Preface

The source of this book is written in mmark and is converted from the original LaTeX source.

The following people made large or small contributions to earlier versions of this book:

Adam J. Gray, Alexander Katasonov, Alexey Chernenkov, Alex Sychev, Andrea Spadaccini, Andrey Mirtchovski, Anthony Magro, Babu Sreekanth, Ben Bullock, Bob Cunningham, Brian Fallik, Cecil New, Cobold, Damian Gryski, Daniele Pala, Dan Kortschak, David Otton, Fabian Becker, Filip Zaludek, Hadi Amiri, Haiping Fan, Iaroslav Tymchenko, Jaap Akkerhuis, JC van Winkel, Jeroen Bulten, Jinpu Hu, John Shahid, Jonathan Kans, Joshua Stein, Makoto Inoue, Marco Ynema, Mayuresh Kathe, Mem, Michael Stapelberg, Nicolas Kaiser, Olexandr Shalakhin, Paulo Pinto, Peter Kleiweg, Philipp Schmidt, Robert Johnson, Russel Winder, Simoc, Sonia Keys, Stefan Schroeder, Thomas Kapplet, T.J. Yang, Uriel†, Vrai Stacey, Xing Xing.

"Learning Go" has been translated into (note that this used the original LaTeX source).

- Chinese, by Xing Xing, 这里是中文译本: http://www.mikespook.com/learning-go/

I hope this book is useful.

Miek Gieben, London, 2015.

This book still sees development, small incremental improvements trickle in from Github.

Miek Gieben, London, 2017.

Learning Go's source has been rewritten in mmark2, but did not see any other changes. This books translates cleanly into an RFC-like document.

Miek Gieben, London, 2018.

L E A R N I N G G O

Chapter 1. Introduction

Is Go an object-oriented language? Yes and no.

FREQUENTLY ASKED QUESTIONS, GO AUTHORS

The Go programming language is an open source project language to make programmers more productive.

According to the website [go_web] "Go is expressive, concise, clean, and efficient". And indeed it is. My initial interest was piqued when I read early announcements about this new language that had built-in concurreny and a C-like syntax (Erlang also has built-in concurrency, but I could never get used to its syntax). Go is a compiled statically typed language that feels like a dynamically typed, interpreted language. My go to (scripting!) language Perl has taken a back seat now that Go is around.

The unique Go language is defined by these principles:

Clean and Simple
: Go strives to keep things small and beautiful. You should be able to do a lot in only a few lines of code.

Concurrent
: Go makes it easy to "fire off" functions to be run as *very* lightweight threads. These threads are called goroutines [1] in Go.

Channels
: Communication with these goroutines is done, either via shared state or via channels [csp].

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

Safe
: Explicit casting and strict rules when converting one type to another. Go has garbage collection. No more `free()` in Go: the language takes care of this.

Standard format
: A Go program can be formatted in (almost) any way the programmers want, but an official format exists. The rule is very simple: The output of the filter `gofmt` *is the officially endorsed format*.

Postfix types
: Types are given *after* the variable name, thus `var a int`, instead of `int a`.

UTF-8
: UTF-8 is everywhere, in strings *and* in the program code. Finally you can use $\Phi = \Phi + 1$ in your source code.

Open Source
: The Go license is completely open source.

Fun
: Programming with Go should be fun!

As I mentioned Erlang also shares some features of Go. A notable difference between Erlang and Go is that Erlang borders on being a functional language, while Go is imperative. And Erlang runs in a virtual machine, while Go is compiled.

How to Read this Book

I've written this book for people who already know some programming languages and how to program. In order to use this book, you (of course) need Go installed on your system, but you can easily try examples online in the Go playground[2]. All exercises in this book work with Go 1, the first stable release of Go – if not, it's a bug.

The best way to learn Go is to create your own programs. Each chapter therefore includes exercises (and answers to exercises) to acquaint you with the language. Each exercise is either *easy*, *intermediate*, or *difficult*. The answers are included after the exercises on a new page. Some exercises don't have an answer; these are marked with an asterisk.

Here's what you can expect from each chapter:

Chapter 2. Basics
: We'll look at the basic types, variables, and control structures available in the language.

Chapter 3. Functions
: Here we look at functions, the basic building blocks of Go programs.

Chapter 4. Packages
: We'll see that functions and data can be grouped together in packages. We'll also see how to document and test our packages.

Chapter 5. Beyond the basics

We'll create our own types. We'll also look at memory allocations in Go.

Chapter 6. Interfaces

We'll learn how to use interfaces. Interfaces are the central concept in Go, as Go does not support object orientation in the traditional sense.

Chapter 7. Concurrency

We'll learn the `go` keyword, which can be used to start function in separate routines (called goroutines). Communication with those goroutines is done via channels.

Chapter 8. Communication

Finally we'll see how to interface with the rest of the world from within a Go program. We'll see how to create files and read and write to and from them. We'll also briefly look into networking.

Official Documentation

There is a substantial amount of documentation written about Go. The Go Tutorial [go_tutorial], the Go Tour (with lots of exercises) and the Effective Go [effective_go] are helpful resources. The website http://golang.org/doc/ is a very good starting point for reading up on Go[3]. Reading these documents is certainly not required, but it is recommended.

> *When searching on the internet use the term "golang" instead of plain "go".*

Go comes with its own documentation in the form of a program called `godoc`[4]. If you are interested in the documentation for the built-ins, simply do this:

```
% godoc builtin
```

To get the documentation of the `hash` package, just:

```
% godoc hash
```

To read the documentation of `fnv` contained in `hash`, you'll need to issue `godoc hash/fnv` as `fnv` is a subdirectory of `hash`.

```
PACKAGE DOCUMENTATION

package fnv
    import "hash/fnv"

    Package fnv implements FNV-1 and FNV-1a, non-cryptographic ha
    ...
```

Chapter 2. Basics

I am interested in this and hope to do something.

ON ADDING COMPLEX NUMBERS TO GO, KEN THOMPSON

In this chapter we will look at the basic building blocks of the Go programming language.

Hello World

In the Go tutorial, you get started with Go in the typical manner: printing "Hello World" (Ken Thompson and Dennis Ritchie started this when they presented the C language in the 1970s). That's a great way to start, so here it is, "Hello World" in Go.

```
package main 1

import "fmt" 2 // Implements formatted I/O.

/* Print something */ 3
func main() {          4
    fmt.Printf("Hello, world.") 5
}
```

Lets look at the program line by line. This first line is just required 1 . All Go files start with `package <something>`, and `package main` is required for a standalone executable.

`import "fmt"` says we need `fmt` in addition to `main` 2 . A package other than `main` is commonly called a library, a familiar concept in many programming languages (see Chapter 4. Packages). The line ends with a comment that begins with `//`.

Next we another comment, but this one is enclosed in `/* */` 3 . When your Go program is executed, the first function called will be `main.main()`, which mimics

the behavior from C. Here we declare that function [4].

Finally we call a function from the package `fmt` to print a string to the screen. The string is enclosed with `"` and may contain non-ASCII characters [5].

Compiling and Running Code

To build a Go program, use the `go` tool. To build `helloworld` we just enter:

```
% go build helloworld.go
```

This results in an executable called `helloworld`.

```
% ./helloworld
Hello, world.
```

You can combine the above and just call `go run helloworld.go`.

Variables, Types and Keywords

In the next few sections we will look at the variables, basic types, keywords, and control structures of our new language.

Go is different from (most) other languages in that the type of a variable is specified *after* the variable name. So not: `int a`, but `a int`. When you declare a variable it is assigned the "natural" null value for the type. This means that after `var a int`, `a` has a value of 0. With `var s string`, `s` is assigned the zero string, which is `""`. Declaring and assigning in Go is a two step process, but they may be combined. Compare the following pieces of code which have the same effect.

```
var a int
var b bool
a = 15
b = false
```

```
a := 15
b := false
```

On the left we use the `var` keyword to declare a variable and *then* assign a value to it. The code on the right uses `:=` to do this in one step (this form may only be used *inside* functions). In that case the variable type is *deduced* from the value. A value of 15 indicates an `int`. A value of `false` tells Go that the type should be `bool`. Multiple `var` declarations may also be grouped; `const` (see Constants) and `import` also allow this. Note the use of parentheses instead of braces:

```
var (
    x int
```

```
    b bool
)
```

Multiple variables of the same type can also be declared on a single line: `var x, y int` makes `x` and `y` both `int` variables. You can also make use of *parallel assignment* `a, b := 20, 16`. This makes `a` and `b` both integer variables and assigns 20 to `a` and 16 to `b`.

A special name for a variable is `_`. Any value assigned to it is discarded (it's similar to `/dev/null` on Unix). In this example we only assign the integer value of 35 to `b` and discard the value 34: `_, b := 34, 35`. Declared but otherwise *unused* variables are a compiler error in Go.

Boolean Types

A boolean type represents the set of boolean truth values denoted by the predeclared constants *true* and *false*. The boolean type is `bool`.

Numerical Types

Go has most of the well-known types such as `int`. The `int` type has the appropriate length for your machine, meaning that on a 32-bit machine it is 32 bits and on a 64-bit machine it is 64 bits. Note: an `int` is either 32 or 64 bits, no other values are defined. Same goes for `uint`, the unsigned int.

If you want to be explicit about the length, you can have that too, with `int32`, or `uint32`. The full list for (signed and unsigned) integers is `int8`, `int16`, `int32`, `int64` and `byte`, `uint8`, `uint16`, `uint32`, `uint64`, with `byte` being an alias for `uint8`. For floating point values there is `float32` and `float64` (there is no `float` type). A 64 bit integer or floating point value is *always* 64 bit, also on 32 bit architectures.

Note that these types are all distinct and assigning variables which mix these types is a compiler error, like in the following code:

```
package main

func main() {
    var a int
    var b int32
    b = a + a
    b = b + 5
}
```

We declare two different integers, a and b where a is an `int` and b is an `int32`. We want to set b to the sum of a and a. This fails and gives the error: `cannot use a + a (type int) as type int32 in assignment`. Adding the constant 5 to b *does* succeed, because constants are not typed.

Constants

Constants in Go are just that — constant. They are created at compile time, and can only be numbers, strings, or booleans; `const x = 42` makes `x` a constant. You can use *iota* [5] to enumerate values.

```
const (
    a = iota
    b
)
```

The first use of `iota` will yield 0, so `a` is equal to 0. Whenever `iota` is used again on a new line its value is incremented with 1, so `b` has a value of 1. Or, as shown here, you can even let Go repeat the use of `iota`. You may also explicitly type a constant: `const b string = "0"`. Now `b` is a `string` type constant.

Strings

Another important built-in type is `string`. Assigning a string is as simple as:

```
s := "Hello World!"
```

Strings in Go are a sequence of UTF-8 characters enclosed in double quotes (“). If you use the single quote (‘) you mean one character (encoded in UTF-8) — which is *not* a `string` in Go.

Once assigned to a variable, the string cannot be changed: strings in Go are immutable. If you are coming from C, note that the following is not legal in Go:

```
var s string = "hello"
s[0] = 'c'
```

To do this in Go you will need the following:

```
s := "hello"
c := []rune(s)       1
c[0] = 'c'           2
s2 := string(c)      3
fmt.Printf("%s\n", s2) 4
```

Here we convert `s` to an array of runes [1]. We change the first element of this array [2]. Then we create a *new* string `s2` with the alteration [3]. Finally, we print the string with `fmt.Printf` [4].

Runes

`Rune` is an alias for `int32`. It is an UTF-8 encoded code point. When is this type useful? One example is when you're iterating over characters in a string. You could loop over each byte (which is only equivalent to a character when strings are encoded in 8-bit ASCII, which they are *not* in Go!). But to get the actual characters you should use the `rune` type.

Complex Numbers

Go has native support for complex numbers. To use them you need a variable of type `complex128` (64 bit real and imaginary parts) or `complex64` (32 bit real and imaginary parts). Complex numbers are written as `re + imi`, where `re` is the real part, `im` is the imaginary part and i is the literal 'i' ($\sqrt{-1}$).

Errors

Any non-trivial program will have the need for error reporting sooner or later. Because of this Go has a builtin type specially for errors, called `error`. `var e error` creates a variable `e` of type `error` with the value `nil`. This error type is an interface – we'll look more at interfaces in Chapter 6. Interfaces. For now you can just assume that `error` is a type just like all other types.

Operators and Built-in Functions

Go supports the normal set of numerical operators. See Table 1 for lists the current ones and their relative precedence. They all associate from left to right.

Precedence	Operator(s)
Highest	`* / % << >> & &^`
	`+ -`
	`== != < <= > >=`
	`<-`
	`&&`
Lowest	`||`

Table 1. Operator precedence.

`+ - * /` and `%` all do what you would expect, `& | ^` and `&^` are bit operators for bitwise *and* bitwise *or* bitwise *xor* and bit clear respectively. The `&&` and `||` operators are logical *and* and logical *or* Not listed in the table is the logical not `!`

Although Go does not support operator overloading (or method overloading for that matter), some of the built-in operators *are* overloaded. For instance, `+` can be used for integers, floats, complex numbers and strings (adding strings is concatenating them).

Go Keywords

Let's start looking at keywords, Table 2 lists all the keywords in Go.

`break`	`default`	`func`	`interface`	`select`
`case`	`defer`	`go`	`map`	`struct`
`chan`	`else`	`goto`	`package`	`switch`
`const`	`fallthrough`	`if`	`range`	`type`
`continue`	`for`	`import`	`return`	`var`

Table 2. Keywords in Go.

We've seen some of these already. We used `var` and `const` in the Variables, Types and Keywords section, and we briefly looked at `package` and `import` in our "Hello World" program at the start of the chapter. Others need more attention and have their own chapter or section:

- `func` is used to declare functions and methods.
- `return` is used to return from functions. We'll look at both `func` and `return` in detail in Chapter 3. Functions.
- `go` is used for concurrency. We'll look at this in Channels.
- `select` used to choose from different types of communication, We'll work with `select` in Channels.
- `interface` is covered in Chapter 6. Interfaces.
- `struct` is used for abstract data types. We'll work with `struct` in Chapter 5. Beyond the basics.
- `type` is also covered in Chapter 5. Beyond the basics.

Control Structures

There are only a few control structures in Go. To write loops we use the `for` keyword, and there is a `switch` and of course an `if`. When working with channels

`select` will be used (see Channels). Parentheses are not required around the condition, and the body must *always* be brace-delimited.

If-Else

In Go an `if` looks like this:

```
if x > 0 {
    return y
} else {
    return x
}
```

Since `if` and `switch` accept an initialization statement, it's common to see one used to set up a (local) variable.

```
if err := SomeFunction(); err == nil {
    // do something
} else {
    return err
}
```

It is idomatic in Go to omit the `else` when the `if` statement's body has a `break`, `continue`, `return` or, `goto`, so the above code would be better written as:

```
if err := SomeFunction(); err != nil {
    return err
}
// do something
```

The opening brace on the first line must be positioned on the same line as the `if` statement. There is no arguing about this, because this is what `gofmt` outputs.

Goto

Go has a `goto` statement - use it wisely. With `goto` you jump to a label which must be defined within the current function. For instance, a loop in disguise:

```
func myfunc() {
    i := 0
Here:
    fmt.Println(i)
    i++
    goto Here
}
```

The string `Here:` indicates a label. A label does not need to start with a capital letter and is case sensitive.

For

The Go `for` loop has three forms, only one of which has semicolons:

- `for init; condition; post { }` - a loop using the syntax borrowed from C;
- `for condition { }` - a while loop, and;
- `for { }` - an endless loop.

Short declarations make it easy to declare the index variable right in the loop.

```
sum := 0
for i := 0; i < 10; i++ {
    sum = sum + i
}
```

Note that the variable `i` ceases to exist after the loop.

Break and Continue

With `break` you can quit loops early. By itself, `break` breaks the current loop.

```
for i := 0; i < 10; i++ {
    if i > 5 {
    break  1
    }
    fmt.Println(i)  2
}
```

Here we `break` the current loop 1 , and don't continue with the `fmt.Println(i)` statement 2 . So we only print 0 to 5. With loops within loop you can specify a label after `break` to identify *which* loop to stop:

```
J:  for j := 0; j < 5; j++ {  1
        for i := 0; i < 10; i++ {
            if i > 5 {
                break J  2
            }
            fmt.Println(i)
        }
    }
```

Here we define a label "J" 1, preceding the `for`-loop there. When we use `break J` 2, we don't break the inner loop but the "J" loop.

With `continue` you begin the next iteration of the loop, skipping any remaining code. In the same way as `break`, `continue` also accepts a label.

Range

The keyword `range` can be used for loops. It can loop over slices, arrays, strings, maps and channels (see Channels). `range` is an iterator that, when called, returns the next key-value pair from the "thing" it loops over. Depending on what that is, `range` returns different things.

When looping over a slice or array, `range` returns the index in the slice as the key and value belonging to that index. Consider this code:

```
list := []string{"a", "b", "c", "d", "e", "f"}
for k, v := range list {
    // do something with k and v
}
```

First we create a slice of strings. Then we use `range` to loop over them. With each iteration, `range` will return the index as an `int` and the key as a `string`. It will start with 0 and "a", so `k` will be 0 through 5, and v will be "a" through "f".

You can also use `range` on strings directly. Then it will break out the individual Unicode characters ^[In the UTF-8 world characters are sometimes called *runes* Mostly, when people talk about characters, they mean 8 bit characters. As UTF-8 characters may be up to 32 bits the word rune is used. In this case the type of `char` is `rune`. and their start position, by parsing the UTF-8. The loop:

```
for pos, char := range "Gő!" {
    fmt.Printf("character '%c' starts at byte position %d\n", cha
}
```

prints

```
character 'G' starts at byte position 0
character 'ő' starts at byte position 1
character '!' starts at byte position 3
```

Note that ő took 2 bytes, so '!' starts at byte 3.

Switch

Go's `switch` is very flexible; you can match on much more than just integers. The cases are evaluated top to bottom until a match is found, and if the `switch` has no expression it switches on `true`. It's therefore possible – and idiomatic – to write an `if-else-if-else` chain as a `switch`.

```
// Convert hexadecimal character to an int value
switch {  1
case '0' <= c && c <= '9':  2
    return c - '0'  3
case 'a' <= c && c <= 'f':  4
    return c - 'a' + 10
case 'A' <= c && c <= 'F':  5
    return c - 'A' + 10
}
return 0
```

A `switch` without a condition is the same as `switch true` 1 . We list the different cases. Each `case` statement has a condition that is either true of false. Here 2 we check if `c` is a number. If `c` is a number we return its value 3 . Check if `c` falls between "a" and "f" 4 . For an "a" we return 10, for "b" we return 11, etc. We also do the same 5 thing for "A" to "F".

There is no automatic fall through, you can use `fallthrough` for that.

```
switch i {
    case 0:  fallthrough
    case 1:  1
        f()
    default:
        g()  2
```

`f()` can be called when `i == 0` 1 . With `default` you can specify an action when none of the other cases match. Here `g()` is called when `i` is not 0 or 1 2 . We could rewrite the above example as:

```
switch i {
    case 0, 1:  1
        f()
    default:
        g()
```

You can list cases on one line 1 , separated by commas.

Built-in Functions

A few functions are predefined, meaning you *don't* have to include any package to get access to them. Table 3 lists them all.[6]

`close`	`new`	`panic`	`complex`
`delete`	`make`	`recover`	`real`
`len`	`append`	`print`	`imag`
`cap`	`copy`	`println`	

Table 3. Pre-defined functions in Go.

These built-in functions are documented in the `builtin` pseudo package that is included in recent Go releases. Let's go over these functions briefly.

`close`
: is used in channel communication. It closes a channel. We'll learn more about this in Channels.

`delete`
: is used for deleting entries in maps.

`len and cap`
: are used on a number of different types, `len` is used to return the lengths of strings, maps, slices, and arrays. In the next section Arrays we'll look at slices, arrays and the function `cap`.

`new`
: is used for allocating memory for user defined data types. See Allocation with new.

`make`
: is used for allocating memory for built-in types (maps, slices, and channels). See Allocation with make.

`copy, append`
: `copy` is for copying slices. And `append` is for concatenating slices. See Slices in this chapter.

`panic, recover`
: are used for an *exception* mechanism. See Panic and recovering for more.

`print, println`
: are low level printing functions that can be used without reverting to the `fmt` package. These are mainly used for debugging. built-in,println)

`complex, real, imag`
: all deal with complex numbers. We will not use complex numbers in this book.

Arrays, Slices, and Maps

To store multiple values in a list, you can use arrays, or their more flexible cousin: slices. A dictionary or hash type is also available. It is called a `map` in Go.

Arrays

An array is defined by: `[n]<type>`, where n is the length of the array and `<type>` is the stuff you want to store. To assign or index an element in the array, you use square brackets:

```
var arr [10]int
arr[0] = 42
arr[1] = 13
fmt.Printf("The first element is %d\n", arr[0])
```

Array types like `var arr [10]int` have a fixed size. The size is *part* of the type. They can't grow, because then they would have a different type. Also arrays are values: Assigning one array to another *copies* all the elements. In particular, if you pass an array to a function it will receive a copy of the array, not a pointer to it.

To declare an array you can use the following: `var a [3]int`. To initialize it to something other than zero, use a *composite literal* `a := [3]int{1, 2, 3}`. This can be shortened to `a := [...]int{1, 2, 3}`, where Go counts the elements automatically.

> *A composite literal allows you to assign a value directly to an array, slice, or map. See Constructors and composite literals for more information.*

When declaring arrays you *always* have to type something in between the square brackets, either a number or three dots (`...`), when using a composite literal. When using multidimensional arrays, you can use the following syntax: `a := [2][2]int{ {1,2}, {3,4} }`. Now that you know about arrays you will be delighted to learn that you will almost never use them in Go, because there is something much more flexible: slices.

Slices

A slice is similar to an array, but it can grow when new elements are added. A slice always refers to an underlying array. What makes slices different from arrays is that a slice is a pointer *to* an array; slices are reference types.

> *Reference types are created with `make`. We detail this further in Chapter 5.*

Beyond the basics.

That means that if you assign one slice to another, both refer to the *same* underlying array. For instance, if a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array. With: `slice := make([]int, 10)`, you create a slice which can hold ten elements. Note that the underlying array isn't specified. A slice is always coupled to an array that has a fixed size. For slices we define a capacity and a length The image below shows the creation of an array, then the creation of a slice. First we create an array of m elements of the type `int`: `var array[m]int` .

Next, we create a slice from this array: `slice := array[:n]` . And now we have:

- `len(slice) == n`
- `cap(slice) == m`
- `len(array) == cap(array) == m`

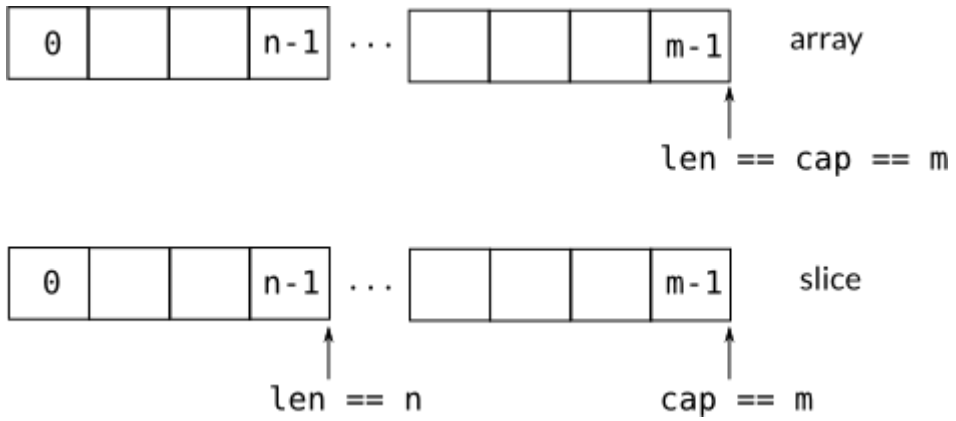

Figure 1. An array versus a slice.

Given an array, or another slice, a new slice is created via `a[n:m]`. This creates a new slice which refers to the variable `a`, starts at index `n`, and ends before index `m`. It has length `n - m`.

```
a := [...]int{1, 2, 3, 4, 5}  1
s1 := a[2:4]  2
s2 := a[1:5]  3
s3 := a[:]  4
s4 := a[:4]  5
s5 := s2[:]  6
s6 := a[2:4:5]  7
```

First we define 1 an array with five elements, from index 0 to 4. From this we create 2 a slice with the elements from index 2 to 3, this slices contains: `3, 4`.

Then we we create another slice 3 from `a`: with the elements from index 1 to 4, this contains: `2, 3, 4, 5`. With `a[:]` 4 we create a slice with all the elements in the array. This is a shorthand for: `a[0:len(a)]`. And with `a[:4]` 5 we create a slice with the elements from index 0 to 3, this is short for: `a[0:4]`, and gives us a slices that contains: `1, 2, 3, 4`. With `s2[:]` we create a slice from the slice `s2` 6, note that `s5` still refers to the array `a`. Finally, we create a slice with the elements from index 3 to 3 *and* also set the cap to 4 7.

When working with slices you can overrun the bounds, consider this code.

```
package main

func main() {
    var array [100]int    1
    slice := array[0:99]  2

    slice[98] = 1  3
    slice[99] = 2  4
}
```

At 1 we create an array with a 100 elements, indexed from 0 to 99. Then at 2 we create a slice that has index 0 to 98. We assign 1 to the 99th element 3 of the slice. This works as expected. But at 4 we dare to do the impossible, and and try to allocate something beyond the length of the slice and we are greeted with a *runtime* error: `Error: "throw: index out of range".`

If you want to extend a slice, there are a couple of built-in functions that make life easier: `append` and `copy`. The append function appends zero or more values to a slice and returns the result: a slice with the same type as the original. If the original slice isn't big enough to fit the added values, append will allocate a new slice that is big enough. So the slice returned by append may refer to a different underlying array than the original slice does. Here's an example:

```
s0 := []int{0, 0}
s1 := append(s0, 2)  1
s2 := append(s1, 3, 5, 7)  2
s3 := append(s2, s0...)  3
```

At 1 we append a single element, making `s1` equal to `[]int{0, 0, 2}`. At 2 we append multiple elements, making `s2` equal to `[]int{0, 0, 2, 3, 5, 7}`. And at 3 we append a slice, giving us `s3` equal to `[]int{0, 0, 2, 3, 5, 7, 0, 0}`. Note the three dots used after `s0...`! This is needed make it clear explicit that you're appending another slice, instead of a single value.

The copy function copies slice elements from a source to a destination, and returns the number of elements it copied. This number is the minimum of the length of the source and the length of the destination. For example:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
n1 := copy(s, a[0:]) 1
n2 := copy(s, s[2:]) 2
```

After 1, `n1` is 6, and `s` is `[]int{0, 1, 2, 3, 4, 5}`. And after 2, `n2` is 4, and `s` is `[]int{2, 3, 4, 5, 4, 5}`.

Maps

Many other languages have a type similar to maps built-in. For instance, Perl has hashes, Python has its dictionaries, and C++ also has maps (as part of the libraries). In Go we have the `map` type. A `map` can be thought of as an array indexed by strings (in its most simple form).

```
monthdays := map[string]int{
    "Jan": 31, "Feb": 28, "Mar": 31,
    "Apr": 30, "May": 31, "Jun": 30,
    "Jul": 31, "Aug": 31, "Sep": 30,
    "Oct": 31, "Nov": 30, "Dec": 31, 1
}
```

The general syntax for defining a map is `map[<from type>]<to type>`. Here, we define a map that converts from a `string` (month abbreviation) to an `int` (number of days in that month). Note that the trailing comma at 1 is *required*.

Use `make` when only declaring a map: `monthdays := make(map[string]int)`. A map is a reference type.

For indexing ("searching") the map, we use square brackets. For example, suppose we want to print the number of days in December:
`fmt.Printf("%d\n", monthdays["Dec"])`

If you are looping over an array, slice, string, or map a, `range` clause will help you again, it returns the key and corresponding value with each invocation.

```
year := 0
for _, days := range monthdays 1
    year += days
}
fmt.Printf("Numbers of days in a year: %d\n", year)
```

At 1 we use the underscore to ignore (assign to nothing) the key returned by `range`. We are only interested in the values from `monthdays`.

To add elements to the map, you would add new month with: `monthdays["Undecim"] = 30`. If you use a key that already exists, the value will be silently overwritten: `monthdays["Feb"] = 29`. To test for existence you would use the following: `value, present := monthdays["Jan"]`. If the key "Jan" exists, `present` will be true. It's more Go like to name `present` "ok", and use: `v, ok := monthdays["Jan"]`. In Go we call this the "comma ok" form.

You can remove elements from the `map`: `delete(monthdays, "Mar")` [7]. In general the syntax `delete(m, x)` will delete the map entry retrieved by the expression `m[x]`.

Exercises

For-loop

1. Create a loop with the `for` construct. Make it loop 10 times and print out the loop counter with the `fmt` package.

2. Rewrite the loop from 1 to use `goto`. The keyword `for` may not be used.

3. Rewrite the loop again so that it fills an array and then prints that array to the screen.

▷ Answer

Average

1. Write code to calculate the average of a `float64` slice. In a later exercise you will make it into a function.

▷ Answer

FizzBuzz

1. Solve this problem, called the Fizz-Buzz [fizzbuzz] problem:

Write a program that prints the numbers from 1 to 100. But for multiples of three print, "Fizz" instead of the number, and for multiples of five, print "Buzz". For numbers which are multiples of both three and five, print "FizzBuzz".

▷ Answer

Chapter 3. Functions

> *I'm always delighted by the light touch and stillness of early programming languages. Not much text; a lot gets done. Old programs read like quiet conversations between a well-spoken research worker and a well- studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication bought such noise?*
>
> Richard P. Gabriel

Functions are the basic building blocks of Go programs; all interesting stuff happens in them.

Here is an example of how you can declare a function:

```
type mytype int
func (p mytype) funcname(q int) (r,s int) { return 0,0 }
1          2          3       4          5           6
```

To declare a function, you use the `func` keyword 1. You can optionally bind 2 to a specific type called receiver (a function with a receiver is usually called a method). This will be explored in Chapter 6. Interfaces. Next 3 you write the name of your function. Here 4 we define that the variable `q` of type `int` is the input parameter. Parameters are passed *pass-by-value*. The variables `r` and `s` 5 are the *named return parameters* (((functions, named return parameters))) for this function. Functions in Go can have multiple return values. This is very useful to return a value *and* error. This removes the need for in-band error returns (such as -1 for `EOF`) and modifying an argument. If you want the return parameters not to be named you only give the types: `(int, int)`. If you have only one value to return you may omit the parentheses. If your function is a subroutine and does not have anything to return you may omit this entirely. Finally, we have the body

6 of the function. Note that `return` is a statement so the braces around the parameter(s) are optional.

As said the return or result parameters of a Go function can be given names and used as regular variables, just like the incoming parameters. When named, they are initialized to the zero values for their types when the function begins. If the function executes a `return` statement with no arguments, the current values of the result parameters are returned. Using these features enables you (again) to do more with less code.[8]

The names are not mandatory but they can make code shorter and clearer: *they are documentation*. However don't overuse this feature, especially in longer functions where it might not be immediately apparent what is returned.

Functions can be declared in any order you wish. The compiler scans the entire file before execution, so function prototyping is a thing of the past in Go. Go does not allow nested functions, but you can work around this with anonymous functions. See the Section Functions as values in this chapter. Recursive functions work just as in other languages:

```
func rec(i int) {
    if i == 10 {  1
        return
    }
    rec(i+1)  2
    fmt.Printf("%d ", i)
}
```

Here 2 we call the same function again, `rec` returns when `i` has the value 10, this is checked on the second line 1. This function prints: `9 8 7 6 5 4 3 2 1 0`, when called as `rec(0)`.

Scope

Variables declared outside any functions are *global* in Go, those defined in functions are *local* to those functions. If names overlap - a local variable is declared with the same name as a global one - the local variable hides the global one when the current function is executed.

In the following example we call `g()` from `f()`:

```
package main

var a int  1
```

```
func main() {
    a = 5
    print(a)
    f()
}

func f() {
    a := 6 2
    print(a)
    g()
}

func g() {
    print(a)
}
```

Here 1, we declare a to be a global variable of type `int`. Then in the `main` function we give the *global* a the value of 5, after printing it we call the function `f`. Then here 2, `a := 6`, we create a *new, local* variable also called a. This new a gets the value of 6, which we then print. Then we call g, which uses the *global* a again and prints a's value set in `main`. Thus the output will be: `565`. A *local* variable is *only* valid when we are executing the function in which it is defined. Note that the `:=` used in line 12 is sometimes hard to spot so it is generally advised *not* to use the same name for global and local variables.

Functions as values

As with almost everything in Go, functions are also *just* values. They can be assigned to variables as follows:

```
import "fmt"

func main() {
    a := func() { 1
        fmt.Println("Hello")
    } 2
    a() 3
}
```

a is defined as an anonymous (nameless) function 1. Note the lack of parentheses `()` after a. If there were, that would be to *call* some function with the name a before we have defined what a is. Once a is defined, then we can *call* it, 3 .

Functions–as–values may be used in other places, for example maps. Here we convert from integers to functions:

```
var xs = map[int]func() int{
    1: func() int { return 10 },
    2: func() int { return 20 },
    3: func() int { return 30 },
}
```

Note that the final comma on second to last line is *mandatory*.

Or you can write a function that takes a function as its parameter, for example a `Map` function that works on `int` slices. This is left as an exercise for the reader; see the exercise Map function.

Callbacks

Because functions are values they are easy to pass to functions, from where they can be used as callbacks. First define a function that does "something" with an integer value:

```
func printit(x int) {
    fmt.Printf("%v\n", x)
}
```

This function does not return a value and just prints its argument. The *signature* of this function is: `func printit(int)`, or without the function name: `func(int)`. To create a new function that uses this one as a callback we need to use this signature:

```
func callback(y int, f func(int)) {
    f(y)
}
```

Here we create a new function that takes two parameters: `y int`, i.e. just an `int` and `f func(int)`, i.e. a function that takes an int and returns nothing. The parameter `f` is the variable holding that function. It can be used as any other function, and we execute the function on line 2 with the parameter `y`: `f(y)`

Deferred Code

Suppose you have a function in which you open a file and perform various writes and reads on it. In such a function there are often spots where you want to return early. If you do that, you will need to close the file descriptor you are working on. This often leads to the following code:

```
func ReadWrite() bool {
    file.Open("file")
    // Do your thing
    if failureX {
        file.Close() 1
        return false
    }

    if failureY {
        file.Close() 1
        return false
    }
    file.Close() 1
    return true  2
}
```

Note that we repeat a lot of code here; you can see the that `file.Close()` is called at 1. To overcome this, Go has the `defer` keyword. After `defer` you specify a function which is called just *before* 2 the current function exits.

With `defer` we can rewrite the above code as follows. It makes the function more readable and it puts the `Close` *right next* to the `Open`.

```
func ReadWrite() bool {
    file.Open("filename")
    defer file.Close() 1
    // Do your thing
    if failureX {
        return false 2
    }
    if failureY {
        return false 2
    }
    return true 2
}
```

At 1 `file.Close()` is added to the defer list. `Close` is now done automatically at 2. This makes the function shorter and more readable. It puts the `Close` right next to the `Open`.

You can put multiple functions on the “defer list”, like this example from

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

Deferred functions are executed in LIFO order, so the above code prints: `4 3 2 1 0`.

With `defer` you can even change return values, provided that you are using named result parameters and a function literal [9], i.e:

```
defer func() {/* ... */}()
```

Here we use a function without a name and specify the body of the function inline, basically we're creating a nameless function on the spot. The final parentheses are needed because `defer` needs a function call, not a function value. If our anonymous function would take an parameter it would be easier to see why we need the parentheses:

```
defer func(x int) {/* ... */}(5)
```

In this (unnamed) function you can access any named return parameter:

```
func f() (ret int)
    defer func() {  1
        ret++
    }()
    return 0
}
```

Here 1 we specify our function, the named return value `ret` is initialized with zero. The nameless function in the defer increments the value of `ret` with 1. The `return 0` on line 5 *will not be the returned value*, because of `defer`. The function `f` will return 1!

Variadic Parameter

Functions that take a variable number of parameters are known as variadic functions. To declare a function as variadic, do something like this:

```
func myfunc(arg ...int) {}
```

The `arg ...int` instructs Go to see this as a function that takes a variable number of arguments. Note that these arguments all have to have the type `int`. In the body of your function the variable `arg` is a slice of ints:

```
for _, n := range arg {
    fmt.Printf("And the number is: %d\n", n)
```

```
}
```

We range over the arguments on the first line. We are not interested in the index as returned by `range`, hence the use of the underscore there. In the body of the `range` we just print the parameters we were given.

If you don't specify the type of the variadic argument it defaults to the empty interface `interface{}` (see Chapter Chapter 6. Interfaces).

Suppose we have another variadic function called `myfunc2`, the following example shows how to pass variadic arguments to it:

```
func myfunc(arg ...int) {
    myfunc2(arg...)
    myfunc2(arg[:2]...)
}
```

With `myfunc2(arg...)` we pass all the parameters to `myfunc2`, but because the variadic parameters is just a slice, we can use some slice tricks as well.

Panic and recovering

Go does not have an exception mechanism: you cannot throw exceptions. Instead it uses a panic-and-recover mechanism. It is worth remembering that you should use this as a last resort, your code will not look, or be, better if it is littered with panics. It's a powerful tool: use it wisely. So, how do you use it? In the words of the Go Authors [go_blog_panic]:

Panic
: is a built-in function that stops the ordinary flow of control and begins panicking. When the function `F` calls `panic`, execution of `F` stops, any deferred functions in `F` are executed normally, and then `F` returns to its caller. To the caller, `F` then behaves like a call to `panic`. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking `panic` directly. They can also be caused by *runtime errors*, such as out-of-bounds array accesses.

Recover
: is a built-in function that regains control of a panicking goroutine. Recover is *only* useful inside *deferred* functions. During normal execution, a call to `recover` will return `nil` and have no other effect. If the current goroutine is panicking, a call to `recover` will capture the value given to `panic` and resume normal execution.

This function checks if the function it gets as argument will panic when it is executed[10]:

```
func Panic(f func()) (b bool) {  1
    defer func() {  2
        if x := recover(); x != nil {
            b = true
        }
    }()
    f()  3
    return  4
}
```

We define a new function `Panic` 1 that takes a function as an argument (see Functions as values). It returns true if `f` panics when run, else false. We then 2 define a `defer` function that utilizes `recover`. If the current goroutine panics, this defer function will notice that. If `recover()` returns non-`nil` we set `b` to true. At 3 Execute the function we received as the argument. And finally 4 we return the value of `b`. Because `b` is a named return parameter.

The following code fragment, shows how we can use this function:

```
func panicy() {
    var a []int
    a[3] = 5
}

func main() {
    fmt.Println(Panic(panicy))
}
```

On line 3 the `a[3] = 5` triggers a *runtime* out of bounds error which results in a panic. Thus this program will print `true`. If we change line 2: `var a []int` to `var a [4]int` the function `panicy` does not panic anymore. Why?

Exercises

Average

1. Write a function that calculates the average of a `float64` slice.

▷ Answer

Bubble sort

1. Write a function that performs a bubble sort on a slice of ints. From [bubblesort]:

> *It works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list.*

It also gives an example in pseudo code:

```
procedure bubbleSort( A : list of sortable items )
  do
    swapped = false
    for each i in 1 to length(A) - 1 inclusive do:
      if A[i-1] > A[i] then
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  while swapped
end procedure
```

▷ Answer

For-loop II

1. Take what you did in exercise to write the for loop and extend it a bit. Put the body of the for loop - the `fmt.Printf` - in a separate function.

▷ Answer

Fibonacci

1. The Fibonacci sequence starts as follows: $1, 1, 2, 3, 5, 8, 13, \ldots$ Or in mathematical terms: $x_1 = 1; x_2 = 1; x_n = x_{n-1} + x_{n-2} \quad \forall n > 2$.

   Write a function that takes an `int` value and gives that many terms of the Fibonacci sequence.

▷ Answer

Var args

1. Write a function that takes a variable number of ints and print each integer on a separate line.

▷ Answer

Functions that return functions

1. Write a function that returns a function that performs a $+2$ on integers. Name the function `plusTwo`. You should then be able do the following:

   ```
   p := plusTwo()
   fmt.Printf("%v\n", p(2))
   ```

   Which should print 4. See Callbacks.

2. Generalize the function from above and create a `plusX(x)` which returns functions that add `x` to an integer.

▷ Answer

Maximum

1. Write a function that finds the maximum value in an `int` slice (`[]int`).

▷ Answer

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:

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

1. Write a simple `map()`-function in Go. It is sufficient for this function only to work for ints.

▷ Answer

Stack

1. Create a simple stack which can hold a fixed number of ints. It does not have to grow beyond this limit. Define `push` – put something on the stack – and `pop` – retrieve something from the stack – functions. The stack should be a LIFO (last in, first out) stack.

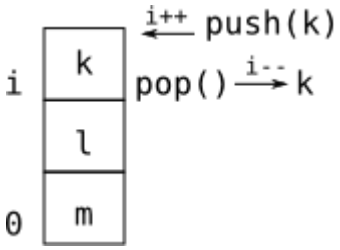

Figure 2. A stack.

2. Write a `String` method which converts the stack to a string representation. The stack in the figure could be represented as: `[0:m] [1:l] [2:k]` .

▷ **Answer**

Chapter 4. Packages

"^"

ANSWER TO WHETHER THERE IS A BIT WISE NEGATION OPERATOR – KEN THOMPSON

A package is a collection of functions and data.

You declare a package with the **`package`** keyword. The filename does not have to match the package name. The convention for package names is to use lowercase characters. Go packages may consist of multiple files, but they share the **`package <name>`** line. Let's define a package **`even`** in the file **`even.go`**.

```
package even

func Even(i int) bool {
    return i%2 == 0
}

func odd(i int) bool {
    return i%2 == 1
}
```

Here 1 we start a new namespace: "even". The function `Even` 2 starts with a capital letter. This means the function is *exported*, and may be used outside our package (more on that later). The function `odd` 3 does not start with a capital letter, so it is a *private* function.

Now we just need to build the package. We create a directory under `$GOPATH`, and copy `even.go` there (see Compiling and Running Code in Chapter 2. Basics).

```
% mkdir $GOPATH/src/even
% cp even.go $GOPATH/src/even
% go build
% go install
```

Now we can use the package in our own program `myeven.go`:

```
package main

import ( 1
    "even" 2
    "fmt"  3
)

func main() {
    i := 5
    fmt.Printf("Is %d even? %v\n", i, even.Even(i)) 4
}
```

Import 1 the following packages. The *local* package `even` is imported here 2. This 3 imports the official `fmt` package. And now we use 4 the function from the `even` package. The syntax for accessing a function from a package is `<package>.FunctionName()`. And finally we can build our program.

```
% go build myeven.go
% ./myeven
Is 5 even? false
```

If we change our `myeven.go` at 4 to use the unexported function `even.odd`: `fmt.Printf("Is %d even? %v\n", i, even.odd(i))` We get an error when compiling, because we are trying to use a *private* function:

```
myeven.go: cannot refer to unexported name even.odd
```

Note that the "starts with capital → exported", "starts with lower-case → private" rule also extends to other names (new types, global variables) defined in

the package. Note that the term “capital” is not limited to US-ASCII – it extends to all bicameral alphabets (Latin, Greek, Cyrillic, Armenian and Coptic).

Identifiers

The Go standard library names some function with the old (Unix) names while others are in CamelCase. The convention is to leave well-known legacy not-quite-words alone rather than try to figure out where the capital letters go: `Atoi`, `Getwd`, `Chmod`. CamelCasing works best when you have whole words to work with: `ReadFile`, `NewWriter`, `MakeSlice`. The convention in Go is to use CamelCase rather than underscores to write multi-word names.

As we did above in our `myeven` program, accessing content from an imported (with `import`) package is done with using the package’s name and then a dot. After `import "bytes"` the importing program can talk about `bytes.Buffer`. A package name should be good, short, concise and evocative. The convention in Go is that package names are lowercase, single word names.

The package name used in the `import` statement is the default name used. But if the need arises (two different packages with the same name for instance), you can override this default: `import bar "bytes"` The function `Buffer` is now accessed as `bar.Buffer`.

Another convention is that the package name is the base name of its source directory; the package in `src/compress/gzip` is imported as `compress/gzip` but has name `gzip`, not `compress/gzip`.

It is important to avoid stuttering when naming things. For instance, the buffered reader type in the `bufio` package is called `Reader`, not `BufReader`, because users see it as `bufio.Reader`, which is a clear, concise name.

Similarly, the function to make new instances of `ring.Ring` (package `container/ring`), would normally be called `NewRing`, but since `Ring` is the only type exported by the package, and since the package is called `ring`, it’s called just `New`. Clients of the package see that as `ring.New`. Use the package structure to help you choose good names.

Another short example is `once.Do` (see package `sync`); `once.Do(setup)` reads well and would not be improved by writing `once.DoOrWaitUntilDone(setup)`. Long names don’t automatically make things more readable.

Documenting packages

When we created our `even` package, we skipped over an important item: documentation. Each package should have a *package comment*, a block

comment preceding the `package` clause. In our case we should extend the beginning of the package, with:

```
// The even package implements a fast function for detecting if a
// is even or not.
package even
```

When running `go doc` this will show up at the top of the page. When a package consists of multiple files the package comment should only appear in one file. A common convention (in really big packages) is to have a separate `doc.go` that only holds the package comment. Here is a snippet from the official `regexp` package:

```
/*
    The regexp package implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
*/
package regexp
```

Each defined (and exported) function should have a small line of text documenting the behavior of the function. Again to extend our `even` package:

```
// Even returns true of i is even. Otherwise false is returned.
func Even(i int) bool {
```

And even though `odd` is not exported, it's good form to document it as well.

```
// odd is the opposite of Even.
func odd(i int) bool {
```

Testing packages

In Go it is customary to write (unit) tests for your package. Writing tests involves the `testing` package and the program `go test`. Both have excellent documentation.

The `go test` program runs all the test functions. Without any defined tests for our `even` package, `go test` yields:

```
% go test
?       even    [no test files]
```

Let us fix this by defining a test in a test file. Test files reside in the package directory and are named `*_test.go`. Those test files are just like other Go programs, but `go test` will only execute the test functions. Each test function has the same signature and its name should start with `Test`: `func TestXxx(t *testing.T)` .

When writing test you will need to tell `go test` whether a test was successful or not. A successful test function just returns. When the test fails you can signal this with the following functions. These are the most important ones (see `go doc testing` or `go help testfunc` for more):

- `func (t *T) Fail()`, `Fail` marks the test function as having failed but continues execution.
- `func (t *T) FailNow()`, `FailNow` marks the test function as having failed and stops its execution. Any remaining tests in this file are skipped, and execution continues with the next test.
- `func (t *T) Log(args ...interface{})`, `Log` formats its arguments using default formatting, analogous to `Print()`, and records the text in the error log.
- `func (t *T) Fatal(args ...interface{})`, `Fatal` is equivalent to `Log()` followed by `FailNow()`.

Putting all this together we can write our test. First we pick a name: `even_test.go`. Then we add the following contents:

```
package even  1

import "testing"  2

func TestEven(t *testing.T) {  3
    if !Even(2) {
        t.Log("2 should be even!")
        t.Fail()
    }
}
```

A test file belongs to the current 1 package. This is not only convenient, but also allows tests of unexported functions and structures. We then 2 import the `testing` package. And finally the test we want to execute. The code here 3

should hold no surprises: we check if the `Even` function works OK. And now, the moment we have been waiting for executing the test.

```
% go test
ok      even    0.001s
```

Our test ran and reported `ok`. Success! If we redefine our test function, we can see the result of a failed test:

```
// Entering the twilight zone
func TestEven(t *testing.T) {
    if Even(2) {
        t.Log("2 should be odd!")
        t.Fail()
    }
}
```

We now get:

```
FAIL    even    0.004s
--- FAIL: TestEven (0.00 seconds)
    2 should be odd!
FAIL
```

And you can act accordingly (by fixing the test for instance).

Writing new packages should go hand in hand with writing (some) documentation and test functions. It will make your code better and it shows that you really put in the effort.

The Go test suite also allows you to incorporate example functions which serve as documentation *and* as tests. These functions need to start with `Example`.

```
func ExampleEven() {
    if Even(2) {
        fmt.Printf("Is even\n")
    }
    // Output: 1
    // Is even
}
```

Those last two comments lines 1 are part of the example, `go test` uses those to check the *generated* output with the text in the comments. If there is a mismatch the test fails.

Useful packages

The standard libary of Go includes a huge number of packages. It is very enlightening to browse the `$GOROOT/src` directory and look at the packages. We cannot comment on each package, but the following are worth a mention: [11]

`fmt`

Package `fmt` implements formatted I/O with functions analogous to C's `printf` and `scanf`. The format verbs are derived from C's but are simpler. Some verbs (%-sequences) that can be used:

- *%v*, the value in a default format. when printing structs, the plus flag (%+v) adds field names.
- *%#v*, a Go-syntax representation of the value.
- *%T*, a Go-syntax representation of the type of the value.

`io`

This package provides basic interfaces to I/O primitives. Its primary job is to wrap existing implementations of such primitives, such as those in package os, into shared public interfaces that abstract the functionality, plus some other related primitives.

`bufio`

This package implements buffered I/O. It wraps an `io.Reader` or `io.Writer` object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.

`sort`

The `sort` package provides primitives for sorting arrays and user-defined collections.

`strconv`

The `strconv` package implements conversions to and from string representations of basic data types.

`os`

The `os` package provides a platform-independent interface to operating system functionality. The design is Unix-like.

`sync`

The package `sync` provides basic synchronization primitives such as mutual exclusion locks.

`flag`

The `flag` package implements command-line flag parsing.

`encoding/json`
: The `encoding/json` package implements encoding and decoding of JSON objects as defined in RFC 4627 [RFC4627].

`html/template`
: Data-driven templates for generating textual output such as HTML.

  Templates are executed by applying them to a data structure. Annotations in the template refer to elements of the data structure (typically a field of a struct or a key in a map) to control execution and derive values to be displayed. The template walks the structure as it executes and the "cursor" @ represents the value at the current location in the structure.

`net/http`
: The `net/http` package implements parsing of HTTP requests, replies, and URLs and provides an extensible HTTP server and a basic HTTP client.

`unsafe`
: The `unsafe` package contains operations that step around the type safety of Go programs. Normally you don't need this package, but it is worth mentioning that *unsafe* Go programs are possible.

`reflect`
: The `reflect` package implements run-time reflection, allowing a program to manipulate objects with arbitrary types. The typical use is to take a value with static type `interface{}` and extract its dynamic type information by calling `TypeOf`, which returns an object with interface type `Type`. See Chapter 6. Interfaces, Section Introspection and reflection.

`os/exec`
: The `os/exec` package runs external commands.

Exercises

Stack as package

1. See the Stack exercise. In this exercise we want to create a separate package for that code. Create a proper package for your stack implementation, `Push`, `Pop` and the `Stack` type need to be exported.

2. Write a simple unit test for this package. You should at least test that a `Pop` works after a `Push`.

▷ **Answer**

Calculator

1. Create a reverse polish calculator. Use your stack package.

▷ **Answer**

Chapter 5. Beyond the basics

> *Go has pointers but not pointer arithmetic. You cannot use a pointer variable to walk through the bytes of a string.*
>
> Go For C++ Programmers – Go Authors

In this chapter we delve deeper into the language.

Go has pointers. There is however no pointer arithmetic, so they act more like references than pointers that you may know from C. Pointers are useful. Remember that when you call a function in Go, the variables are *pass-by-value*. So, for efficiency and the possibility to modify a passed value *in* functions we have pointers.

You declare a pointer by prefixing the type with an '`*`': `var p *int`. Now `p` is a pointer to an integer value. All newly declared variables are assigned their zero value and pointers are no different. A newly declared pointer, or just a pointer that points to nothing, has a nil-value . In other languages this is often called a NULL pointer in Go it is just `nil`. To make a pointer point to something you can use the address-of operator (`&`), which we demonstrate here:

```
var p *int
fmt.Printf("%v", p)
```

```
var i int          2
p = &i             3

fmt.Printf("%v", p) 4
```

This 1 Prints `nil`. Declare 2 an integer variable `i`. Make `p` point 3 to `i`, i.e. take the address of `i`. And this 4 will print something like `0x7ff96b81c000a`. De-referencing a pointer is done by prefixing the pointer variable with `*`.

As said, there is no pointer arithmetic, so if you write: `*p++`, it is interpreted as `(*p)++`: first reference and then increment the value.

Allocation

Go also has garbage collection, meaning that you don't have to worry about memory deallocation.[12]

To allocate memory Go has two primitives, `new` and `make`. They do different things and apply to different types, which can be confusing, but the rules are simple. The following sections show how to handle allocation in Go and hopefully clarifies the somewhat artificial distinction between `new` and `make` .

Allocation with new

The built-in function `new` is essentially the same as its namesakes in other languages: `new(T)` allocates zeroed storage for a new item of type `T` and returns its address, a value of type `*T`. Or in other words, it returns a pointer to a newly allocated zero value of type `T`. This is important to remember.

The documentation for `bytes.Buffer` states that "the zero value for Buffer is an empty buffer ready to use.". Similarly, `sync.Mutex` does not have an explicit constructor or Init method. Instead, the zero value for a `sync.Mutex` is defined to be an unlocked mutex.

Allocation with make

The built-in function `make(T, args)` serves a purpose different from `new(T)`. It creates slices, maps, and channels *only*, and it returns an initialized (not zero!) value of type `T`, and not a pointer: `*T`. The reason for the distinction is that these three types are, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity; until those items are initialized, the slice is `nil`. For slices, maps, and channels, `make` initializes the internal data structure and prepares the value for use.

For instance, `make([]int, 10, 100)` allocates an array of 100 ints and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. In contrast, `new([]int)` returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a `nil` slice value. These examples illustrate the difference between `new` and `make`.

```
var p *[]int = new([]int)       1
var v  []int = make([]int, 100) 2

var p *[]int = new([]int)       3
*p = make([]int, 100, 100)

v := make([]int, 100)           4
```

Allocates 1 slice structure; rarely useful. `v` 2 refers to a new array of 100 ints. At 3 we make it unnecessarily complex, 4 is more idiomatic.

Remember that `make` applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with `new`.

> ***new*** *allocates;* ***make*** *initializes.*
>
> *The above two paragraphs can be summarized as:*
>
> - *`new(T)` returns `*T` pointing to a zeroed `T`*
> - *`make(T)` returns an initialized `T`*
>
> *And of course* ***`make`*** *is only used for slices, maps and channels.*

Constructors and composite literals

Sometimes the zero value isn't good enough and an initializing constructor is necessary, as in this example taken from the package `os`.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
```

```
    return f
}
```

There's a lot of boiler plate in there. We can simplify it using a *composite literal* , which is an expression that creates a new instance each time it is evaluated.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f    1
}
```

It is OK to return the address of a local variable 1 the storage associated with the variable survives after the function returns.

In fact, taking the address of a composite literal allocates a fresh instance each time it is evaluated, so we can combine these last two lines.[13]

```
return &File{fd, name, nil, 0}
```

The items (called fields) of a composite literal are laid out in order and must all be present. However, by labeling the elements explicitly as field:value pairs, the initializers can appear in any order, with the missing ones left as their respective zero values. Thus we could say

```
return &File{fd: fd, name: name}
```

As a limiting case, if a composite literal contains no fields at all, it creates a zero value for the type. The expressions `new(File)` and `&File{}` are equivalent. In fact the use of `new` is discouraged.

Composite literals can also be created for arrays, slices, and maps, with the field labels being indices or map keys as appropriate. In these examples, the initializations work regardless of the values of `Enone`, and `Einval`, as long as they are distinct:

```
ar := [...]string{Enone: "no error", Einval: "invalid argument"}
sl := []string{Enone: "no error", Einval: "invalid argument"}
ma := map[int]string {Enone: "no error", Einval: "invalid argumen
```

Defining your own types

Of course Go allows you to define new types, it does this with the `type` keyword: `type foo int`

This creates a new type `foo` which acts like an `int`. Creating more sophisticated types is done with the `struct` keyword. An example would be when we want record somebody's name (`string`) and age (`int`) in a single structure and make it a new type:

```
package main

import "fmt"

type NameAge struct {
    name string // Both non exported fields.
    age  int
}

func main() {
    a := new(NameAge)
    a.name = "Pete"
    a.age = 42
    fmt.Printf("%v\n", a)
}
```

Apropos, the output of `fmt.Printf("%v\n", a)` is `&{Pete 42}`

That is nice! Go knows how to print your structure. If you only want to print one, or a few, fields of the structure you'll need to use `.<field name>`. For example to only print the name:

```
fmt.Printf("%s", a.name)
```

More on structure fields

As said each item in a structure is called a field. A struct with no fields: `struct {}`. Or one with four fields:

```
struct {
    x, y int
    A *[]int
    F func()
}
```

If you omit the name for a field, you create an anonymous field (((field, anonymous))), for instance:

```
struct {
    T1        // Field name is T1.
    *T2       // Field name is T2.
    P.T3      // Field name is T3.
    x, y int  // Field names are x and y.
}
```

Note that field names that start with a capital letter are exported, i.e. can be set or read from other packages. Field names that start with a lowercase are private to the current package. The same goes for functions defined in packages, see Chapter 4. Packages for the details.

Methods

If you create functions that work on your newly defined type, you can take two routes:

1. Create a function that takes the type as an argument.

```
func doSomething(n1 *NameAge, n2 int) { /* */ }
```

2. Create a function that works on the type (see *receiver* in Chapter 3. Functions):

```
func (n1 *NameAge) doSomething(n2 int) { /* */ }
```

This is a method call, which can be used as:

```
var n *NameAge
n.doSomething(2)
```

Whether to use a function or method is entirely up to the programmer, but if you want to satisfy an interface (see the next chapter) you must use methods. If no such requirement exists it is a matter of taste whether to use functions or methods.

But keep the following in mind, this is quoted from [go_spec]:

> *If `x` is addressable and `&x`'s method set contains `m`, `x.m()` is shorthand for `(&x).m()`.*

In the above case this means that the following is *not* an error:

```
var n NameAge       // Not a pointer
n.doSomething(2)
```

Here Go will search the method list for `n` of type `NameAge`, come up empty and will then *also* search the method list for the type `*NameAge` and will translate this call to `(&n).doSomething(2)`.

There is a subtle but major difference between the following type declarations. Also see the Section "Type Declarations" [go_spec]. Suppose we have:

```
// A Mutex is a data type with two methods, Lock and Unlock.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock impl. */ }
func (m *Mutex) Unlock()  { /* Unlock impl. */ }
```

We now create two types in two different manners:

- `type NewMutex Mutex`.
- `type PrintableMutex struct{Mutex}`.

`NewMutex` is equal to `Mutex`, but it *does not* have *any* of the methods of `Mutex`. In other words its method set is empty. But `PrintableMutex` *has inherited* the method set from `Mutex`. The Go term for this is *embedding* . In the words of [go_spec]:

> *The method set of `*PrintableMutex` contains the methods `Lock` and `Unlock` bound to its anonymous field `Mutex`.*

Conversions

Sometimes you want to convert a type to another type. This is possible in Go, but there are some rules. For starters, converting from one value to another is done by operators (that look like functions: `byte()`) and not all conversions are allowed.

From	b []byte	i []int	r []rune	s string	f f]
To					
[]byte	·			[]byte(s)	
[]int		·		[]int(s)	
[]rune				[]rune(s)	

From	b []byte	i []int	r []rune	s string	f f]
string	string(b)	string(i)	string(r)	·	
float32					·
int					ir

Table 4. Valid conversions, `float64` works the same as `float32`.

- From a `string` to a slice of bytes or runes.

```
mystring := "hello this is string"
byteslice := []byte(mystring)
```

Converts to a `byte` slice, each `byte` contains the integer value of the corresponding byte in the string. Note that as strings in Go are encoded in UTF-8 some characters in the string may end up in 1, 2, 3 or 4 bytes.

```
runeslice  := []rune(mystring)
```

Converts to an `rune` slice, each `rune` contains a Unicode code point. Every character from the string corresponds to one rune.

- From a slice of bytes or runes to a `string`.

```
b := []byte{'h','e','l','l','o'} // Composite literal.
s := string(b)
i := []rune{257,1024,65}
r := string(i)
```

For numeric values the following conversions are defined:

- Convert to an integer with a specific (bit) length: `uint8(int)`
- From floating point to an integer value: `int(float32)`. This discards the fraction part from the floating point value.
- And the other way around: `float32(int)`.

User defined types and conversions

How can you convert between the types you have defined yourself? We create two types here `Foo` and `Bar`, where `Bar` is an alias for `Foo`:

```
type foo struct { int }  // Anonymous struct field.
type bar foo             // bar is an alias for foo.
```

Then we:

```
var b bar = bar{1} // Declare `b` to be a `bar`.
var f foo = b      // Assign `b` to `f`.
```

Which fails on the last line with: `cannot use b (type bar) as type foo in assignment`

This can be fixed with a conversion: `var f foo = foo(b)`

Note that converting structures that are not identical in their fields is more difficult. Also note that converting `b` to a plain `int` also fails; an integer is not the same as a structure containing an integer.

Exercises

Map function with interfaces

1. Use the answer from the earlier map exercise but now make it generic using interfaces. Make it at least work for ints and strings.

▷ Answer

Pointers

1. Suppose we have defined the following structure:

   ```
   type Person struct {
       name string
       age  int
   }
   ```

   What is the difference between the following two lines?

   ```
   var p1 Person
   p2 := new(Person)
   ```

2. What is the difference between the following two allocations?

   ```
   func Set(t *T) {
       x = t
   ```

```
}
```

and

```
func Set(t T) {
    x= &t
}
```

▷ Answer

Linked List

1. Make use of the package `container/list` to create a (doubly) linked list. Push the values 1, 2 and 4 to the list and then print it.

2. Create your own linked list implementation. And perform the same actions as above.

▷ Answer

Cat

1. Write a program which mimics the Unix program `cat`.

2. Make it support the `-n` flag, where each line is numbered.

3. The solution to the above question given in contains a bug. Can you spot and fix it?

▷ Answer

Method calls

1. Suppose we have the following program. Note the package `container/vector` was once part of Go, but was removed when the `append` built-in was introduced. However, for this question this isn't important. The package implemented a stack-like structure, with push and pop methods.

```
package main


import "container/vector"


func main() {
    k1 := vector.IntVector{}
```

```
    k2 := &vector.IntVector{}
    k3 := new(vector.IntVector)
    k1.Push(2)
    k2.Push(3)
    k3.Push(4)
}
```

What are the types of `k1`, `k2` and `k3`?

2. Now, this program compiles and runs OK. All the `Push` operations work even though the variables are of a different type. The documentation for `Push` says:

> *`func (p *IntVector) Push(x int)` Push appends x to the end of the vector.*

So the receiver has to be of type `*IntVector`, why does the code above (the Push statements) work correctly then?

▷ **Answer**

Chapter 6. Interfaces

> *I have this phobia about having my body penetrated surgically. You know what I mean?*
>
> eXistenZ – Ted Pikul

In Go, the word *interface* is overloaded to mean several different things. Every type has an interface, which is the *set of methods defined* for that type. This bit of code defines a struct type `S` with one field, and defines two methods for `S`. [15]

```
type S struct { i int }
func (p *S) Get() int  { return p.i }
func (p *S) Put(v int) { p.i = v }
```

Figure 3. Defining a struct and methods on it.

You can also define an interface type, which is simply a set of methods. This defines an interface `I` with two methods:

```
type I interface {
    Get() int
    Put(int)
}
```

`S` is a valid *implementation* for interface `I`, because it defines the two methods which `I` requires. Note that this is true even though there is no explicit declaration that `S` implements `I`.

A Go program can use this fact via yet another meaning of interface, which is an interface value:

```
func f(p I) {  1
    fmt.Println(p.Get())  2
    p.Put(1)  3
}
```

At 1 we declare a function that takes an interface type as the argument. Because `p` implements `I`, it *must* have the `Get()` method, which we call at 2. And the same holds true for the `Put()` method at 3. Because `S` implements `I`, we can call the function `f` passing in a pointer to a value of type `S`: `var s S; f(&s)`

The reason we need to take the address of `s`, rather than a value of type `S`, is because we defined the methods on `s` to operate on pointers, see the definition in the code above. This is not a requirement – we could have defined the methods to take values – but then the `Put` method would not work as expected.

The fact that you do not need to declare whether or not a type implements an interface means that Go implements a form of duck typing [duck_typing]. This is not pure duck typing, because when possible the Go compiler will statically check whether the type implements the interface. However, Go does have a purely dynamic aspect, in that you can convert from one interface type to another. In the general case, that conversion is checked at run time. If the conversion is invalid – if the type of the value stored in the existing interface value does not satisfy the interface to which it is being converted – the program will fail with a run time error.

Interfaces in Go are similar to ideas in several other programming languages: pure abstract virtual base classes in C++, typeclasses in Haskell or duck typing in Python. However there is no other language which combines interface values, static type checking, dynamic run time conversion, and no requirement for

explicitly declaring that a type satisfies an interface. The result in Go is powerful, flexible, efficient, and easy to write.

Which is what?

Let's define another type `R` that also implements the interface `I`:

```
type R struct { i int }
func (p *R) Get() int  { return p.i }
func (p *R) Put(v int) { p.i = v }
```

The function `f` can now accept variables of type `R` and `S`.

Suppose you need to know the actual type in the function `f`. In Go you can figure that out by using a type switch.

```
func f(p I) {
    switch t := p.(type) {  1
        case *S:  2
        case *R:  2
        default:  3
    }
}
```

At 1 we use the type switch, note that the `.(type)` syntax is *only* valid within a `switch` statement. We store the value in the variable `t`. The subsequent cases 2 each check for a different *actual* type. And we can even have a `default` 3 clause. It is worth pointing out that both `case R` and `case s` aren't possible, because `p` needs to be a pointer in order to satisfy `i`.

A type switch isn't the only way to discover the type at *run-time*.

```
if t, ok := something.(I); ok {  1
    // ...
}
```

You can also use a "comma, ok" form 1 to see if an interface type implements a specific interface. If `ok` is true, `t` will hold the type of `something`. When you are sure a variable implements an interface you can use: `t := something.(I)` .

Empty interface

Since every type satisfies the empty interface: `interface{}` we can create a generic function which has an empty interface as its argument:

```
func g(something interface{}) int {
    return something.(I).Get()
}
```

The `return something.(I).Get()` is the tricky bit in this function. The value `something` has type `interface{}`, meaning no guarantee of any methods at all: it could contain any type. The `.(I)` is a type assertion which converts `something` to an interface of type `I`. If we have that type we can invoke the `Get()` function. So if we create a new variable of the type `*S`, we can just call `g()`, because `*S` also implements the empty interface.

```
s = new(S)
fmt.Println(g(s));
```

The call to `g` will work fine and will print 0. If we however invoke `g()` with a value that does not implement `I` we have a problem:

```
var i int
fmt.Println(g(i))
```

This compiles, but when we run this we get slammed with: "panic: interface conversion: int is not main.I: missing method Get".

Which is completely true, the built-in type `int` does not have a `Get()` method.

Methods

Methods are functions that have a receiver (see Chapter 3. Functions). You can define methods on any type (except on non-local types, this includes built-in types: the type `int` can not have methods). You can however make a new integer type with its own methods. For example:

```
type Foo int

func (self Foo) Emit() {
    fmt.Printf("%v", self)
}

type Emitter interface {
    Emit()
}
```

Doing this on non-local (types defined in other packages) types yields an error "cannot define new methods on non-local type int".

Methods on interface types

An interface defines a set of methods. A method contains the actual code. In other words, an interface is the definition and the methods are the implementation. So a receiver can not be an interface type, doing so results in a "invalid receiver type …" compiler error. The authoritative word from the language spec [go_spec]:

> *The receiver type must be of the form `T` or `*T` where `T` is a type name. `T` is called the receiver base type or just base type. The base type must not be a pointer or interface type and must be declared in the same package as the method.*

> *Creating a pointer to an interface value is a useless action in Go. It is in fact illegal to create a pointer to an interface value. The release notes for an earlier Go release that made them illegal leave no room for doubt:*
>
> > *The language change is that uses of pointers to interface values no longer automatically de-reference the pointer. A pointer to an interface value is more often a beginner's bug than correct code.*

Interface names

By convention, one-method interfaces are named by the method name plus the *-er* suffix: Read*er*, Writ*er*, Formatt*er* etc.

There are a number of such names and it's productive to honor them and the function names they capture. `Read`, `Write`, `Close`, `Flush`, `String` and so on have canonical signatures and meanings. To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method `String` not `ToString`. [16]

A sorting example

Recall the Bubblesort exercise, where we sorted an array of integers:

```
func bubblesort(n []int) {
    for i := 0; i < len(n)-1; i++ {
        for j := i + 1; j < len(n); j++ {
            if n[j] < n[i] {
                n[i], n[j] = n[j], n[i]
            }
```

```
        }
    }
}
```

A version that sorts strings is identical except for the signature of the function: `func bubblesortString(n []string) { /* ... */ }`. Using this approach would lead to two functions, one for each type. By using interfaces we can make this more generic. Let's create a new function that will sort both strings and integers, something along the lines of this non-working example:

```
func sort(i []interface{}) {   1
    switch i.(type) {          2
    case string:               3
        // ...
    case int:
        // ...
    }
    return /* ... */           4
}
```

Our function will receive a slice of empty interfaces at 1. We then 2 use a type switch to find out what the actual type of the input is. And then 3 then sort accordingly. And, when done, return 4 the sorted slice.

But when we call this function with `sort([]int{1, 4, 5})`, it fails with: "cannot use i (type []int) as type []interface { } in function argument"

This is because Go can not easily convert to a *slice* of interfaces. Just converting to an interface is easy, but to a slice is much more costly. The full mailing list discussion on this subject can be found at [go_nuts_interfaces]. To keep a long story short: Go does not (implicitly) convert slices for you.

So what is the Go way of creating such a "generic" function? Instead of doing the type inference ourselves with a type switch, we let Go do it implicitly: The following steps are required:

- Define an interface type (called `Sorter` here) with a number of methods needed for sorting. We will at least need a function to get the length of the slice, a function to compare two values and a swap function.

  ```
  type Sorter interface {
      Len() int           // len() as a method.
      Less(i, j int) bool // p[j] < p[i] as a method.
      Swap(i, j int)      // p[i], p[j] = p[j], p[i] as a metho
  }
  ```

- Define new types for the slices we want to sort. Note that we declare slice types:

```
type Xi []int
type Xs []string
```

- Implementation of the methods of the `Sorter` interface. For integers:

```
func (p Xi) Len() int               {return len(p)}
func (p Xi) Less(i int, j int) bool {return p[j] < p[i]}
func (p Xi) Swap(i int, j int)      {p[i], p[j] = p[j], p[i]}
```

  And for strings:

```
func (p Xs) Len() int               {return len(p)}
func (p Xs) Less(i int, j int) bool {return p[j] < p[i]}
func (p Xs) Swap(i int, j int)      {p[i], p[j] = p[j], p[i]}
```

- Write a *generic* Sort function that works on the `Sorter` interface.

```
func Sort(x Sorter) {  1
    for i := 0; i < x.Len() - 1; i++ {  2
        for j := i + 1; j < x.Len(); j++ {
            if x.Less(i, j) {
                x.Swap(i, j)
            }
        }
    }
}
```

  At 1 `x` is now of the `Sorter` type and using the defined methods for this interface we implement Bubblesort at 2.

  Now we can use our *generic* `Sort` function as follows:

```
ints := Xi{44, 67, 3, 17, 89, 10, 73, 9, 14, 8}
strings := Xs{"nut", "ape", "elephant", "zoo", "go"}


Sort(ints)
fmt.Printf("%v\n", ints)
Sort(strings)
fmt.Printf("%v\n", strings)
```

Listing interfaces in interfaces

Take a look at the following example of an interface definition, this one is from the package `container/heap`:

```
type Interface interface {
    sort.Interface
    Push(x interface{})
    Pop() interface{}
}
```

Here another interface is listed inside the definition of `heap.Interface`, this may look odd, but is perfectly valid, remember that on the surface an interface is nothing more than a listing of methods. `sort.Interface` is also such a listing, so it is perfectly legal to include it in the interface.

Introspection and reflection

In the following example we want to look at the "tag" (here named "namestr") defined in the type definition of `Person`. To do this we need the `reflect` package (there is no other way in Go). Keep in mind that looking at a tag means going back to the *type* definition. So we use the `reflect` package to figure out the type of the variable and *then* access the tag.

```
type Person struct {
    name string "namestr"
    age  int
}

func ShowTag(i interface{}) {  1
    switch t := reflect.TypeOf(i); t.Kind() {
    case reflect.Ptr:  2
        tag := t.Elem().Field(0).Tag
    //              <<3>>      <<4>>        <<5>>
```

Figure 4. Introspection using reflection.

We are calling `ShowTag` at 1 with a `*Person`, so at 2 we're expecting a `reflect.Ptr`. We are dealing with a `Type` 3 and according to the documentation [17]:

> *Elem returns a type's element type. It panics if the type's Kind is not Array, Chan, Map, Ptr, or Slice.*

So on `t` we use `Elem()` to get the value the pointer points to. We have now dereferenced the pointer and are "inside" our structure. We then 4 use `Field(0)` to access the zeroth field.

The struct `StructField` has a `Tag` member which returns the tag-name as a string. So on the 0^{th} field we can unleash `.Tag` 5 to access this name: `Field(0).Tag`. This gives us `namestr`.

To make the difference between types and values more clear, take a look at the following code:

```
func show(i interface{}) {
    switch t := i.(type) {
    case *Person:
        t := reflect.TypeOf(i)   1
        v := reflect.ValueOf(i)  2
        tag := t.Elem().Field(0).Tag  3
        name := v.Elem().Field(0).String()  4
    }
}
```

Figure 5. Reflection and the type and value.

At 1 we create `t` the type data of `i`, and `v` gets the actual values at 2. Here at 3 we want to get to the "tag". So we need `Elem()` to redirect the pointer, access the first field and get the tag. Note that we operate on `t` a `reflect.Type`. Now 4 we want to get access to the *value* of one of the members and we employ `Elem()` on `v` to do the redirection. we have "arrived" at the structure. Then we go to the first field `Field(0)` and invoke the `String()` method on it.

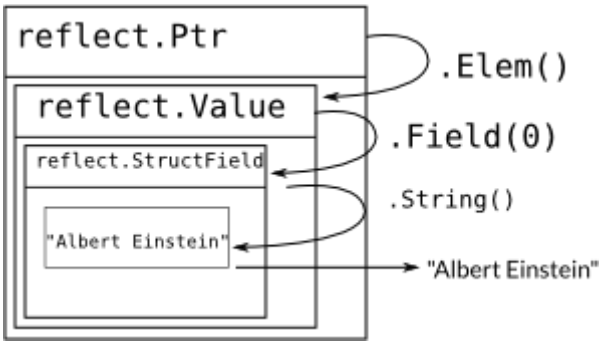

Figure 6. Peeling away the layers using reflection. Going from a `*Person` via `Elem` using the methods described in `go doc reflect` to get the actual `string` contained within.")

Setting a value works similarly as getting a value, but only works on *exported* members. Again some code:

```
type Person struct {
    name string
    age  int
}

func Set(i interface{}) {
    switch i.(type) {
    case *Person:
        r := reflect.ValueOf(i)
        r.Elem(0).Field(0).SetString("Albert Einstein")
    }
}
```

Figure 7. Reflect with *private* member.

```
type Person struct {
    Name string
    age  int
}

func Set(i interface{}) {
    switch i.(type) {
    case *Person:
        r := reflect.ValueOf(i)
        r.Elem().Field(0).SetString("Albert Einstein")
    }
}
```

Figure 8. Reflect with *public* member.

The first program compiles and runs, but when you run it, you are greeted with a stack trace and a *run time* error: “panic: reflect.Value.SetString using value obtained using unexported field”.

The second program works OK and sets the member `Name` to “Albert Einstein”. Of course this only works when you call `Set()` with a pointer argument.

Exercises

Interfaces and max()

In the maximum exercise we created a max function that works on a slice of integers. The question now is to create a program that shows the maximum number and that works for both integers and floats. Try to make your program as generic as possible, although that is quite difficult in this case.

▷ Answer

Pointers and reflection

One of the last paragraphs in section Introspection and reflection has the following words:

> *The code on the right works OK and sets the member* `Name` *to "Albert Einstein". Of course this only works when you call* `Set()` *with a pointer argument.*

Why is this the case?

▷ **Answer**

Chapter 7. Concurrency

> *Parallelism is about performance.*
> *Concurrency is about program design.*
>
> GOOGLE I/O 2010 – ROB PIKE

In this chapter we will show off Go's ability for concurrent programming using channels and goroutines. Goroutines are the central entity in Go's ability for concurrency.

But what *is* a goroutine, from [effective_go]:

> *They're called goroutines because the existing terms – threads, coroutines, processes, and so on – convey inaccurate connotations. A goroutine has a simple model: it is a function executing in parallel with other goroutines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.*

A goroutine is a normal function, except that you start it with the keyword `go`.

```
ready("Tea", 2)     // Normal function call.
go ready("Tea", 2)  // ... as goroutine.
```

```
func ready(w string, sec int) {
    time.Sleep(time.Duration(sec) * time.Second)
    fmt.Println(w, "is ready!")
}

func main() {
    go ready("Tea", 2)    //<1>
    go ready("Coffee", 1) //<1>
    fmt.Println("I'm waiting")
    time.Sleep(5 * time.Second) //<2>
```

Figure 9. Go routines in action.

The following idea for a program was taken from [go_course_day3]. We run a function as two goroutines, the goroutines wait for an amount of time and then print something to the screen. At 1 we start the goroutines. The `main` function waits long enough at 2, so that both goroutines will have printed their text. Right now we wait for 5 seconds, but in fact we have no idea how long we should wait until all goroutines have exited. This outputs:

```
I'm waiting          // Right away
Coffee is ready!     // After 1 second
Tea is ready!        // After 2 seconds
```

If we did not wait for the goroutines (i.e. remove the last line at 2) the program would be terminated immediately and any running goroutines would *die with it*.

To fix this we need some kind of mechanism which allows us to communicate with the goroutines. This mechanism is available to us in the form of channels . A channel can be compared to a two-way pipe in Unix shells: you can send to and receive values from it. Those values can only be of a specific type: the type of the channel. If we define a channel, we must also define the type of the values we can send on the channel. Note that we must use `make` to create a channel:

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

Makes `ci` a channel on which we can send and receive integers, makes `cs` a channel for strings and `cf` a channel for types that satisfy the empty interface. Sending on a channel and receiving from it, is done with the same operator: `<-`.

Depending on the operands it figures out what to do:

```
ci <- 1   // *Send* the integer 1 to the channel ci.
<-ci      // *Receive* an integer from the channel ci.
i := <-ci // *Receive* from the channel ci and store it in i.
```

Let's put this to use.

```
var c chan int 1

func ready(w string, sec int) {
    time.Sleep(time.Duration(sec) * time.Second)
    fmt.Println(w, "is ready!")
    c <- 1  2
}

func main() {
    c = make(chan int) 3
    go ready("Tea", 2) 4
    go ready("Coffee", 1) 4
    fmt.Println("I'm waiting, but not too long")
    <-c 5
    <-c 5
}
```

At 1 we declare `c` to be a variable that is a channel of ints. That is: this channel can move integers. Note that this variable is global so that the goroutines have access to it. At 2 in the `ready` function we send the integer 1 on the channel. In our `main` function we initialize `c` at 3 and start our goroutines 4. At 5 we Wait until we receive a value from the channel, the value we receive is discarded. We have started two goroutines, so we expect two values to receive.

There is still some remaining ugliness; we have to read twice from the channel 5). This is OK in this case, but what if we don't know how many goroutines we started? This is where another Go built-in comes in: `select` (((keywords, select))). With `select` you can (among other things) listen for incoming data on a channel.

Using `select` in our program does not really make it shorter, because we run too few go-routines. We remove last lines and replace them with the following:

```
L: for {
    select {
    case <-c:
        i++
        if i > 1 {
            break L
        }
```

```
    }
}
```

We will now wait as long as it takes. Only when we have received more than one reply on the channel `c` will we exit the loop `L`.

Make it run in parallel

While our goroutines were running concurrently, they were not running in parallel. When you do not tell Go anything there can only be one goroutine running at a time. With `runtime.GOMAXPROCS(n)` you can set the number of goroutines that can run in parallel. From the documentation:

> *GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting. If n < 1, it does not change the current setting. This call will go away when the scheduler improves.*

If you do not want to change any source code you can also set an environment variable `GOMAXPROCS` to the desired value.

Note that the above discussion relates to older versions of Go. From version 1.5 and above, `GOMAXPROCS` defaults to the number of CPU cores[go_1_5_release_notes].

More on channels

When you create a channel in Go with `ch := make(chan bool)`, an unbuffered channel for bools is created. What does this mean for your program? For one, if you read (`value := <-ch`) it will block until there is data to receive. Secondly anything sending (`ch <- true`) will block until there is somebody to read it. Unbuffered channels make a perfect tool for synchronizing multiple goroutines.

But Go allows you to specify the buffer size of a channel, which is quite simply how many elements a channel can hold. `ch := make(chan bool, 4)`, creates a buffered channel of bools that can hold 4 elements. The first 4 elements in this channel are written without any blocking. When you write the 5^{th} element, your code *will* block, until another goroutine reads some elements from the channel to make room.

In conclusion, the following is true in Go:

$$\text{ch} := \text{make(chan type, value)} \begin{cases} value == 0 & \rightarrow \text{unbuffered} \\ value > 0 & \rightarrow \text{buffer } value \text{ elements} \end{cases}$$

When a channel is closed the reading side needs to know this. The following code will check if a channel is closed.

```
x, ok = <-ch
```

Where `ok` is set to `true` the channel is not closed *and* we've read something. Otherwise `ok` is set to `false`. In that case the channel was closed and the value received is a zero value of the channel's type.

Exercises

Channels

1. Modify the program you created in exercise For-loop to use channels, in other words, the function called in the body should now be a goroutine and communication should happen via channels. You should not worry yourself on how the goroutine terminates.

2. There are a few annoying issues left if you resolve question 1 above. One of the problems is that the goroutine isn't neatly cleaned up when `main.main()` exits. And worse, due to a race condition between the exit of `main.main()` and `main.shower()` not all numbers are printed. It should print up until 9, but sometimes it prints only to 8. Adding a second quit-channel you can remedy both issues. Do this.

▷ **Answer**

Fibonacci II

This is the same exercise as an earlier one Fibonacci in exercise. For completeness the complete question:

> *The Fibonacci sequence starts as follows:* $1, 1, 2, 3, 5, 8, 13, \ldots$ *Or in mathematical terms:* $x_1 = 1; x_2 = 1; x_n = x_{n-1} + > x_{n-2} \quad \forall n > 2$.
>
> *Write a function that takes an* `int` *value and gives that many terms of the Fibonacci sequence.*

But now the twist: You must use channels.

▷ **Answer**

Chapter 8.

Communication

Good communication is as stimulating as black coffee, and just as hard to sleep after.

– Anne Morrow Lindbergh

In this chapter we are going to look at the building blocks in Go for communicating with the outside world. We will look at files, directories, networking and executing other programs. Central to Go's I/O are the interfaces `io.Reader` and `io.Writer`. The `io.Reader` interface specifies one method `Read(p []byte) (n int, err err)`.

Reading from (and writing to) files is easy in Go. This program only uses the `os` package to read data from the file `/etc/passwd`.

```
package main

import (
    "log"
    "os"
)

func main() {
    buf := make([]byte, 1024)
    f, e := os.Open("/etc/passwd") 1
    if e != nil {
        log.Fatalf(e)
    }
    defer f.Close() 2
    for {
        n, e := f.Read(buf) 3
        if e != nil {
            log.Fatalf(e) 4
        }
        if n == 0 { 5
            break
        }
        os.Stdout.Write(buf[:n]) 6
```

```
    }
}
```

We open the file at 1 with `os.Open` that returns a `*os.File` `*os.File` implements `io.Reader` and `io.Writer` interface. After the `Open` we directly put the `f.Close()` which we defer until the function return. At 3 we call `Read` on `f` and read up to 1024 bytes at the time. If anything fails we bail out at 4. If the number of bytes read is 0 we've read the end of the file 5. And at 6 we output the buffer to standard output.

If you want to use buffered I/O there is the `bufio` package:

```
package main

import (
    "bufio"
    "log"
    "os"
)

func main() {
    buf := make([]byte, 1024)
    f, e := os.Open("/etc/passwd") 1
    if e != nil {
        log.Fatalf(e)
    }
    defer f.Close()
    r := bufio.NewReader(f) 2
    w := bufio.NewWriter(os.Stdout)
    defer w.Flush() 3
    for {
        n, e := r.Read(buf) 4
        if e != nil {
            log.Fatalf(e)
        }
        if n == 0 {
            break
        }
        w.Write(buf[0:n]) 5
    }
}
```

Again, we open 1 the file. Then at 2 we Turn `f` into a buffered `Reader`. `NewReader` expects an `io.Reader`, so you this will work. Then at 4 we read and at 5 we write. We also call `Flush()` at 3 to flush all output. This entire program could be optimized further by using `io.Copy`.

io.Reader

As mentioned above the `io.Reader` is an important interface in the language Go. A lot (if not all) functions that need to read from something take an `io.Reader` as input. To fulfill the interface a type needs to implement that one method. The writing side `io.Writer`, has the `Write` method.

If you think of a new type in your program or package and you make it fulfill the `io.Reader` or `io.Writer` interface, *the whole standard Go library can be used* on that type!

Some examples

The previous program reads a file in its entirety, but a more common scenario is that you want to read a file on a line-by-line basis. The following snippet shows a way to do just that (we're discarding the error returned from `os.Open` here to keep the examples smaller – don't ever do this in real life code).

```
f, _ := os.Open("/etc/passwd"); defer f.Close()
r := bufio.NewReader(f) 1
s, ok := r.ReadString('\n') 2
```

At 1 make `f` a `bufio` to have access to the `ReadString` method. Then at 2 we read a line from the input, `s` now holds a string which we can manipulate with, for instance, the `strings` package.

A more robust method (but slightly more complicated) is `ReadLine`, see the documentation of the `bufio` package.

A common scenario in shell scripting is that you want to check if a directory exists and if not, create one.

```
if [ ! -e name ]; then
    mkdir name
else
    # error
fi
```

```
if f, e := os.Stat("name"); e != 
    os.Mkdir("name", 0755)
} else {
    // error
}
```

The similarity between these two examples (and with other scripting languages) have prompted comments that Go has a "script"-like feel to it, i.e. programming in Go can be compared to programming in an interpreted language (Python, Ruby, Perl or PHP).

Command line arguments

Arguments from the command line are available inside your program via the string slice `os.Args`, provided you have imported the package `os`. The `flag` package has a more sophisticated interface, and also provides a way to parse flags. Take this example from a DNS query tool:

```
dnssec := flag.Bool("dnssec", false, "Request DNSSEC records") 1
port := flag.String("port", "53", "Set the query port") 2
flag.Usage = func() {    3
    fmt.Fprintf(os.Stderr, "Usage: %s [OPTIONS] [name ...]\n", os
    flag.PrintDefaults() 4
}
flag.Parse() 4
```

At 1 we define a `bool` flag `-dnssec`. Note that this function returns a *pointer* to the value, the `dnssec` is now a pointer to a `bool`. At 2 we define an `strings` flag. Then at 3 we *redefine* the `Usage` variable of the flag package so we can add some extra text. The `PrintDefaults` at 4 will output the default help for the flags that are defined. Note even without redefining a `flag.Usage` the flag `-h` is supported and will just output the help text for each of the flags. Finally at 4 we call `Parse` that parses the command line and fills the variables.

After the flags have been parsed you can used them: `if *dnssec { ... }`

Executing commands

The `os/exec` package has functions to run external commands, and is the premier way to execute commands from within a Go program. It works by defining a `*exec.Cmd` structure for which it defines a number of methods. Let's execute `ls -l`:

```
import "os/exec"

cmd := exec.Command("/bin/ls", "-l")
err := cmd.Run()
```

The above example just runs "ls -l" without doing anything with the returned data, capturing the standard output from a command is done as follows:

```
cmd := exec.Command("/bin/ls", "-l")
buf, err := cmd.Output()
```

And `buf` is byte slice, that you can further use in your program.

Networking

All network related types and functions can be found in the package `net`. One of the most important functions in there is `Dial`. When you `Dial` into a remote system the function returns a `Conn` interface type, which can be used to send and receive information. The function `Dial` neatly abstracts away the network family and transport. So IPv4 or IPv6, TCP or UDP can all share a common interface.

Dialing a remote system (port 80) over TCP, then UDP and lastly TCP over IPv6 looks like this[18]:

```
conn, e := Dial("tcp", "192.0.32.10:80")
conn, e := Dial("udp", "192.0.32.10:80")
conn, e := Dial("tcp", "[2620:0:2d0:200::10]:80")
```

If there were no errors (returned in `e`), you can use `conn` to read and write. And `conn` implements the `io.Reader` and `io.Writer` interface. [19]

But these are the low level nooks and crannies, you will almost always use higher level packages, such as the `http` package. For instance a simple Get for http:

```
package main

import (
    "fmt"
    "http"
    "io/ioutil"
)

func main() {
    r, err := http.Get("http://www.google.com/robots.txt")
    if err != nil {
        fmt.Printf("%s\n", err.String())
        return
    }
    b, err := ioutil.ReadAll(r.Body)
    r.Body.Close()
    if err == nil {
        fmt.Printf("%s", string(b))
    }
}
```

Exercises

Finger daemon

Write a finger daemon that works with the finger(1) command.

From the Debian package description:

> *Fingerd is a simple daemon based on RFC 1196 [RFC1196] that provides an interface to the "finger" program at most network sites. The program is supposed to return a friendly, human-oriented status report on either the system at the moment or a particular person in depth.*

Stick to the basics and only support a username argument. If the user has a `.plan` file show the contents of that file. So your program needs to be able to figure out:

- Does the user exist?
- If the user exists, show the contents of the `.plan` file.

▷ Answer

Echo server

Write a simple echo server. Make it listen to TCP port number 8053 on localhost. It should be able to read a line (up to the newline), echo back that line and then close the connection.

Make the server concurrent so that every request is taken care of in a separate goroutine.

▷ Answer

Word and Letter Count

Write a small program that reads text from standard input and performs the following actions:

- Count the number of characters (including spaces).
- Count the number of words.
- Count the numbers of lines

In other words implement wc(1) (check you local manual page), however you only have to read from standard input.

▷ Answer

Uniq

Write a Go program that mimics the function of the Unix `uniq` command. This program should work as follows, given a list with the following items:

```
'a' 'b' 'a' 'a' 'a' 'c' 'd' 'e' 'f' 'g'
```

it should print only those items which don't have the same successor:

```
'a' 'b' 'a' 'c' 'd' 'e' 'f' 'g'
```

The next listing is a Perl implementation of the algorithm.

```
#!/usr/bin/perl
my @a = qw/a b a a a c d e f g/;
print my $first = shift @a;
foreach (@a) {
    if ($first ne $_) { print; $first = $_; }
}
```

▷ Answer

Quine

A *Quine* is a program that prints itself. Write a Quine in Go.

▷ Answer

Processes

Write a program that takes a list of all running processes and prints how many child processes each parent has spawned. The output should look like:

```
Pid 0 has 2 children: [1 2]
Pid 490 has 2 children: [1199 26524]
Pid 1824 has 1 child: [7293]
```

- For acquiring the process list, you'll need to capture the output of `ps -e -opid,ppid,comm`. This output looks like:

  ```
    PID  PPID COMMAND
   9024  9023 zsh
  19560  9024 ps
  ```

- If a parent has one child you must print `child`, if there is more than one print `children`.

- The process list must be numerically sorted, so you start with pid 0 and work your way up.

Here is a Perl version to help you on your way (or to create complete and utter confusion).

```
#!/usr/bin/perl -l
my (%child, $pid, $parent);
my @ps=`ps -e -opid,ppid,comm`;   # capture the output from `ps`
foreach (@ps[1..$#ps]) {      # discard the header line
  ($pid, $parent, undef) = split; # split the line, discard 'comm
  push @{$child{$parent}}, $pid;  # save the child PIDs on a list
}
# Walk through the sorted PPIDs
foreach (sort { $a <=> $b } keys %child) {
  print "Pid ", $_, " has ", @{$child{$_}}+0, " child",
    @{$child{$_}} == 1 ? ": " : "ren: ", "[@{$child{$_}}]";
}
```

▷ Answer

Number cruncher

- Pick six (6) random numbers from this list:
  $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100$ Numbers may be picked multiple times.
- Pick one (1) random number (i) in the range: $1 \ldots 1000$.
- Tell how, by combining the first 6 numbers (or a subset thereof) with the operators `+`,`-`,`*` and `/`, you can make i.

An example. We have picked the numbers: 1, 6, 7, 8, 8 and 75. And i is 977. This can be done in many different ways, one way is:
$((((1 * 6) * 8) + 75) * 8) - 7 = 977$ or
$(8 * (75 + (8 * 6))) - (7/1) = 977$

Implement a number cruncher that works like that. Make it print the solution in a similar format (i.e. output should be infix with parenthesis) as used above.

Calculate *all* possible solutions and show them (or only show how many there are). In the example above there are 544 ways to do it.

▷ Answer

Bibliography

[RFC1196]

[RFC4627]

[bubblesort]
Wikipedia Bubble sort 2010

[csp]
C. A. R. Hoare Communicating sequential processes (csp) 1985

[duck_typing]
Wikipedia Duck typing 2010

[effective_go]
Go Authors Effective Go 2010

[fizzbuzz]
Imran On Tech Using fizzbuzz to find developers... 2010

[go_1_5_release_notes]
Go Authors Go 1.5 Release Notes 2010

[go_blog_panic]
Go Authors Defer, panic, and recover 2010

[go_course_day3]
Rob Pike The Go programming language, day 3 2010

[go_interfaces]
Ian Lance Taylor Go interfaces 2010

[go_nuts_interfaces]
Go Community Function accepting a slice of interface types 2010

[go_spec]
Go Authors Go language specification 2010

[go_tutorial]
Go Authors Go tutorial 2010

[go_web]
Go Authors Go website 2010

[iota]
Wikipedia Iota 2010

Footnotes

1. Yes, that sounds a lot like *co*routines, but goroutines are slightly different as we will see in Chapter 8. Communication. [return]
2. http://play.golang.org. [return]
3. http://golang.org/doc/ itself is served by `godoc`. [return]
4. When building from source it must be installed separately with `go get golang.org/x/tools/cmd/godoc`. [return]
5. The word iota is used in a common English phrase, 'not one iota', meaning 'not the slightest difference', in reference to a phrase in the New Testament: "*until heaven and earth pass away, not an iota, not a dot, will pass from the Law*." *[iota]* [return]
6. You can use the command `godoc builtin` to read the online documentation about the built-in types and functions. [return]
7. Always rainy in March anyway. [return]
8. This is a motto of Go; "Do *more* with *less* code". [return]
9. A function literal is sometimes called a closure . [return]
10. Modified from a presentation of Eleanor McHugh. [return]
11. The descriptions are copied from the packages' `go doc`. [return]
12. The downside is that you know have to worry about garbage collection. If you really need it garbage collection in a Go program can be disabled by running it with the environment variable `GOGC` set to `off`: `GOGC=off ./myprogram`. [return]
13. Taking the address of a composite literal tells the compiler to allocate it on the heap, not the stack. [return]
14. Also see Methods in this chapter. [return]
15. The following text is partly from *[go_interfaces]*. [return]
16. Text copied from *[effective_go]*. [return]
17. `go doc reflect` [return]
18. In case you are wondering, 192.0.32.10 and 2620:0:2d0:200::10 are http://www.example.org. [return]
19. The variable `conn` also implements a `close` method, this really makes it an `io.ReadWriteCloser`. [return]

Index

d

- duck
  - typing [go]

f

- field [go]
- functions
  - receiver [go]
  - method [go]
  - pass-by-value [go]
  - as values [go]
  - literals [go]
  - signature [go]
  - literal [go]
  - variadic [go]
  - exported [go]
  - private [go]
  - public [go]

g

- generic [go]
- goroutine [go] [go]

i

- interface [go]
  - set of methods [go]
  - type [go]
  - value [go]

- io
  - buffered [go]

- io.Reader [go]

k

- keywords
  - iota [go]
  - if [go]
  - return [go]
  - else [go]
  - goto [go]
  - for [go]

    - break [go]
    - continue [go]
    - range [go] [go] [go] [go]
    - switch [go]
    - fallthrough [go]
    - default [go]
    - map [go]
    - map adding elements [go]
    - map existence [go]
    - map remove elements [go]
    - defer [go]
    - defer list [go]
    - package [go]
    - import [go]
    - type [go]
    - struct [go]
    - go [go]

l

- label [go]
- literal
    - composite [go] [go]

m

- methods
    - inherited [go]

n

- networking
    - Dial [go]

- nil [go]

o

- operators
    - bitwise and [go]
    - bitwise or [go]
    - bit wise xor [go]
    - bitwise clear [go]
    - and [go]
    - or [go]

    - not [go]
    - address-of [go]
    - increment [go]
    - channel [go]

p

- package [go]
    - builtin [go]
    - fmt [go] [go]
    - bytes [go]
    - bufio [go] [go] [go]
    - ring [go]
    - io [go] [go]
    - sort [go]
    - strconv [go]
    - os [go]
    - sync [go]
    - flag [go] [go]
    - encoding/json [go]
    - html/template [go]
    - net/http [go]
    - unsafe [go]
    - reflect [go] [go]
    - os/exec [go] [go]

r

- reference types [go]
- runes [go] [go]

s

- scope
    - local [go] [go]

- slice
    - capacity [go]
    - length [go]

- structures
    - embed [go]

t

- tooling
  - go [go]
  - go build [go]
  - go run [go]
  - go test [go]

- type assertion [go]
- type switch [go]

v

- variables
  - declaring [go]
  - assigning [go]
  - parallel assignment [go]
  - underscore [go]
