2 Go

Go ist eine schlanke Programmiersprache und kommt mit 25 Keywords aus. Go ist weder ausdrücklich funktional noch ausdrücklich objektorientiert. Dennoch lassen sich beide Paradigmen in Go umsetzen. Dieses Kapitel führt in die Grundlagen der Programmiersprache Go ein.

Das Kapitel beginnt mit der Installation einer Go-Entwicklungsumgebung und der Beschreibung des Go-Workspace. Anschließend werden die Basiskonstrukte von Go beschrieben: Packages strukturieren Go-Dateien auf oberster Ebene und definieren die Sichtbarkeit enthaltener Elemente. Funktionen implementieren Programmlogik mithilfe von Schleifen, Bedingungen und Variablen. Typen stehen entweder in Form von Standard- oder benutzerdefinierten Typen zur Verfügung.

Nach Einführung dieser grundlegenden und aus anderen Programmiersprachen bekannten Sprachkonstrukte fährt das Kapitel mit den Besonderheiten von Go fort: Pointer, Interfaces und Methoden. Pointer ermöglichen die explizite Unterscheidung von Werte- und Referenztypen. Interfaces trennen Schnittstelle und Implementierung von Typen. Methoden implementieren Funktionen auf benutzerdefinierten Typen, die auf Instanzen des jeweiligen Typs gebunden werden.

Im letzten Abschnitt werden mit Arrays und Maps zwei der neben den Standardtypen wichtigsten Go-Typen beschrieben. Arrays implementieren typisierte Listen fester Länge. Maps dienen der Verwaltung typisierter Key-/Value-Paare.

2.1 Installation

Go-Distributionen stehen zum Binär-Download für MacOS, Windows und Linux unter *https://golang.org/dl/* bereit. Unter MacOS wird die Go-Distribution nach */usr/local/go* installiert. Damit die Go-Tools im Terminal funktionieren, muss die PATH-Variable um den Pfad */usr/local/go/bin* erweitert werden. Führen Sie das Go-Tool in einem neuen Terminalfenster mit dem Parameter *version* aus:

```
$ go version
go version go1.11 darwin/amd64
```

Entspricht die Ausgabe dem Beispiel, dann hat alles funktioniert, und die Go-Entwicklungsumgebung steht bereit. Weitere Installationshinweise sowie Anleitungen für die Go-Installation unter Windows und Linux finden Sie hier *https://golang.org/doc/install*.

2.2 Der Go-Workspace

Das grundlegende Konzept einer Go-Entwicklungsumgebung ist der *Go-Workspace*. Der Go-Workspace ist ein zentrales Verzeichnis, das über die Umgebungsvariable *GOPATH* referenziert wird. Unterhalb des GOPATH finden sich die drei Unterverzeichnisse *bin*, *pkg* und *src*:

```
bin/
    restvoice
pkg/
    linux_amd64/
        github.com/rwirdemann/restvoice/
            usecase.a
src/
    github.com/rwirdemann/restvoice/
        main.go
        usecase/
            create_invoice.go
```

Go-Programme werden compiliert und zu einem statischen Binary gelinkt. Die resultierende Binärdatei landet im Verzeichnis *bin* des Go-Workspace. Zusammengehörige Dateien, sogenannte *Packages*, werden zu einer *package.a*-Datei übersetzt und in eines der plattformspezifischen Unterverzeichnisse von *pkg* abgelegt. Das Unterverzeichnis *src* enthält die Dateien mit den Go-Sourcen, geordnet in Unterverzeichnisse, die die Package-Struktur widerspiegeln.

Die zugrunde liegende Idee des Go-Workspace ist, dass alle Projekte inklusive der benötigten Abhängigkeiten an zentraler Stelle abgelegt werden. Projekte werden schnell gefunden und können sich benötigte Bibliotheken teilen. Die Nutzung mehrerer Workspaces ist durch die Verwendung der Umgebungsvariable GOPATH einfach möglich.

Der Go-Workspace ist prinzipiell eine gute Idee, birgt aber auch eine Reihe praktischer Probleme. So ist es nicht ohne Weiteres möglich, unterschiedliche Versionen eines Packages im selben Workspace zu verwalten. Es ist auch nicht möglich, eine Bibliothek zu veröffentlichen, die eine bestimmte Version einer abhängigen Bibliothek benötigt.

Mit Version 1.5 wurde Go um eine Dependency-Management-Technik, das sogenannte *Vendoring* erweitert. Mit Vendoring kann man einem Projekt weitere Abhängigkeiten aus einem projektlokalen vendor-Verzeichnis zufügen. Seit Go-Version 1.11 steht mit *Go Modules* eine Ablösung des Vendoring ins Haus. Die Beispiele in diesem Buch funktionieren GOPATH-basiert. Die Beschäftigung mit Vendoring oder besser *Go Modules* ist an dieser Stelle nicht erforderlich. Spätestens wenn es in die Praxis geht, müssen Sie sich mit dem Thema Dependency Management beschäftigen. Das Go-Wiki enthält eine gute Einführung zur Funktionsweise und Benutzung von Go Modules [The18].

2.3 Test der Installation

Erstellen Sie den Go-Workspace mit den drei Unterverzeichnissen *bin*, *pkg* und *src* und exportieren Sie dessen Verzeichnis in der Umgebungsvariable GOPATH:

```
$ cd $HOME
$ mkdir -p go/bin go/pkg go/src
$ export GOPATH=$HOME/go
```

Anschließend wird im Verzeichnis *src* eine Datei *main.go* mit folgendem Inhalt erstellt:

```
1  package main
2
3  func main() {
4      println("Hello, Jo")
5  }
```

Die Funktion *main* im Package *main* ist der Einstiegspunkt in ein Go-Programm und wird beim Start des Programms ausgeführt. Das Kommando *go run* compiliert und führt das Programm aus:

```
$ cd $GOPATH/src
$ go run main
Hello, Jo
```

Das Kommandozeilentool *go* ist das zentrale Werkzeug einer Go-Distribution. Entsprechend parametrisiert lassen sich damit alle wichtigen Entwicklungsaufgaben ausführen. Einige Beispiele:

```
go build main.go   # Übersetzt und legt das Binary im aktuellen Verzeichnis ab
go install main.go # Übersetzt und legt das Binary im Verzeichnis "bin" ab
go test            # Führt die Tests im aktuellen Package aus
```

Ein vollständige Dokumentation der Go-Tools finden Sie unter *https://golang.org/cmd/go*.

2.4 Programmstruktur

Go-Quellcode ist auf Dateien mit dem Suffix *.go* verteilt. Die Struktur einer Go-Datei wird auf oberster Ebene durch sechs Keywords bestimmt:

```
package
import
var
const
type
func
```

Das erste Statement einer Go-Datei ist immer *package*, gefolgt von einem oder mehreren *import*-Statements zum Import referenzierter Packages. Anschließend folgen Variablen- und Konstantendeklarationen sowie Typ- und Funktionsdefinitionen in beliebiger Reihenfolge und Anzahl.

2.5 Packages

Go-Files werden in Packages organisiert. Alle Dateien eines Verzeichnisses gehören zum selben Package. Der letzte Teil des Pfadnamens bestimmt den Package-Namen. Gemäß dieser Konvention liegt die Datei *memory.go* im Verzeichnis *$GOPATH/go101/cache* und gehört zum Package *cache*:

```
1  package cache
2
3  func Write(key string, value string) {
4      ...
5  }
```

Packages definieren Namensräume. Die Sichtbarkeit von Typen, Funktionen, Variablen und Konstanten wird mithilfe von Groß- und Kleinschreibung definiert. Alles Kleingeschriebene ist ausschließlich innerhalb des Packages sichtbar. Alles Großgeschriebene ist auch außerhalb des Packages sichtbar. Packages werden über das *import*-Statement importiert. Öffentliche Typen, Variablen und Funktionen werden genutzt, indem der Package-Name dem importierten Element vorangestellt wird:

```
1  package main
2
3  import "cache"
4
5  func main() {
6      cache.Write("1", "Kalle")
7  }
```

Die Funktion *main* importiert das Package *cache* und ruft die exportierte Funktion *cache.Write* auf. Das main-Beispiel zeigt eine Ausnahme in Bezug auf die Namenskonvention für Packages: Die Go-Datei mit der Funktion *main* gehört zum Package main, unabhängig vom Verzeichnis, in dem sie liegt.

Initialisierung

Jede Datei eines Packages kann eine *init*-Funktion enthalten, in der paketweite Initialisierungsaufgaben ausgeführt werden. Init-Funktionen werden beim Laden eines Packages automatisch ausgeführt:

```
1  package auth
2
3  var kf []byte
4
5  func init() {
6      if kf, err := ioutil.ReadFile("public.key"); err != nil {
7          log.Fatalf("Could not open public key file: %s", "public.key")
8      }
9  }
```

Init-Funktionen sind Package-intern und können aus keinem anderen Package heraus aufgerufen werden. Ein anonymer Package-Import sorgt für die Ausführung von *init*, ohne dass eine öffentliche Funktion des importierten Packages aufgerufen werden muss. Dies ist zum Beispiel dann sinnvoll, wenn eine Bibliothek Initialisierungscode enthält, der nie explizit aufgerufen wird, für das Funktionieren des Programmes aber einmalig ausgeführt werden muss. Das Laden eines Datenbanktreibers ist ein Beispiel:

```
import _ "github.com/go-sql-driver/mysql"
```

Der Import lädt ausschließlich den MySQL-Datenbanktreiber. Anschließend werden die SQL-Funktionen des Go-Standard-Package *database/sql* genutzt.

2.6 Funktionen

Funktionen in Go werden mit dem Schlüsselwort *func*, gefolgt vom Namen, einer optionalen Parameterliste, keinem, einem oder mehreren Rückgabewerten deklariert:

```
1 func inc(i int) int {
2     return i + 1
3 }
```

Funktionen mit mehreren Rückgabewerten müssen deren Typen in Klammern angeben:

```
1 func swap(x int, y int) (int, int) {
2     return y, x
3 }
```

Return-Werte können Namen haben. Benannte Return-Werte können ohne explizite Deklaration im Funktions-Body verwendet werden. Ein parameterloses Return-Statement liefert den zuletzt an diese Variable zugewiesenen Wert zurück:

```
1 func inc(i int) (j int) {
2     j = i + 1
3     return      // Liefert "j"
4 }
```

2.7 Variablen

Variablen werden mit dem Schlüsselwort *var* oder per Direktzuweisung deklariert:

```
1 var firstname string
2
3 func name() string{
4     lastname := "Brunner"
5     return firstname + " " + lastname
6 }
```

Bei der Direktzuweisung erkennt der Compiler den Typ der Variablen, sodass die explizite Angabe des Typs *string* entfallen kann und die Variable direkt initialisiert wird. Eine Direktzuweisung funktioniert nur innerhalb von Funktionen. Außerhalb von Funktionen deklarierte Variablen müssen mit *var* deklariert werden. Wird eine per *var* deklarierte Variable direkt initialisiert, kann die Typangabe entfallen:

```
var firstname = "Jo"
```

Die Sichtbarkeit einer Variablen wird durch den sie umgebenden Block bestimmt. Außerhalb von Funktionen deklarierte Variablen sind in allen Funktionen des Packages sichtbar. Ist die Variable groß geschrieben, dann ist sie außerhalb ihres Packages sichtbar und wird durch den vorangestellten Package-Namen adressiert. Im folgenden Beispiel exportiert das Package *mysql* die Variable *Db*:

```
1  package mysql
2
3  import (
4      "database/sql"
5  )
6
7  var Db *sql.DB
8
9  func init() {
10     Db, _ = sql.Open("mysql", "user:password@/dbname")
11 }
```

Die exportierte Variable *Db* wird im Package *main* durch Voranstellen des Package-Namens *mysql* verwendet:

```
1  package main
2
3  import (
4      "go-basics/mysql"
5  )
6
7  func main() {
8      mysql.Db.Ping()
9  }
```

2.8 Typen

Go ist eine statische, getypte Programmiersprache. Jede Variable, jeder Funktionsparameter und jeder Rückgabewert hat einen Typ, dessen korrekte Verwendung vom Compiler sichergestellt wird. Das Typsystem von Go ist leichtgewichtig. Typinferenz sorgt dafür, dass ein Typ nur dann anzugeben ist, wenn der Compiler ihn nicht aus dem Kontext schließen kann.

Tabelle 2.1 Die Standardtypen von Go

Typ	Nullwert	Hinweis
bool	false	
string	␣	Leerzeichen
int int8 int16 int32 int64	0	
uint, uint8, uint16, uint32 uint64 uintptr	0	
byte	0	Alias für uint8
rune	0	Alias für int32, Unicode Code Point
float32	0.0	
float64	0.0	
complex64	0+0i	
complex128	0+0i	

Go unterscheidet zwischen Standard- und benutzerdefinierten Typen. Tabelle 2.1 gibt einen Überblick über die Go-Standardtypen mit ihren jeweiligen Nullwerten. Der Nullwert eines Typs ist der Wert, mit dem nicht explizit initialisierte Variablen belegt werden.

Aufbauend auf die Go-Standardtypen können benutzerdefinierte Typen definiert werden. Das folgende Beispiel definiert den benutzerdefinierten Typ *Location* basierend auf *string*:

```
1  type Location string
2
3  func foo()  {
4      var city Location
5      city = "Hamburg"
6      fmt.Printf("City: %s\n", city)
7  }
```

Zusammengesetzte Typen

Mithilfe des Keyword *struct* werden mehrere Attribute zu einem zusammengesetzten Typen kombiniert:

```
1  type Address struct {
2      Street string
3      City   Location
4  }
```

Zusammengesetzte Typen werden per Literal instanziiert, indem dem Typnamen eine in geschweiften Klammern angegebene Parameterliste übergeben wird:

```
address := Address{Street: "Oxford Street", City: "London"}
```

Die Angabe der Parameternamen ist nur bei partieller Initialisierung erforderlich. Nicht explizit initialisierte Attribute werden mit ihrem Nullwert initialisiert. Bei vollständiger

Initialisierung reicht die Kurzschreibweise:

```
address := Address{"Oxford Street", "London"}
```

Zusammengesetzte Typen können beliebig geschachtelt werden:

```
type Contact struct {
    Name   string
    Mobile string
    Home   Address
}

address := Address{"Oxford Street", "London"}
contact := Contact{Name: "Jo", Home: address}
```

Der Zugriff auf die Attribute eines zusammengesetzten Typs erfolgt über die Punktnotation:

```
fmt.Println(contact.Name)
fmt.Println(contact.Home.Street)
```

Alternativ zur Angabe eines expliziten Attributnamens lassen sich zusammengesetzte Typen auch anonym einbetten. Ein anonym eingebettetes Struct besteht nur aus seinem Typ:

```
type Contact struct {
    Name   string
    Mobile string
    Address
}
```

Die Felder eingebetteter Structs werden direkt referenziert:

```
fmt.Println(contact.Street)
```

Eingebettete Structs werden per Literal oder Zuweisung initialisiert:

```
contact := Contact{Address: Address{"Oxford Street", "London"}}
contact.Street = "Oxford Street"
```

2.9 Schleifen

Go kennt nur eine Schleife, die for-Schleife. Je nach Parameter erfüllt die Schleife unterschiedliche Zwecke:

```
for i := 0; i < 10; i++  // Index-basierte Iteration
for i < 10               // while
for                      // for ever
for i, v := range foo    // Index- und Werte-basierte Iteration
```

Die for-Bedingung ist immer ohne Klammern, der for-Body immer in geschweiften Klammern anzugeben:

```
1 i := 0
2 for i < 10 {
3     fmt.Println(i)
4     i++
5 }
```

In Kombination mit *range* iteriert *for* über eine Liste von Werten und liefert für jeden Schleifendurchlauf den jeweiligen Listenwert sowie dessen Listenindex:

```
1 for i, v := range []string{"a", "b", "c"} {
2     fmt.Println("Index:", i, "Wert:", v)
3 }
```

Wird die Angabe der Werte-Variablen weggelassen, liefert *range* nur den jeweiligen Schleifenindex:

```
for i := range []string{"a", "b", "c"}
```

Wird statt des Index nur der Wert benötigt, muss die Index-Variable als zu ignorierende Variable mit einem Unterstrich deklariert werden:

```
for _, v := range []string{"a", "b", "c"} {
```

2.10 Verzweigungen

Für Verzweigungen kennt Go die beiden Schlüsselworte *if* und *switch*. If-Statements bestehen aus einer Bedingung, einem Codeblock sowie einem optionalen else-Zweig:

```
1 i := math.Pow(10, 2)
2 if i < j {
3     fmt.Println("i < j")
4 } else {
5     fmt.Println("i >= j")
6 }
```

Die Bedingung wird immer ohne Klammern, der if- und else-Block immer in geschweiften Klammern angegeben. Der if-Bedingung kann ein Statement vorangestellt werden, dessen Ergebnis in der Bedingung verwendet wird:

```
1 if i := math.Pow(10, 2); i < j {
2     fmt.Println("i < j")
3 } else {
4     fmt.Println("i >= j")
5 }
```

Das vorangestellte Statement führt zu einer kompakteren Schreibweise und reduziertem Variablen-Scope. Die Schreibweise wird häufig zum Aufruf von Funktionen mit potenziell fehlerhaftem Ausgang verwendet:

```
1 if b, err := json.Marshal(contact); err != nil {
2     log.Error(err)
3 } else {
4     send(b)
5 }
```

Der Kontakt wird nur gesendet, wenn er fehlerfrei serialisiert werden konnte. Die Variablen *b* und *err* sind nur im if- und else-Branch sichtbar.

Switch-Statements bestehen aus einer Bedingung, gefolgt von einer Liste Case-Blöcken. Ein Case-Block besteht aus einer oder mehren Anweisungen und endet mit dem Beginn des folgenden Case-Blocks:

```
1 switch r.Method {
2 case "GET":
3     log.Info("GET")
4     handleGetRequest(r)
5 case "POST":
6     ...
7 default:
8     ...
9 }
```

Das Schlüsselwort *break* ist optional und kann verwendet werden, um einen Case-Block vor dessen kompletter Abarbeitung abzuschließen. Das Schlüsselwort *fallthrough* kann am Ende eines Case-Blocks eingesetzt werden, um den nachfolgenden Block in die weitere Auswertung mit einzubeziehen, auch wenn dessen Bedingung nicht erfüllt ist.

Eine weitere Verwendungsart von *switch* ist die Ersetzung der Variablen im Switch-Teil durch einzelne Bedingungen in den Case-Zweigen:

```
1 switch {
2 case height <= 4:
3     fmt.Println("Short")
4 case height <= 5:
5     fmt.Println("Normal")
6 case height > 5:
7     fmt.Println("Tall")
8 }
```

Diese Variante des Switch-Statements wird gerne als besser lesbare Version hintereinandergereihter If-Statements verwendet.

2.11 Methoden

Eine Methode ist eine auf ein Objekt[1] gebundene Funktion. Der Begriff *binden* stammt aus der objektorientierten Programmierung und bedeutet soviel wie dem Objekt eine Nachricht oder einen Auftrag zu senden. Das folgende Beispiel erzeugt einen Kreis und ruft darauf die Methode *draw* auf:

```
1 c := Circle{x:50, y: 50, radius:100}
2 c.draw()
```

Eine Funktion wird zu einer Methode, indem dem Funktionsnamen der Typ vorangestellt wird, auf dem die Methode definiert wird:

```
1 func (c Circle) draw() {
2     graphics.Circle(c.x, c.y, c.radius)
3 }
```

Innerhalb des Methodenrumpfs wird das Objekt, auf dem die Methode arbeitet, durch die dem Typ vorangestellte Variable (hier *c*) repräsentiert.

Die Verwendung von Methoden statt Funktionen ist ein wesentliches Merkmal objektorientierter Systeme. Dabei ist es dort eher die Regel denn die Ausnahme, dass eine Methode einen Seiteneffekt auf dem gebundenen Objekt erzeugt. Dieser Seiteneffekt ist gewünscht, da viele Methoden den Zustand des Objektes ändern sollen. Ein Beispiel für Methoden mit Seiteneffekt ist ein Setter, der einem privaten Attribut einen Wert zuweist:

```
1 func (c Circle ) setX(x int)  {
2     c.x = x
3 }
4
5 func main() {
6     c := Circle{}
7     c.setX(50)
8     fmt.Println(c)
9 }


$ go run main.go
x: 0 y: 0, r: 0
```

Der Setter *setX* weist dem Attribut x der Circle-Instanz *c* den Wert 50 zu. Der anschließende Println-Aufruf soll die Zuweisung verifizieren, gibt aber für *x* statt der erwarteten 50 den Initialwert 0 aus. Der Grund für die nicht erfolgte Zuweisung ist die Call-By-Value-Semantik von Go. Entsprechend arbeiten Funktionen und Methoden auf Kopien ihrer Parameter bzw. Objekte. Gemäß dieser Semantik ist die Methode *setX* seiteneffektfrei, da sie nicht auf dem in *main* instanziierten *c*, sondern auf einer Kopie davon arbeitet. Seiteneffekte werden in Go durch den Einsatz von Pointern erzwungen.

[1] In Go gibt es das Konzept *Objekt* nicht. Die Instanzen eines Typs sind eher Werte. Der Begriff *Objekt* wird synonym für einen Go-Wert benutzt, weil sich *Objekt* besser lesen und sprechen lässt.

2.12 Pointer

Ein Pointer ist eine Variable, die statt eines konkreten Wertes eine Speicheradresse enthält. Diese Adresse zeigt auf die Stelle im Hauptspeicher, an der der eigentliche Wert gespeichert ist. Im folgenden Beispiel enthält *p* die Adresse, an der *x* gespeichert ist:

```
1 func main() {
2     x := 1
3     p := &x
4     fmt.Println(p) // => 0xc420014080
5 }
```

Ein Pointer wird entweder über die Funktion *new* oder wie im Beispiel über den Operator & erzeugt. Der Zugriff auf einen von einem Pointer referenzierten Wert erfolgt durch einen vorangestellten Stern, der die Variable dereferenziert:

```
fmt.Println(*p) // => 1
```

Seiteneffekte werden erzwungen, indem Funktionsparameter als Pointer-Variablen, d. h. mit einem dem Typ vorangestellten Stern, deklariert werden:

```
1 func inc(p *int) {
2     *p = *p +1
3 }
```

Die Funktion *inc* arbeitet jetzt nicht mehr auf einer Kopie der an sie übergebenen Variablen. Stattdessen wird *inc* über den Pointer-Parameter *p* mitgeteilt, an welcher Stelle der zu inkrementierende Wert im Speicher steht. Dieser Wert wird dereferenziert, inkrementiert und an die Speicherstelle zurückgeschrieben. *inc* erzeugt den erzwungenen Seiteneffekt:

```
1 func main() {
2     x := 1
3     p := &x
4     inc(p)
5     fmt.Println(x) // => 2
6 }
```

Methoden werden auf eine Call-By-Reference-Semantik umgestellt, indem dem Typ des gebundenen Objekts ein Stern vorangestellt wird. Auf Pointern deklarierte Methoden werden *Pointer Receiver* genannt:

```
1 func (c *Circle) setX(x int) {
2     c.x = x
3 }
4
5 func main() {
6     c := Circle{}
7     c.setX(50)
8     fmt.Println(c.x) // -> 50
9 }
```

Die Methode *setX* arbeitet nicht mehr auf einer Kopie von Circle, sondern auf der in *main* instanziierten Version. Die Variable *c* muss dafür nicht explizit als Pointer deklariert werden. Allein die Deklaration von *setX* als Pointer-Receiver sorgt dafür, dass der Methode ein Pointer auf *c* übergeben wird. Entsprechend können Methoden auf Pointer-Typen die an sie gebundenen Objekte verändern.

Ein Vorteil von der Call-By-Value-Semantik von Go ist, dass Funktions- und Methodenaufrufe immer seiteneffektfrei sind, sofern sie keinen globalen Zustand verändern. Die Seiteneffektfreiheit führt zu robusteren Programmen, da insbesondere unerwünschte Seiteneffekte vermieden werden. Die Verwendung von Pointern macht Seiteneffekte explizit. Der Code macht deutlich, dass ein Seiteneffekt gewünscht und bewusst programmiert wurde.

2.13 Interfaces

Ein Interface beschreibt die Schnittstelle eines Objekts. Ein Interface sagt, was ein Objekt kann, ohne festzulegen, wie es das tut. Technisch ist ein Interface eine Sammlung von Methoden im Sinne einer Verhaltensbeschreibung:

```
1 type Figure interface {
2     Draw()
3     Erase()
4 }
```

Das Interface *Figure* beschreibt Objekte, die sich selber zeichnen und löschen können. Interfaces können in Variablen-Deklarationen, als Rückgabewerte oder als Funktionsparameter verwendet werden:

```
1 func DrawOnCanvas(f Figure) {
2     f.Draw()
3 }
```

Zur Laufzeit muss sich hinter *f* ein konkretes Objekt verbergen, das sich tatsächlich zeichnen und löschen kann. In Go spricht man davon, dass das Objekt bzw. dessen Typ das Interface *Figure* erfüllen muss. Ein Typ erfüllt ein Interface, wenn er alle Methoden des Interfaces implementiert. Entsprechend erfüllt der Typ *Circle* das Interface *Figure*, wenn er die Methoden *Draw* und *Erase* implementiert:

```
1 type Circle struct {
2 }
3
4 func (c Circle) Draw() { ... }
5
6 func (c Circle) Erase() { ... }
```

Circle erfüllt *Figure*, ohne dass die Implementierung von *Circle* einen Hinweis auf diese Beziehung enthält. *Circle* kann überall dort verwendet werden, wo *Figure*-Instanzen akzeptiert werden:

```
1 func main() {
2     c := Circle{}
3     DrawOnCanvas(c)
4     ...
5 }
```

Die Verwendung von *Figure* entkoppelt *DrawOnCanvas* von *Circle*. Die Funktion *DrawOnCanvas* funktioniert mit allen Typen, die *Draw* und *Erase* implementieren, zum Beispiel *Rectangle* oder *Square*.

Interface-Deklarationen akzeptieren Pointer und Werte, ohne dass ein Interface als Pointer-Variable deklariert werden muss. Entsprechend kann der Funktion

```
DrawOnCanvas(f Figure)
```

sowohl ein Wert als auch ein Pointer übergeben werden. Die Implementierung eines Interface unterscheidet hingegen zwischen Werte- und Pointer-Implementierung. Werden beispielsweise *Draw* und *Erase* als Pointer Receiver implementiert, liefert das folgende Beispiel einen Übersetzungsfehler:

```
1 func (c *Circle) Draw() { ... }
2
3 func (c *Circle) Erase() { ... }
4
5 func main() {
6     c := Circle{}
7     DrawOnCanvas(c)
8     ...
9 }
```

```
$ go build main.go

./main.go:22:14: cannot use c (type Circle) as type Figure in argument
to DrawOnCanvas: Circle does not implement Figure (Draw method has pointer receiver)
```

Die Fehlermeldung liefert den Grund für das Problem: Der Wertetyp *Circle* implementiert *Figure* nicht mehr. Wird der Funktion *DrawOnCanvas* statt eines Werts ein Pointer auf Circle übergeben, lässt sich das Programm wieder übersetzen, da **Circle* das erwartete Interface implementiert:

```
1 func main() {
2     c := &Circle{}
3     DrawOnCanvas(c)
4     ...
5 }
```

2.14 Arrays und Slices

Neben den bekannten Basistypen enthält Go mit Arrays und Slices zwei weitere wichtige Typen.

2.14.1 Arrays

Arrays sind Listen gleichen Typs mit fester Länge.

```
var a [3]string
```

Das Array *a* hat eine Länge von 3 und enthält Elemente vom Typ *string*. Der Zugriff auf Elemente erfolgt indexbasiert:

```
fmt.Println(a[0])
```

Arrays können per Literal erzeugt und initialisiert werden. Das folgende Array-Literal erzeugt ein 4-elementiges String-Array und initialisiert die ersten drei Elemente. Das vierte, nicht initialisierte Element wird mit dem Defaultwert ␣ für Strings initialisiert:

```
var a = [4]string{"Kalle", "Ruscha", "Penny"}
```

Die Längenangabe ist optional und kann durch ... ersetzt werden. Die Länge des Arrays ergibt sich dann aus der Anzahl der initialisierten Elemente:

```
1 var a = [...]string{"Kalle", "Ruscha", "Penny"}
2 fmt.Println(len(a)) // => 3
```

Die Build-In-Funktion *len* liefert die Länge eines Arrays, die zum Beispiel zum Iterieren über die enthaltenen Elemente genutzt werden kann:

```
1 for i := 0; i < len(a); i++  {
2     fmt.Println(a[i])
3 }
```

Eleganter wird die Iteration mit der Build-In-Funktion *range*, die neben dem Wert auch die Index-Position des Elements liefert:

```
1 for i, v := range a {
2     fmt.Println(i, ":",  v)
3 }
```

2.14.2 Slices

Die eigentliche Arbeit auf Arrays erfolgt mithilfe von Slices. Eine Slice ist eine dynamische Sicht auf ein Array. Das Array einer Slice wird als das *unterliegende Array* der Slice bezeichnet. Eine Slice wird erzeugt, indem dem unterliegenden Array zwei Indizes *i, j* in eckigen

Klammern übergeben werden:

```
var a = [...]string{"Kalle", "Ruscha", "Penny", "Lotta"}
var s = a[1:3]
fmt.Println(s[0]) // => Ruscha
fmt.Println(s[1]) // => Penny
```

Die Slice *s* enthält die Elemente Ruscha und Penny des Arrays *a*. Slices haben eine Länge (len) und eine Kapazität (cap). Die Länge ist die Anzahl der enthaltenen Elemente, die Kapazität die mögliche Obergrenze, bis zu der die Slice erweitert werden kann.

```
var s = a[1:3]
fmt.Println(len(s)) // => 2
fmt.Println(cap(s)) // => 3
```

Die Slice *s* enthält zwei Elemente, hat aber noch Luft für ein weiteres. Übersteigt die Länge einer Slice deren Kapazität, bricht das Programm mit einem *Out of bounds*-Fehler ab.

```
s := a[1:5] // => invalid slice index 5 (out of bounds for 4-element array)
```

Das Slice-Literal *[]type* erzeugt eine Slice sowie ein zugehöriges unterliegendes Array, im Beispiel vom Typ *string* mit einer Länge von 4:

```
var a = []string{"Kalle", "Ruscha", "Penny", "Lotta"}
```

Der einzige Unterschied zwischen Array- und Slice-Erzeugung ist die fehlende Größenangabe in den eckigen Klammern. Alternativ zur Literalschreibweise kann eine Slice mit der Build-In-Funktion *make* erzeugt werden, die eine Parametrisierung von Länge und Kapazität der Slice erlaubt:

```
var s = make([]int, 0, 100)
fmt.Println(len(s)) // => 0
fmt.Println(cap(s)) // => 100
```

Die Erzeugung mit *make* ist dann sinnvoll, wenn von vornherein klar ist, dass die Anzahl der Elemente im unterliegenden Array ansteigend ist:

```
var s = make([]int, 0, 100)
for i := 0; i < 100; i++ {
    s = append(s, rand.Int())
}
```

Die Build-In-Funktion *append* hängt ein neues Element an die im ersten Parameter übergebene Slice an. Das neue Element wird im unterliegenden Array gespeichert. Dessen Länge entspricht der Slice-Kapazität, im Beispiel also 100. Arrays sind immutable[2] und können nur verändert werden, indem sie kopiert werden. Sobald *append* die Länge des unterliegenden Arrays überschreitet, erzeugt die Funktion ein neues, größeres Array und kopiert die alten sowie das neue Element in das neu erzeugte Array. Dieser Schritt wird vermieden, wenn von vornherein ein ausreichend großes Array erzeugt wird.

[2] Ein Objekt ist immutable, wenn es nach seiner Erzeugung nicht mehr verändert werden kann.

2.14.3 Polymorphe Arrays

Der Typ eines Arrays ist bestimmt durch Länge und Elementtyp. Die Verwendung eines Interface als Array-Typ ermöglicht die polymorphe[3] Verwendung des Arrays:

```
1  type Figure interface {
2      Draw()
3  }
4
5  type Circle struct {...}
6
7  func (c Circle) Draw() {...}
8
9  type Rectangle struct {...}
10
11 func (r Rectangle) Draw() {...}
12
13 var figures = [...]Figure{Circle{}, Circle{}, Rectangle{}}
14 for _, f := range figures {
15     f.Draw()
16 }
```

Das Array *figures* weiß zur Laufzeit nicht, ob das jeweilige Element ein Kreis oder Rechteck ist. Das einzig Wichtige ist das Vorhandensein einer *Draw*-Methode.

2.15 Maps

Neben Arrays sind Maps der zweite wichtige Container-Datentyp in Go. Maps enthalten Key-Value-Paare und werden mithilfe des Map-Literals erzeugt:

```
1  var m = map[int]string {
2      1: "Kalle",
3      2: "Penny",
4  }
```

Der in eckigen Klammern angegebene Wert ist der Typ des Keys und der darauffolgende Wert der Typ der Map-Elemente. Alle Elemente einer Map haben denselben Key- und Werte-Typ. Das Beispiel erzeugt eine Map mit Integer-Keys, die Strings enthält. Alternativ zur Literalschreibweise können Maps mit *make* erzeugt werden:

```
1  var m = make(map[int]string, 2)
```

Der optionale Längenparameter bestimmt die Größe der initial allokierten Map. Wird er weggelassen, wird initial eine sehr kleine Map erzeugt.

[3] Polymorph bedeutet, dass ein Typ zur Laufzeit verschiedene Formen annehmen kann. Im Beispiel können die *Figure*-Elemente des Arrays die Formen Kreis oder Rechteck annehmen.

Die Zuweisung von Elementen erfolgt durch Angabe des Keys in eckigen Klammern, gefolgt vom Zuweisungsoperator:

```
m[1] = "Kalle"
```

Bereits vorhandene Werte werden überschrieben. Der Zugriff auf Map-Elemente erfolgt in umgekehrter Richtung:

```
m[1] = "Kalle"
fmt.Println(m[1]) // => Kalle
```

Enthält die Map unter dem angegebenen Key keinen Wert, liefert der Zugriff den Defaultwert des jeweiligen Typs:

```
fmt.Println(m[100]) // => ""
```

Das ist nicht immer gewünscht, denn ein leerer String ist ein gültiger Wert und eignet sich nicht für das Prüfen auf das Vorhandensein eines Elements. Aus diesem Grund erfolgt die Überprüfung, ob die Map einen Wert für einen bestimmten Key enthält, über das sogenannte *Ok-Idiom*:

```
if v, ok := m[1]; ok {
    fmt.Println(v) // => Kalle
}
```

Das Keyword *range* iteriert durch die Elemente einer Map und liefert für alle Elemente Key und Wert:

```
for key, value := range m {
    fmt.Println(key, value)
}
```

2.16 Verzögerungen, Panic und Recover

Eine interessante Variante zur Steuerung des Kontrollflusses ist die Verwendung von *defer*. Defer verzögert einen Funktionsaufruf solange, bis der umschließende Funktionsblock beendet ist:

```
func readFile() (int, []byte) {
    f, err := os.Open("/tmp/contact.json")
    defer f.Close()
    check(err)
    ...
}
```

Egal, was in *readFile* nach dem Öffnen der Datei passiert, *f.Close* wird erst am Ende von *readFile* ausgeführt. Die Verwendung von *defer* hat zwei Vorteile: Zum einen werden Aufräumarbeiten direkt an die Stelle gerückt, auf die sie sich beziehen. So lässt sich einfacher erkennen, dass eine geöffnete Datei auch wieder geschlossen wird. Zum anderen werden verzögerte Funktionsaufrufe auch dann ausgeführt, wenn die Funktion in ihrem Verlauf crashed (aka *panicked*).

Ein Go-Programm kann aus zwei Gründen panicken: Entweder entdeckt die Go-Runtime einen Laufzeitfehler und beendet das Programm mit Ausgabe eines Stacktrace. Oder das Programm wird explizit durch Aufruf der Build-In-Funktion *panic* beendet. Go-Programmierer benutzen *panic* für Fehler, die quasi unmöglich sind, aber doch irgendwie behandelt werden müssen. Zum Beispiel kann die Serialisierung eines Kontakts eigentlich nicht schiefgehen, aber was, wenn doch?

```
1 if b, err := json.Marshal(contact); err == nil {
2     send(b)
3 } else {
4     panic(err)
5 }
```

Ein panic-Aufruf beendet das komplette Programm, was häufig, aber nicht immer gewünscht ist. Panicked beispielsweise eine Go-Routine[4], dann soll das nicht zwingend zum Abbruch der Main-Funktion führen. Mithilfe der Build-In-Funktion *recover* lässt sich auf *panic* reagieren:

```
1  func foo() (err error)  {
2      defer func() {
3          if p := recover(); p != nil {
4              err = fmt.Errorf("%v", p)
5          }
6      }()
7      ...
8      panic("help")
9      return nil
10 }
```

Ein Recover-Aufruf findet immer in einer verzögert ausgeführten Funktion statt. Panicked die umschließende Funktion, dann werden alle verzögerten Funktionen in umgekehrter Reihenfolge ausgeführt. Das verzögerte Recover im Beispiel bricht den Panic-Aufruf ab und liefert den ursprünglich an Panic übergebenen Wert. Dieser wird dem benannten Return-Wert *err* zugewiesen und unmittelbar zurückgeliefert.

Sinnvolle Recovery-Fälle sind Aufräumarbeiten vor endgültigem Programmabbruch, Handling von Panic-Aufrufen in Third-Party-Bibliotheken oder das Abfangen von Panic-Aufrufen aus HTTP-Handlern heraus. Letzteres ist im Go-Package *net/http* so umgesetzt und verhindert ein Beenden des kompletten HTTP-Servers, wenn ein einzelner Request-Handler mit *panic* abbricht.

[4] Eine Go-Routine ist ein parallel ablaufender Strang innerhalb eines Go-Programms. Go-Routinen werden in Abschnitt 10.4.1 des Kapitels *Skalierung* beschrieben.

2.17 Was sonst noch?

Dieses Kapitel versteht sich als Grundlagenkapitel. Das Kapitel beschreibt gerade soviel Go wie nötig ist, um die folgenden Kapitel zu verstehen. Weitergehende Konzepte wie Closures, Middleware, Concurrency oder Kontexte werden im weiteren Verlauf des Buches immer dann eingeführt, wenn sie konkret benötigt werden.