

### Einbindung in gängige Editoren

Für einige der gängigen Editoren und Entwicklungsumgebungen wie *Emacs*, *vim*, *BEdit*, *Kate* und *XCode* werden Unterstützungen mitgeliefert. Weitere Programme wie *TextMate* oder *Eclipse* werden durch die Community unterstützt. Auf der Seite <http://go-lang.cat-v.org/text-editors/> finden Sie entsprechende Links.

## 1.6 Etwas mehr Struktur

So schön das Beispiel ist, so eingeschränkt ist es auch in seinen Fähigkeiten. Diese sollen nun Zug um Zug ausgebaut werden. Im ersten Schritt soll etwas Struktur in das Programm gebracht werden. So soll es nicht nur einen Gruß geben, sondern unterschiedliche Grußformeln. Dazu sollen diese in ein Package ausgelagert werden. Dieses Package kann anschließend direkt durch ein Hauptprogramm oder weitere Packages genutzt werden.

### 1.6.1 Die erste eigene Bibliothek

Unser erstes Package trägt seiner Aufgabe entsprechend den passenden Namen `greetings`. Aktuell besteht es nur aus einer Datei, kann jedoch bei entsprechender Größe auf mehrere Dateien aufgeteilt werden. Sie sind nur innerhalb eines Verzeichnisses zu speichern und müssen die gleiche `package`-Anweisung enthalten.

Unsere Bibliothek enthält verschiedene Funktionen, die jeweils einen unterschiedlichen Gruß ausgeben. Diese Funktionen erhalten jeweils den Namen der zu grüßenden Person als Parameter.

```
Quelltext 1.3      /* Ein Gruß-Package */
Ein Gruß-Package package greetings

import "fmt"

// Hallo sagen.
func Hello(name string) {
    greeting := "Hello, %v!\n"

    fmt.Printf(greeting, name)
}
```

```
// Verabschieden.  
func Cheerio(name string) {  
    greeting := "Cheerio, %v!\n"  
  
    fmt.Printf(greeting, name)  
}
```

Der generelle Aufbau des Packages ähnelt dem des Hauptprogramms im Quelltext 1.2 sehr. Der Name ist, wie gewünscht, in `greetings` geändert worden, und es enthält neben zweier Grußfunktionen keine Hauptfunktion `main()`. Dafür zeigt es einleitend, dass für Kommentare wie in C/C++ zudem noch die Form `/*...*/` zur Verfügung steht.

Das Package `fmt` wird wie bereits vom ersten Beispiel bekannt importiert, jedoch existiert kein expliziter Export der eigenen Funktionen. Werden also immer alle Funktionen, Typen oder Variablen exportiert? Oder woher weiß das System, welche Funktionen für andere Systeme nutzbar sind?

Im Beispiel oben beginnen `Hello()` und `Cheerio()` im Gegensatz zur bekannten Hauptfunktion mit einem Großbuchstaben. Und genau dies ist das implizite Verfahren für den Export von Bezeichnern. Alle kleingeschriebenen Variablen, Konstanten, Typen, Channels und Funktionen sind nur innerhalb des jeweiligen Packages sichtbar, auch über Dateigrenzen hinweg. Alle großgeschriebenen können hingegen im importierenden Package genutzt werden. Dies haben Sie bereits bei der ersten Verwendung von `Printf()` auf Seite 7 gesehen.

Packages selbst bilden zudem einen Namensraum. Die in ihm definierten Bezeichner dürfen nur einmal vorkommen. Es kann also keine weitere Funktion mit dem Namen `Hello()` definiert werden. Das Gleiche gilt für alle weiteren Bezeichner. Sie alle müssen innerhalb eines Packages eindeutig sein. Bei der Verwendung verschiedener Packages dürfen hingegen gleiche Namen auftreten. Sie werden dort über den Packagenamen oder dessen möglichen Alias qualifiziert. Dieser Alias kommt auch zum Einsatz, wenn mehrere Packages gleichen Namens importiert werden soll. Zur Vermeidung dieser Problematik gibt es jedoch zudem noch eine Namenskonvention, die in Abschnitt 3.1.2 näher erläutert wird.

## 1.6.2 Nutzung der eigenen Bibliothek

Nun kann ein Hauptprogramm unser neues Package `greetings` importieren. Wurde das Package einfach nur kompiliert und das einbindende Hauptpaket befindet sich im gleichen Verzeichnis, so ist ein `./` voranzustellen. Bei einer Übersetzung via `make` wird hingegen der dort genutzte *Target Name* genutzt. Hierzu später mehr.

Nach dem Import kann das Hauptprogramm unsere exportierten Funktionen nutzen. Wie schon zuvor mit der Nutzung des Packages `fmt` im Quelltext 1.2 werden die Funktionen über den Namen des Packages und den Funktionsnamen, getrennt durch einen Punkt, adressiert.

**Quelltext 1.4** `package main`  
*»Hello, world« mit Bibliothek* `import "./greetings"`

```

func main() {
    var world string = "improved world"

    greetings.Hello(world)
    greetings.Cheerio(world)
}

```

### 1.6.3 Kontrollkonstrukte

Verschiedene Kontrollkonstrukte können uns nun helfen, den Verlauf des Programms dynamisch in Abhängigkeit von Werten zu verändern. So steht für Verzweigungen das bekannte `if` zur Verfügung, bei Bedarf natürlich auch mit einem `else`-Zweig. Die Unterscheidung zwischen mehreren Werten erfolgt hingegen mittels `switch`, das den bisherigen Nutzern von C-Sprachen ebenfalls bekannt sein dürfte. Neu ist jedoch `select` für das Lesen aus mehreren Channels im Rahmen der Nebenläufigkeit. Hierzu finden Sie ab Seite 33 mehr.

Für Schleifen bietet Go das `for`-Konstrukt. Es ist vielfältig und kann sowohl für Zähler als auch für an Bedingungen geknüpfte oder endlose Schleifen eingesetzt werden.

Damit können wir unser Grußpaket nun flexibler gestalten. Die Funktion `Hello()` soll um unterschiedliche Anreden – abhängig vom Geschlecht der Person – erweitert werden. Dieses kann vom Nutzer über ein zusätzliches Attribut ausgewählt werden.

**Quelltext 1.5** `func Hello(gender int, name string) {`  
*Hello mit Anrede* `var addr string`

```

switch gender {
default:
    addr = ""
case 1:
    addr = "Ms. "
}
}

```

```
    case 2:
        addr = "Mr. "
    }

    fmt.Printf("Hello, %v%v!\n", addr, name)
}
```

Kennern anderer C-Sprachen fällt auf, dass Go eine Positionierung des Standardzweigs mit **default**: am Anfang erlaubt. Es werden immer alle **case**-Ausdrücke von oben nach unten und von links nach rechts ausgewertet, erst dann der Standardzweig. Weiterhin fehlt das aus C bekannte **break**, es ist in Go an dieser Stelle nicht notwendig.

### 1.6.4 Variable Parameteranzahl

Als wir `Printf()` oben das erste Mal genutzt haben, hatte es nur zwei Parameter: die Format-Zeichenkette sowie eine einzusetzende Variable. Dieses Mal sind es hingegen drei Parameter, da noch eine weitere einzusetzende Variable hinzugekommen ist. Hierbei handelt es sich aber um keine andere Funktion mit gleichem Namen. Go erlaubt es nicht, den gleichen Funktionsnamen mit unterschiedlichen Parametern mehrfach zu verwenden, ihn also zu überladen.

Dafür ist jedoch die Nutzung von `...T` als letztem Parameter einer Funktion erlaubt. Der Buchstabe *T* steht für einen Typ, und die drei Punkte zeigen an, dass hiervon kein bis beliebig viele Parameter übergeben werden dürfen. Handelt es sich hierbei um den Typ `interface{}` dürfen sogar beliebige Parameter übergeben werden, bunt untereinander gemischt. Go stellt Konstrukte zur Auswertung dieser Typen zur Verfügung; sie werden später noch vorgestellt. Die Funktion `Printf()` nutzt dieses Verhalten.

Mit diesem Wissen bietet sich eine weitere Variante der Funktion `Hello()` an. Sie soll nun eine beliebige Anzahl an Grüßen ausgeben können. Hierfür sei ein kurzer Vorgriff auf die Definition eigener Typen erlaubt – in diesem Fall handelt es sich um ein Struct mit zwei Feldern. Mehr zu diesen zusammengesetzten Typen und weiteren folgt dann ab Seite 18.

Neben der Typdefinition enthält das Beispiel einige weitere Neuigkeiten. Die Variable `who` steht innerhalb der Funktion als eine Liste vom Typ `Greeting` zur Verfügung. Der hier tatsächlich zum Einsatz kommende Typ und wie mit ihm umgegangen werden kann, wird in Abschnitt 2.4 vorgestellt. Für den Moment soll es genügen, dass `who` zwischen 0 und einer nicht spezifizierten Anzahl an `Greetings` enthalten kann, je nach Anzahl der beim Aufruf übergebenen Parameter.

Quelltext 1.6  
Mehrfaches Hello

```

type Greeting struct {
    Gender int
    Name   string
}

func Hello(who ...Greeting) {
    var addr string

    for idx, g := range who {
        switch g.Gender {
        default:
            addr = ""
        case 1:
            addr = "Ms. "
        case 2:
            addr = "Mr. "
        }

        fmt.Printf("(%v) Hello, %v%v!\n", idx, addr,
            g.Name)
    }
}

```

Die einzelnen Elemente können wir komfortabel mit einer **for**-Schleife zusammen mit dem Schlüsselwort **range** auslesen. Als Ergebnis wird hier **idx** der jeweils aktuelle Index und **g** der dazugehörige Gruß vom Typ **Greeting** zugeordnet. Der innere Block der Schleife hat sich kaum verändert, nur dass nun auf die beiden Felder von **g** zugegriffen wird.

Nun kann **Hello()** mit einer beliebigen Anzahl von Kombinationen aus Geschlecht und Name aufgerufen werden. Ohne Parameter wird nichts ausgegeben, andernfalls pro Paar jeweils eine Zeile.

Quelltext 1.7  
Aufruf des  
mehrfachen Hello

```

package main

import . "./greetings"

// Hauptprogramm.
func main() {
    // Drei Grüße.

    var (
        gA Greeting = Greeting{1, "Miller"}
        gB Greeting = Greeting{2, "Smith"}
        gC Greeting = Greeting{99, "Jones"}
    )
}

```

```
// GrüÙe aussprechen.  
  
Hello()  
Hello(gA, gB)  
Hello(gC)  
}
```

Neben der Nutzung der geänderten Funktion `Hello()` mit unterschiedlichen Anzahlen an Parametern fällt noch die Deklaration und Initialisierung der drei Variablen auf. Anstatt jede durch das Kommando `var` einzuleiten, können mehrere durch runde Klammern gebündelt werden. Diese Einsparung an Tipparbeit ist auch bei Importen sowie bei der Definition von Konstanten und globalen Variablen möglich.

Auch für die Zuweisung der Werte in den Structs wird eine Abkürzung genutzt. Anstatt jedes Feld einzeln zu initialisieren, erfolgt dies in einer Anweisung über eine Werteliste in geschweiften Klammern. Die Werte müssen in der Reihenfolge der Felder angegeben werden. Alternativ lassen sich auch die Bezeichner, gefolgt von einem Doppelpunkt und dem Wert, angeben. Diese komfortable Variante kennt noch weitere Formen und lässt sich auch bei den Speicherstrukturen *Array*, *Slice* und *Map*, die in Kapitel 2 vorgestellt werden, anwenden.

### Verzicht auf Packagenamen

Generell werden importierte Typen, Funktionen, Konstanten und Variablen über eine Kombination aus dem Packagenamen und dem Bezeichner angesprochen. Wird eine große Anzahl an Packages in einem umfangreichen Programm eingebunden, so ist dies alleine aus Gründen der Wartbarkeit sinnvoll. Beim Import kann jedoch auch ein Alias oder der Verzicht definiert werden, letzteres wie im Beispiel über einen vorangestellten Punkt. Mehr Details hierzu finden Sie in Abschnitt 3.1.2.

## 1.7 Konstanten

Das Geschlecht wurde im oben aufgeführten Beispiel über eine Zahl bestimmt. Schöner ist jedoch der Einsatz sprechender Konstanten. Sie sagen mehr über den Sinn des Codes aus. Über das Schlüsselwort `const` werden wir sie daher ebenfalls in der Bibliothek `greetings` deklarieren und für die Nutzung außerhalb exportieren.

Der Quelltext 1.8 ist so deutlich besser lesbar. Im aktuellen Fall sind die Konstanten nicht speziell typisiert. Der `const`-Ausdruck definiert sie implizit als Typ `int`. Ist der Ausdruck eindeutig, so ist