

# Contents

<b>Contents</b>	<b>1</b>
<b>1 Intro</b>	<b>2</b>
1.1 Official documentation . . . . .	2
1.2 Compiling and running code . . . . .	3
<b>2 Go basics and features</b>	<b>4</b>
2.1 Hello World . . . . .	5
<b>3 Undefined</b>	<b>6</b>
3.1 Exercises . . . . .	6
3.2 Answers . . . . .	6
<b>4 Colophon</b>	<b>9</b>

# 1

## Intro

"I am interested in this and hope to do something."

---

Ken Thompson on adding complex numbers to Go.

This is an introduction into the Go language from Google. Its aim is to provide a short .... and provide exercises to .... Its intended audience is people who are familiar with programming and know different languages, be it C, C++, Java, Erlang or Haskell. The exercises included should make you familiar with the syntax of Go..... So we are not starting with the basics of programming, we starting with the basics of Go and then go right of to the cool stuff.

Each chapter concludes with a number of exercises. Each exercise is numbered **Q***n*, where *n* is a number. After the exercise number another number in parentheses displays the difficulty of this particular assignment. This difficulty ranges from 0 to 9, where 0 is easy and 9 is extremely difficult (or at least it should be). Then in brackets a short name is given. An example might be:

**Q1.** (4) [A map function] ...

giving a question of a level 4 difficulty, concerning a `map()`-function. The answers to all exercises are included in the final chapter, the numbering and setup of the answer is identical to the exercises.

All the included source code (either Go or shell) is tested and should compile. The following conventions are used throughout this book:

- Code is displayed in a *courier* font;
- Keywords are displayed in a ***courier bold font***;
- Comments are displayed in a *courier italic font*;
- Lines that are too long to display are cut off with a ↵ which means it originaly was one line.

### 1.1 Official documentation

The reader is assumed to have read (and somewhat understand), the Go Tutorial<sup>1</sup>, and the Effective Go document<sup>2</sup>.

Didn't read those documents? Go read them now and stop wasting your time with this doc.

---

<sup>1</sup><http://www.golang.org/dnsjk>

<sup>2</sup><http://www.golang.org/fdmkd>

## 1.2 Compiling and running code

This is a very concise instruction on how to get your code compiled and get it running. The shortest program you can write in Go looks like this:

Listing 1.1: Shortest Go program

```
package main                                1
func main() {}                               2
```

In which we define the **package** `main` and supply it with one function also called `main()`. The fully qualified function name is `main.main()`. This is the function that is called first. To compile we do the following

```
% 8g short.go # compiles to short.8 (for 32 bit Intel)
% 8l -o short short.8
```

For 64 bit Intel you should use `6g` and `6l`, this will generate `short.6`. You can then execute the program `short`, which of course does absolute nothing. ..X better, use the official go way

## 2

# Go basics and features

A few things that make Go different than most other language out there.

### Concurrent

Go makes it easy to "fire off" functions to be run as *very* lightweight threads. These threads are called *go-routines* in Go;

### Channels

Communication with these go-routines is done via *channels*<sup>1</sup>

### Fast

Compilation is fast and execution as fast. The aim is to be as fast as C. Compilation time is measured in seconds;

### Safe

Go has garbage collection, no more `malloc()` in Go, the language takes care of this;

### Standard formatting

A Go program can be formatted in (almost) any way the programmers want, *but* an official format exist. The rule is very simple:

The output of the filter `gofmt` is the official endorsed format.

### Types are postfix

Types are given *after* the variable name, thus `var a int`, instead of `int a`; as in C;

### UTF-8

UTF-8 is all over the place, in strings *and* in the program code. Finally you can use  $\Phi$  in your source code;

### Open Source

The Go license is completely open source;

### Fun

Programming with Go should be fun again!

It must be said the Erlang and Scala (ohja) also share some of the features of Go. Notible differences between Erlang and Go is that Erlang border on being a functional language, where Go is an imperative one. And Erlang is running a virtual machine, while Go is compiled. Erlang is much more mature.

I have no clue what Scala does, but it still runs on a virtual machine, in this case the Java Virtual Machine (JVM).

Well, on with Go, as that it the one you want to XXX todo

---

<sup>1</sup>See [http://en.wikipedia.org/wiki/Communicating\\_sequential\\_processes](http://en.wikipedia.org/wiki/Communicating_sequential_processes) for more information.

## 2.1 Hello World

In the Go tutorial, Go is presented to the world in the typical manner: letting it print "Hello World" (Heck! Ken Thompson and Dennis Ritchie started this when they presented the C language in the nineteen seventies). So here it is, a slightly modified "Hello World" in Go.<sup>2</sup>,

Listing 2.1: Hello World

```
package main                                1
import fmt "fmt" // Implements formatted I/O. 3
func main() {                                5
    fmt.Printf("Hello, world; or Καλημέρα κόσμε\n") 6
}                                              7
```

<sup>2</sup>The original version also includes Japanese output, but this is very hard to reproduce in  $\text{\LaTeX}$ .

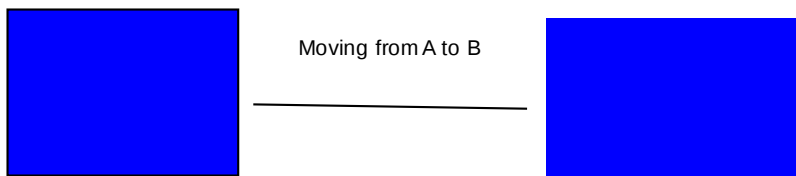


Figure 2.1: een verhaaltje

# 3

## Undefined

### 3.1 Exercises

**Q1.** (4) [Map function] A `map()`-function is a function that takes a function and a list. The function is applied to each member in the list and a new list containing these calculated values is returned. Thus:

$$\text{map}(f(), (a_1, a_2, \dots, a_{n-1}, a_n)) = (f(a_1), f(a_2), \dots, f(a_{n-1}), f(a_n))$$

1. Write a simple `map()`-function in Go. It is sufficient for this function only to work for `ints`.
2. Expand your code to also work on a list of `strings`.
3. Now make it generic using interfaces.

**Q2.** (5) [Fibonacci] The Fibonacci sequence starts as follows: 1, 1, 2, 3, 5, 8, 13, ... Or in mathematical terms:  $n = (n - 1) + (n - 2)$ .

1. Write a function that takes an `int` value and gives to Fibonacci sequence up to that value.

### 3.2 Answers

**A1.** (4) [Map function] First answer Second answer

Listing 3.1: A map function in Go

```
package main 1
import "fmt" 2

/* define the empty interface as a type */ 4
type e interface{} 5

func mult2(f e) e { 7
    switch f.(type) { 8
    case int: 9
        return f.(int) * 2 10
    case string: 11
        return f.(string) + f.(string) + f.(string) + f.( 12
            string)
    } 13
    return f 14
} 15

func Map(n []e, f func(e) e) { 17
    for k, v := range n { 18
```

```

        n[k] = f(v)
    }
}

func main() {
    m := [...]e{1, 2, 3, 4}
    s := [...]e{"a", "b", "c", "d"}
    Map(&m, mult2)
    Map(&s, mult2)
    fmt.Printf("%v\n", m)
    fmt.Printf("%v\n", s)
}

```

## A2. (5) [Fibonacci]

Listing 3.2: A Fibonacci function in Go

```

package main
import "fmt"

func dup3(in <-chan int) (<-chan int, <-chan int, <-chan
int) {
    a, b, c := make(chan int, 2), make(chan int, 2),
make(chan int, 2)
    go func() {
        for {
            x := <-in
            a <- x
            b <- x
            c <- x
        }
    }()
    return a, b, c
}

func fib() <-chan int {
    x := make(chan int, 2)
    a, b, out := dup3(x)
    go func() {
        x <- 0
        x <- 1
        <-a
        for {
            x <- <-a+<-b
        }
    }()
    return out
}

func main() {
    x := fib()
    for i := 0; i < 10; i++ {
        fmt.Println(<-x)
    }
}

```

} 36

// See [sdh33b.blogspot.com/2009/12/fibonacci-in-go.html](http://sdh33b.blogspot.com/2009/12/fibonacci-in-go.html) 38

## Colophon

This work was created with  $\LaTeX$ . The serif font is Liberation Serif, all typewriter text is typeset in Courier New.

Primary author is Miek Gieben<sup>1</sup>.

---

<sup>1</sup>miek@miek.nl