
13     fmt.Println("Wird nie ausgeführt.")
14   }
15
16   // Vergleiche liefern ebenfalls Wahrheitswerte.
17   if 3 > 2 {
18     fmt.Println("3 ist größer als 2.")
19   }
20
21   // else leitet eine alternative Verzweigung ein,
22   // wenn die vorherige Bedingung nicht zutreffend war.
23   if 3*3 == 10 {
24     fmt.Println("3 mal 3 ist gleich 10.")
25   } else {
26     fmt.Println("3 mal 3 ist gleich:", 3*3)
27   }
28
29   // else if fragt weitere konkrete Bedingungen ab
30   name := "Klaus"
31   if name == "Bob" {
32     fmt.Println("Ich kenne dich, du bist Bob.")
33   } else if name == "Klaus" {
34     fmt.Println("Ich kenne dich, du bist Klaus.")
35   } else {
36     fmt.Println("Ich kenne dich noch nicht. Du bist:", name)
37   }
38 }
```

8 Schleifen (for loop)

Wir widmen uns nun einer Kontrollstruktur, die neben if/else-Verzweigungen zum grundlegenden Handwerkszeug eines Programmierers gehört: den Schleifen. Um einen Programmablauf zu steuern, benötigt man neben bedingten Anweisungssequenzen auch die Möglichkeit, Programmteile wiederholt ablaufen zu lassen. Das hat vor allem zwei Gründe:

1. Man möchte gleichförmigen Code nicht X-mal wiederholen. Wenn ich zum Beispiel einen Countdown runter zählen will, mache ich im Grunde immer wieder das Gleiche: Eine Zahl um eins verringern und dann ausgeben.

2. Man weiß oftmals zum Zeitpunkt des Programmierens nicht exakt, wie oft ein Ablauf wiederholt wird. Stell dir vor, du möchtest eine Eingabe vom Anwender einlesen: Typischerweise wird die Eingabe auf Gültigkeit geprüft. Wenn der Nutzer sein Alter als Zahl eingeben soll, dies jedoch zweimal falsch macht, indem er Worte eingibt, wollen wir ihn solange auffordern eine korrekte Eingabe zu machen, bis wir sie lesen können.

Bitte betrachte den unten aufgeführten Code und gib ihn im Go Playground ein. Führe ihn ruhig schon aus, um zu sehen was er bewirkt. Man sieht die einzige Schleifenform, die Go kennt: die for-Schleife in zwei Varianten.

Wir analysieren das Programm wieder Schritt für Schritt:

- Der erste Abschnitt zeigt eine sehr einfache, aber recht langwierige Schreibweise. Wir deklarieren und initialisieren eine Variable zum Hochzählen einer ganzen Zahl (daher Typ int). Es schließt sich der Schleifenkopf an. Dieser wird durch das Schlüsselwort „for" eingeleitet. Es folgt, durch ein Leerzeichen getrennt, ein Ausdruck, der zu einem booleschen Wert evaluieren muss. Hey, das kennen wir doch schon! Richtig, genau wie bei einer Fallunterscheidung mit if/else gibt eine Bedingung an, ob die Anweisungen im Schleifenrumpf (weiter) ausgeführt werden. Wir haben hier einen Vergleich geschrieben, der den Inhalt der Variablen Zahl auf „kleiner als" 10 prüft. Da wir die Variable mit 0 initialisiert haben, ist das Ergebnis zunächst true. Es schließt sich der Schleifenrumpf an. Wie wir bereits gelernt haben, wird die Anweisungssequenz in geschweifte Klammern gehüllt. Die Schleife wird also zum ersten mal ausgeführt und es geschehen zwei Dinge. Erstens geben wir den aktuellen Wert der Variablen zahl aus. Dies ist offenbar 0. Anschließend zählen wir sie um eins hoch. Wir erinnern uns an Zuweisungen aus dem früheren Kapitel „Variablen". Mittels Zuweisungen wird der Variablen auf der linken Seite ein neuer Wert, definiert auf der rechten Seite als alter Wert plus

Listing 8.1: Schleifen (for-Schleife)

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // Schleifen ermöglichen die wiederholte Ausführung von Anweisungen.
7    // Eine Schleife wird solange ausgeführt, wie die zugehörige Bedingung
         wahr ist.
8    var zahl int = 0
9    for zahl < 10 {
10     fmt.Println(zahl)
11     zahl = zahl + 1
12   }
13
14   // Kompaktere Schreibweise:
15   // Kopf umfasst:
16   //  Deklaration und Initialisierung der Schleifenvariablen (wird einmal
         zu Beginn ausgeführt)
17   //  Bedingung der Ausführung (wird vor jedem Durchlauf geprüft)
18   //  Update-Anweisung (wird nach jedem Durchlauf ausgeführt)
19   for zahl := 0; zahl < 10; zahl++ {
20     fmt.Println(zahl)
21   }
22 }
```

eins, zugewiesen. Nachdem der Schleifenrumpf durchlaufen wurde, setzt sich der Kontrollfluss im Schleifenkopf fort. Er wiederholt sich also gegebenenfalls wie man es von einer Schlafe erwartet. Das Wort „for" kannst du umgangssprachlich als „solange" betrachten. Solange zahl einen Wert kleiner als 10 beinhaltet, wird die Anweisungssequenz erneut ausgeführt. Das ist auch beim zweiten Durchlauf mit Wert 1 der Fall. Wir wiederholen also den Ablauf von Ausgabe und Erhöhung noch einige Male. Schließlich sind wir beim Wert 9 angelangt. Auch diesmal wird der Wert von zahl erhöht. Nun sind wir beim Wert 10, der der die Bedingung zahl $<$ 10 jedoch zu false werden lässt. Wir durchlaufen die Schleife nicht mehr und setzen den Programmablauf hinter der Schleife (schließende geschweifte Klammer) fort. Die Schleife hat also die Zahlen 0 bis 9 ausgegeben. Es ist wichtig zu verstehen, dass 9 hier die letzte ausgegeben Zahl war, obwohl zahl bereits den Wert 10 hat. Das liegt an der Reihenfolge der Aktionen: Die Prüfung erfolgt vor Ausgabe und erneutem Erhöhen. Wir sehen, dass wiederholte Ausführungen mit Hilfe der for-Schleife sehr kompakt auszudrücken sind.

- Im zweiten Abschnitt sieht man eine verkürzte und sehr viel üblichere Form der for-Schleife. Sie mag auf dich zunächst komplizierter wirken, du solltest sie aber bevorzugt verwenden. Der Schleifenkopf wird hier erweitert. Nach dem bekannten for-Schlüsselwort zur Einleitung folgt die Deklaration und Initialisierung einer

Schleifenvariablen. Wenn du dir erneut die erste for-Schleife ansiehst, wirst du feststellen, dass dies lediglich die Variablendeklaration und -Initialisierung ist, die wir zuvor separat geschrieben hatten. Es folgt, getrennt durch ein Semikolon, die Bedingung, wie wir sie bereits kennen. Nun schließt sich, ebenfalls durch ein Semikolon getrennt, eine Update-Anweisung an. Sie entspricht funktional der Erhöhung unserer Variablen um eins aus der ersten Schleife (Inkrement). variable++ ist lediglich eine Kurzform für variable = variable + 1 und wird gerne benutzt. Der Schleifenkörper gestaltet sich wie bereits bekannt. Gehen wir nun näher auf die zwei neuen Elemente im Schleifenkopf ein. Die Deklaration und Initialisierung der Schleifenvariablen (direkt hinter „for") wird genau einmal, vor dem ersten Prüfen der Schleifenbedingung und dem ersten Durchlauf ausgeführt. Aus diesem Grund nutzt man diese Stelle für das bekannt machen und vorbelegen einer Schleifenvariablen. Hinter der uns bereits bekannten Bedingung folgt die Update-Anweisung. Sie wird immer am Ende eines jeden Durchlaufs ausgeführt, unmittelbar bevor die Schleifenbedingung neu geprüft wird. Somit kann man hier Aktualisierungen, wie das Hochzählen unterbringen. Zur besseren Übersicht für dich nochmal der genaue Ablauf in Kurzform:

1. Deklaration und Initialisierung der Schleifenvariablen: zahl wird bekannt gemacht und mit dem Wert 0 vorbelegt
2. Prüfung der Bedingung zur Ausführung („Schleifeninvariante"): Der Wert von zahl wird gegen „kleiner" 10 verglichen. Wenn die Bedingung true (wahr) ist, gehe zu 3. sonst gehe zu 6.
3. Ausführung des Schleifenrumpfes mit allen enthaltenen Anweisungen: Ausgabe des aktuellen Wertes von zahl.
4. Durchführung der Update-Anweisung: zahl wird um eins erhöht (inkrementiert)
5. Gehe wieder zu 2. (Prüfen der Bedingung)
6. Ende der Schleife, da Bedingung nicht (länger) erfüllt. Fortsetzung hinter der schließenden geschweiften Klammer.

Genau wie die vorherigen Kapitel, solltest du auch das Konzept der Schleife gut verinnerlichen. Ich empfehle dir ein paar eigene Schleifen zu schreiben. Gib z.B. hundert mal deinen Namen aus. Oder zähle einen Countdown von 1000 herunter. Es ist vor allem wichtig, dass du den genauen Ablauf verinnerlichst.

9 Übung

Wir haben nun die wichtigsten Konstrukte der Programmierung kennengelernt. Keine Angst, wir sind noch nicht am Ende. Jedoch sind wir nun an einem Punkt, an dem es Sinn macht, das Gelernte zu kombinieren. Nur indem man gelernte Theorie kombiniert und anwendet, kann man wirklich Programmieren lernen. Auswendig lernen funktioniert dabei nicht. Das ist das Schöne daran! Wir werden daher eine kleine Übung starten, die du zunächst allein bearbeiten sollst. Anschließend analysieren wir eine mögliche Lösung. Du wirst danach einen besseren Eindruck vom Nutzen der zuvor behandelten Konzepte haben. Weiterhin wirst du feststellen, wie komplex bereits ein recht kleines Programm mit einer Hand voll Konstrukten werden kann. Das liegt vor allem in der Natur der Kombinatorik begründet. Deine Aufgabe lautet wie folgt:

- Gib alle ganzen Zahlen von 1 bis 100 (jeweils einschließlich) untereinander aus. Dabei gelten folgende Ausnahmeregeln:
  - Wenn eine Zahl ohne Rest durch 3 teilbar ist, gib anstelle der Zahl "Programmieren" aus.
  - Wenn eine Zahl ohne Rest durch 5 teilbar ist, gib anstelle der zahl "Go" aus.
  - Wenn eine Zahl ohne Rest sowohl durch 3, als auch durch 5 teilbar ist, gib statt der Zahl "Programmieren in Go" aus.

Zum besseren Verständnis zeige ich dir die Ausgabe, welche durch das Programm erzeugt werden soll:

Du solltest die Aufgabenstellung nun verstanden haben. Bitte versuche jetzt, eine Lösungen zu entwerfen und programmiere sie aus. Geh ruhig in kleinen Schritten vor und probiere immer wieder aus, ob dein Programm so läuft, wie du es erwartest. Wenn nicht, geh ein Stück zurück und korrigiere die Fehler. Es empfiehlt sich in kleinen Schritten vorzugehen und die Aufgabe in Teilprobleme zu zerlegen. In kleinen Schritten kommt man schneller und sicherer ans Ziel! Eine sinnvolle Unterteilung in Teilschritte könnte wie folgt lauten:

1. Ausgabe der Zahlen von 1 bis 100 untereinander, zunächst ohne Wortersetzung.
2. Ermitteln ob eine Zahl durch 3 teilbar ist. Zum Beispiel kannst du hinter jeder Zahl zunächst ausgeben, ob sie durch 3 teilbar ist. Hierzu eine kleine Berechnungsvorschrift, mit der du bestimmen kannst, ob eine Zahl durch 3 teilbar ist:
   a) Nimm eine Kopie der aktuellen Zahl

Listing 9.1: Erwartete Ausgabe zur Übung

```
1  1
2  2
3  Programmieren
4  4
5  Go
6  Programmieren
7  7
8  8
9  Programmieren
10 Go
11 11
12 Programmieren
13 13
14 14
15 Programmieren in Go
16 16
17 17
18 Programmieren
19 19
20 Go
21 ... (usw. bis einschließlich 100, ersetzt durch Go)
```

b) Solange die Kopie deiner aktuellen Zahl > 0 ist:

i. Subtrahiere 3 und prüfe wieder b

c) Die Zahl ist nun entweder gleich 0, dann ist sie ohne Rest durch 3 teilbar. Wenn sie kleiner als 0, also negativ ist, ist sie nicht durch 3 teilbar.

3. Anwendung der Bestimmung der Teilbarkeit auf die zuvor aufgestellte Teilbarkeit durch 3.

4. Prüfung von Teilbarkeit auf 3 UND (AND mittels &&-Operator) 5. Dies muss nun vor der Bestimmung von 2. und 3. geschehen.

Mit Hilfe der Vorgehensbeschreibung solltest du nun in der Lage sein, die Aufgabe zu lösen. Gib nicht zu schnell auf! Schlage ruhig nochmal das ein oder andere frühere Beispiel nach. Wenn du eine Lösung gefunden hast oder wirklich nicht mehr weiter weißt, analysieren wir nachfolgend eine mögliche Lösung:

Bitte nimm dir etwas Zeit, um die Lösung genau anzusehen. Kommen dir die Bestandteile bekannt vor? Sieht deine Lösung ähnlich aus, vielleicht sogar kürzer? Ich werde die Lösung erläutern. Wir beginnen im Kopf der ersten for-Schleife. Jeder Stichpunkt danach bezieht sich dann auf einen Abschnitt bis zur nächsten Leerzeile.

- Wir schreiben eine „große" for-Schleife, da wir die Zahlen von 1 bis 100 nicht händisch aufschreiben wollen und müssen. Es gibt viel bessere Wege dieses Ziel zu erreichen. Ein guter Ausgangspunkt ist daher eine Schleife, die für sich genommen

Listing 9.2: Erste Lösung zur Übung

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    for zahl := 1; zahl <= 100; zahl++ {
7
8      var teilbarDrei bool = false
9      var teilbarFünf bool = false
10
11     var zahlKopie int = zahl
12     for zahlKopie > 0 {
13       zahlKopie = zahlKopie - 3
14     }
15     teilbarDrei = zahlKopie == 0
16
17     zahlKopie = zahl
18     for zahlKopie > 0 {
19       zahlKopie = zahlKopie - 5
20     }
21     teilbarFünf = zahlKopie == 0
22
23     if teilbarDrei && teilbarFünf {
24       fmt.Println("Programmieren in Go")
25     } else if teilbarDrei {
26       fmt.Println("Programmieren")
27     } else if teilbarFünf {
28       fmt.Println("Go")
29     } else {
30       fmt.Println(zahl)
31     }
32   }
33 }
```

einfach nur von 1 bis 100 hoch zählt. Wichtig ist, dass wir bei 1 beginnen und der Vergleich <= (kleiner oder gleich lautet) Nur so läuft die Schleife für alle Zahlen bis einschließlich 100 durch. Bei 101 wird der Rumpf nicht mehr ausgeführt.

- Wir wissen, dass wir für die aktuelle Zahl, unsere Schleifenvariable, Teilbarkeiten bestimmen sollen. Um die Teilbarkeit zu ermitteln und später zur modifizierten Ausgabe abzufragen, müssen wir die Antwort auf die Frage der Teilbarkeit irgendwo aufbewahren. Dazu deklarieren und initialisieren wir uns zwei Variablen vom Typ „bool" da eine Antwort nur ja (true) oder nein (false) lauten kann. Als Initialwert wählen wir false, da wir erst einmal davon ausgehen müssen, dass die aktuelle Zahl nicht teilbar ist. Andernfalls weisen wir der Variablen später den Wert true zu. Wir legen jeweils für die Teilbarkeit durch 3 und 5 eine eigene Variable an und geben

ihnen einen sprechenden Namen. Das macht das Programm besser lesbar. Du fragst dich vielleicht, warum wir die Variablen immer wieder anlegen müssen, das heißt in jedem Schleifendurchlauf. Das liegt, daran dass die beiden Variablen zur Teilbarkeit nur innerhalb eines Durchlaufs des Rumpfes gültig sind. Variablen haben stets eine begrenzte Gültigkeit, die durch den umschließenden Block begrenzt wird. Die for-Schleifen-Variable ist für die Dauer der gesamten Schleifenausführung gültig. Nach Abarbeitung der gesamten Schleife ist die Variable also nicht mehr bekannt. Der Versuch sie zu verwenden scheitert. Der Compiler meldet eine unbekannte Variable. Man könnte also auch eine neue Variable z.B. mit gleichem Namen anlegen. Die Variablen, die innerhalb der Schleife deklariert werden, sind wiederum nur innerhalb selbiger gültig. In einem neuen Durchlauf sind sie also nicht mehr bekannt und wir legen sie erneut an.

- Nun wollen wir zuerst die Teilbarkeit durch 3 bestimmen. Du kannst natürlich auch mit 5 anfangen und die Teilbarkeit durch 3 hinterher bestimmen. Wir halten uns zur Bestimmung an die Berechnungsvorschrift von oben. Wir legen uns eine neue Variable an, welche eine Kopie der aktuellen Zahl ist. Dies tun wir, da wir sie verändern müssen und wir nicht unsere Schleifenvariable modifizieren wollen. Würden wir mit ihr arbeiten, wäre die Abarbeitung der Schleife im folgenden beendet. Wir legen also eine Kopie an und belegen sie mit dem aktuellen Schleifenvariablen-Wert. Nun schreiben wir eine kleine for-Schleife, deren Bedingung der Vergleich der Kopie mit „größer als" 0 ist. Das heißt, wir haben eine innere, kleine Schleife, die solange durchlaufen wird, wie zahlKopie größer 0 ist. Es ist wichtig, dass wir zahlKopie in jedem Durchlauf verringern. Für die Teilbarkeit durch 3 muss dies 3 sein. Wenn wir keine Verringerung durchführen würden, wäre die Schleifenbedingung ewig erfüllt und wir hätten eine Endlosschleife! Das passiert vor allem Anfängern häufiger. Doch keine Angst. Du kannst so ein „hängendes" Programm beenden. Im Falle des Go Playground hast du entweder einen Button, um das Programm abzubrechen oder du schließt einfach dein Browser-Fenster (Bitte kopiere dir aber vorher deinen Stand) und öffnest ein neues. Mit wachsender Erfahrung wird dir so ein klassischer Fehler nicht mehr unterlaufen! Wenn unsere innere Schleife endet, wissen wir, dass zahlKopie entweder 0 oder negativ ist. Also führen wir im Anschluss einen Vergleich durch (rechts vom einfachen Gleichheitszeichen). zahlKopie wird auf Gleichheit mit 0 verglichen. Das boolesche Ergebnis dieses Vergleichs wird der Variablen auf der linken Seite (links vom einfachen Gleichheitszeichen) zugewiesen. Der Ausdruck wird also von rechts nach links ausgewertet, wie wir bereits über Variablen und Zuweisungen gelernt haben. teilbarDrei enthält nun entweder den Wert true (z.B. für die Zahl 3) oder false (z.B. für 1).
- Wir wiederholen nun das Verfahren analog für die Teilbarkeit durch 5. zahlKopie können wir wiederverwenden, indem wir sie wieder auf den aktuellen Schleifenvariablen-Wert setzen, denn wir sind ja immer noch im selben Durchlauf. In der inneren Schleife müssen wir nun jedoch jeweils 5 subtrahieren. Der

Vergleich gegen 0 liefert uns das Ergebnis für unsere zu Beginn angelegte Variable (teilbarFünf).

- Der anspruchsvollste Teil ist geschafft! Es bleibt nur noch festzustellen, welche Teilbarkeit zutrifft. Dazu verwenden wir mehrere if-else-Verzweigungen. Wichtig ist hier vor allem die Reihenfolge. Zuerst testen wir auf Teilbarkeit durch 3 UND (AND) 5. Dafür verwenden wir den &&-Operator. Dieser führt einen logischen AND Vergleich aus, wie er im Kapitel über Datentypen behandelt wurde. Wenn der Fall zutrifft, geben wir „Programmieren in Go" aus. Es ist wichtig, dass wir diesen Fall zuerst behandeln, da uns sonst die Ausgabe für Teilbarkeit durch 5 oder 3 zuvorkommen könnte ohne dass wir die Prüfung auf beides bereits durchgeführt hätten. Trifft dieser Fall nicht zu, prüfen wir auf Teilbarkeit durch 3 und anschließend 5 (oder umgekehrt). Die letzte Alternative unterliegt keinen weiteren Bedingungen und gibt somit einfach die Zahl aus.

Du hast nun gesehen, wie man die einzelnen Elemente der Programmierung auf die Lösung eines konkreten Problems anwenden kann. Da es (fast) immer mehrere Möglichkeiten gibt zum Ziel zu kommen, möchte ich dir eine weitere Lösung vorstellen. Diese nutzt einen Operator, den wir bisher noch nicht behandelt haben, den Modulo-Operator. Der Modulo Operator ermittelt den Rest einer ganzzahligen Division. Damit können wir unser Programm deutlich vereinfachen, was stets ein erstrebenswertes Ziel ist. Programme so einfach wie möglich zu halten ist bei realen (sehr großen) Programmen eine große Herausforderung. Je einfacher ein Programm jedoch zu verstehen ist, desto leichter ist die Wartung. Dazu gehört z.B. die Suche und das Beheben von Fehlern oder das Hinzufügen neuer Funktionen. Nachfolgend ist die zweite, verbesserte Version des Programms aufgeführt. Nur zu, probiere es aus. Dann bekommst du ein Gefühl dafür, dass es sich hierbei wirklich um ein einfacheres Programm handelt.

Wenn du genau hinsiehst, bemerkst du, dass die beiden Schleifen zur (aufwändigen) Bestimmung der Teilbarkeit weggefallen sind. Stattdessen können wir die Teilbarkeit in jeweils einer Zeile bestimmen. Der Modulo-Operator wird mit dem Prozent-Zeichen (%) angegeben und erwartet links und rechts von sich eine ganze Zahl. Das Ergebnis vergleichen wir mittels Gleichheitsoperator (==) auf Rest 0. Das boolesche Ergebnis rechts von := wird der neuen Variable teilbarDrei bzw. teilbarFünf zugewiesen. Wir erinnern uns: Zuweisungen werden von rechts nach links ausgewertet und mit Hilfe der Kurzform := können wir eine Variable deklarieren und initialisieren ohne den Typ explizit anzugeben. Der Rest des Programms bleibt unverändert.

Listing 9.3: Verbesserte Lösung zur Übung

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // Bessere Lösung:
7    // Modulo Operator % ermittelt Rest einer Division
8    for zahl := 1; zahl <= 100; zahl++ {
9
10     teilbarDrei := zahl%3 == 0
11     teilbarFünf := zahl%5 == 0
12
13     if teilbarDrei && teilbarFünf {
14       fmt.Println("Programmieren in Go")
15     } else if teilbarDrei {
16       fmt.Println("Programmieren")
17     } else if teilbarFünf {
18       fmt.Println("Go")
19     } else {
20       fmt.Println(zahl)
21     }
22   }
23 }
```

10 Was guten Programmcode ausmacht

Anhand der vorherigen Übung siehst du, dass die Art und Weise, wie man eine Aufgabe löst, einen großen Einfluss auf den Quelltext hat. Nicht immer kann man objektiv feststellen, ob ein Lösungsweg besser oder schlechter ist als ein anderer. Als grobe Richtschnur möchte ich dir jedoch ein paar (relativ) objektive Kriterien auflisten, nach denen man ein Programm bewerten kann.

1. Funktionalität: Es ist wichtig, dass ein Programm das tut, wofür es geschaffen wurde. Es soll also den Anforderungen genügen. Es sollte eine angemessene Umsetzung der Anforderungen bereitstellen. Diese sollte weder „vergoldet" noch ungenügend ausfallen. Auch die Benutzbarkeit einer Anwendung ist maßgeblich für die Akzeptanz durch den Nutzer. Wie sieht die Anwendung aus? Wie fühlt sich das Programm an? Ist es intuitiv zu bedienen?

2. Lesbarkeit: Lesbarkeit zählt zu den wichtigsten Kriterien für ein hervorragendes Programm. Stell dir vor, ein anderer Programmierer müsste eines deiner Programme lesen und abändern. Wie lange würde er dafür brauchen? Würde er es schaffen? Würde er fluchen beim Lesen oder wäre für ihn alles glasklar? Damit man ein Programm ändern kann, muss man es zunächst verstehen. Verständnis setzt Lesbarkeit voraus, genau wie bei einem Buch. Es ist kinderleicht kryptischen, unverständlichen Code zu schreiben, der bald weggeworfen werden muss, weil ihn selbst der Entwickler nach ein paar Wochen Pause nicht mehr versteht. Auch wenn sich das komisch anhört, hierdurch können große (wirtschaftliche) Schäden entstehen. Stell dir vor, ein Autor würde nach kurzer Zeit sein eigenes Buch nicht mehr verstehen. Klingt komisch? Ist in der Programmierung aber durchaus üblich. Woran macht sich nun gute Lesbarkeit fest? Dazu gehören viele Kleinigkeiten. Vor allem gehört auch ein gutes Stück Erfahrung dazu. Nachfolgend sind ein paar Punkte aufgezählt:

    a) Sprechende Namen: Egal ob Variablennamen, Funktionsnamen, Packages oder andere Bezeichner. Ein Name, unter dem sich jeder etwas vorstellen kann, ist immer besser als ein kryptischer (kurzer) Name. Dementsprechend solltest du einen Zähler nicht i nennen, wie vielleicht in der Mathematik üblich, sondern besser counter (oder zähler). Eine Adresse sollt nicht adr heißen, weil das eine coole Abkürzung ist, die für viele andere Wörter stehen könnte, sondern address. In jeder Programmiersprache gibt es eine Konvention zur Formatierung von zusammengesetzten Wörtern. Go verwendet hier CamelCase, wobei man den ersten Anfangsbuchstaben klein,

Anfangsbuchstaben von weiteren Wörtern groß schreib. Ein Beispiel wäre “timeInSeconds”, “nachName” oder “zubehoerTeileNummer”.

b) Einhaltung der Formatierungsregeln. Dazu zählen Einrückungen, das Setzen von Klammern und ähnliche Dinge. Während viele Regeln Projekt-spezifisch sind, hat Go eine Standard-Formatierung, die befolgt werden sollte. Sie kann auch automatisch korrigiert werden. Im Go Playground kannst du dazu den Button “format” verwenden.

c) Sinnvolle Unterteilung des Programms in Teilprogramme. Dahinter steht das alte Prinzip „Teile und herrsche”. Wir werden darauf in folgenden Kapiteln noch näher eingehen. Es sei schon mal gesagt, dass Funktionen nach ihren jeweiligen Zuständigkeiten getrennt werden sollten. Andernfalls entsteht sogenannter “Spaghetti-Code”, ein Programm dass alles ineinander vermischt tut und in dem kein roter Faden zu erkennen ist.

3. Änderbarkeit: Änderbarkeit drückt aus wie leicht oder schwer ein Programm angepasst werden kann. Wenn du den Punkt Lesbarkeit beachtest, ist das Fundament für Änderbarkeit bereits geebnet. Es gibt jedoch weitere Merkmale die Änderbarkeit einfacher oder komplizierter machen.

   a) Wie leicht oder kompliziert ist es Änderungen einer bestimmten Kategorie zu machen? Wenn sehr typische Anforderungsänderungen auftauchen, sollten dafür am besten nur geringfügige Anpassungen notwendig sein. Ein einfaches Beispiel: Wenn der Nutzer beispielsweise eine andere Hintergrundfarbe will, musst du dann an 20 Stellen die Hintergrundfarbe ändern? Musst du nur an einer Stelle die Hintergrundfarbe ändern? Kann der Nutzer die Einstellung vielleicht sogar selbst ändern?

   b) Wie leicht ist es das geänderte Programm zu prüfen? Nach einer Änderung will man nicht nur wissen, ob die neue Funktionalität die neuen Anforderungen erfüllt. Man will auch sicher sein, dass die bereits früher implementierte Funktionalität nicht versehentlich zerstört wurde. Das kann bei komplexen Programmen leicht passieren und erfordert gute (automatisierte) Tests und vor allem gute Testbarkeit des Programms. Dieses Thema ist mindestens ein eigenes Buch wert.

   c) Wie leicht lässt sich das Programm installieren? Wie leicht lässt es sich aktualisieren? Ist es plattformunabhängig? Ist es leicht zu portieren oder hat man sich bewusst für eine Plattform entschieden?

4. Zuverlässigkeit: Hierunter versteht man alle Eigenschaften, die ein Programm robust machen. Dazu gehören z.B. Antworten auf folgende Fragen: Wie häufig tauchen Fehler auf? Wie kritisch sind diese Fehler? Wie hoch ist die Wahrscheinlichkeit eines Systemausfalls?

5. Effizienz: Funktionalität kann auf verschiedene Wege erreicht werden. Programmabläufe werden als „Algorithmen” bezeichnet. Ein Algorithmus

beschreibt detailliert wie ein Programm ein gegebenes Problem löst. Je nach den verwendeten Algorithmen, kann ein Programm mehr oder weniger viele Rechen-Ressourcen verbrauchen. Beispielsweise können hohe CPU-Lasten zu einer Rechenzeit führen, die als unangenehm empfunden wird. Auch ein zu hoher Speicherverbrauch und ähnliche Konsequenzen können negative Auswirkungen haben. Die Effizienz von Algorithmen und deren Implementierungen kann erheblich variieren. Es lohnt sich also für rechenintensive Operationen effiziente Algorithmen nachzuschlagen. Im Zweifelsfall sind konkrete Messungen die erste Maßnahme für Performance-Optimierungen. Man sollte sich jedoch davor hüten, im Voraus beliebige Annahmen über die Effizienz zu machen. Viel unlesbarer Code wird „im Namen der Effizienz" produziert, obwohl das Programm dadurch gar nicht schneller wird.

Das waren die wesentlichen Kriterien für die Qualität von Programmen. Du siehst, es gibt viele Ansatzpunkte und je nach Zweck können unterschiedliche Punkte wichtig sein. Man sollte jedoch keinen der Punkte komplett vernachlässigen, da irgendein Nachteil sonst massiv spürbar sein wird und zu einer hohen Unzufriedenheit der Anwender oder Entwickler führen kann.

11 Funktionen

Wir widmen uns nun einem Thema, das wir bei unserem ersten Programm (Hello world!) bereits kurz angesprochen haben. Es geht um Funktionen als zentrales Konzept, um Code zu kapseln. Eine Funktion dient (genau) einem Zweck und umfasst somit ein kleines Teilprogramm. Nehmen wir die bisher stets verwendete main-Funktion. Sie definiert den Einstiegspunkt des Programms und dient in einem sauber strukturierten Programm daher nur diesem Zweck. Lass dich nicht dadurch täuschen, dass wir bisher sämtlichen Code in diese Funktion geschrieben haben. Bei sehr kleinen Programmen und Übungen ist das okay. Auch ein kurzes Kommandozeilen-Programm kann man komplett in main abhandeln. Wenn ein Programm jedoch wächst und in Bereiche von z.B. über 100 Zeilen kommt, sollte die main-Funktion nicht mehr als „Topf für alles" dienen. Der Grund ist, du ahnst es bereits, die Lesbarkeit und Wartbarkeit des Programms. Wir werden das an nachfolgendem Beispiel untersuchen. Nehmen wir erneut die Aufgabenstellung aus der letzten Übung. Wenn du dir das Programm aus der Perspektive von Teilproblemen ansiehst, kommen wir zu folgender Unterteilung:

1. Laufen über den Zahlenbereich von 1 bis 100 (äußere for-Schleife).
2. Bestimmen der Teilbarkeit durch 3
3. Bestimmen der Teilbarkeit durch 5
4. Ausgabe des Ergebnisses (Zahl oder Wort, je nach Teilbarkeit)

Diese Aufteilung erscheint sehr natürlich, wenn man in Teilaufgaben oder Teilprogrammen denkt, meinst du nicht? Wenn wir „teile und herrsche" anwenden, nutzen wir verschiedene Vorteile:

- Der menschliche Verstand kann komplexe Probleme nicht komplett und auf einmal erfassen. Es ist viel einfacher, sich Teilaspekte nacheinander anzusehen um hinterher ein grobes „Gesamtbild" zu erhalten. Das ein komplexes Problem überhaupt verstanden wird, hängt oftmals davon ab, ob man es auf eine Reihe von Teilproblemen zurückführen kann.
- Nur mit Teilprogrammen und Modularisierung hat man überhaupt eine Chance, große Programme wartbar zu halten. Du kannst dir das wie einen Baukasten vorstellen. Wenn du etwas aus wohldefinierten Bausteinen baust, kannst du es leicht verändern. Wenn du jedoch einen Gegenstand „aus einem Guss" vor dir hast, kannst du ihn nicht einfach an deine (neuen) Wünsche anpassen.

- Teilprogramme ermöglichen die Wiederverwendung von Code. Stell dir vor, du möchtest eine Wand streichen. Du musst nur einmal lernen, wie man eine Wand streicht. Danach kannst du Wände in allen Farben streichen. Ähnlich verhält es sich in der Programmierung. Die Duplizierung (Wiederholung) von ähnlichem Code führt zu vielen Problemen. Besser ist es, Funktionen allgemeiner zu halten und von der konkreten Farbe zu „abstrahieren", da sie für das Vorgehen nicht entscheidend ist.

- Der Entwicklungsfortschritt ist leichter zu erfassen und Arbeit besser zu verteilen. Wenn man einen langen Weg in viele Etappen unterteilt, ist es nicht nur motivierender dem Ziel näher zu kommen, man kann es auch viel leichter messen. Softwareprojekte zu managen ist ein komplexes Thema. Jedoch sei hier gesagt, dass die Ressourcen-Planung, insbesondere bzgl. Zeit, eine sehr große Herausforderung ist. Dass Software-Projekte erst oberhalb der vereinbarten Zeiten und Kosten fertiggestellt oder sogar abgebrochen werden müssen, ist leider immer noch häufig die Realität. Darüber hinaus wird komplexe Software in Teams entwickelt. Wenn du ein großes Programm in Teilprogramme unterteilst, lässt sich die Entwicklung besser auf mehrere Entwickler verteilen.

Sehen wir uns nun ein Beispielprogramm an, dass Funktionen verwendet um Teilprogramme auszugliedern:

Vermutlich erkennst du sehr schnell, dass der eigentliche Code, innerhalb der Funktionskörper, unserer letzten Lösung zur Übung entspricht. Lediglich die Aufteilung hat sich verändert. Wenn wir uns ausschließlich die main-Funktion ansehen, fällt auf, dass diese sehr leicht verständlich ausdrückt, was das Programm tut. Es geht die Zahlen von 1 bis 100 durch. Danach wird die Teilbarkeit ohne Rest durch 3 bestimmt. Im Anschluss wird das Verfahren für die Zahl 5 wiederholt. Schließlich wird ein Ergebnis ausgegeben und die Schleife beginnt von vorn. Dieser High-Level-Blick erleichtert anderen Entwicklern (und nach einiger Zeit auch dir selbst) das Verständnis. Voraussetzung dafür ist jedoch, dass man den Umfang von Funktionen sinnvoll schneidet und sprechende Namen vergibt.

- Sehen wir uns die erste neue Funktion an: istTeilbarOhneRest. Wie die main-Funktion auch, wird eine neue Funktion mit dem Schlüsselwort „func" eingeleitet. Es folgt ein beliebig wählbarer (Ausnahme main-Funktion), vorzugsweise sprechender, Bezeichner. istTeilbarOhneRest ist ein sprechender Bezeichner, da er kurz und klar ausdrückt, was die Funktion tut. Ein guter Funktionsname führt zu keinen bösen Überraschungen, da die Funktion genau das tut, wofür ihr Name steht. Es folgt eine Liste von Parametern, begrenzt durch runde Klammern am Anfang und Ende der Liste sowie Kommas zwischen einzelnen Parametern. Parameter haben Ähnlichkeit mit Variablen. Sie haben einen Namen und einen Typ. Beide müssen explizit angegeben werde, anders als bei Variablen, bei denen der Typ gegebenenfalls abgeleitet werden kann. Parameter enthalten Daten zur Interaktion mit der „Außenwelt". Über die Angabe von divisor und divident kann man die Funktion flexibel verwenden. So muss ich nicht für die

Teilbarkeit durch 3, 5 oder andere Werte neuen Code schreiben. Nein, ich kann dieselbe Funktion einfach mit anderen Parametern aufrufen. Die konkreten Werte setze ich einfach an die entsprechende Stelle in die runden Klammern beim Aufruf. Doch zurück zu Funktionsdefinition. Hinter den Übergabeparametern folgen optionale Rückgabeparameter. Sie werden zwischen der schließenden runden Klammer und der öffnenden geschweiften angegeben. Wir geben hier, wie üblich nur den Typen an, Go erlaubt hier optional auch die Angabe eines Namens. Im Gegensatz zu vielen anderen Programmiersprachen darf man in Go sogar mehrere Rückgabewerte liefern. Sie werden dann, ähnlich wie die Übergabeparameter, mit Kommas voneinander getrennt angegeben. Bei der Verwendung von Namen in Rückgabeparametern werden hier auch runde Klammern fällig. Gleiches gilt für die Definition von mehreren Rückgabewerten. Rückgabewerte dienen dazu ein Ergebnis der Funktion nach außen bekannt zu machen. Im Gegensatz dazu, werden Übergabeparameter verwendet um Werte beim Aufruf von außen hineinzugeben. Bei beiden ist die Reihenfolge zwar bei der Definition beliebig, beim Aufruf jedoch stimmig zu halten. Das liegt daran, das anhand der Position des konkreten Wertes beim Aufruf („Argument") der entsprechende Parameter zugeordnet wird. Das bedeutet hier konkret, beim Aufruf muss ich darauf achten, dass ich zuerst den Divisor, dahinter den Divident angebe. Andernfalls bekomme ich ein falsches Ergebnis. Die Funktion selbst arbeitet mit den Parametern wie mit Variablen. Beim Aufruf werden die konkreten Werte den Parametern als Kopie zugewiesen und sind innerhalb der Funktion gültig. Es ist wichtig zu wissen, dass die übergebenen Werte kopiert (!) werden. Man übergibt also nicht das Original. Stell dir vor, du überreichst eine Kopie einer Urkunde an eine Person als Nachweis. Die Person kann zwar mit den Daten arbeiten und die Kopie verändern, eine Änderung der Kopie wirkt sich jedoch nicht auf dein Originaldokument aus. Daher machen wir das Ergebnis der Berechnung als Rückgabewert verfügbar und nicht als „leere Eingabe". Das geschieht mit Hilfe des Schlüsselworts „return". Return beendet eine Funktion sofort und gibt die angegeben Werte als Rückgabewerte nach außen. Die Programmausführung setzt also hinter der Stelle des Aufrufs (hier in main) fort. Der zurückgegebene Wert kann dann optional verwendet werden, z.B. indem man ihn sofort an eine Variable zuweist. Der Compiler fordert für jede Funktion, die einen Rückgabewert definiert, auch mindestens ein entsprechendes return. Wer angibt, Werte zurückzuliefern, muss das auch tun! Anders verhält es sich bei Funktionen ohne Rückgabewert (siehe z.B. main). Es ist kein return notwendig. Die Ausführung endet an der schließenden geschweiften Klammer.

- Sehen wir uns nun die zweite neue Funktion an: ergebnisAusgeben. Wir fragen uns bei der Programmierung: Welche Daten braucht diese Funktion, um ein passendes Ergebnis auszugeben? Nun, sie muss wissen ob die Teilbarkeit durch 3 gegeben ist, ebenso für 5 sowie die eigentliche Zahl kennen. So können wir alle 3 Ausgabeszenarien handhaben. Dies geschieht wie bekannt mit unserer if-else-Kaskade. Je nachdem, welche konkreten Werte für die Parameter übergeben wurden, lauten die Inhalte der Parameter entsprechend. Wir fragen sie ab und

handhaben sie wie erwartet. Da wir lediglich Werte ausgeben und kein Ergebnis im Aufrufkontext brauchen, benötigen wir auch keinen Rückgabewert und kein return-Statement.

Du siehst, Funktionen verbessern ein Programm spürbar. Es ist dazu jedoch erforderlich, dass man gründlich darüber nachdenkt, wie man seine Funktionen schreibt. Es lohnt sich daher, ab und zu kurz inne zuhalten oder auch nachträglich Funktionen zu schreiben und Teilprogramme so auszulagern. Andere Entwickler und später auch du selbst, werden es dir danken oder es ab einem gewissen Niveau auch erwarten! Abschließend möchte ich dir noch ein paar kurze Kriterien an die Hand geben, mit denen du die Güte einer Funktion überprüfen kannst:

1. Trägt die Funktion einen sprechenden Namen? Drückt dieser Name kurz und bündig aus, was die Funktion (ausschließlich) bezweckt? Das „Wie" spielt hierbei keine Rolle. Orientiere dich an den Anforderungen und nimm dabei eine „Vogelperspektive" ein.

2. Erfüllt die Funktion genau einen Zweck oder vermischt sie Verantwortlichkeiten? Der Zweck in unserem Beispiel ist „Die Bestimmung der Teilbarkeit ohne Rest" sowie „Die Ausgabe eines Wertes in Abhängigkeit der Teilbarkeit". Zweck der main-Funktion ist die Ausgabe substituierter Zahlenwerte von 1 bis 100. Es handelt sich dabei um genau einen Zweck, aus der für den Funktionsnamen passenden Perspektive. Beispielsweise würde eine Funktion "bestimmeTeilbarkeitUndGibAus" gegen dieses Prinzip verstoßen. Als Hinweis: Die main-Funktion tut dies nicht. Sie arbeitet auf einer abstrakteren Perspektive, der Ausgabe von Zahlen/Wort-werten.

3. Ist die Funktion kurz? Eine gute Funktion ist relativ kurz. Das bedeutet, dass kurze Funktionen besser sind als lange, da sie leichter zu überblicken sind und weniger zu vermischten Verantwortlichkeiten und Perspektiven tendieren. Hier gibt es keine festen Werte. Als grober Richtwert sei gesagt, dass maximal 15 Zeilen gut und der Regelfall sein sollten. Auch bis zu 30 Zeilen können je nach Situation okay sein. 50 bis 100 Zeilen sollten nur in seltenen, begründeten Fällen akzeptiert werden, da die Lesbarkeit schon massiv leidet. Mehr als 100 Zeilen sollten in jedem Fall vermieden werden. Auch hier kann es Fälle geben, die das rechtfertigen. Doch oftmals sind das nur Ausreden um ein mühsames „Nachbessern" zu vermeiden. Es lohnt sich aber langfristg, hier Aufräumarbeiten anzugehen, bevor die Funktion weiter wuchert und irgendwann nicht mehr wartbar ist.

Du hast nun das elementare Konzept zur sauberen Strukturierung von Programmcode kennengelernt, die Funktion. Du wirst in deinem Lernprozess noch weitere kenne lernen. Doch bis dahin gewöhne dir bitte an, das Konzept von Funktionen nutzbringend anzuwenden.

Listing 11.1: Funktionen zur Strukturierung

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // Mit Hilfe von Funtionen lagert man Teilprogramme aus.
7    // Dies verbessert im Allgemeinen die Lesbarkeit und
         Wiederverwendbarkeit von Programmcode.
8    // Weitere Vereinfachung unseres vorherigen Programms:
9    for zahl := 1; zahl <= 100; zahl++ {
10
11     teilbarDrei := istTeilbarOhneRest(zahl, 3)
12     teilbarFünf := istTeilbarOhneRest(zahl, 5)
13
14     ergebnisAusgeben(teilbarDrei, teilbarFünf, zahl)
15   }
16 }
17
18 // Funktionen erhalten Werte per Parameter übergeben.
19 // Auch für Parameter muss ein Name und Typ angegeben werden.
20 // Funktionen können Werte zurückgeben.
21 func istTeilbarOhneRest(divisor int, divident int) bool {
22   return divisor%divident == 0
23 }
24
25 func ergebnisAusgeben(teilbar3 bool, teilbar5 bool, zahl int) {
26   if teilbar3 && teilbar5 {
27     fmt.Println("Programmieren in Go")
28   } else if teilbar3 {
29     fmt.Println("Programmieren")
30   } else if teilbar5 {
31     fmt.Println("Go")
32   } else {
33     fmt.Println(zahl)
34   }
35 }
```

12 Listen (Slices)

Bestimmt ist dir schon aufgefallen, dass wir bisher nur sehr einfache Datentypen verwendet haben. In jedem dieser Datentypen passt immer nur „eine" Information. Sicher hast du dich schon gefragt, wie man denn damit komplexe Daten behandeln soll. Denken wir z.B. an eine einfache Namensliste. Mit dem bisher Gelernten müssten wir für jeden Namen eine eigene Variable anlegen. Das ist nicht nur mühsam, es führt auch zu schlecht wartbarem Code. Die Antwort lautet daher: Es gibt eine ganze Reihe von Datenstrukturen, welche die Handhabung komplexerer Daten, auch nach einem bestimmten System, ermöglichen. Wir wollen uns in diesem Kapitel einem dieser elementaren Datenstrukturen widmen, den Listen. In Go sind Listen unter dem Begriff „Slices" bekannt. Ein Slice hält eine beliebig lange Liste von Werten eines Datentyps. Genau genommen ist es damit nur eine „Aneinanderreihung" von Speicherplatz für Variablen eines Typs. Go unterstützt Slices als fest eingebaute Datenstruktur für jeden möglichen Datentyp. Werfen wir zunächst einen Blick auf das Code-Beispiel.

Wir gehen das Programm nun wieder Schritt für Schritt durch. Jeder Stichpunkt bezieht sich auf einen Abschnitt zwischen zwei Absätzen mit Kommentaren. Ich empfehle dir, das Beispiel auch Schritt für Schritt auszuprobieren.

- Wir legen uns zunächst ein neues Slice an. Das Prinzip ist dasselbe, wie bei der Deklaration und Initialisierung von Variablen. Wir benötigen einen Namen sowie einen Datentyp und gegebenenfalls das Schlüsselwort var. Der Datentyp setzt sich zusammen aus einem paar eckiger Klammern, als Zeichen für ein Slice und im Anschluss den Datentypen für die abzuspeichernden Elemente. Wir wollen eine Liste aus Strings erstellen, also geben wir []string an. Auf der rechten Seite des Gleichheitszeichens geschieht nun die Initialisierung. Mit Hilfe der eingebauten Basisfunktion „make" werden in Go unter anderem (leere) Slices erstellt. Dabei wird als erster Parameter der Datentyp erwartet (wieder []string) und im Anschluss die Anzahl der initial zu erzeugenden Felder, das heißt Einträge, im Slice. Da wir sofort wissen, dass wir genau 3 Einträge vornehmen wollen, geben wir die Länge 3 an. Eine kurze Testausgabe zeigt uns die Inhalte in den 3 Feldern des neu angelegten Slices. Die Inhalte werden in eckige Klammern eingehüllt. Hier steht jeweils der leere String (""), d.h. die Elemente werden mit dem uns bereits bekannten Zero-Value des Datentyps vorbelegt.

- Da wir mit den vorbelegten Zero-Values nicht viel anfangen können, wollen wir nun einige konkrete Werte in den Elementfeldern ablegen. Wir können uns das Slice als einen Aktenordner mit 3 Fächern vorstellen. In der Informatik und somit auch in der Programmierung ist es üblich, dass Elemente bei 0 beginnend

durchnummeriert werden. Das bedeutet, das erste Feld trägt die Nummer („Index”) 0, das zweite 1, das dritte 2. Das klingt zunächst verwirrend, wird dir aber nach einiger Zeit ganz normal vorkommen. Indem wir den Bezeichner des Slices gefolgt vom gewünschten Index in eckigen Klammern angeben, können wir auf das Feld an der entsprechenden Stelle zugreifen. Mit Hilfe des Zuweisungsoperators (=) weisen wir einen Wert zu. Eine erneute Ausgabe zeigt uns, dass nun unsere zugewiesenen Namen im Slice stehen. Die Reihenfolge entspricht den Indizes. In Feld 0 steht “Alice”, in Feld 1 “Bob” und in Feld 2 “Charlie”.

- Wir haben nun gelernt, dass die Feldindizes bei 0 beginnen. Das bedeutet logischerweise, dass ein Slice mit 3 Feldern keinen Index größer als 2 vorsieht. Zum Testen nimm bitte die Kommentar-Slashes vor namen[3] = "Kracht!" weg und führe dein Programm aus. Der Compiler beschwert sich nicht, das Programm beginnt zu laufen und stößt dann an dieser Stelle auf einen Laufzeitfehler: panic: runtime error: index out of range Dies ist ein häufiger Anfängerfehler. Wer mit dem Zählen von 0 bis Länge-1 noch nicht so vertraut ist, wird öfter mal versuchen auf ein Feld zuzugreifen, dass nicht existiert. Das ist nicht schlimm. Du wirst diesen Fehler nach einiger Zeit nicht mehr machen. Was dann passiert, ist dass die Laufzeigumgebung der Sprache während der Ausführung feststellt, dass du auf ein Feld zugreifen willst, das nicht existiert. Da dieser Fehler nicht einfach ignoriert werden kann und zu einer Fehlfunktion führt, bricht das Programm ab. Der Fehler ist hier ziemlich klar. Wenn du „index out of range” liest, weißt du, dass der Fehler ein ungültiger Index ist. Weiterhin bekommst du einen Hinweis zur Zeilennummer. Bei mir sieht das wie im nachfolgenden Listing aus. Der Hinweis „.../main.go:21...” sagt mir, dass das Programm zuletzt in der Datei „main.go” in Zeile 21 aktiv war, als der Fehler aufgetreten ist. Du bekommst immer so einen „Stacktrace”, eine Auflistung der zuletzt ausgeführten Code-Zeilen, wenn dein Programm zur Laufzeit abstürzt. Das macht die oftmals mühsame Fehlersuche, auch in größeren Programmen, deutlich leichter. Bevor wir nun weitermachen, kommentiere die fehlerhafte Zeile bitte wieder aus oder lösche sie. Slices (oder allgemein Listen als Datenstruktur) haben die Eigenschaft, dass ihre Länge nicht in Stein gemeißelt ist. Die Anzahl der möglichen Elemente kann bei Bedarf vergrößert oder verkleinert werden, was in den meisten Anwendungen auch absolut notwendig ist. Stell dir ein Telefonbuch vor. Es muss je nach Zeitpunkt wachsen oder schrumpfen können, damit man es vernünftig handhaben kann und immer nur so viel Platz verbraucht wird, wie notwendig ist. Wir möchten nun einen weiteren Namen an unser Slice anhängen ohne ein neues mit mehr Feldern anlegen zu müssen. Hierzu existiert die fest eingebaute Funktion „append”. Sie dient dem „Anhängen” von Elementen an ein gegebenes Slice. Somit können wir unsere Liste verlängern. append erwartet das zu verlängernde Slice (bei uns namen) sowie die anzuhängenden Elemente als Aufzählung. Das ist hier einfach ein weiterer Name. Der Rückgabewert (das Ergebnis) von append ist dann die verlängerte Liste. Wir weisen sie einfach wieder unserer bekannten Variable zu und haben so unser Slice um ein Element verlängert. Eine Testausgabe zeigt uns, dass das Element auch tatsächlich angehangen wurde.

- Häufig möchte man wissen, welche Größe ein Slice aktuell hat. Dazu gibt es die fest eingebaute Funktion „len“. len erwartet als einzigen Parameter das zu messende Slice. Rückgabewert ist die Anzahl der Felder im Slice. Wenn du also ein Slice nur deklarierst und keine Elemente hinzufügst und auch kein make aufrufst, ist der Rückgabewert 0. Nach einem make entspricht die Länge der Anzahl der Felder, die du als zweiten Parameter für make angegeben hast. In unserem Fall sehen wir, dass wir aktuell 4 Felder im Slice haben. Nach dem Anhängen eines weiteren Elements mit append ist die Länge 5. Auch hier sei noch einmal darauf hingewiesen, dass bei einer Länge von 5 die einzelnen Felder von 0 bis 4 durchnummeriert werden. Es gibt hier also keinen Index 5!

- Die Behandlung von Slices ist oftmals eng mit dem Konzept der Schleife verbunden. Es liegt in der Natur der Schleife, dass sie für die gleichförmige Verarbeitung einer Menge von Elementen geeignet ist. Wir haben das bereits im Kontext eines Zahlenbereichs gesehen. Im vorliegenden Fall laufen wir per index über alle Felder des Slices („Iteration”). Damit wir wissen, wann wir am Ende angekommen sind, verwenden wir die Funktion len. Auch hier sieht man wieder sehr deutlich, dass die Indizierung bei 0 beginnt und bei Länge - 1 endet, da wir auf kleiner vergleichen. Für die Länge 5 wird der Schleifenrumpf nicht mehr ausgeführt, da dieser Index die Länge überschritten hat! Wir sehen, dass Variablen uns wieder sehr zu Hilfe kommen um flexibel Werte zu repräsentieren. Anstelle konkreter Werte können wir genauso gut eine Variable vom Typ int in die eckigen Klammern setzen, denn so eine Variable steht für eine ganze Zahl, die einen Index repräsentieren kann. Die Schleife zeigt zum Beispiel, wie man einfach alle Werte eines Slices abgreifen und verändern kann. Eine Testausgabe zeigt uns, dass die Werte übernommen wurden.

- Da die zuletzt gezeigte Form der Schleife, das Iterieren über alle Elemente, häufig benötigt wird, bietet Go hierfür eine vereinfachte Variante an. Mit Hilfe der fest eingebauten Funktion „range” geben wir an, dass wir über alle Elemente laufen wollen. range liefert uns jeden Wert nacheinander, ohne dass wir das „Hochzählen” und Abbrechen immer wieder selbst definieren müssen. Es ist äquivalent zum ersten Schleifenkopf. Einen Unterschied gibt es jedoch. range liefert zwei Rückgabewerte. Der erste Rückgabewert ist der aktuelle Index (0 bis len - 1). Der zweite Rückgabewert ist der aktuelle Wert an dieser Stelle. In Go können mehrere Rückgabewerte einfach über eine Aufzählung der Variablen zugewiesen werden. Hinter dem Komma steht anstelle eines zweiten Variablennamens bei uns jedoch ein Unterstrich (_). Dieser Unterstrich ist eine „Wildcard” und bedeutet, dass wir den Wert nicht benötigen. Es ist also keine Variable, sondern ein expliziter Verzicht auf den Wert. Dies ist erforderlich, da der Go Compiler streng auf ungenutzte Variablen achtet. Du kannst es gerne ausprobieren. Die Zuweisung an eine Variable lässt der Compiler nur dann zu, wenn du sie auch verwendest und z.B. den Wert ausgibst. Da ungenutzter Code jedoch zu schlecht wartbarem Code führt, ist der Compiler hier sehr streng. Zu diesem Zweck dient der Unterstrich. Man verwendet ihn dort als Platzhalter, wo mehrere Werte geliefert werden, wir

jedoch einen oder einige davon nicht brauchen. Zum Schluss führen wir noch eine Testausgabe durch und sehen: Das Überschreiben hat auch mit der for range-Schleife funktioniert.

Du hast nun einen kleinen Einblick in die Verwendung von Slices bekommen. Die Verwendung bietet sich immer dann an, wenn man es mit einer Reihe von Werten gleichen Typs zu tun hat. Sie bieten sich auch dann besonders an, wenn man eine Reihenfolge benötigt. Es gibt noch weitere Möglichkeiten Daten zu organisieren. Einen Teil werden wir später noch behandeln. Außerdem möchte ich noch erwähnen, dass es weitere Operationen für Slices gibt. Zum Beispiel kann man einen Teil des Slices „heraus trennen". Eigenschaften wie diese heben Go, unter anderem, von vielen anderen Sprachen ab, wo diese Operationen selbst programmiert werden müssen. Da es hier jedoch nicht in erster Linie um Go, sondern um eine Einführung in die Programmierung geht, möchte ich darauf nicht zu detailliert eingehen. Ich möchte dich aber ermutigen, dich in Zukunft weiter mit Slices auseinanderzusetzen. Im Internet findest du zahlreiche hervorragende Beschreibungen und Beispiele dazu. Weiterhin sei erwähnt, dass Go, wie andere Sprachen auch, Arrays kennt. Arrays sind ebenfalls indizierte Strukturen. Sie sind jedoch gegenüber Slices (oder Listen) sehr eingeschränkt und weniger komfortabel. Das liegt vor allem daran, dass ihre Länge von Anfang an fest vorgeschrieben ist. Sie können nicht wachsen oder schrumpfen, was ihre Verwendung sehr unflexibel macht. Es gibt nur wenige Situationen, in denen man Arrays Slices (oder Listen) vorzieht und bewusst auf den Komfort verzichtet. In anderen Sprachen sind das oftmals Performance-Gründe. In Go sind Slices jedoch sehr effizient und „fest in der Sprache" implementiert. Du kannst sie also frei für deine Zwecke verwenden.

Listing 12.1: Slices

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // Oft möchte man eine ganze Reihe von Werten als Liste verwalten.
7    // Hierzu gibt es in Go Slices.
8    // Ein Slice definiert eine Liste von Speicherbereichen, die per Index
         angesprochen werden.
9    // Der Index beginnt bei 0 und endet bei der Länge - 1.
10   // Erstellung eines Slices mit Anfangslänge 3. Die Elemente sind leer (
         Zero-Values).
11   var namen []string = make([]string, 3)
12   fmt.Println(namen)
13
14   // Zuweisung von Werten an den einzelnen Positionen.
15   namen[0] = "Alice"
16   namen[1] = "Bob"
17   namen[2] = "Charlie"
18   fmt.Println(namen)
19
20   // Ist das Slice zu kurz für den gewählten Index, gibt es einen Fehler.
21   // namen[3] = "Kracht!"
22   // Abhilfe schafft das einfache Anhängen von Elementen.
23   // Das Slice wird entsprechend vergrößert und das neue Element angehängt
         .
24   namen = append(namen, "Karl")
25   fmt.Println(namen)
26
27   // Mit Hilfe der Funktion len kann man die Länge eines Slices bestimmen.
28   fmt.Println(len(namen))
29   namen = append(namen, "Toni")
30   fmt.Println(len(namen))
31
32   // Slices können oftmals sinnvoll in Kombination mit Schleifen
         eingesetzt werden.
33   // Beispiel: Überschreiben aller Namen.
34   for index := 0; index < len(namen); index++ {
35     namen[index] = "Überschrieben"
36   }
37   fmt.Println(namen)
38
39   // Einfacher:
40   for index, _ := range namen {
41     namen[index] = "Neu_Überschrieben"
42   }
43   fmt.Println(namen)
44
45   // Anmerkung: Es gibt noch weitere Slice-Operationen, z.B. zum
         Heraustrennen bestimmter Bereiche.
46   // Arrays sind ähnliche Strukturen, jedoch stärker eingeschränkt. Sie kö
         nnen nicht wachsen.
47 }
```

Listing 12.2: Laufzeitfehler “index out of range” mit Zeilennummer

```
1 [  ]
2 [Alice Bob Charlie]
3 panic: runtime error: index out of range
4
5 goroutine 1 [running]: main.main()
6   /tmp/sandbox977287488/main.go:21 +0xa20
```

13 Schlüssel-Wert-Paare (Maps)

Neben den Listen (Slices in Go) gibt es weitere Standardkonzepte zur strukturierten Speicherung von Daten. Dazu zählen Maps. Sie können beliebig viele Schlüssel-Wert-Paare aufnehmen. Listen bieten die Verwaltung von Elementen in geordneter Reihenfolge. Schlüssel-Wert-Speicher hingegen, sind „zum Nachschlagen" von Elementen gedacht. Eine sehr treffende Analogie sind Wörterbücher. Man hat einen Schlüssel, welchen man zur Suche verwendet. Ausgangspunkt des Nachschlagens ist also ein Wort in der Quellsprache (der Schlüssel). Zu jedem Schlüssel existiert ein Wert, der gesucht wird. Das ist bei einem Wörterbuch das Wort in der Zielsprache. Du kannst dir Maps also wie Wörterbücher vorstellen, nur dass man beliebige Typen als Schlüssel und Wert verwenden kann. Denkbar wäre z.B. unter einer Telefonnummer eine Adresse nachzuschlagen. Wichtig ist jedoch, dass ein Schlüssel eindeutig sein muss, so wie beispielsweise eine Telefonnummer. Ein Schlüssel muss eindeutig sein, damit die Zuordnung zum Wert eindeutig ist. Da Schlüssel-Wert-Paare neben Listen zu den elementarsten und wichtigsten Datenstrukturen zählen, bietet Go hierfür den fest eingebauten Typ „map". Maps sind daher sehr leicht zu verwenden. Wir sehen uns dazu das unten aufgeführte Code-Beispiel an.

Die Ausgabe sieht wie folgt aus:

Wir analysieren das Programm wieder Abschnitt für Abschnitt:

- Wir beginnen damit uns eine neue Map anzulegen. Dies geschieht analog zu Slices mit der eingebauten Funktion „make". Anders als bei Slices, geben wir dahinter jedoch als Parameter den Typ der Map an. Dieser setzt sich zusammen aus dem Typ der Schlüssel sowie dem Typen der Werte. Nach dem Schlüsselwort map folgt in eckigen Klammern stets der Schlüssel. Wir wollen im folgenden Hausnummern als Schlüssel verwenden und geben folglich int an. Hinter den eckigen Klammern folgt unmittelbar der Typ für die abzulegenden Werte. Da wir jeder Hausnummer einen Familiennamen zuordnen wollen, verwenden wir string. Damit wurde unsere Map erfolgreich erstellt und wir können sie im folgenden verwenden.

- Wenn wir einen Eintrag in der Map anlegen wollen (oder allgemein ansprechen wollen), müssen wir stets hinter der Map-Variablen in eckigen Klammern den Schlüssel angeben. Das Einfügen von Einträgen geschieht ganz einfach unter Angabe des (neuen) Schlüssels in den Klammern und wird mit einer Zuweisung vollzogen. Unter der Hausnummer 1 wollen wir den Familiennamen "Schulz" ablegen. Wir legen noch ein paar weitere Schlüssel-Wert Paare auf die gleiche Weise ab. Es ist wichtig zu wissen, dass Map Einträge keiner Reihenfolge unterliegen, im Gegensatz zu Slices. Die einzige Ordnung auf ihnen ist die Zuordnung eines

eindeutigen Schlüssels zu einem Wert. Demzufolge können wir auch einfach Einträge auslassen, was wir für die Hausnummern 5 und 6 auch tun. Eine Testausgabe mittels fmt.Println zeigt sehr schön die Map-Inhalte als Schlüssel-Wert Paare. Der erste Wert ist logischerweise immer ein Schlüssel, gefolgt von einem Doppelpunkt und dem Wert. Die Einträge sind jeweils durch ein Leerzeichen voneinander getrennt.

- Im nächsten Abschnitt sehen wir nun einen der großen Vorteile von Map-Strukturen. Wir können gezielt nach Einträgen fragen. In einem Slice müssten wir dazu erst aufwändig suchen über eine for-Schleife. Das wäre nicht nur ineffizient, sondern auch mühsam und würde unnötig viel Code erzeugen. Über den Schlüssel können wir jedoch ganz einfach den entsprechenden Wert auslesen. Wenn für diesen Schlüssel kein Wert existiert, wird übrigens der Zero-Value für den Wert-Typ zurückgegeben. Das wäre in unserem Fall der leere String (“”).

- Manchmal möchte man zunächst prüfen, ob ein Eintrag existiert, bevor man vielleicht einen Zero-Value in der Hand hält. Dazu liefert der Zugriff auf einen Eintrag bei Bedarf einen zweiten Rückgabewert. Wie wir in einem früheren Kapitel gesehen haben, müssen vorherige Werte abgefragt werden. Mit Hilfe der Wildcard _ können wir den ersten Wert jedoch ohne Variable einfach ignorieren. Hinter dem Komma folgt eine neue Variable für den Wert, der uns mehr interessiert: die Existenz des Eintrags. Als zweiter Wert wird also die Existenz als boolescher Wert zurückgegeben. Die Testausgabe zeigt uns, dass dieser Eintrag mit Schlüssel 6 nicht existiert (false).

- Nun wollen wir weitere Werte hinzufügen. Die Hausnummer 4 steht jetzt nicht mehr leer und wir legen einen Eintrag per Zuweisung an.

- Mit einer Zuweisung können wir außerdem existierende Einträge überschreiben. Dabei wird genau genommen nur der Wert eines existierenden Schlüssel-Wert-Paares überschrieben. Äußerlich unterscheidet sich dies nicht vom Hinzufügen und verwendet ebenfalls die Zuweisung. Wenn man zunächst wissen möchte, ob für den Schlüssel bereits ein Wert existiert, so kann man zuvor auf den bereits beschriebenen Existenztest zurückgreifen. Wir verzichten an dieser Stelle darauf, da wir wissen, dass die alten Bewohner aus- und Familie Holz stattdessen eingezogen ist.

- Weiterhin möchte man manchmal Einträge gänzlich aus der Map entfernen. Hierzu existiert die fest eingebaute Funktion delete. Ihr übergibt man zunächst die Map und anschließend den Schlüssel. Im Anschluss existiert der Eintrag nicht mehr, wie unsere Testausgabe beweist.

- Abschließend sehen wir uns die hervorragende Unterstützung für Schleifen an. Mit Hilfe des bereits kennengelernten Schlüsselwortes „range” können wir über alle Schlüssel-Wert-Paare laufen und diese beliebig handhaben. Wir nutzen hier sowohl den Schlüssel, als auch den Wert. Möchte man auf eines verzichten, kann man

wieder die Wildcard, repräsentiert durch den Unterstrich (_) einsetzen. Wie bereits erwähnt, beschwert sich der Compiler, wenn man eine unbenutzte Variable nicht verwendet. Dann sollte man von der Wildcard Gebrauch machen.

Du hast nun gelernt, wozu Maps geeignet sind und wie man sie in Go verwendet. Slices und Maps sind die einzigen (fest eingebauten) Datenstrukturen in Go. Trotz ihrer einfachen Handhabung lassen sich damit bereits sehr elegante Programme schreiben. Wie immer, empfehle ich dir das neu Gelernte anzuwenden. Gerne kannst du auch probieren maps in Kombination mit Slices (z.B. als Wert-Einträge) zu verwenden. Abschließend merke dir bitte, welche Datenstruktur wofür geeignet ist:

1. Für eine geordnete Reihe von Elementen eignen sich Slices (Listen). Die einzelnen Elemente sind durchnummeriert und werden immer per Index, beginnend bei 0 und endend bei Länge - 1, angesprochen.

2. Zum Nachschlagen von Elementen anhand eines eindeutigen Schlüsselwertes eignen sich Schlüssel-Wert-Paare. Die einzelnen Elemente sind ungeordnet und werden immer per Schlüssel angesprochen.

Listing 13.1: Maps

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // Wir wollen jeder Hausnummer in unserer Straße eine Reihe von
         Bewohnern zuordnen.
7    anwohner := make(map[int]string)
8
9    anwohner[1] = "Schulz"
10   anwohner[2] = "Bäcker"
11   // Nummer 3 und 4 sind unbewohnt, also lassen wir sie weg.
12   anwohner[5] = "Mustermann"
13   // Nummer 6 wurde noch nicht gebaut, also lassen wir sie weg.
14   anwohner[7] = "Müller"
15   fmt.Println(anwohner)
16
17   // Wir können nun fragen: Wer wohnt in Haus nummer 5?
18   fmt.Println(anwohner[5])
19
20   // Außerdem können wir prüfen ob ein Eintrag (Schlüssel) existiert.
21   // Dazu nutzen wir den zweiten Rückgabewert.
22   _, existiert := anwohner[6]
23   fmt.Println("Existiert Eintrag 6:", existiert)
24
25   // Wir können neue Werte hinzufügen.
26   // Nummer 4 steht nun nicht mehr leer.
27   anwohner[4] = "Baumann"
28   fmt.Println(anwohner)
29
30   // Wir können bestehende Einträge überschreiben indem wir sie einfach
         neu setzen.
31   anwohner[1] = "Holz"
32   fmt.Println(anwohner)
33
34   // Weiterhin können wir Einträge löschen mit der eingebauten Funktion
         delete.
35   delete(anwohner, 2)
36   fmt.Println(anwohner)
37
38   // Maps lassen sich ebenfalls sehr gut in Schleifen verwenden.
39   // Hier laufen wir über alle Einträge.
40   for schlüssel, wert := range anwohner {
41     fmt.Println(schlüssel, wert)
42   }
43 }
```

Listing 13.2: Ausgabe des Programms zu Schlüssel-Wert-Paaren

```
1  map[1:Schulz 2:Bäcker 5:Mustermann 7:Müller]
2  Mustermann
3  Existiert Eintrag 6: false
4  map[1:Schulz 2:Bäcker 5:Mustermann 7:Müller 4:Baumann]
5  map[4:Baumann 1:Holz 2:Bäcker 5:Mustermann 7:Müller]
6  map[1:Holz 5:Mustermann 7:Müller 4:Baumann]
7  1 Holz
8  5 Mustermann
9  7 Müller
10 4 Baumann
```

14 Strukturierte Datentypen (Structs)

Wir haben in diesem Buch bisher die elementaren Datentypen sowie Slices und Maps kennengelernt. Es folgt nun eine Erläuterung zur Definition eigener Datentypen. Dabei handelt es sich in der Regel um sogenannte „Structs". Structs sind strukturierte Datentypen, deren Bestandteile sich aus anderen Datentypen zusammensetzen. Beispielsweise wollen wir eine Reihe von Adressen verwalten. Das bisher Gelernte reicht dazu jedoch nicht aus. Wir können uns zwar Listen (Slices) anlegen. Doch was würden wir darin ablegen? Alle Informationen einer Adresse als Strings? Das wäre keine gute Lösung. Klüger ist es, man definiert eigene Typen, welche diese Datenstruktur abbilden. Es gehört zum Alltag eines jeden Entwicklers neue Typen zu definieren. Wir behandeln in diesem Kapitel lediglich deren Zusammensetzung und Erzeugung. Wir werden später noch darauf eingehen, wie man sie mit eigenem Verhalten ausstatten kann. Betrachten wir folgendes Code-Beispiel:

Wir gehen das Programm wieder Abschnitt für Abschnitt durch und analysieren es. Wir beginnen unter dem import-Statement, wo der Typ "Adresse" deklariert wird.

- Die Deklaration eines neuen Typs geschieht außerhalb von Funktionen oder anderen Blöcken und wird mit dem Schlüsselwort „type" eingeleitet. Darauf folgt der von uns gewählte Typname. Typen, die außerhalb des Packages sichtbar sein sollen, werden, ebenso wie Funktionsnamen, groß begonnen. Da wir uns hier nicht weiter um eine etwaige Einschränkung kümmern wollen, beginnt der Typ mit einem Großbuchstaben. Es folgt ein Typ, auf dessen Basis wir unseren eigenen Typ definieren. In unserem Falle ist das „struct". Wie bereits erwähnt, kapselt ein struct auf strukturierte Art und Weise einen Satz zusammengehörender Variablen, von dem jeder Wert vom Typ struct eine Ausprägung besitzt. Diese Variablen nennt man Felder („struct fields"). Zur Abbildung einer einfachen Adressinformation benötigen wir einen Namen, eine Postleitzahl sowie einen Ort. Alle drei Felder müssen mit einem Typ versehen werden und sind durch eine neue Zeile voneinander getrennt. Es bedarf keines Kommas oder ähnlicher Trenner, wir verwenden lediglich einen Zeilenumbruch. Um mit den Formatierungsregeln konform zu gehen und die Lesbarkeit hoch zu halten, rücken wir die Felder ein, denn sie sind Bestandteile des struct-Typen. Wie von anderen Konstrukten bekannt, werden die deklarierten Inhalte (die Felder) in geschweifte Klammern gehüllt. Fertig! Wir haben unseren ersten eigenen Typ deklariert.

- Beginnen wir nun in der main-Funktion. Wir können unseren Typen wie jeden anderen Typen verwenden. Dazu gehört selbstverständlich die Deklaration einer Variablen unseres Typs Adresse. Wenn wir keine Initialisierung vornehmen, sind

alle Felder mit ihren jeweiligen Zero-Values belegt. Eine Testausgabe beweist uns das. Wir sehen eine 0 in geschweiften Klammern. Dies ist der Zero-Value der Postleitzahl. Auch Name und Ort werden ausgegeben. Da der leere String jedoch keine Zeichen enthält, ist das nicht direkt zu erkennen.

- Wir wollen unserer Variablen nun einen Wert vom Typ struct mit echten Werten zuweisen. Wir erzeugen eines unter Angabe des Typnamen, gefolgt von geschweiften Klammern. Ähnlich, wie wir es von der Angabe von Argumenten für Parameter kennen, werden die Werte einfach hintereinander aufgelistet. Die Trennung erfolgt durch ein Komma. Wenn wir adresse nun ausgeben, sehen wir die Werte, wie zuvor gesetzt.

- Alternativ zur Angabe der Feldwerte anhand ihrer Position, kann man gezielt einzelne Felder per Name ansprechen. Somit müssen wir auch nicht die Reihenfolge in der Typdeklaration einhalten. Die Testausgabe zeigt, dass alle zugehörigen Werte, wie erwartet, gesetzt wurden.

- Die Angabe von Feldwerten per Name ist vor allem dann nützlich, wenn wir Felder leer (auf dem Zero-Value) belassen wollen. Das Beispiel zeigt, dass wir die Postleitzahl nicht angeben. Dennoch kann ich die anderen Werte benannt setzen. Die Testausgabe zeigt, dass die Postleitzahl den Wert 0 aufweist und die anderen Werte wie erwartet gefüllt sind.

- Wir können auf die Felder von struct-Werten, ähnlich wie auf Variablen, zugreifen. Dazu brauchen wir lediglich den Bezeichner des entsprechenden struct-Wertes vor dem Feldnamen aufzuführen und einen Punkt anzuschließen. Man greift so qualifiziert auf den Feldinhalt zu. Das Beispiel zeigt, wie wir einzelne Felder auslesen oder anpassen können.

Du hast nun eine kurze Anleitung zur Definition von eigenen Typen in Form von structs erhalten. Wir werden später noch auf weitere Eigenschaften von structs eingehen. Sie stellen ein sehr häufig benötigtes und nützliches Konstrukt der Sprache dar. Bevor du weiter liest, versuche bitte einige eigene struct-Typen zu definieren und experimentiere damit ein wenig. Hier sind einige Anregungen zum Ausprobieren:

- Probiere einen struct Typen in Kombination mit einem Slice aus und iteriere darüber mit einer for-Schleife (z.B. um Werte hineinzulegen, anzupassen oder auszugeben).

- Definiere zwei struct Typen und setze einen Typen als Feld-Typen des anderen Struct-Typen ein, sodass eine Schachtelung eintritt. Beispiel: Eine Person hat eine Adresse. Definiere einen Typ „Person” mit Name und Alter und setze zusätzlich eine Adresse (ohne Name) hinein. Lege dir ein paar Werte an und erfahre, wie du sie setzen und auslesen kannst.

- Verwende einen struct Typ als Parameter (Übergabe- oder Rückgabeparameter) für eine eigene Funktion. Auf diese Art und Weise können strukturierte

Informationen gekapselt und in einem Kontext übergeben werden. Anstelle von 3 oder mehr Einzel-Parametern kannst du einfach einen Wert vom Typ „Adresse" übergeben. Die Funktion nimmt sich dann die für sie relevanten Werte heraus.

Listing 14.1: Structs zur Definition eigener Datentypen

```
1  package main
2
3  import "fmt"
4
5  // Häufig möchte man mehr als nur einfache Datentypen speichern. Zum
       Beispiel Adressen.
6  // Dazu kann man neue Typen definieren, die sich aus den elementaren
       Datentypen zusammensetzen.
7  type Adresse struct {
8    Name string
9    Plz  int
10   Ort  string
11 }
12
13 func main() {
14   // Die Deklaration erfolgt wie bei anderen Typen auch.
15   // Wenn nicht initialisert, sind Zero-Values gesetzt.
16   var adresse Adresse
17   fmt.Println(adresse)
18
19   // Ein Struct wird durch angabe des Typ-Namens und geschweiften Klammern
20   // mit den entsprechenden Werten für die Felder erzeugt.
21   adresse = Adresse{"Klaus", 21149, "Hamburg"}
22   fmt.Println(adresse)
23
24   // Felder können auch per Name statt per Reihenfolge angegeben werden.
25   adresse = Adresse{Plz: 21149, Ort: "Hamburg", Name: "Klaus"}
26   fmt.Println(adresse)
27
28   // Werte können auch weggelassen werden, wenn alle angegebenen Felder
         benannt werden.
29   // Die nicht angegebenen Werte sind dann leere Werte (Zero-Values). Hier
          ist die Plz dann 0.
30   adresse = Adresse{Ort: "Hamburg", Name: "Klaus"}
31   fmt.Println(adresse)
32
33   // Feldinhalte können ähnlich wie Variablen behandelt werden.
34   // Man kann sie auslesen und ggf. verändern.
35   fmt.Println(adresse.Name + " wohnt in " + adresse.Ort)
36   adresse.Ort = "München"
37   fmt.Println("Jetzt wohnt " + adresse.Name + " in " + adresse.Ort)
38 }
```

15 Wert- vs Referenzsemantik und Zeiger (Pointer)

Kommen wir nun zu einem weiteren grundlegenden Thema der Programmierung. Wir sind bisher immer davon ausgegangen, dass Werte, die wir erhalten oder übergeben Kopien darstellen. Denken wir z.B. an die Übung mit der Zählschleife, als wir eine Variable als Kopie unseres aktuellen Zählschleifenwertes verwendet haben. Weiterhin haben wir bei den Themen Funktionen und Parameter gelernt, dass diese als Kopie übergeben wurden. Das war auch der Grund, warum wir Rückgabewerte von Funktionen für Ergebnisse verwendet haben. Die „Eingabeparameter" hingegen geben wir als Kopie in die Funktion. Dieses Verhalten einer Sprache, nämlich Kopien zu verwenden, nennt man „Wertsemantik". Wie wir gelernt haben, kann sie sehr nützlich sein. Beispielsweise möchte man bewusst auf einer Kopie arbeiten und nicht das Original verändern. Nun gibt es aber auch Situationen, in denen ein anderes Verhalten gewünscht ist. Man möchte keine Kopien anfertigen, sondern ein und dasselbe Element referenzieren. Dazu bieten Programmiersprachen zumeist auch Referenzsemantik an. In Go (und anderen Sprachen) wird Referenzsemantik auf beliebige Typen mit dem technischen Konzept „Zeiger" (pointer) realisiert. Einige Typen, darunter Slices haben automatisch Referenzsemantik. Zum Thema Zeiger sehen wir uns zunächst das untere Code-Beispiel an:

Bevor wir den praktischen Nutzen von Referenzsemantik (bzw. Zeigern) anhand des enthaltenen Beispiels erläutern, sehen wir uns kurz die grundlegenden Eigenschaften von Zeiger an:

1. Zeiger enthalten Speicheradressen. Sie referenzieren über diese Adresse einen konkreten Speicherbereich.

2. Zeiger haben Typen, die sich an dem Typ des Werts orientieren, den sie referenzieren (*Referenztyp). Ein Zeiger für ints kann ausschließlich eine Adresse enthalten, die auf einen Wert vom Typ int im Speicher zeigt.

3. Ein Zeigertyp ist ein eigener Datentyp und nicht mit dem referenzierten Typen gleichzusetzen. Ein Zeiger für ints ist ein anderer Datentyp als int. Wir erinnern uns: int Zeiger enthalten Speicheradressen auf einen int, keine konkreten int-Werte.

4. Mit Hilfe des Referenzierungsoperators & erhält man die Adresse und somit den Zeiger auf einen tatsächlichen Wert.

5. Mit Hilfe des Dereferenzierungsoperators * gelangt man vom Zeiger zum referenzierten, tatsächlichen, Wert. (Umkehrung von Punkt 4)

6. Alle Zeigertypen besitzen den Zero-Value nil, wenn sie nicht gesetzt. Der Zeiger zeigt dann nirgendwohin.

Diese Grundregeln sollen uns nun bei der Analyse des Quellcodes behilflich sein. Wir widmen wieder jedem Abschnitt in main sowie jeder Funktion unter main einen eigenen Stichpunkt:

- Wir beginnen mit der Deklaration und Initialisierung einer gewöhnlichen Variablen zahl. Wir weisen ihr den Wert 10 zu und wollen im folgenden die Speicheradresse ermitteln. Dies tun wir in der nächsten Zeile direkt vor der Übergabe an die println-Funktion als Argument. Wir benötigen dazu den Referenzierungsoperator &. Er liefert uns die Speicheradresse, wie wir z.B. ausgeben können. Wenn du das Programm ausführst, wirst du als ersten Wert eine Adresse ähnlich der folgenden vorfinden: 0x104382e0 Die genauen Zahlen (und Buchstaben) können abweichen, da unterschiedliche Speicheradressen verwendet werden. Das Format wird jedoch das gleiche sein, das heißt du wirst den Präfix 0x und darauf folgend Zahlen im Bereich 0 bis 9 und eventuell Buchstaben im Bereich a bis f vorfinden. Speicheradressen werden im Hexadezimalformat angegeben. Die Kenntnis des genauen Formats ist hier nicht entscheidend. Wichtig für dich ist zu wissen, dass es sich dabei um Speicheradressen handelt. Wenn man sich näher mit der Informationstechnologie beschäftigt, ist es auf jeden Fall sinnvoll die üblichen Zahlenformate, wie das Binärformat und das Hexadezimalformat zu kennen. Da die Grundlagen der Informatik jedoch nicht Schwerpunkt dieses Buches sind, möchte ich dich auf das Internet verweisen, wo du zahlreiche gute Erklärungen dazu finden kannst.

- Hier sehen wir die Deklaration eines Zeigers. Das Konzept ist dasselbe, wie bei den bisher gesehenen Deklarationen. Wir erinnern uns an die Regeln Nummer 2 und 3. Demnach sind Zeigertypen an den zu referenzierenden Wert-Typ gebunden, sind jedoch nicht mit diesem Typ gleichzusetzen. Ein Zeigertyp wird durch ein Sternchen (*) gefolgt vom referenzierten Typen deklariert. Wir legen uns hier einen Zeiger vom Typ „Zeiger auf int" an. Dieser Zeiger kann ausschließlich auf Werte vom Typ int Zeigen. Verstöße gegen diese Regel meldet der Compiler. Aus Neugier geben wir nun auch diese frisch deklarierte Variable aus. Da es sich bereits um einen Zeiger (eine Adresse) handelt, benötigen wir dafür auch keinen weiteren Operator. Als Inhalt wird <nil> ausgegeben. Die eckigen Klammern gehören zur Ausgabeformatierung, der eigentliche Wert lautet nil. Gemäß Regel 6 ist nil der Zero-Value („nicht gesetzter Wert") aller Zeigertypen. nil signalisiert, dass der Zeiger „ins Leere" zeigt. Das ist ein durchaus üblicher Zustand z.B. vor, nach oder zwischen der Verwendung von Zeigern. Man muss in diesen Fällen jedoch wissen, ob ein Zeiger tatsächlich einen Wert referenziert oder nicht und das gegebenenfalls mittels if meinZeiger == nil ... abtesten. Andernfalls kann ein berühmter Laufzeitfehler entstehen, die ungültige Referenz. In unserem Fall ist das kein Problem, da println auch mit der Referenz ins leere „umgehen" kann. Wenn eine Funktion jedoch damit rechnet, dass sich hier ein echter Wert befindet

(z.B. eine Zahl) und diese aber nicht vorfindet, kann das Programm lediglich einen Fehler werfen und sich beenden.

- Wir betrachten nun die Zuweisung einer Adresse an eine Zeigervariable. Wir verwenden erneut den Referenzierungsoperator zur Ermittlung der Adresse. Diese können wir dann unserer Zeigervariablen zuweisen, da der Typ auf der rechten Seite (Zeiger auf int), dem Typ auf der linken Seite gerecht wird. Man spricht auch von „Zuweisungskompatibilität". Andernfalls würde der Compiler einen Fehler melden. Du kannst es gerne ausprobieren. Die Ausgabe unserer Zeigervariablen zeigt die gleiche Adresse, wie die Ausgabe im ersten Abschnitt. Das ist auch logisch, denn wir referenzieren hier ein und dieselbe Variable! Was hier einfach klingt wird gerne vergessen, nicht nur von Anfängern. Es handelt sich um denselben Wert im Speicher. Nämlich den Wert hinter der Variablen zahl. Hätte ich dahingegen eine andere Variable, ebenfalls mit Wert 10 referenziert, wäre die Adresse eine andere. Warum? Nun, es handelt sich zwar um die gleiche Zahl aber unterschiedliche Variablen und unterschiedliche Stellen im Speicher. Merke dir: Gleichheit und Identität sind zwei verschiedene Dinge in der Programmierung. Du kannst es dir umgangssprachlich wie folgt merken: Wenn du „dasselbe" essen willst, wie jemand anders, teilst du dir das Essen mit ihm. Oftmals meinen wir jedoch „das Gleiche", also das gleiche Gericht aber ein eigenes Exemplar. Analoges gilt für Variablen und Werte. Gleiche Werte oder Variablen sind Kopien und benötigen Wertsemantik (keine Zeiger). Identische Werte oder Variablen referenzieren ein und denselben Speicherbereich und benötigen Referenzsemantik (Zeiger).

- Wir wollen das bisher Gelernte nun anhand eines Beispiels im Code verdeutlichen. Wir möchten dazu über eine Funktion (zwecks sauberer Code-Trennung) den Inhalt zweier Variablen austauschen. a wird mit 1 initialisiert und b mit 2. Nach dem Funktionsaufruf soll a 2 enthalten und b 1. Wie die Ausgabe des Programms zeigt, misslingt der erste Versuch über eine Funktion, der wir die Variablen als normale Werte übergeben. Der Aufruf der zweiten Funktion hingegen, erfüllt unsere Anforderung. Gehen wir im Folgenden auf die Funktionen ein.

- Die erste Funktion ist naiv geschrieben und geht davon aus, dass wir die übergebenen Werte der Parameter für a und b einfach tauschen können. Natürlich benötigen wir eine temporäre Variable zur Zwischenablage. Andernfalls haben wir nach der ersten Zuweisung in beiden Variablen die gleichen Werte. Also speichern wir einen Wert in tmp zwischen, machen die erste Zuweisung und schließen den Tausch mit dem (mittlerweile überschriebenen Wert aus a) ab. Hier haben wir uns über tmp übrigens einer Kopie für unsere Zwecke bedient! Leider funktioniert die Funktion nicht richtig. Warum ist das so? Nun, wie wir sehen, werden als Parameter normale Werte-Typen übergeben. Diese werden stets kopiert beim Aufruf. Wir tauschen zwar die Werte der lokalen Kopien in der Funktion aus, aber das bringt nichts, da sich das Ergebnis nicht nach außen (in main) bemerkbar macht. Das ist, als wenn jemand versucht seinen Kontostand zu erhöhen, indem er auf einem

ausgedruckten Auszug herumkritzelt. Auf der Kopie mag er wie ein Millionär aussehen, an dem Original (dem Konto) hat sich jedoch nichts geändert.

- Um unser Problem zu lösen, benötigen wir Zeiger als Parameter. Im Funktionskopf ändern wir daher die Parametertypen zu Zeigern auf ints sowie den Namen der Funktion (da Funktionsnamen eindeutig sein müssen). Die Umsetzung innerhalb der Funktion ähnelt der vorherigen Funktion. Das Konzept der Zwischenspeicherung über tmp bleibt bestehen. Der einzige Unterschied ist, dass wir den Dereferenzierungsoperator benötigen (*). Wenn wir den Dereferenzierungsoperator vor eine Adresse (einen Zeiger / Zeigervariable) schreiben, verfolgen wir die enthaltene Speicheradresse zum tatsächlichen Element. Es ist, als wenn du auf ein Schild siehst um eine Sehenswürdigkeit zu erreichen. Das Schild ist in diesem Fall der Zeiger, die Sehenswürdigkeit das tatsächlich referenzierte Element. Bitte verwechsle den Dereferenzierungsoperator nicht mit der Deklaration eines Zeigers. Im Fall der Deklaration als Variable oder Parameter folgt immer der referenzierte Typ. Hinter dem Dereferenzierungsoperator steht jedoch ein Wert oder analog eine Variable. Wir weisen tmp also den konkreten Wert zu, auf den a zeigt. Danach weisen wir den von b referenzierten Wert dem von a referenzierten Wert zu. Wir müssen auf beiden Seiten dereferenzieren, damit die Typen kompatibel sind. Wir wollen einen int einem anderen int zuweisen. Deshalb dereferenzieren wir auch links von der Zuweisung auf das konkrete Element, das dann ersetzt wird. Abschließend weisen wir den zwischengespeicherten Wert (ursprünglicher Wert von a) dem referenzierten Element von b zu. Da tmp nur innerhalb der Funktion benötigt wird, haben wir ihn nicht als Zeiger angelegt. Daher müssen wir tmp auch nicht dereferenzieren.

Du hast nun das Wesentliche zu Zeigern (Referenzen) in Go kennengelernt. Das Thema wird, unabhängig von der konkrete Sprache, oftmals als besonders schwierig von Anfängern empfunden. Das liegt meistens daran, dass sie das Gesamtkonzept hinter Zeigern (nämlich die Referenzierung, dasselbe VS das gleiche) nicht verstanden haben. Es ist also zunächst wichtig, das Konzept dahinter zu verstehen. Anschließend sollte man die Handwerksregeln, wie Referenzierung und Dereferenzierung, verinnerlichen. Dabei ist es stets von Bedeutung, das Gelernte anzuwenden. Beginne also am besten gleich damit dir eigenen Beispiele auszudenken und zu programmieren. Darüber hinaus schadet es nicht, dich weiter mit dem Thema auseinanderzusetzen. Jede Suchmaschine liefert dir eine Menge nützlicher Links zum Thema „Referenzsemantik", „Zeiger" oder „Pointer". Bei Bedarf kannst du dir natürlich auch dieses Kapitel nochmal in aller Ruhe durchlesen und nachprogrammieren. Abschließend möchte ich dir noch eine übersichtliche Entscheidungshilfe an die Hand geben. Für Anfänger ist es nämlich gar nicht leicht zu erkennen, ob Kopieren oder Referenzieren der passende Weg ist.

1. Normale Kopien (Wertsemantik) sind angebracht, wenn mindestens eine der folgenden Bedingungen erfüllt ist:
   a) Der Inhalt einer Variablen wird nur im lokalen Kontext benötigt. Das heißt,

sie ist nur innerhalb der aktuellen Funktion von Bedeutung. Es findet keine Übergabe als Parameter an andere Funktionen oder z.B. structs statt.

b) Der Inhalt einer Variablen wird in anderem Kontext (z.B. Funktion oder struct) ausschließlich lesend oder als Kopie verwendet. Es soll keine Veränderung des Inhalts erfolgen, die „nach außen hin" sichtbar wäre.

2. Zeiger (Referenzsemantik) sind angebracht, wenn mindestens eine der folgenden Bedingungen erfüllt sind:

   a) Der Inhalt einer Variablen wird auch außerhalb des lokalen Kontexts, z.B. in anderen Funktionen oder structs, verwendet und soll dort verändert werden. Die geänderten Werte sollen natürlich auch „außerhalb" sichtbar sein.

   b) Werte sollen aus Gründen der Einzigartigkeit oder Performance nicht kopiert werden.

3. Darüber hinaus gibt es einige Go-Typen, die automatisch (ohne die Verwendung von Zeigern) mit Referenzsemantik arbeiten.

   a) Dazu zählen Slices und Maps, die wir bereits kennengelernt haben.

   b) Weiterhin haben Channels automatisch Referenzsemantik. Sie zählen jedoch zu den fortgeschrittenen Konzepten und werden hier deshalb nicht behandelt.

Wenn man es also sehr kurz zusammenfassen soll: Zeiger sind immer dann angebracht, wenn Änderbarkeit in anderen Kontexten (Funktionen, structs etc.) am selben Element möglich sein soll. Andernfalls sind Kopien die sichere (und einfachere) Variante. Bei einigen Typen, insbesondere Slices und Maps, nutzt Go automatisch Referenzsemantik. Wenn ich also ein Slice an eine Funktion übergebe, können die Inhalte (Elemente) geändert werden.

Listing 15.1: Zeiger zum Referenzieren

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    // In einigen Sprachen, insbesondere den hardware-nahen, gibt es Zeiger.
7    // Zeiger sind Adressen, die Speicheradressen markieren.
8    // Mit Hilfe des Operators & bestimmt man die Speicheradresse einer
         Variablen. (also einen Zeiger darauf)
9    var zahl int = 10
10   fmt.Println(&zahl)
11
12   // Zeiger sind an einen Datentypen gebunden.
13   // Sie werden durch ein vorangestelltes * markiert.
14   // Ein Zeiger ohne zugewiesene Adresse enthält den leeren Wert nil. (
         zeigt also nirgendwohin)
15   var zahlZeiger *int
16   fmt.Println(zahlZeiger)
17
18   // Zuweisung der Adresse der Variablen Zahl an die Zeigervariable.
19   zahlZeiger = &zahl
20   fmt.Println(zahlZeiger)
21
22   // Zeiger werden in Go verwendet wenn Werte nicht kopiert, sondern
         referenziert werden sollen.
23   // Ein Beispiel:
24   a := 1
25   b := 2
26   tauschVersuch(a, b)
27   fmt.Println("Tauschversuch:", a, b)
28
29   tauscheMitZeigern(&a, &b)
30   fmt.Println("Tausche mit Zeigern:", a, b)
31 }
32
33 // Funktioniert nicht, da die Werte beim Aufruf hineinkopiert werden.
34 // Es werden innerhalb der Funktion die Kopien vertauscht, nicht jedoch
       die Originalwerte.
35 func tauschVersuch(a int, b int) {
36   tmp := a
37   a = b
38   b = tmp
39 }
40
41 // Hier werden nicht die int Werte hineinkopiert, sondern die Adressen (
       Zeiger).
42 // Mittels Dereferenzierung kann nun der Inhalt der Speicherbereiche
       vertauscht werden.
43 func tauscheMitZeigern(a *int, b *int) {
44   tmp := *a // Dereferenzierung: Von der Adresse zum Speicherinhalt.
45   *a = *b
46   *b = tmp
47 }
```

16 Methoden

Wir beschäftigen uns nun mit dem Konzept der Methoden. Methoden sind ein elementares Konzept der objektorientierten Programmierung. Kurz gesagt, betrachtet das Paradigma der objektorientierten Programmierung (OOP) Programme als ein Geflecht aus Objekten. Jedes Objekt hat dabei eine eigene Identität (sprich eine eigene Speicheradresse) und unterscheidet sich so von anderen Objekten, die eventuell gleiche Werte enthalten (Wir erinnern uns an Gleichheit vs Identität). Objekte interagieren miteinander, indem sie sich Nachrichten schicken. Diese Nachrichten sind Methodenaufrufe. Methoden sind eine Art Code zu kapseln, ähnlich den Funktionen. Was Methoden im Wesentlichen ausmacht werden wir nun anhand eines Beispiels in Go kennenlernen. Es sei noch kurz gesagt, dass Go keine (reine) objektorientierte Sprache ist. Vielmehr wurden bei der Sprachentwicklung die als am nützlichsten betrachteten Konzepte aus verschiedenen Konzepten aufgegriffen, so auch aus der OOP.

In Go gibt es zwei wesentliche Gründe, warum man das Schreiben einer Methode einer Funktion vorziehen kann:

1. Man möchte Typ-Spezifisches Verhalten möglichst nah an den Typ koppeln. Die Ausführung der Methode bezieht sich in erster Linie immer auf ein Exemplar des Typs selbst. Beispiel: Ein Auto hat eine ganz konkrete Art sich fortzubewegen. Es fährt, unter anderem, auf vier Rädern, hat einen Motor usw. Den Programmcode des Fahrens wollen wir daher direkt „am Typ" definieren, da er speziell für Werte vom Typ „Auto" gedacht ist.

2. Man kann bequem auf den Zustand des Exemplars des Typen zugreifen und diesen bei Bedarf sogar verändern. Anstelle der Übergabe als Parameter greift eine Methode sehr einfach auf die Werte im Objekt zu, auf dem die Methode aufgerufen werden soll. Beispiel: Das Auto soll fahren und muss dafür diverse elektronische Werte abfragen und ändern.

Platt ausgedrückt kann man den konzeptionellen Unterschied zwischen der Verwendung einer Funktion und einer Methode wie folgt formulieren:

- Eine Funktion tut etwas „unter Verwendung" möglicher Parameter. Die konkreten Argumente für die Parameter müssen bei jedem Aufruf mitgegeben werden.

- Eine Methode tut etwas „auf" einem Objekt (d.h. ein konkretes Exemplar eines Typs mit eigener Speicheradresse). Alle Informationen aus dem Objekt selbst werden nicht als Parameter übergeben, da eine Methode zum Objekt gehört und somit automatisch Zugriff auf den Objektzustand (z.B. Felder im struct) hat.

Konzeptionell kann man auch sagen, das Objekt wendet die Methode auf sich selbst an.

Wirf nun bitte einen Blick auf das Beispielprogramm. Unser Ziel ist es, ein Auto mit seinem Durchschnittsverbrauch und aktuellen Tankinhalt abzubilden. Wir lassen es dazu testweise fahren und tanken. Beim Fahren kann es passieren, dass uns der Kraftstoff ausgeht. In diesem Fall wollen wir eine entsprechende Meldung ausgeben.

Wir analysieren nun wieder das Programm Schritt schritt für Schritt. Wir beginnen dabei bei der Typdefinition für Auto. Im Anschluss widmet sich jeder Stichpunkt einer Methode bzw. Funktion (Fahren, Tanken und dann main).

- Für eine einfache Abbildung eines Autos definieren wir einen neuen Typ „Auto" auf Basis eines structs. Wir beschränken uns auf die benötigten Felder für den aktuellen Tankinhalt sowie Verbrauch. Als Datentypen verwenden wir float32 (32 Bit Genauigkeit im Gegensatz zu float64). Hiermit können wir auch Gleitkommazahlen speichern. Als Genauigkeit reicht uns jedoch der kleinere float-Typ aus. Alles zur Typdefinition mit structs sollte dir bereits aus dem früheren Kapitel bekannt sein.

- Wir sehen uns nun die erste Methode des Typs Auto an, Fahren. Methoden sehen Funktionen ziemlich ähnlich. Genau genommen sind sie auch „nur" spezielle Funktionen. Das Spezielle an ihnen ist, dass jedes Exemplar des zugehörigen Typen eine solche Methode „besitzt" und auf sich selbst anwenden kann. Wenn du dir den Kopf (Signatur) genau ansiehst, wirst du den Unterschied zu normalen Funktionen erkennen. Hinter dem func-Schlüsselwort steht eine Variable vom zugehörigen Typ oder dessen Zeiger-Typ in runden Klammern. Dahinter folgt, wie gewohnt, der Name sowie eine Parameterliste und optionale Rückgabewerte (hier nicht vorhanden). Den Teil zwischen func-Schlüsselwort und Namen nennt man Empfänger („receiver"). Der Empfänger einer Methode ist immer (!) das Objekt, auf dem ich die Methode aufrufen möchte und entweder vom Typ des Objekts oder vom entsprechenden Zeigertyp. Warum haben wir hier den Zeigertyp *Auto gewählt? Nun, vielleicht kannst du es dir anhand des Kapitels über Zeiger und Referenzsemantik denken: Wir wollen das Objekt innerhalb des Methodenrumpfs modifizieren können. Das heißt konkret, wenn wir die Feldinhalte unserer Auto-Exemplare anpassen können wollen, muss der Empfänger Typ ein Zeiger sein. Wenn wir wissen, dass wir nur lesenden Zugriff benötigen, kann man auch den „normalen" Typ, ohne Zeiger, verwenden. Wichtig ist, dass beim Aufruf, den wir uns später noch ansehen, der entsprechende Typ vorliegt. Man sollte daher nach Möglichkeit die Empfängertypen nicht mischen, damit die Aufrufe konsistent erfolgen können. Du kannst also getrost konsistent zum Zeigertypen greifen, wenn wenigstens eine Methode Schreibzugriff auf das Objekt braucht, was meistens der Fall ist. Wir sehen uns nun den Methodenrumpf an, der sich nicht weiter von Funktionen unterscheidet. Mit dem Empfänger a (unser Objekt selbst) können wir analog zu einem Parameter umgehen. Nun ein paar erklärende Worte zur Programmlogik. Zum Fahren geben wir ein paar Informationen aus. Das ist zuerst die übergebene Entfernung. Danach berechnen wir den Gesamtverbrauch auf dieser Strecke. Das

ist einfach der angegebene Verbrauch, multipliziert mit der Entfernung, geteilt durch 100 Kilometer (da der Verbrauch in Liter je 100 Kilometer angegeben wird). Wichtig ist, dass wir zuerst (!) die Entfernung durch 100 teilen, bevor wir sie mit dem Verbrauch multiplizieren. Go (und andere Programmiersprachen) kennen zwar die mathematische Regel „Punktrechnung vor Strichrechnung", doch da wir hier zwei Punktrechnungen haben, müssen wir die Reihenfolge mit Klammern klar machen. Wenn du schon mal einen Taschenrechner bedient hast, sollte dir das bekannt vorkommen. Hier gibt es noch eine kleine Besonderheit zum Thema Datentypen zu beachten: Da die Entfernung eine ganze Zahl vom Typ int ist, müssen wir sie zunächst in eine Gleitkommazahl umwandeln. Dazu bedienen wir uns einer Typkonvertierung („cast"). Eine Typumwandlung wird in Go wie folgt ausgedrückt: Man gibt den Zieltyp an gefolgt vom umzuwandelnden Wert in Runden Klammern. float32(entfernung) wandelt unseren int entfernung in eine Gleitkommazahl um, die wir dann problemlos mit den anderen Gleitkommazahlen verrechnen können. Go ist hier sogar sehr vorbildlich und liefert dir einen Compiler-Fehler, wenn du die Typumwandlung vergessen solltest. Er weist dich extra daraufhin, dass du nicht mit gemischten Typen rechnen kannst. Nachdem wir nun den Gesamtverbrauch für die Strecke errechnet haben, können wir diesen vom aktuellen Tankinhalt abziehen. Wie bereits erwähnt, können wir auf den Empfänger wie auf einen Parameter zugreifen. An dieser Stelle benötigen wir den Zeigertypen, damit die Änderung des Tankinhalts (schreibender Zugriff per Zuweisung) auch „nach außen" sichtbar wird und nicht nur auf einer Kopie arbeitet. Abschließend prüfen wir noch, ob der Tankinhalt über 0 liegt und geben eine passende Meldung für beide Fälle aus. Hier sei nochmal darauf hingewiesen, dass auch der Vergleich im if Statement auf kompatiblen Typen arbeiten muss. Daher vergleichen wir mit einem float (0.0). Außerdem enthalten Gleitkommazahlen in Computern immer eine gewisse Ungenauigkeit. Die Prüfung auf (exakte) Gleichheit sollte man daher stets vermeiden. Das ist hier jedoch ohnehin kein Problem, da wir auch mit negativen Werten rechnen müssen.

- Wenn du die erste Methode „Fahren" verstanden hast, sollte dir die Methode „Tanken" keine Probleme bereiten. Wir verwenden wieder den Empfängertyp *Auto, da wir eine Methode auf Objekten vom Typ Auto definieren wollen und einen Zeiger, da wir das Objekt selbst (seine Felder) ändern wollen (und um konsistent zwischen den Methoden zu bleiben). Der Tankinhalt wird entsprechend der als Parameter übergebenen Menge erhöht und der neue Inhalt als Information ausgegeben.

- Wenden wir uns nun der altbekannten main-Funktion zu, die nach wie vor den Einstiegspunkt in ein Programm definiert. Hier wollen wir einfach nur unseren neuen Typen Auto und die Methoden darauf ausprobieren. Daher legen wir uns ein neues Objekt ("Exemplar" oder "Instanz") vom Typ Auto an und geben Beispielwerte für Verbrauch und Tankinhalt an. Wir verwenden den Referenzierungsoperator & vor der Erstellung um uns die Adresse und somit

einen Zeiger auf das neue Auto liefern zu lassen. Der Typ der Variablen auto ist daher *Auto. Das tun wir, da die Empfängertypen der Methoden von Auto vom selben Typ sind und wir so die Methoden einfach mit dem passenden Empfänger aufrufen können. Wir könnten die Referenzierung bei der Erstellung auch einfach weglassen und zunächst auf den Zeiger verzichten. Dann müssten wir jedoch bei jedem Methodenaufruf die Adresse holen (ebenfalls mit &), was weniger elegant ist. Nach der Erstellung eines Objekts vom Typ Auto probieren wir die Methoden darauf aus. Hier zeigt sich auch warum ich immer davon gesprochen habe, dass eine Methode „auf" einem Objekt arbeitet. Wir rufen eine Methode immer für ein konkretes Objekt auf. Dies geschieht durch Angabe der Variablen, gefolgt von einem Punkt und der Auswahl der Methode. Der Unterschied zur Funktion ist also, dass ein konkretes Objekt vorangestellt wird, das eine Methode „auf sich selbst" ausführt. Damit wäre auch geklärt, wo der konkrete Empfänger aus der Methodendefinition herkommt und warum dieser nicht als Parameter übergeben wird. Wenn du beispielsweise eine Aufgabe erledigst, für die du nur dich selbst und deine Fähigkeiten benötigst, brauchst du keinen weiteren „Input" für dich selbst. Du kannst selbstständig auf dein Gehirn und deinen Körper zugreifen. Es ist wichtig zu wissen, dass Methoden immer (!) auf genau einem konkreten Objekt arbeiten. Das heißt unsere Methoden „Fahren" und „Tanken" arbeiten hier immer auf den Feldinhalten eines konkreten Auto-Objekts. Nur zu, probiere es aus. Wenn du weitere Auto-Objekte definierst und die Methoden auf ihnen aufrufst, so hat das keine Auswirkungen auf die anderen Auto-Objekte. Dies ist ein weitere Unterschied zu Parametern. Parameter kann man zwischen Funktionsaufrufen teilen (insbesondere mit Zeigern). Der Empfänger einer Methode ist jedoch immer nur auf sich selbst bezogen. Mit Blick auf unsere Analogie zum Menschen bedeutet das: Jeder Mensch kann zum Beispiel laufen, essen, schlafen und sprechen. Wir sind uns dabei sogar recht ähnlich. Doch jeder tut es ausschließlich unter Zuhilfenahme seines eigenen Körpers und Verstandes. Sieht man genau hin, entstehen dadurch auch die Unterschiede.

Du hast nun gelernt, wie man Methoden zur Definition von Objekt-spezifischem Verhalten auf Typen (hier structs) definieren und anwenden kann. Dabei handelt es sich um ein elementares objektorientiertes Konzept. In einigen Sprachen ist es sogar die einzige Möglichkeit so etwas wie Funktionen zu definieren. So weit geht Go jedoch nicht. Die Sprache überlässt es dem Programmierer, je nach Anwendungsfall zu entscheiden, was mehr Sinn macht: eine simple Funktion oder eine Methode auf einem Objekt. Eine kleine Entscheidungshilfe hast du ja bereits an die Hand bekommen. Auf jeden Fall erweitern Methoden das Konzept von eigenen Typen und structs. Ich halte sie daher für wichtig zu verstehen, selbst für Einsteiger. Zur Festigung empfiehlt es sich wie immer, wenn du selbst ein bisschen herumprogrammierst. Probiere Methoden aus und zieh gegebenenfalls weitere Quellen zu Rate.

Listing 16.1: Methoden zur Definition von Verhalten auf Typen (z.B. structs)

```
1  package main
2
3  import "fmt"
4
5  type Auto struct {
6    Verbrauch  float32 // in Liter je 100 Kilometer
7    Tankinhalt float32 // in Liter
8  }
9
10 func (a *Auto) Fahren(entfernung int) {
11   fmt.Println() // Leerzeile schreiben zur besseren Lesbarkeit.
12   fmt.Println("Starte Fahrt. Entfernung bis zum Ziel:", entfernung)
13
14   gesamtVerbrauch := (float32(entfernung) / 100.0) * a.Verbrauch
15   a.Tankinhalt = a.Tankinhalt - gesamtVerbrauch
16
17   if a.Tankinhalt <= 0.0 {
18     fmt.Println("Liegen geblieben! Fehlende Liter bis zum Ziel:", a.
         Tankinhalt*-1)
19   } else {
20     fmt.Println("Ziel erreicht. Restlicher Tankinhalt in Litern:", a.
         Tankinhalt)
21   }
22 }
23
24 func (a *Auto) Tanken(liter float32) {
25   a.Tankinhalt = a.Tankinhalt + liter
26   fmt.Println("Aufgetankt. Neuer Tankinhalt in Litern:", a.Tankinhalt)
27 }
28
29 func main() {
30   auto := &Auto{Verbrauch: 7.5, Tankinhalt: 62}
31
32   auto.Fahren(350)
33   auto.Fahren(16)
34   auto.Fahren(28)
35   auto.Fahren(413)
36   auto.Tanken(31.5)
37
38   auto.Fahren(163)
39   auto.Fahren(3)
40   auto.Tanken(18.6)
41
42   auto.Fahren(530)
43 }
```

17 Was du gelernt hast

Damit kommen wir langsam zum Ende dieses Buchs. Mein Ziel war es, dir einen praktischen Einstieg in die grundlegenden Konzepte der Programmierung zu ermöglichen. Dazu haben wir uns Schritt für Schritt mit den wesentlichen Konstrukten beschäftigt, deren Zweck beleuchtet und anhand von Programmierbeispielen ausprobiert. Ich möchte noch einmal kurz zusammenfassen, was du für dich herausgefunden haben solltest:

1. Was steckt hinter der Programmierung?
2. Wie fühlt sich Programmierung an?
3. Interessiert mich das, sodass ich mich weiter damit beschäftigen möchte?

Ich hoffe, dass du diese Fragen für dich persönlich nach dem Durcharbeiten dieses Buchs beantworten kannst. Allgemein lässt sich wohl sagen, dass Programmierung keine Magie ist. Sie erfordert aber ein paar grundlegende Eigenschaften, die man dafür mitbringen sollte:

1. Interesse am Thema. Wenn man nicht mag, womit man sich beschäftigt, wird man langfristig auch nicht zu guten Ergebnissen kommen. Daher halte ich Interesse für die wichtigste Bedingung.
2. Grundlegendes logisches und strukturiertes Denken. Diese Fähigkeit kann man trainieren. Jedoch sollte man ein bisschen davon bereits mitbringen.
3. Die Bereitschaft kontinuierlich dazuzulernen, auch langfristig. Kaum ein Fachgebiet entwickelt sich so schnell weiter, wie die IT und damit auch die Softwareentwicklung. Selbst wenn du eine Ausbildung durchgezogen hast und einer beruflichen Tätigkeit nachgehst, ist dieser Punkt immer noch entscheidend.
4. Zumindest grundlegende Englischkenntnisse. Da Anleitungen, Beispiele und Programme meistens in Englisch verfasst sind, sollte man grundlegende englische Sprachkenntnisse besitzen. Aber auch hier gilt: Man kann es lernen und ausbauen.

Das sind für mich die wesentlichen Eigenschaften, die man mitbringen sollte, um sich intensiver mit dem Thema Programmierung und Softwareentwicklung im Allgemeinen auseinanderzusetzen. Die Eintrittshürde ist also gar nicht so hoch. Alles andere kann man lernen, wenn man denn will! Das muss nicht immer über die klassischen Wege, wie Ausbildung oder Studium laufen. Gerade im IT-Bereich gibt es auch viele Quereinsteiger. Man kann aus vielen Quellen nützliches Wissen ziehen. Das können z.B. Bücher,

Seminare, Videos oder erfahrene Entwickler sein. Mindestens genauso wichtig wie die Theorie, ist jedoch die Anwendung. Egal wie viele Bücher du liest, wenn du die gelernte Theorie nicht anwendest, kannst du daraus keinen realen Nutzen ziehen. Wer nie viel programmiert hat, kann kein Softwareentwickler sein. Und wer nur vor langer Zeit mal programmiert hat, wird zumindest langfristig kein Softwareentwickler mehr sein, da er die Weiterentwicklung verpasst.

Obwohl das hart klingen mag, gibt es doch eine ganze Reihe von handfesten Vorteilen für Softwareentwickler. Für mich persönlich ist es ein sehr befriedigendes Erlebnis, wenn ich an einem Programm mitgewirkt habe, dass nach viel Arbeit tolle Leistungen erbringt. Auch ist es eine angenehme Herausforderung seine eigenen Fähigkeiten weiterzuentwickeln und Schritt für Schritt anspruchsvollere Probleme zu lösen. Nicht zuletzt kann man mit der Programmierung auch kreativ sein. Dies wird vielleicht im Berufsalltag keine große Rolle spielen. Dafür gibt es im privaten Bereich und in Open-Source Projekten längst auch die Möglichkeit sich kreativ und mit Spaß an der Tätigkeit einzubringen. Und überhaupt, mit welcher anderen Tätigkeit kann man Maschinen dazu bringen, den Menschen zu unterstützen? Okay, es gibt Maschinenbauer und Elektrotechniker, die auf jeden Fall vergleichbares leisten. Das Faszinierende an der Software bleibt für mich jedoch, dass ich dafür keine Geräte bauen oder anpassen muss. Stattdessen erstellt man eine Art „Gehirn" für vorhandene Maschinen. Ich hoffe, ich konnte ein bisschen Begeisterung bei dir wecken. Wenn nicht, ist das auch nicht schlimm, denn du weißt dann immerhin, dass du hier in Zukunft Zeit sparen kannst. Wenn doch, möchte ich dich ermutigen, noch das letzte Kapitel zu lesen. Hier gehe ich darauf ein, mit welchen Schritten du sehr gut weitermachen kannst.

18 Wie geht es weiter?

Auch wenn dieses Buch hier endet, hat das Lernen erst angefangen, wenn du dich weiter mit dem Thema Programmierung beschäftigen möchtest. Damit dir das Fortsetzen möglichst leicht fällt, will ich dir noch ein paar konkrete Tipps an die Hand geben:

1. Eine sehr gute Website, die mir ebenfalls dabei geholfen hat die Sprache Go zu lernen ist https://gobyexample.com/. Anhand vieler kleiner Programmbeispiele wird man mit allen Konstrukten der Sprache schrittweise vertraut gemacht. Die Seite konzentriert sich dabei auf das Erlernen von Go. Daher sollte man ein paar Grundkenntnisse zur Programmierung oder von anderen Sprachen mitbringen. Nach der erfolgreichen Bearbeitung dieses Buches solltest du diese jedoch besitzen. Die Beispiele stellen für dich eine sehr gute Möglichkeit dar, dein Wissen zu festigen und zu erweitern.

2. Wenn du mehr programmieren möchtest und größere Programme schreiben willst, benötigst du eine lokale Entwicklungsumgebung. Dazu ist es erforderlich, sich Go-Werkzeuge zunächst herunterzuladen und zu installieren. Zu den Werkzeugen gehören z.B. der Go-Compiler sowie die Standard-Bibliothek. Im Internet gibt es dazu diverse Anleitungen. Am besten ist allerdings, du hältst dich an die offizielle Website. Die Installation ist hier beschrieben: https://golang.org/doc/install. Die anschließende Konfiguration ist hier dokumentiert: https://golang.org/doc/code.html. Zum Erstellen und Anpassen von Quellcode eignet sich theoretisch jeder beliebige Editor. Jedoch ist das wenig komfortabel, da Features wie Syntax-Highlighting, Auto-Vervollständigung usw. fehlen. Anders verhält es sich bei einer IDE (Integrated Development Environment). Ich persönlich benutze die hervorragende LiteIDE. Sie bietet viele Komfort-Features und fühlt sich doch sehr schlank an. Die offizielle Installationsanleitung findest du hier: https://github.com/visualfc/liteide/blob/master/liteidex/deploy/welcome/en/install.mdWenn du die Installation von Go und einer Entwicklungsumgebung geschafft hast, solltest du dir genügend Zeit nehmen dich an sie zu gewöhnen. Langfristig kann man durch eine passende IDE sehr viel produktiver werden.

3. Bilde dich selbstständig weiter. Egal ob es um Go geht oder Programmierung im Allgemeinen bzw. mit anderen Sprachen. Eigenständige Weiterbildung ist immer eine gute Idee. Wahrscheinlich weißt du wie man eine Suchmaschine bedient. Im Internet findest du unzählige Möglichkeiten dich weiterzubilden, auch im Bezug auf Programmierung.

4. Programmiere! Man kann es nicht oft genug betonen. Um Wissen zu erhalten und auszubauen, ist es wichtig, dass du es anwendest. Du könntest dir z.B. eigene kleine Aufgaben stellen und diese Umsetzen. Alternativ kannst du nach Programmieraufgaben für Einsteiger suchen. Vielleicht kennst du auch jemanden, der mit dir zusammen programmieren möchte oder schon ein bisschen mehr Erfahrung hat. Der gegenseitige Austausch, direkt beim Programmieren, kann sehr sinnvoll sein. Ihr könntet sogar voneinander lernen. Wenn dich das Konzept interessiert, such doch mal nach dem Begriff „Pair Programming".

5. Sieh dir die Standardbibliothek von Go an. Die genaue Dokumentation findest du hier: `https://golang.org/pkg/`. Eine wesentliche Eigenschaft erfahrener Programmierer ist, dass sie Lösungen zu „Standardproblemen" nicht selbst programmieren, sondern vorhandenen Code nutzen. In der Go Standardbibliothek gibt es viele nützliche Funktionen, die häufig benötigt werden. Ob es das Untersuchen von String-Inhalten, der Umgang mit Dateien oder die formatierte Ausgabe ist, ein Blick in die Standardfunktionen lohnt sich fast immer! Auch hier gilt wieder: Übung macht den Meister. Ausprobieren hilft beim Erlernen.

Wenn dieses Buch nützlich für dich war, würde ich mich sehr freuen, wenn du es mich wissen lässt. Auch negative Kritik ist erwünscht, wenn sie konstruktiv ist. Denn nur aus sachlicher Kritik kann man lernen. Hinterlasse mir doch einfach einen Kommentar als Rezension, Blog-Kommentar oder E-Mail. Wenn dieses Buch auch für dich nützlich war, kann ich mir gut vorstellen, ein weiteres zu schreiben. Es würde dann auf dem bisher Gelernten aufbauen und fortgeschrittene Themen behandeln.

Bis dahin wünsche ich dir viel Erfolg bei weiteren Programmier-Sessions!

Listings

Abbildungsverzeichnis

Tabellenverzeichnis