Learning Go

Go 1

http://golang.org

Author:
Miek Gieben

Thanks to:
Go Authors, Google

With the help and contributions from:

(in alphabetical order)

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

"Learning Go" has been translated into:

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

Learning as we Go (*1.0*)

Supports the Go 1 release

Contents

List of Figures

List of Code Examples

List of Exercises

Preface

> "Is Go an object-oriented language? Yes and no."
>
> *Frequently asked questions*
> GO AUTHORS

Audience

This is an introduction to the Go language from Google. Its aim is to provide a guide to this new and innovative language.

The intended audience of this book is people who are familiar with programming and know some programming languages, be it C[6], C++[28], Perl[8], Java[21], Erlang[7], Scala[22] or Haskell[1]. This is *not* a book which teaches you how to program, this is a book that just teaches you how to use Go.

As with learning new things, probably the best way to do this is to discover it for yourself by creating your own programs. Each chapter therefore includes a number of exercises (and answers) to acquaint you with the language. An exercise is numbered as **Q**n, where n is a number. After the exercise number another number in parentheses displays the difficulty of this particular assignment. This difficulty ranges from 0 to 2:

0. easy;
1. intermediate;
2. difficult.

Then a short name is given, for easier reference. For example:

> **Q1**. (1) A map function ...

introduces a question numbered **Q1** of a level 1 difficulty, concerning a `map()`-function. The answers are included after the exercises on a new page. The numbering and setup of the answers is identical to the exercises, except that an answer starts with **A**n, where the number n corresponds with the number of the exercise. Some exercises don't have an answer; these are marked with an asterisk.

Book layout

Chapter 1: Introduction
: A short introduction and history of Go. It describes how to get the source code of Go itself. It assumes a Unix-like environment, although Go should be fully usable on Windows.

Chapter 2: Basics
: Describes the basic types, variables and control structures available in the language.

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

Chapter 4: Packages
: In chapter 4 we see that functions and data can be grouped together in packages. You will also see how to document and test your packages.

Chapter 5: Beyond the basics
: After that we look at creating your own types in chapter 5. It also looks at allocation in Go.

Chapter 6: Interfaces
: Go does not support object orientation in the traditional sense. In Go the central concept is interfaces.

Chapter 7: Concurrency
: With the `go` keyword functions can be started in separate routines (called goroutines). Communication with those goroutines is done via channels.

Chapter 8: Communication
: In the last chapter we show how to interface with the rest of the world from within a Go program. How to create files and read and write from and to them. We also briefly look into networking.

I hope you will enjoy this book and the language Go.

Settings used in this book

- Go itself is installed in `~/go`, and `$GOROOT` is set to `GOROOT=~/go` ;
- Go source code we want to compile ourself is placed in `~/g/src` and `$GOPATH` is set to `GOPATH=~/g` .

Translations

The content of this book is freely available. This has already led to translations:

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

Miek Gieben, 2011, 2012 – `miek@miek.nl`

1 Introduction

> "I am interested in this and hope to do something."
>
> *On adding complex numbers to Go*
> KEN THOMPSON

What is Go? From the website [17]:

> *The Go programming language is an open source project to make programmers more productive. Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multi core and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.*

Go 1 is the first stable release of the language Go. This document and all exercises work with Go 1 – if not, it's a bug.

The following convention is used throughout this book:

- Code is displayed in `DejaVu Mono`;
- Keywords are displayed in **`DejaVu Mono Bold`**;
- Comments are displayed in *`DejaVu Mono Italic`*;
- Extra remarks in the code ← *Are displayed like this*;
- Longer remarks get a number – ❶ – with the explanation following;
- Line numbers are printed on the right side;
- Shell examples use a **%** as prompt;
- User entered text in shell examples **`is in bold`**, system responses are in a `typewriter font`;
- An emphasized paragraph is indented and has a vertical bar on the left.

Official documentation

There is already a substantial amount of documentation written about Go. The Go Tutorial [16], and the Effective Go document [11]. The website `http://golang.org/doc/` is a very good starting point for reading up on Go[a]. Reading these documents is certainly not required, but it is recommended.

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

[a] `http://golang.org/doc/` itself is served by `go doc`.

Go 1 comes with its own documentation in the form of a program called `go doc`. If you are interested in the documentation for the built-ins (see "Operators and built-in functions" in the next chapter) you can fire it up, like so:

```
% go doc builtin
```

How to create your own package documentation is explained in chapter 4.

Origins

Go has it origins in Inferno [2] (which in turn was based upon Plan 9 [3]). Inferno included a language called Limbo [4]. Quoting from the Limbo paper:

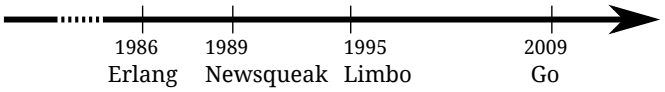
> *Limbo is a programming language intended for applications running distributed systems on small computers. It supports modular programming, strong type checking at compile- and run-time,* inter process communication over typed channels*, automatic* garbage collection*, and simple abstract data types. It is designed for safe execution even on small machines without hardware memory protection.*

A feature Go inherited from Limbo is channels (see chapter 7). Again from the Limbo documentation:

> *[A channel] is a communication mechanism capable of sending and receiving objects of the specified type to another agent in the system. Channels may be used to communicate between local processes; using library procedures, they may be connected to named destinations. In either case send and receive operations may be directed to them.*

The channels in Go are easier to use than those in Limbo. If we dig deeper into the history of Go we also find references to "Newsqueak" [25], which pioneered the use of channel communication in a C–like language. Channels aren't unique to these languages; a big non–C–like language which uses them is Erlang [7].

Figure 1.1. Chronology of Go



The whole of idea of using channels to communicate with other processes is called Communicating Sequential Processes (CSP) and was conceived by C. A. R. Hoare [24], who incidentally is the same man that invented QuickSort [23].

> Go is the first C–like language that is widely available, runs on many different platforms and makes concurrency easy (or easier).

Getting Go

In this section we explain how to install Go locally on your machine, but you can also compile Go code online at `http://play.golang.org/`. To quickly play with code this is by far the easiest route.

You can also get pre-compiled binaries from [18].

Ubuntu and Debian both have a Go package in their repositories (look for the package "golang"), but there are still some minor issues being worked out. For now we will stick to the installation from source.

So we will have to retrieve the code from the mercurial archive and compile Go yourself. For other Unix-like systems the procedure is the same.

- First install Mercurial (to get the `hg` command). In Ubuntu/Debian/Fedora you must install the `mercurial` package;
- For building Go you need the packages: `bison`, `gcc`, `libc6-dev`, `ed`, `gawk` and `make`;
- Set the environment variable `GOROOT` to the root of your Go install:
  `% export GOROOT=~/go`
- Then retrieve the latest release (= Go 1) source code:
  `% hg clone -r release https://go.googlecode.com/hg/ $GOROOT`
- Set your PATH to so that the shell can find the Go binaries:
  `% export PATH=$GOROOT/bin:$PATH`
- Compile Go
  `% cd $GOROOT/src`
  `% ./all.bash`

If all goes well, you should see the following at the end:

```
--- cd ../test
0 known bugs; 0 unexpected bugs

ALL TESTS PASSED

---
Installed Go for linux/amd64 in /home/go
Installed commands in /home/go/bin
```

Getting Go for Windows

The best way is to follow the instructions from [18], which are repeated here for your convience.

- Download Go 1 from: `http://code.google.com/p/go/downloads/list?q=OpSys-Windows+Type%3DArchive`;
- Unpack it to your `C:\` drive;

- Make sure that the contents are `C:\Go`. Note: this directory should be created when you unpacked the zip;
- Add `C:\Go\bin` to your $PATH:
  `set PATH=%PATH%;C:\Go\bin`

Exercises

Q1. (1) Documentation

1. Go's documentation can be read with the `go doc` program, which is included the Go distribution.

   `go doc hash` gives information about the *hash* package:

   ```
   % go doc hash
   PACKAGE

   package hash

   ...
   ...
   ...

   SUBDIRECTORIES

           adler32
           crc32
           crc64
           fnv
   ```

   With which `go doc` command can you read the documentation of *fnv* contained in *hash*?

Answers

A1. (1) Documentation

1. The package *fnv* is in a *subdirectory* of *hash*, so you will only need `go doc hash/fnv`.

   Specific functions inside the "Go manual" can also be accessed. For instance the function `Printf` is described in *fmt*. To view the documentation for this function alone use: `go doc fmt Printf` .

   All the built-in functions are also accesible by using `go doc builtin`.

2 Basics

> "In Go, the code does exactly what it says on the page."
>
> *Go Nuts mailing list*
> ANDREW GERRAND

There are a few things that make Go different from other languages.

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 [a] in Go;

Channels
: Communication with these goroutines is done via channels [32, 24];

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;` as one would in C;

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

Open Source
: The Go license is completely open source, see the file LICENSE in the Go source code distribution;

Fun
: Programming with Go should be fun!

[a] Yes, that sounds a lot like *co*routines, but goroutines are slightly different as we will see in chapter 7.

Erlang [7] also shares some of the features of Go. Notable differences between Erlang and Go is that Erlang borders on being a functional language, where Go is an imperative one. And Erlang runs in a virtual machine, while Go is compiled. Go also has a much more Unix-like feel to it.

Hello World

In the Go tutorial, Go is presented to the world in the typical manner: letting it print "Hello World" (Ken Thompson and Dennis Ritchie started this when they presented the C language in the 1970s). We don't think we can do better, so here it is, "Hello World" in Go.

Listing 2.1. Hello world

```
package main ❶                                                    1

import "fmt"  // Implements formatted I/O. ❷                      3

/* Print something */ ❸                                          5
func main() { ❹                                                   6
❺                                                                 7
 fmt.Printf("Hello, world; or καλημέρα  κόσμε; or こんにちは世界")  8
}                                                                 9
```

Lets look at the program line by line.

❶ This first line is just required. All Go files start with **package** `<something>`, **package** `main` is required for a standalone executable;

❷ This says we need *fmt* in addition to *main*. A package other than *main* is commonly called a library, a familiar concept in many programming languages (see chapter 4). The line ends with a comment which is started with `//`;

❸ This is also a comment, but this one is enclosed in `/*` and `*/`;

❹ Just as **package** `main` was required to be first, **import** may come next. In Go, **package** is always first, then **import**, then everything else. 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;

❺ On line 8 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. Here we use Greek and Japanese.

Compiling and running code

The preferred way to build a Go program is to use the `go` tool. To build `helloworld` we just enter:

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

This results in an executable called `helloworld`.

```
% ./helloworld
Hello, world; or καλημέρα  κόσμε; or こんにちは世界
```

Settings used in this book

- Go itself is installed in `~/go`, and `$GOROOT` is set to `GOROOT=~/go` ;
- Go source code we want to compile ourself is placed in `~/g/src` and `$GOPATH` is set to `GOPATH=~/g` . This variable comes into play when we start using packages (chapter 4).

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 has a C-like feel when it comes to its syntax. If you want to put two (or more) statements on one line, they must be separated with a semicolon (';'). Normally you don't need the semicolon.

Go is different from other languages in that the type of a variable is specified *after* the variable name. So not: `int a`, but `a int`. When declaring 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.

Listing 2.2. Declaration with =

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

Listing 2.3. Declaration with :=

```
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` and `import` also allow this. Note the use of parentheses:

```
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
```

Which makes `a` and `b` both integer variables and assigns 20 to `a` and 16 to `b`.

A special name for a variable is _ (underscore). Any value assigned to it is discarded. 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. The following code generates this error: `i declared and not used`

```
package main
func main() {
    var i int
}
```

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 the well known types such as **`int`**. This 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`**.

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

Listing 2.4. Familiar types are still distinct

```
package main                                                          1

func main() {                                                         3
        var a int           ⟵ Generic integer type                    4
        var b int32         ⟵ 32 bits integer type                    5
        a = 15                                                        6
        b = a + a           ⟵ Illegal mixing of these types           7
        b = b + 5           ⟵ 5 is a (typeless) constant, so this is OK 8
}                                                                     9
```

Gives the error on the assignment on line 7:

```
types.go:7: cannot use a + a (type int) as type int32 in assignment
```

The assigned values may be denoted using octal, hexadecimal or the scientific notation: `077`, `0xFF`, `1e3` or `6.022e23` are all valid.

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`** [b] to enumerate values.

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

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.

You can even do the following, let Go repeat the use of **`= iota`**:

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

You may also explicitly type a constant, if you need that:

```
const (
        a = 0               ⟵ Is an int now
        b string = "0"
)
```

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 can not be changed: strings in Go are immutable. For people coming from C, the following is not legal in Go:

```
var s string = "hello"
s[0] = 'c'      ⟵ Change first char. to 'c', this is an error
```

To do this in Go you will need the following:

```
s := "hello"
c := []rune(s)      ❶
c[0] = 'c'          ❷
s2 := string(c)     ❸
fmt.Printf("%s\n", s2) ❹
```

[b] 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*.” [34]

❶ Convert `s` to an array of runes, see chapter 5 section “Conversions” on page 63;

❶ Change the first element of this array;

❷ Create a *new* string `s2` with the alteration;

❸ print the string with `fmt.Printf`.

Multi-line strings

Due to the insertion of semicolons (see [11] section “Semicolons”), you need to be careful with using multi line strings. If you write:

```
s := "Starting part"
    + "Ending part"
```

This is transformed into:

```
s := "Starting part";
    + "Ending part";
```

Which is not valid syntax, you need to write:

```
s := "Starting part" +
     "Ending part"
```

Then Go will not insert the semicolons in the wrong places. Another way would be to use *raw* string literals by using backquotes (`):

```
s := `Starting part
     Ending part`
```

Be aware that in this last example `s` now also contains the newline. Unlike *interpreted* string literals the value of a raw string literal is composed of the *uninterpreted* characters between the quotes.

Runes

`Rune` is an alias for int32. It is an UTF-8 encoded code point. When is this type useful? For instance, when iterating over characters in a string. You can 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!). So 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’ ($\sqrt{-1}$). An example of using complex numbers:

```
var c complex64 = 5+5i;fmt.Printf("Value is: %v", c)
```

will print: `(5+5i)`

The Printf() verb %v, means “print the value in its default format”.

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`.

Operators and built-in functions

Go supports the normal set of numerical operators. Table 2.1 lists the current ones and their relative precedence. They all associate from left to right.

Table 2.1. Operator precedence

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

`+ - * /` 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

Table 2.2. 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 2.2 lists all the keywords in Go. We will cover them in the following paragraphs and chapters. Some of them we have seen already.

- For **`var`** and **`const`** see “Variables, types and keywords” on page 8;
- **`package`** and **`import`** are briefly touched on in section “Hello World”. In chapter 4 they are documented in more detail.

Others deserve more text and have their own chapter/section:

- **func** is used to declare functions and methods;
- **return** is used to return from functions, for both **func** and **return** see chapter 3 for the details;
- **go** is used for concurrency, see chapter 7;
- **select** used to choose from different types of communication, see chapter 7;
- **interface** see chapter 6;
- **struct** is used for abstract data types, see chapter 5;
- **type** also see chapter 5.

Control structures

There are only a few control structures in Go [c]. For instance there is no do or while loop, only a **for**. There is a (flexible) **switch** statement and **if** and **switch** accept an optional initialization statement like that of **for**. There also is something called a type switch and a multiway communications multiplexer, **select** (see chapter 7). The syntax is different (from that in C): parentheses are not required and the body must *always* be brace-delimited.

If-else

In Go an **if** looks like this:

```
if x > 0 {          ⟵ { is mandatory
    return y
} else {
    return x
}
```

Mandatory braces encourage writing simple **if** statements on multiple lines. It is good style to do so anyway, especially when the body contains a control statement such as a **return** or **break**.

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

```
if err := file.Chmod(0664); err != nil {      ⟵ nil is like C's NULL
    fmt.Printf(err)     ⟵ Scope of err is limited to if's body
    return err
}
```

You can use the logical operators (see table 2.1) as you would normally:

[c] This section is copied from [11].

```
if true && true  {
    fmt.Println("true")
}
if ! false {
    fmt.Println("true")
}
```

In the Go libraries, you will find that when an **if** statement doesn't flow into the next statement – that is, the body ends in **break**, **continue**, **goto**, or **return** – the unnecessary **else** is omitted.

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
doSomething(f)
```

This is an example of a common situation where code must analyze a sequence of error possibilities. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in **return** statements, the resulting code needs no **else** statements.

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    return err
}
doSomething(f, d)
```

Syntax-wise the following is illegal in Go:

```
if err != nil
{                       ← Must be on the same line as the if
    return err
}
```

See [11] section "Semicolons" for the deeper reasons behind this.

> Ending with **if-then-else**
>
> Note that if you end a function like this:
>
> ```
> if err != nil {
>     return err
> } else {
>     return nil
> }
> ```
>
> It will not compile. This is a bug in the Go compiler. See [20] for an extended problem description and hopefully a fix.

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:         ← First word on a line ending with a colon is a label
        println(i)
        i++
        goto Here     ← Jump
}
```

The name of the label is case sensitive.

For

The Go **for** loop has three forms, only one of which has semicolons.

```
for init; condition; post { }     ← Like a C for

for condition { }                 ← Like a while

for { }                           ←  Like a C for(;;) (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 += i        ← Short for sum = sum + i
}       ← i ceases to exist after the loop
```

Finally, since Go has no comma operator and ++ and - - are statements not expressions, if you want to run multiple variables in a **for** you should use parallel assignment.

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {     ← Parallel assignment
    a[i], a[j] = a[j], a[i]     ← Here too
}
```

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     ← Stop this loop, making it only print 0 to 5
    }
    println(i)
}
```

With loops within loops you can specify a label after **break**. Making the label identify *which* loop to stop:

```
J:  for j := 0; j < 5; j++ {
        for i := 0; i < 10; i++ {
            if i > 5 {
                break J     ⟵ Now it breaks the j-loop, not the i one
            }
            println(i)
        }
    }
```

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. The following loop prints 0 to 5.

```
for i := 0; i < 10; i++ {
    if i > 5 {
        continue    ⟵ Skip the rest of the remaining code in the loop
    }
    println(i)
}
```

Range

The keyword **range** can be used for loops. It can loop over slices, arrays, strings, maps and channels (see chapter 7). **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"}      ❶
for k, v := range list {          ❷
    // do what you want with k and v ❸
}
```

❶ Create a slice (see “Arrays, slices and maps” on page 19) of strings.

❷ Use **range** to loop over them. With each iteration **range** will return the index as an **int** and the key as a **string**, starting with 0 and “a”.

❸ k will have the value 0...5, and v will loop through “a”...“f”.

You can also use **range** on strings directly. Then it will break out the individual Unicode characters [d] and their start position, by parsing the UTF-8. The loop:

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

prints

[d] 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**.

```
character 'a' starts at byte position 0
character 'Φ' starts at byte position 1
character 'x' starts at byte position 3    ← Φ took 2 bytes
```

Switch

Go's **switch** is very flexible. The expressions need not be constants or even 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**.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

There is no automatic fall through, you can however use **fallthrough** to do just that. Without **fallthrough**:

```
switch i {
    case 0:  // empty case body
    case 1:
        f()  // f is not called when i == 0!
}
```

And with:

```
switch i {
    case 0:  fallthrough
    case 1:
        f()  // f is called when i == 0!
}
```

With **default** you can specify an action when none of the other cases match.

```
switch i {
    case 0:
    case 1:
        f()
    default:
        g()     // called when i is not 0 or 1
}
```

Cases can be presented in comma-separated lists.

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+':     ⟵ , as "or"
        return true
    }
    return false
}
```

Here's a comparison routine for byte arrays that uses two **switch** statements:

```
// Compare returns an integer comparing the two byte arrays
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    // Strings are equal except for possible tail
    switch {
    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0    // Strings are equal
}
```

Built-in functions

A small number of functions are predefined, meaning you *don't* have to include any package to get access to them. Table 2.3 lists them all.[e]

Table 2.3. Pre–defined functions in Go

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

These built-in functions are documented in the *builtin* pseudo package that is included in recent Go releases.

[e] You can use the command `go doc builtin` to read the online documentation about the built-in types and functions.

close is used in channel communication. It closes a channel, see chapter 7 for more on this.

delete is used for deleting entries in maps.

len and *cap* are used on a number of different types, `len` is used for returning the length of strings and the length of slices and arrays. See section “Arrays, slices and maps” for the details of slices and arrays and the function `cap`.

new is used for allocating memory for user defined data types. See section “Allocation with new” on page 59.

make is used for allocating memory for built-in types (maps, slices and channels). See section “Allocation with make” on page 59.

copy is used for copying slices. See section “Slices” in this chapter.

append is for concatenating slices. See section “Slices” in this chapter.

panic and *recover* are used for an *exception* mechanism. See the section “Panic and recovering” on page 36 for more.

print and *println* are low level printing functions that can be used without reverting to the *fmt* package. These are mainly used for debugging.

complex, *real* and *imag* all deal with complex numbers. Apart from the simple example we gave, we will not further explain complex numbers.

Arrays, slices and maps

Storing multiple values in a list can be done by utilizing 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. Assigning or indexing an element in the array is done with 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. Note that all fields must be specified. So if you are using multidimensional arrays you have to do quite some typing:

A composite literal allows you to assign a value directly to an array, slice or map. See the section "Constructors and composite literals" on page 60 for more.

```
a := [2][2]int{ [2]int{1,2}, [2]int{3,4} }
```

Which is the same as:

```
a := [2][2]int{ [...]int{1,2}, [...]int{3,4} }
```

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. Since release `2010-10-27` this syntax was further simplified. From the release notes:

Go release 2010-10-27 [15].

> *The syntax for arrays, slices, and maps of composite literals has been simplified. Within a composite literal of array, slice, or map type, elements that are themselves composite literals may elide the type if it is identical to the outer literal's element type.*

This means our example can become:

```
a := [2][2]int{ {1,2}, {3,4} }
```

Slices

TODO
Add push/pop to this section as container/vector will be deprecated.

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, which 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:

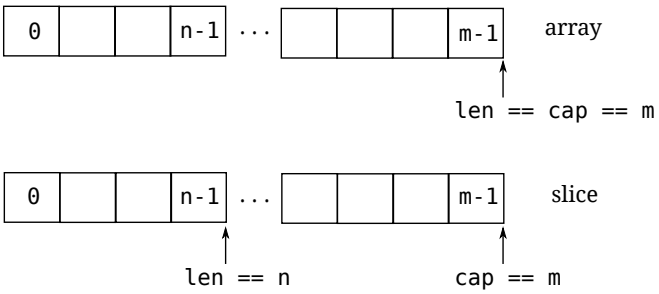
Reference types are created with ***make****.*

```
sl := 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. Figure 2.1 depicts the following Go code. 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[0:n]`
And now we have:

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

Figure 2.1. Array versus slice



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

```
// array[n:m], create a slice from array with elements n to m-1
a := [...]int{1, 2, 3, 4, 5} ❶
s1 := a[2:4] ❶
s2 := a[1:5] ❷
s3 := a[:]   ❸
s4 := a[:4]  ❹
s5 := s2[:]  ❺
```

❶ Define an array with 5 elements, from index 0 to 4;

❶ Create a slice with the elements from index 2 to 3, this contains: `3, 4`;

❷ Create a slice with the elements from index 1 to 4, contains: `2, 3, 4, 5`;

❸ Create a slice with all the elements of the array in it. This is a shorthand for: `a[0:len(a)]`;

❹ Create a slice with the elements from index 0 to 3, this is thus short for: `a[0:4]`, and yields: `1, 2, 3, 4`;

❺ Create a slice from the slice `s2`, note that `s5` still refers to the array `a`.

In the code listed in 2.5 we dare to do the impossible on line 8 and try to allocate something beyond the capacity (maximum length of the underlying array) and we are greeted with a *runtime* error.

Listing 2.5. Arrays and slices

```
package main                                                          1

func main() {                                                         3
        var array [100]int     // Create array, index from 0 to 99    4
        slice := array[0:99]   // Create slice, index from 0 to 98    5

        slice[98] = 'a'        // OK                                  7
        slice[99] = 'a'        // Error: "throw: index out of range"  8
}                                                                     9
```

If you want to extend a slice, there are a couple of built-in functions that make life easier: **append** and **copy**. From [13]:

> *The function **append** appends zero or more values `x` to a slice `s` and returns the resulting slice, with the same type as `s`. If the capacity of `s` is not large enough to fit the additional values, **append** allocates a new, sufficiently large slice that fits both the existing slice elements and the additional values. Thus, the returned slice may refer to a different underlying array.*

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

❶ append a single element, `s1 == []int{0, 0, 2}`;

❷ append multiple elements, `s2 == []int{0, 0, 2, 3, 5, 7}`;

❸ append a slice, `s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}`. Note the three dots!

And

> *The function **copy** copies slice elements from a source `src` to a destination `dst` and returns the number of elements copied. Source and destination may overlap. The number of elements copied is the minimum of **len**(src) and **len**(dst).*

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

Maps

Many other languages have a similar type 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). In the following listing we define a **map** which converts from a **string** (month abbreviation) to an **int** – the number of days in that month. The generic way to define a map is with: `map[<from type>]<to type>`

```
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,    ⟵ The comma here is required
}
```

Note to use **make** when only declaring a **map**: `monthdays := make(map[string]int)`

For indexing (searching) in 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, which returns the key and corresponding value with each invocation.

```
year := 0
for _, days := range monthdays {        ⟵ Key is not used, hence _, days
    year += days
}
fmt.Printf("Numbers of days in a year: %d\n", year)
```

Adding elements to the **map** would be done as:

```
monthdays["Undecim"] = 30           ⟵ Add a month
monthdays["Feb"]     = 29           ⟵ Overwrite entry - for leap years
```

To test for existence , you would use the following[26]:

```
var value int
var present bool

value, present = monthdays["Jan"]     ⟵ If exist, present has the value true
                                      ⟵ Or better and more Go like
v, ok := monthdays["Jan"]             ⟵ Hence, the "comma ok" form
```

And finally you can remove elements from the **map**:

```
delete(monthdays, "Mar")               ⟵ Deletes "Mar", always rainy anyway
```

In general the syntax `delete(m, x)` will delete the map entry retrieved by the expression `m[x]`.

Exercises

Q2. (0) For-loop

1. Create a simple 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.

Q3. (0) FizzBuzz

1. Solve this problem, called the Fizz-Buzz [30] 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 the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".*

Q4. (1) Strings

1. Create a Go program that prints the following (up to 100 characters):

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
...
```

2. Create a program that counts the number of characters in this string:

```
asSASA ddd dsjkdsjs dk
```

   In addition, make it output the number of bytes in that string. *Hint*: Check out the *utf8* package.

3. Extend the program from the previous question to replace the three runes at position 4 with 'abc'.

4. Write a Go program that reverses a string, so "foobar" is printed as "raboof". *Hint*: You will need to know about conversion; skip ahead to section "Conversions" on page 63.

Q5. (1) Average

1. Write code to calculate the average of a **`float64`** slice. In a later exercise (Q6 you will make it into a function.

Answers

A2. (0) For-loop

1. There are a multitude of possibilities, one of the solutions could be:

Listing 2.6. Simple for loop

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {     ⟵ See section For on page 15
                fmt.Printf("%d\n", i)
        }
}
```

Let's compile this and look at the output.

```
% go run for.go
% ./for
0
1
.
.
.
9
```

2. Rewriting the loop results in code that should look something like this (only showing the `main`-function):

```
func main() {
        i := 0              ⟵ Define our loop variable
I:                          ⟵ Define a label
        fmt.Printf("%d\n", i)
        i++
        if i < 10 {
                goto I      ⟵ Jump back to the label
        }
}
```

3. The following is one possible solution:

Listing 2.7. For loop with an array

```
func main() {
        var arr [10]int     ⟵ Create an array with 10 elements
        for i := 0; i < 10; i++ {
                arr[i] = i     ⟵ Fill it one by one
        }
        fmt.Printf("%v", arr)     ⟵ With %v Go prints the type
}
```

You could even do this in one fell swoop by using a composite literal:

```
a := [...]int{0,1,2,3,4,5,6,7,8,9}     ⟵ With [...] you let Go count
fmt.Printf("%v\n", a)
```

A3. (0) FizzBuzz

1. A possible solution to this simple problem is the following program.

Listing 2.8. Fizz-Buzz

```
package main

import "fmt"

func main() {
        const (
                FIZZ = 3 ❶
                BUZZ = 5
        )
        var p bool      ❷
        for i := 1; i < 100; i++ { ❸;
                p = false
                if i%FIZZ == 0 { ❹
                        fmt.Printf("Fizz")
                        p = true
                }
                if i%BUZZ == 0 { ❺
                        fmt.Printf("Buzz")
                        p = true
                }
                if !p { ❻
                        fmt.Printf("%v", i)
                }
                fmt.Println() ❼
        }
}
```

❶ Define two constants to make the code more readable. See section ”Constants”;

❷ Holds if we already printed something;

❸ for-loop, see section ”For”

❹ If divisible by FIZZ, print ”Fizz”;

❺ And if divisble by BUZZ, print ”Buzz”. Note that we have also taken care of the FizzBuzz case;

❻ If neither FIZZ nor BUZZ printed, print the value;

❼ Format each output on a new line.

A4. (1) Strings

1. This program is a solution:

Listing 2.9. Strings

```
package main

import "fmt"

func main() {
        str := "A"
        for i := 0; i < 100; i++ {
                fmt.Printf("%s\n", str)
                str = str + "A"     ⟵ String concatenation
        }
}
```

2. To answer this question we need some help from the *unicode/utf8* package. First we check the documentation with `go doc unicode/utf8 | less`. When we read the documentation we notice `func RuneCount(p []byte)int`. Secondly we can convert *string* to a `byte` slice with

```
str := "hello"
b   := []byte(str)     ⟵ Conversion, see page 63
```

Putting this together leads to the following program.

Listing 2.10. Runes in strings

```
package main

import (
        "fmt"
        "unicode/utf8"
)

func main() {
        str := "dsjkdshdjsdh....js"
        fmt.Printf("String %s\nLength: %d, Runes: %d\n", str,
                len([]byte(str)), utf8.RuneCount([]byte(str)))
}
```

3. Reversing a string can be done as follows. We start from the left (`i`) and the right (`j`) and swap the characters as we see them:

Listing 2.11. Reverse a string

```
import "fmt"

func main() {
        s := "foobar"
        a := []byte(s)     ⟵ Again a conversion
        // Reverse a
        for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
                a[i], a[j] = a[j], a[i]     ⟵ Parallel assignment
        }
        fmt.Printf("%s\n", string(a))     ⟵ Convert it back
}
```

A5. (1) Average

1. The following code calculates the average.

```
sum := 0.0
switch len(xs) {
case 0:                 ❶
        avg = 0
default:                ❷
        for _, v := range xs {
                sum += v
        }
        avg = sum / float64(len(xs)) ❸
}
```

❶ If the length is zero, we return 0;

❷ Otherwise we calculate the average;

❸ We have to convert the value to a **`float64`** to make the division work.

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. A function is declared as follows:

Listing 3.1. A function declaration

```
type mytype int     ← New type, see chapter 5

func (p mytype) funcname(q int) (r,s int) { return 0,0 }
     ❶          ❷       ❸       ❹          ❺
❶ The keyword func is used to declare a function;

❶ A function can optionally be bound to a specific type. This is called the receiver. A function with a receiver is a method. This will be explored in chapter 6;

❷ funcname is the name of your function;

❸ The variable q of type int is the input parameter. The parameters are passed pass-by-value meaning they are copied;

❹ The variables r and s are the named return parameters for this function. Functions in Go can have multiple return values, see section "Multiple return values" on page 32. 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;

❺ This is the function's body. Note that return is a statement so the braces around the parameter(s) are optional.
```

Here are two examples. On the left is a function without a return value, while on the right is a simple function that returns its input.

```
func subroutine(in int) {
    return
}
```

```
func identity(in int) int {
    return in
}
```

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 disallows nested functions, but you can work around this with anonymous functions. See section “Functions as values” on page 35 in this chapter.

Recursive functions work just as in other languages:

Listing 3.2. Recursive function

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

This prints: `9 8 7 6 5 4 3 2 1 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.

Listing 3.3. Local scope

```
package main

var a = 6

func main() {
    p()
    q()
    p()
}

func p() {
    println(a)
}

func q() {
    a := 5          ← Definition
    println(a)
}
```

Listing 3.4. Global scope

```
package main

var a = 6

func main() {
    p()
    q()
    p()
}

func p() {
    println(a)
}

func q() {
    a = 5     ← Assignment
    println(a)
}
```

In listing 3.3 we introduce a local variable `a` in the function `q()`. The local `a` is only visible in `q()`. This is why the code will print: `656`. In listing 3.4 no new variables are introduced, there is only a global `a`. Assigning a new value to `a` will be globally visible. This code will print: `655`

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

Listing 3.5. Scope when calling functions from functions

```
package main

var a int

func main() {
        a = 5
        println(a)
        f()
}

func f() {
        a := 6
        println(a)
        g()
}

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

The output will be: `565`. A *local* variable is *only* valid when we are executing the function in which it is defined.

Multiple return values

One of Go's unusual (for compiled languages) features is that functions and methods can return multiple values (Python and Perl can do this too). This can be used to improve on a couple of clumsy idioms in C programs: in-band error returns (such as -1 for `EOF`) and modifying an argument. In Go, `Write` returns a count and an error: "Yes, you wrote some bytes but not all of them because you filled the device". The signature of `*File.Write` in package *os* is:

```
func (file *File) Write(b []byte) (n int, err error)
```

and as the documentation says, it returns the number of bytes written and a non-`nil` `error` when `n != len(b)`. This is a common style in Go.

In the absence of tuples as a native type, multiple return values are the next best thing. You can return precisely what you want without overloading the domain space with special values to signal errors.

Named result parameters

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 [a].

The names are not mandatory but they can make code shorter and clearer: *they are documentation*. If we name the results of `nextInt` it becomes obvious which returned **`int`** is which.

```
func nextInt(b []byte, pos int) (value, nextPos int) { /* ... */ }
```

Because named results are initialized and tied to an unadorned **`return`**, they can simplify as well as clarify. Here's a version of `io.ReadFull` that uses them well:

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:len(buf)]
    }
    return
}
```

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:

Listing 3.6. Without defer

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

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

A lot of code is repeated here. To overcome this Go has the **`defer`** statement. After **`defer`** you specify a function which is called just *before* the current function exits.

The code above could be rewritten as follows. This makes the function more readable, shorter and puts the `Close` right next to the `Open`.

[a] This is a motto of Go; "Do *more* with *less* code"

Listing 3.7. With defer

```
func ReadWrite() bool {
    file.Open("file")
    defer file.Close()      ⟵ file.Close() is added to the defer list
    // Do your thing
    if failureX {
        return false        ⟵ Close() is now done automatically
    }
    if failureY {
        return false        ⟵ And here too
    }
    return true
}
```

You can put multiple functions on the "deferred list", like this example from [11]:

```
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[b], i.e:

Listing 3.8. Function literal

```
defer func() {
        /* ... */
}()                 ⟵ () is needed here
```

Or this example which makes it easier to understand why and where you need the braces:

Listing 3.9. Function literal with parameters

```
defer func(x int) {
        /* ... */
}(5)                ⟵ Give the input variable x the value 5
```

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

Listing 3.10. Access return values within defer

```
func f() (ret int) {        ⟵ ret is initialized with zero
        defer func() {
                ret++       ⟵ Increment ret with 1
        }()
        return 0            ⟵ 1 not 0 will be returned!
}
```

[b] A function literal is sometimes called a closure.

Variadic parameters

Functions that take a variable number of parameters are known as variadic functions. To declare a function as variadic:

```
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 the type **`int`**. Inside your function's body the variable `arg` is a slice of ints:

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

If you don't specify the type of the variadic argument it defaults to the empty interface `interface{}` (see chapter 6). 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...)      ⟵ Pass it as-is
    myfunc2(arg[:2]...)      ⟵ Slice it
}
```

Functions as values

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

Listing 3.11. Anonymous function

```
func main() {
        a := func() {         ⟵ Define a nameless function and assign to a
          println("Hello")
        }                     ⟵ No () here
        a()                   ⟵ Call the function
}
```

If we use `fmt.Printf("%T\n", a)` to print the type of `a`, it prints `func()`.

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

Listing 3.12. Functions as values in maps

```
var xs = map[int]func() int{
    1: func() int { return 10 },
    2: func() int { return 20 },
    3: func() int { return 30 },     ⟵ 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 exercise Q12 on page 38).

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) {           ⟵ Function returns nothing
    fmt.Printf("%v\n", x)        ⟵ Just print it
}
```

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 will hold the function
    f(y)        ⟵ Call the callback f with y
}
```

Panic and recovering

Go does not have an exception mechanism, like that in Java for instance: 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?

The following description was taken from [10]:

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 *run-time 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[c]:

```
func throwsPanic(f func()) (b bool) { ❶
    defer func() { ❷
        if x := recover(); x != nil {
            b = true
        }
    }()
```

[c] Copied from a presentation of Eleanor McHugh.

```
    f() ❷
    return ❸
}
```

❶ We define a new function `throwsPanic` that takes a function as an argument (see "Functions as values"). It returns true if `f` panics when run, else false;

❶ We 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;

❷ Execute the function we received as the argument;

❸ Return the value of `b`. Because `b` is a named return parameter (page 32), we don't specify `b`.

Exercises

Q6. (0) Average

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

Q7. (0) Integer ordering

1. Write a function that returns its (two) parameters in the right, numerical (ascending) order:
   `f(7,2)` → `2,7`
   `f(2,7)` → `2,7`

Q8. (1) Scope

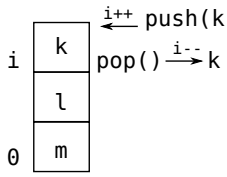
1. What is wrong with the following program?

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                fmt.Printf("%v\n", i)
        }
        fmt.Printf("%v\n", i)
}
```

Q9. (1) 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 3.1. A simple LIFO stack



2. Bonus. Write a `String` method which converts the stack to a string representation. This way you can print the stack using: `fmt.Printf("My stack %v\n", stack)`
   The stack in the figure could be represented as: `[0:m] [1:l] [2:k]`

Q10. (1) Var args

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

Q11. (1) 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.

Q12. (1) Map function A `map()`-function is a function that takes a function and a list. The function is applied to each member in the list and a new list containing these calculated values is returned. Thus:

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

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

Q13. (0) Minimum and maximum

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

Q14. (1) Bubble sort

1. Write a function that performs a bubble sort on a slice of ints. From [31]:

   > *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.*

[31] 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
```

Q15. (1) 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 section Callbacks on page 36 for information about this topic.

2. Generalize the function from 1, and create a `plusX(x)` which returns functions that add `x` to an integer.

Answers

A6. (0) Average

1. The following function calculates the average:

Listing 3.13. Average function in Go

```
func average(xs []float64) (avg float64) { ❶
        sum := 0.0
        switch len(xs) {
        case 0:                 ❷
                avg = 0
        default:                ❸
                for _, v := range xs {
                        sum += v
                }
                avg = sum / float64(len(xs)) ❹
        }
        return  ❺
}
```

❶ We use a named return parameter;

❷ If the length is zero, we return 0;

❸ Otherwise we calculate the average;

❹ We have to convert the value to a **`float64`** to make the division work;

❺ We have an avarage, return it.

A7. (0) Integer ordering

1. Here we can use the multiple return values (section “Multiple return values”) from Go:

```
func order(a, b int) (int, int) {
        if a > b {
                return b,a
        }
        return a,b
}
```

A8. (1) Scope

1. The program does not even compile, because `i` on line 9 is not defined: `i` is only defined within the **`for`**-loop. To fix this the function `main()` should become:

```
func main() {
        var i int
        for i = 0; i < 10; i++ {
                fmt.Printf("%v\n", i)
```

```
        }
        fmt.Printf("%v\n", i)
}
```

Now `i` is defined outside the **`for`**-loop and still visible afterwards. This code will print the numbers 0 through 10.

A9. (1) Stack

1. First we define a new type that represents a stack; we need an array (to hold the keys) and an index, which points to the last element. Our small stack can only hold 10 elements.

```
type stack struct {     ⟵ stack is not exported
    i    int
    data [10]int
}
```

Next we need the `push` and `pop` functions to actually use the thing. *First we show the* wrong *solution!* In Go data passed to functions is *passed-by-value* meaning a copy is created and given to the function. The first stab for the function `push` could be:

```
func (s stack) push(k int) {     ⟵ Works on copy of argument
        if s.i+1 > 9 {
                return
        }
        s.data[s.i] = k
        s.i++
}
```

The function works on the `s` which is of the type **`stack`**. To use this we just call `s.push(50)`, to push the integer 50 on the stack. But the push function gets a copy of `s`, so it is *not* working the *real* thing. Nothing gets pushed to our stack this way, for example the following code:

```
var s stack     ⟵ make s a simple stack variable
s.push(25)
fmt.Printf("stack %v\n", s);
s.push(14)
fmt.Printf("stack %v\n", s);
```

prints:

```
stack [0:0]
stack [0:0]
```

To solve this we need to give the function `push` a pointer to the stack. This means we need to change `push` from

`func (s stack)push(k int)` → `func (s *stack)push(k int)`

We should now use `new()` (see "Allocation with new" in chapter 5) to create a *pointer* to a newly allocated **`stack`**, so line 1 from the example above needs to be `s := new(stack)`

And our two functions become:

```
func (s *stack) push(k int) {
        s.data[s.i] = k
        s.i++
}

func (s *stack) pop() int {
        s.i--
        return s.data[s.i]
}
```

Which we then use as follows

```
func main() {
        var s stack
        s.push(25)
        s.push(14)
        fmt.Printf("stack %v\n", s)
}
```

2. While this was a bonus question, having the ability to print the stack was very valuable when writing the code for this exercise. According to the Go documentation `fmt.Printf("%v")` can print any value (`%v`) that satisfies the `Stringer` interface. For this to work we only need to define a `String()` function for our type:

Listing 3.14. stack.String()

```
func (s stack) String() string {
        var str string
        for i := 0; i <= s.i; i++ {
                str = str + "[" +
                        strconv.Itoa(i) + ":" + strconv.Itoa(s.data[i
                             ]) + "]"
        }
        return str
}
```

A10. (1) Var args

1. For this we need the `...`-syntax to signal we define a function that takes an arbitrary number of arguments.

Listing 3.15. A function with variable number of arguments

```
package main

import "fmt"

func main() {
        printthem(1, 4, 5, 7, 4)
        printthem(1, 2, 4)
}
```

```
func printthem(numbers ... int) {     ← numbers is now a slice of ints
        for _, d := range numbers {
                fmt.Printf("%d\n", d)
        }
}
```

A11. (1) Fibonacci

1. The following program calculates Fibonacci numbers:

Listing 3.16. Fibonacci function in Go

```
package main

import "fmt"

func fibonacci(value int) []int {
        x := make([]int, value) ❶
        x[0], x[1] = 1, 1 ❷
        for n := 2; n < value; n++ {
                x[n] = x[n-1] + x[n-2] ❸
        }
        return x ❹
}

func main() {
        for _, term := range fibonacci(10) { ❺
                fmt.Printf("%v ", term)
        }
}
```

❶ We create an **array** to hold the integers up to the value given in the function call;

❷ Starting point of the Fibonacci calculation;

❸ $x_n = x_{n-1} + x_{n-2}$;

❹ Return the *entire* array;

❺ Using the **range** keyword we "walk" the numbers returned by the Fibonacci function. Here up to 10. And we print them.

A12. (1) Map function

Listing 3.17. A Map function

1.
```
func Map(f func(int) int, l []int) []int {
        j := make([]int, len(l))
        for k, v := range l {
                j[k] = f(v)
        }
```

```
        return j
}

func main() {
        m := []int{1, 3, 4}
        f := func(i int) int {
                return i * i
        }
        fmt.Printf("%v", (Map(f, m)))
}
```

2. Answer to question but now with strings

A13. (0) Minimum and maximum

1. This function returns the largest int in the slice `l`:

```
func max(l []int) (max int) {    ❶
        max = l[0]
        for _, v := range l {    ❷
                if v > max {     ❸
                        max = v
                }
        }
        return                   ❹
}
```

❶ We use a named return parameter;

❷ Loop over `l`. The index of the element is not important;

❸ If we find a new maximum, remember it;

❹ A "lone" return, the current value of `max` is now returned.

2. This function returns the smallest int in the slice `l`. It is almost identical to `max`:

```
func min(l []int) (min int) {
        min = l[0]
        for _, v := range l {
                if v < min {
                        min = v
                }
        }
        return
}
```

The interested reader may combine `max` and `min` into one function with a selector that lets you choose between the minimum or the maximum, or one that returns both values.

A14. (1) Bubble sort

1. Bubble sort isn't terribly efficient, for n elements it scales $O(n^2)$. See QuickSort [23] for a better sorting algorithm.

   But bubble sort is easy to implement:

   Listing 3.18. Bubble sort

```
func main() {
        n := []int{5, -1, 0, 12, 3, 5}
        fmt.Printf("unsorted %v\n", n)
        bubblesort(n)
        fmt.Printf("sorted %v\n", n)
}

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]
                        }
                }
        }
}
```

   Because a slice is a reference type the `bubblesort` function works and does not need to return a sorted slice.

A15. (1) Functions that return functions

1.
```
func main() {
        p2 := plusTwo()
        fmt.Printf("%v\n",p2(2))
}

func plusTwo() func(int) int { ❶
        return func(x int) int { return x + 2 } ❶
}
```

   ❶ Define a new function that returns a function. See how you you can just write down what you mean;

   ❶ Function literals at work, we define the +2–function right there in the return statement.

2. Here we use a closure:

```
func plusX(x int) func(int) int { ❶
        return func(y int) int { return x + y } ❶
}
```

   ❶ Again define a function that returns a function;

   ❶ Use the *local* variable `x` in the function literal.

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.

Listing 4.1. A small package

```
package even                ⟵ Start our own namespace

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

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

Names that start with a capital letter are *exported* and may be used outside your package (more on that later).

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).

```
% mkdir $GOPATH/src/even     ⟵ Create top-level directory
% cp even.go $GOPATH/src/even      ⟵ Copy the package file
% go build                         ⟵ Build it
% go install                       ⟵ Install it to ../pkg
```

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

Listing 4.2. Use of the even package

```
package main

import (                ❶
        "even"          ❷
        "fmt"           ❸
)

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

❶ Import the following packages;

❶ The *local* package *even* is imported here;

❷ The official *fmt* package gets imported;

❸ Use the function from the *even* package. The syntax for accessing a function from a package is `<package>.Function()`.

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

In Go, a function from a package is exported (visible outside the package, i.e. public) when the first letter of the function name is a capital, hence the function name `Even`. If we change our `myeven.go` on line 10 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:10: cannot refer to unexported name even.odd
```

To summarize:

- Public functions begin with a *capital* letter;
- Private functions begin with a *lowercase* letter.

This convention holds true for other labels (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

Names are as important in Go as in any other language. In some cases they even have semantic effect: for instance, the visibility of a name outside its package is determined by the case of its first character. It’s therefore worth spending a little time talking about naming conventions in Go programs.

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`.

Package names

When a package is imported (with **`import`**), the package name becomes the accessor for the contents. After

```
import "bytes"
```

the importing package can talk about `bytes.Buffer`. It's helpful if everyone using the package can use the same name to refer to its contents, so the package name should be good: short, concise and evocative. By convention packages are given lower-case single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity (since everyone using your package will be typing its name), and don't worry about collisions a priori.

The package name is only the default name for imports. You can override the default accessor by providing your own name to the import statement:

```
import bar "bytes"
```

The function `Buffer` is now accessed as `bar.Buffer`. This means that the package name does not need to be globally unique; in the rare case of a collision the importing code can choose a different name to use locally. In any case confusion is rare because the file name in the import determines just which package is being used.

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

The importer of a package will use the name to refer to its contents, so exported names in the package can use that fact to avoid stutter. 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. Moreover, because imported entities are always addressed by their package name, `bufio.Reader` does not conflict with `io.Reader`. Similarly, the function to make new instances of `ring.Ring` (package *container/ring*) —which is the definition of a constructor in Go—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. If the name represents something intricate or subtle, it's usually better to write a helpful doc comment than to attempt to put all the information into the name.

Finally, the convention in Go is to use MixedCaps or mixedCaps rather than underscores to write multi-word names.

Documenting packages

This text is copied from [11].

Every package should have a *package comment*, a block comment preceding the **`package`** clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the `go doc` page and should set up the detailed documentation that follows. An example 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. An example from the *fmt* package:

```
// Printf formats according to a format specifier and writes to standard
// output. It returns the number of bytes written and any write error
// encountered.
func Printf(format string, a ...interface) (n int, err error)
```

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)     ← Test<Capital>restOftheName
```

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 [14]. 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:

Listing 4.3. Test file for even package

```
package even                                                    1

import "testing"                                                3

func TestEven(t *testing.T) {                                   5
        if ! Even(2) {                                          6
                t.Log("2 should be even!")                      7
                t.Fail()                                        8
        }                                                       9
}                                                               10
```

Note that we use **`package`** `even` on line 1 - the tests fall in the same namespace as the package we are testing. This is not only convenient, but also allows tests of unexported functions and structures. We then import the *testing* package, and on line 5 we define the only test function in this file. The displayed Go code should not hold any surprises: we check if the `Even` function works OK. Now, the moment we've 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.

Useful packages

The standard Go repository includes a huge number of packages and it is even possible to install more alongside your current Go installation. It is very enlightening to browse the `$GOROOT/src/pkg` directory and look at the packages. We cannot comment on each package, but the following are worth a mention: [a]

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. *See "Command line arguments" on page 92.*

[a] The descriptions are copied from the packages' `go doc`. Extra remarks are type set in italic.

encoding/json
: The *encoding/json* package implements encoding and decoding of JSON objects as defined in RFC 4627 [5].

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.*

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, section "Introspection and reflection".

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

Exercises

Q16. (0) Stack as package

1. See the Q9 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`.

Q17. (2) Calculator

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

Answers

A16. (0) Stack as package

1. There are a few details that should be changed to make a proper package for our stack. First, the exported functions should begin with a capital letter and so should **`Stack`**. The package file is named `stack-as-package.go` and contains:

Listing 4.4. Stack in a package

```
package stack

// Stack holds the items.
type Stack struct {
        i    int
        data [10]int
}

// Push pushes an item on the stack.
func (s *Stack) Push(k int) {
        s.data[s.i] = k
        s.i++
}

// Pop pops an item from the stack.
func (s *Stack) Pop() (ret int) {
        s.i--
        ret = s.data[s.i]
        return
}
```

2. To make the unit testing work properly you need to do some preparations. We'll come to those in a minute. First the actual unit test. Create a file with the name `pushpop_test.go`, with the following contents:

Listing 4.5. Push/Pop test

```
package stack

import "testing"

func TestPushPop(t *testing.T) {
        c := new(Stack)
        c.Push(5)
        if c.Pop() != 5 {
                t.Log("Pop doesn't give 5")
                t.Fail()
        }
}
```

For `go test` to work we need to put our package files in a directory under `$GOPATH/src`:

```
% mkdir $GOPATH/src/stack
% cp pushpop_test.go $GOPATH/src/stack
% cp stack-as-package.go $GOPATH/src/stack
```

Yields:

```
% go test stack
ok      stack   0.001s
```

A17. (2) Calculator

1. This is one answer:

Listing 4.6. A (rpn) calculator

```
package main

import ( "bufio"; "os"; "strconv"; "fmt")

var reader *bufio.Reader = bufio.NewReader(os.Stdin)
var st = new(Stack)

type Stack struct {
        i    int
        data [10]int
}

func (s *Stack) push(k int) {
        if s.i+1 > 9 { return }
        s.data[s.i] = k
        s.i++
}

func (s *Stack) pop() (ret int) {
        s.i--
        if s.i < 0 { s.i = 0; return }
        ret = s.data[s.i]
        return
}

func main() {
        for {
                s, err := reader.ReadString('\n')
                var token string
                if err != nil { return }
                for _, c := range s {
                        switch {
                        case c >= '0' && c <= '9':
                                token = token + string(c)
                        case c == ' ':
                                r, _ := strconv.Atoi(token)
                                st.push(r)
                                token = ""
                        case c == '+':
```

```
                        fmt.Printf("%d\n", st.pop()+st.pop())
                    case c == '*':
                        fmt.Printf("%d\n", st.pop()*st.pop())
                    case c == '-':
                        p := st.pop()
                        q := st.pop()
                        fmt.Printf("%d\n", q-p)
                    case c == 'q':
                        return
                    default:
                        //error
                    }
                }
            }
        }
```

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

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:

Listing 5.1. Use of a pointer

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

var i int               ⟵ Declare integer variable i
p = &i                  ⟵ Make p point to i

fmt.Printf("%v", p)     ⟵ Prints something like 0x7ff96b81c000a
```

De-referencing a pointer is done by prefixing the pointer variable with '*':

Listing 5.2. Dereferencing a pointer

```
p = &i                      ⟵ Take the address of i
*p = 8                      ⟵ Change the value of i
fmt.Printf("%v\n", *p)      ⟵ Prints 8
fmt.Printf("%v\n", i)       ⟵ Idem
```

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

Allocation

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

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

[a] See exercise 18.

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**. In Go terminology, it returns a pointer to a newly allocated zero value of type **T**. This is important to remember:

> **new** returns a *pointer*.

This means a user of the data structure can create one with **new** and get right to work. For example, 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.

The zero-value-is-useful property works transitively. Consider this type declaration. See section "Defining your own types" on page 61.

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

Values of type **SyncedBuffer** are also ready to use immediately upon allocation or just declaration. In this snippet, both `p` and `v` will work correctly without further arrangement.

```
p := new(SyncedBuffer)      ⟵ Type *SyncedBuffer, ready to use
var v SyncedBuffer          ⟵ Type SyncedBuffer, idem
```

Allocation with make

Back to allocation. 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**, not ***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.

> **make** returns initialized (non zero) *values*.

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)           ⟵ Allocates slice structure; *p == nil
                                    ⟵ Rarely useful
var v  []int = make([]int, 100)     ⟵ v refers to a new array of 100 ints
```

```
var p *[]int = new([]int)           ⟵ Unnecessarily complex
*p = make([]int, 100, 100)

v := make([]int, 100)               ⟵ 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}     ⟵ Create a new File
    return &f                       ⟵ Return the address of f
}
```

It is OK to return the address of a local variable; 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.[b]

[b] Taking the address of a composite literal tells the compiler to allocate it on the heap, not the stack.

```
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.

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 argument"}
```

Defining your own types

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

```
type foo int
```

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:

Listing 5.3. Structures

```
package main
import "fmt"

type NameAge struct {
        name string        ← Not exported
        age  int           ← Not exported
}

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)     ← %s formats a string
```

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[c] fields:

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

If you omit the name for a field, you create an anonymous field, 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 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(in1 *NameAge, in2 int) { /* ... */ }
   ```

   This is (you might have guessed) a *function call*.

2. Create a function that works on the type (see *receiver* in listing 3.1):

   ```
   func (in1 *NameAge) doSomething(in2 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 [13]:

> *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:

[c] Yes, four (4).

```
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 [13, section "Type Declarations"]. 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 implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }
```

We now create two types in two different manners:

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

Now `NewMutux` 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`. In the words of [13]:

> *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.

*Table 5.1. Valid conversions, **`float64`** works the same as **`float32`***

From	`xb []byte`	`xi []int`	`xr []rune`	`s string`	`f float32`	`i int`
To						
`[]byte`	×			`[]byte(s)`		
`[]int`		×		`[]int(s)`		
`[]rune`			×	`[]rune(s)`		
`string`	`string(xb)`	`string(xi)`	`string(xr)`	×		
`float32`					×	`float32(i)`
`int`					`int(f)`	×

- 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;
- 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

Q18. (1) Pointer arithmetic

1. In the main text on page 58 there is the following text:

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

   When you increment a value like this, for which types will it work?

2. Why doesn't it work for all types?

Q19. (2) Map function with interfaces

1. Use the answer from exercise Q12, but now make it generic using interfaces. Make it at least work for ints and strings.

Q20. (1) 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
   }
   ```

Q21. (1) Linked List

1. Make use of the package *container/list* to create a (double) 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 in question 1

Q22. (1) Cat

1. Write a program which mimics the Unix program `cat`. For those who don't know this program, the following invocation displays the contents of the file `blah`:

   ```
   % cat blah
   ```

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

Q23. (2) Method calls

1. Suppose we have the following program. Note the package *container/vector* was once part of Go, but has been 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 correct then?

Answers

A18. (1) Pointer arithmetic

1. This will only work for pointers to point to numerical (**`int, uint`**, etc) values.
2. The ++ is only defined for numerical types and because there is no operator overloading in Go it fails (compilation error) otherwise.

A19. (2) Map function with interfaces

Listing 5.4. A generic map function in Go

```
1. package main

   import "fmt"

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

   func mult2(f e) e {
           switch f.(type) {
           case int:
                   return f.(int) * 2
           case string:
                   return f.(string) + f.(string) + f.(string) + f.(
                        string)
           }
           return f
   }

   func Map(n []e, f func(e) e) []e {
           m := make([]e, len(n))
           for k, v := range n {
                   m[k] = f(v)
           }
           return m
   }

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

A20. (1) Pointers

1. In first line: **`var`** `p1 Person` allocates a `Person`-*value* to `p1`. The type of `p1` is **`Person`**.

   The second line: `p2 :=` **`new`**`(Person)` allocates memory and assigns a *pointer* to `p2`. The type of `p2` is **`*Person`**.

2. In the second function, `x` points to a new (heap-allocated) variable `t` which contains a copy of whatever the actual argument value is.

   In the first function, `x` points to the same thing that `t` does, which is the same thing that the actual argument points to.

   So in the second function, we have an "extra" variable containing a copy of the interesting value.

A21. (1) Linked List

1.

2.

A22. (1) Cat

1. The following is implemention of `cat` which also supports a n flag to number each line.

Listing 5.5. A cat program

```
package main

❶
import (
        "io"
        "os"
        "fmt"
        "bufio"
        "flag"
)

var numberFlag = flag.Bool("n", false, "number each line") ❶

❷
func cat(r *bufio.Reader) {
        i := 1
        for {
                buf, e := r.ReadBytes('\n')      ❸
                if e == io.EOF {                 ❹
                        break
                }
                if *numberFlag {                 ❺
                        fmt.Fprintf(os.Stdout, "%5d  %s", i, buf)
                        i++
                } else {                         ❻
                        fmt.Fprintf(os.Stdout, "%s", buf)
                }
```

```
        }
        return
}

func main() {
        flag.Parse()
        if flag.NArg() == 0 {
                cat(bufio.NewReader(os.Stdin))
        }
        for i := 0; i < flag.NArg(); i++ {
                f, e := os.Open(flag.Arg(i), os.O_RDONLY, 0)
                if e != nil {
                        fmt.Fprintf(os.Stderr, "%s: error reading from
                              %s: %s\n",
                                os.Args[0], flag.Arg(i), e.String())
                        continue
                }
                cat(bufio.NewReader(f))
        }
}
```

❶ Include all the packages we need;

❶ Define a new flag ”n”, which defaults to off. Note that we get the help for free;

❷ Start the function that actually reads the file’s contents and displays it;

❸ Read one line at the time;

❹ Or stop if we hit the end;

❺ If we should number each line, print the line number and then the line itself;

❻ Otherwise we could just print the line.

A23. (2) Method calls

1. The type of `k1` is **`vector.IntVector`**. Why? We use a composite literal (the `{}`), so we get a value of that type back. The variable `k2` is of ***`vector.IntVector`**, because we take the address (`&`) of the composite literal. And finally `k3` has also the type ***`vector.IntVector`**, because `new` returns a pointer to the type.

2. The answer is given in [13] in the section “Calls”, where among other things it says:

   > *A method call `x.m()` is valid if the method set of (the type of) `x` contains `m` and the argument list can be assigned to the parameter list of `m`. If `x` is addressable and `&x`’s method set contains `m`, `x.m()` is shorthand for `(&x).m()`.*

   In other words because `k1` is addressable and ***`vector.IntVector`** *does* have the `Push` method, the call `k1.Push(2)` is translated by Go into `(&k1).Push(2)` which makes the type system happy again (and you too — now you know this).[d]

[d] Also see section “Methods” in this chapter.

6 Interfaces

I have this phobia about having my body penetrated surgically. You know what I mean?

eXistenZ
TED PIKUL

The following text is from [29]. Written by Ian Lance Taylor — one of the authors of Go.

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`**.

Listing 6.1. Defining a struct and methods on it

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

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) {  ❶
    fmt.Println(p.Get()) ❷
    p.Put(1) ❸
}
```

❶ Declare a function that takes an interface type as the argument;

❷ As `p` implements interface **`I`** it *must* have the `Get()` method;

❸ Same holds for the `Put()` method.

Here the variable `p` holds a value of interface type. Because **`S`** implements **`I`**, we can call `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 code above in listing 6.1. 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[33]. 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 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) { ❶
        case *S: ❷
        case *R: ❸
        case S: ❹
        case R: ❺
        default: ❻
    }
}
```

❶ The type switch. Use **(type)** in a **switch** statement. We store the type in the variable `t`;

❷ The actual type of `p` is a pointer to **S**;

❸ The actual type of `p` is a pointer to **R**;

❹ The actual type of `p` is a **S**;

❺ The actual type of `p` is a **R**;

❻ It's another type that implements **I**.

Using **(type)** outside a **switch** is illegal. A type switch isn't the only way to discover the type at *run-time*. You can also use a "comma, ok" form to see if an interface type implements a specific interface:

```
if t, ok := something.(I); ok {
   // something implements the interface I
   // t is the type it has
}
```

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:

Listing 6.2. A function with an empty interface 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:

Listing 6.3. Failing to implement an interface

```
i := 5              ⟵ Make i a ``lousy'' 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). 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:

Listing 6.4. Failure extending built-in types

```
func (i int) Emit() {
  fmt.Printf("%d", i)
}
```

```
cannot define new methods
on non-local type int
```

Listing 6.5. Failure extending non-local types

```
func (a *net.AddrError) Emit() {
  fmt.Printf("%v", a)
}
```

```
cannot define new methods
on non-local type net.AddrError
```

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 [13]:

> *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.*

Pointers to interfaces

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 the release `2010-10-13` 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.*

From the [12]. If not for this restriction, this code:

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

would copy standard input into a copy of `buf`, not into `buf` itself. This is almost never the desired behavior.

Interface names

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

Text copied from [11].

A sorting example

Recall the Bubblesort exercise (Q14), 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{}) { ❶
    switch i.(type) {        ❷
        case string:         ❸
            // ...
        case int:
            // ...
    }
    return /* ... */ ❹
}
```

❶ Our function will receive a slice of empty interfaces;

❷ Using a type switch we find out what the actual type is of the input;

❸ And then sort accordingly;

❹ Return 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. To keep a long story short: Go does not (implicitly) convert slices for you.

The full mailing list discussion on this subject can be found at [19].

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:

1. 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 method
}
```

2. Define new types for the slices we want to sort. Note that we declare slice types;

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

3. 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] }
```

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

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

❶ `x` is now of the `Sorter` type;

❷ Using the defined functions, we implement Bubblesort.

We can now use your 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.

Listing 6.6. Introspection using reflection

```
type Person struct {
    name string "namestr"     ⟵ "namestr" is the tag
    age  int
}

p1 := new(Person)       ⟵ new returns a pointer to Person
ShowTag(p1)             ⟵ ShowTag() is now called with this pointer

func ShowTag(i interface{}) {
 switch t := reflect.TypeOf(i); t.Kind() {     ⟵ Get type, switch on Kind()
 case reflect.Ptr:           ⟵ Its a pointer, hence a reflect.Ptr
        tag := t.Elem().Field(0).Tag
                   ❶      ❷   ❸
```

❶ We are dealing with a **`Type`** and according to the documentation[a]:

[a] `go doc reflect`

```
// Elem returns a type's element type.
// It panics if the type's Kind is not Array, Chan, Map, Ptr, or Slice.
Elem() Type
```

So on `t` we use `Elem()` to get the value the pointer points to;

❶ We have now dereferenced the pointer and are ”inside” our structure. We now use `Field(0)` to access the zeroth field;

❷ 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` 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:

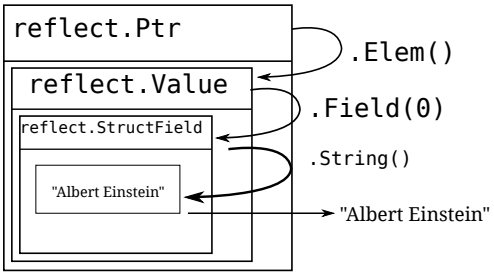
Listing 6.7. Reflection and the type and value

```
func show(i interface{}) {
    switch t := i.(type) {
      case *Person:
        t := reflect.TypeOf(i)      ⟵ Go for type meta data
        v := reflect.ValueOf(i)     ⟵ Go for the actual values
        tag := t.Elem().Field(0).Tag ⓪
        name := v.Elem().Field(0).String() ❶
    }
}
```

⓪ Here we want to get to the “tag”. So we need `Elem()` to redirect the pointer, access the first field and get the tag. Note we operate on `t` a **`reflect.Type`**;

❶ Now we want to get access to the *value* of one of the members and we employ `Elem()` on `v` to do the redirection. Now we have arrived at the structure. Then we go the the first field `Field(0)` and invoke the `String()` method on it.

*Figure 6.1. 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:

Listing 6.8. Reflect with private member

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

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

Listing 6.9. Reflect with public member

```
type Person struct {
 Name string "namestr"     ⟵ Name
 age  int
}

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

The code on the left 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 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.

Exercises

Q24. (1) Interfaces and compilation

1. The code in listing 6.3 on page 72 compiles OK — as stated in the text. But when you run it you'll get a runtime error, so something *is* wrong. Why does the code compile cleanly then?

Q25. (1) Pointers and reflection

1. One of the last paragraphs in section "Introspection and reflection" on page 76, 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?

Q26. (2) Interfaces and max()

1. In exercise Q13 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.

Answers

A24. (1) Interfaces and compilation

1. The code compiles because an integer type implements the empty interface and that is the check that happens at compile time.

   A proper way to fix this is to test if such an empty interface can be converted and, if so, call the appropriate method. The Go code that defines the function `g` in listing 6.2 – repeated here:

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

   Should be changed to become:

   ```
   func g(any interface{}) int {
       if v, ok := any.(I); ok {   // Check if any can be converted
           return v.Get()          // If so invoke Get()
       }
       return -1                   // Just so we return anything
   }
   ```

   If `g()` is called now there are no run-time errors anymore. The idiom used is called "comma ok" in Go.

A25. (1) Pointers and reflection

1. When called with a non-pointer argument the variable is a copy (call-by-value). So you are doing the reflection voodoo on a copy. And thus you are *not* changing the original value, but only this copy.

A26. (2) Interfaces and max()

1. The following program calculates a maximum. It is as generic as you can get with Go.

Listing 6.10. Generic way of calculating a maximum

```
package main

func Less(l, r interface{}) bool { ❶
        switch l.(type) {
        case int:
                if _, ok := r.(int); ok {
                        return l.(int) < r.(int) ❷
                }
        case float32:
                if _, ok := r.(float32); ok {
                        return l.(float32) < r.(float32) ❸
                }
        }
        return false
}
```

```
func main() {
        var a, b, c int = 5, 15, 0
        var x, y, z float32 = 5.4, 29.3, 0.0

        if c = a; Less(a, b) { ❸
                c = b
        }
        if z = x; Less(x, y) { ❹
                z = y
        }
        println(c, z)
}
```

❶ We could have choosen to make the return type of this function a **`interface{}`**, but that would mean that a caller would always have to a type assertion to extra the actual type from the interface;

❶ All parameters are confirmed to be integers. Now perform the comparison;

❷ Parameters are **`float32`**;

❸ Get the maximum of `a` and `b`;

❹ Same for the floats.

7 Concurrency

- "Parallelism is about performance;
- Concurrency is about program design."

Google IO 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 [11]:

> *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)      ⟵ ready() started as goroutine
```

The following idea for a program was taken from [27]. We run a function as two goroutines, the goroutines wait for an amount of time and then print something to the screen. On the lines 14 and 15 we start the goroutines. The `main` function waits long enough, so that both goroutines will have printed their text. Right now we wait for 5 seconds on line 17, but in fact we have no idea how long we should wait until all goroutines have exited.

Listing 7.1. Go routines in action

```
func ready(w string, sec int) {                                    8
        time.Sleep(time.Duration(sec) * time.Second)               9
        fmt.Println(w, "is ready!")                                10
}                                                                  11

func main() {                                                      13
        go ready("Tea", 2)                                         14
        go ready("Coffee", 1)                                      15
        fmt.Println("I'm waiting")                                 16
        time.Sleep(5 * time.Second)                                17
}                                                                  18
```

Listing 7.1 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 line 17) 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.

Listing 7.2. Go routines and a channel

```
var c chan int ❶

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

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

❶ 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;

❷ Send the integer 1 on the channel `c`;

❸ Initialize `c`;

❹ Start the goroutines with the keyword **go**;

❺ Wait until we receive a value from the channel. Note that the value we receive is discarded;

❺ Two goroutines, two values to receive.

There is still some remaining ugliness; we have to read twice from the channel (lines 14 and 15). 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`**. 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 goroutines. We remove the lines 14 and 15 and replace them with the following:

Listing 7.3. Using select

```
L: for {                                                    14
        select {                                            15
        case <-c:                                           16
                i++                                         17
                if i > 1 {                                  18
                        break L                             19
                }                                           20
        }                                                   21
}                                                           22
```

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.

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<-5`) 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:

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

Closing channels

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.

Read-only or write-only channels

TODO
more needs to be written

Exercises

Q27. (1) Channels

1. Modify the program you created in exercise Q2 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. 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.[a]

Q28. (2) Fibonacci II

1. This is the same exercise as the one given page 38 in exercise 11. 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.

[a] You will need the `select` statement.

Answers

A27. (1) Channels

1. A possible program is:

Listing 7.4. Channels in Go

```
package main                                                    1

import "fmt"                                                    3

func main() {                                                   5
        ch := make(chan int)                                    6
        go shower(ch)                                           7
        for i := 0; i < 10; i++ {                               8
                ch <- i                                         9
        }                                                       10
}                                                               11

func shower(c chan int) {                                       13
        for {                                                   14
                j := <-c                                        15
                fmt.Printf("%d\n", j)                           16
        }                                                       17
}                                                               18
```

We start of in the usual way, then at line 6 we create a new channel of ints. In the next line we fire off the function `shower` with the `ch` variable as it argument, so that we may communicate with it. Next we start our for-loop (lines 8-10) and in the loop we send (with **<-**) our number to the function (now a goroutine) `shower`.

In the function `shower` we wait (as this blocks) until we receive a number (line 15). Any received number is printed (line 16) and then continue the endless loop started on line 14.

2. An answer is

Listing 7.5. Adding an extra quit channel

```
package main                                                    1

import "fmt"                                                    3

func main() {                                                   5
        ch := make(chan int)                                    6
        quit := make(chan bool)                                 7
        go shower(ch, quit)                                     8
        for i := 0; i < 10; i++ {                               9
                ch <- i                                         10
        }                                                       11
        quit <- false   // or true, does not matter             12
}                                                               13
```

```
func shower(c chan int, quit chan bool) {                    15
        for {                                                16
                select {                                     17
                case j := <-c:                               18
                        fmt.Printf("%d\n", j)                19
                case <-quit:                                 20
                        break                                21
                }                                            22
        }                                                    23
}                                                            24
```

On line 20 we read from the quit channel and we discard the value we read. We could have used `q := <-quit`, but then we would have used the variable only once — which is illegal in Go. Another trick you might have pulled out of your hat may be: `_ = <-quit`. This is valid in Go, but the Go idiom favors the one given on line 20.

A28. (2) Fibonacci II

1. The following program calculates the Fibonacci numbers using channels.

Listing 7.6. A Fibonacci function in Go

```
package main
import "fmt"

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

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

```
        }()
        return out
}

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

// See sdh33b.blogspot.com/2009/12/fibonacci-in-go.html
```

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`**.

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`.

Listing 8.1. Reading from a file (unbuffered)

```
package main
import "os"

func main() {
        buf := make([]byte, 1024)
        f, _ := os.Open("/etc/passwd") ❶
        defer f.Close() ❷
        for {
                n, _ := f.Read(buf) ❸
                if n == 0 { break } ❹
                os.Stdout.Write(buf[:n]) ❺
        }
}
```

The following is happening here:

❶ Open the file, `os.Open` returns a ***`os.File`**, which implements **`io.Reader`** and **`io.Writer`**;

❷ Make sure we close `f` again;

❸ Read up to 1024 bytes at the time;

❹ We have reached the end of the file;

❺ Write the contents to **`os.Stdout`**

If you want to use buffered IO there is the *bufio* package:

Listing 8.2. Reading from a file (buffered)

```
package main
import ( "os"; "bufio")

func main() {
        buf := make([]byte, 1024)
```

```
        f, _ := os.Open("/etc/passwd") ❶
        defer f.Close()
        r := bufio.NewReader(f) ❷
        w := bufio.NewWriter(os.Stdout)
        defer w.Flush()
        for {
                n, _ := r.Read(buf)      ❸
                if n == 0 { break }
                w.Write(buf[0:n])
        }
}
```

❶ Open the file;

❷ Turn `f` into a buffered `Reader`. `NewReader` expects an **`io.Reader`**, so you might think this will fail. But it does not. *Anything* that has such a `Read()` function implements this interface. And from listing 8.1 we can see that ***`os.File`** indeed does so;

❸ Read from the Reader and write to the Writer, and thus print the file to the screen.

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 only one method: `Read(p []byte) (n int, err error)`. The writing side is (you may have guessed) an **`io.Writer`**, which 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:

```
f, _ := os.Open("/etc/passwd"); defer f.Close()
r := bufio.NewReader(f)     ⟵ Make it a bufio to access the ReadString method
s, ok := r.ReadString('\n')     ⟵ Read a line from the input
// ...     ⟵ s holds the string, with the strings package you can parse it
```

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.

Listing 8.3. Create a directory with the shell

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

Listing 8.4. Create a directory with Go

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

The similarity between these two examples 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") ❶
port := flag.String("port", "53", "Set the query port")       ❷
flag.Usage = func() {                                         ❸
    fmt.Fprintf(os.Stderr, "Usage: %s [OPTIONS] [name ...]\n", os.Args[0])
    flag.PrintDefaults()                                      ❹
}
flag.Parse()                                                  ❺
```

❶ Define a `bool` flag, `-dnssec`. The variable must be a pointer otherwise the package can not set its value;

❷ Idem, but for a `port` option;

❸ Slightly redefine the `Usage` function, to be a little more verbose;

❹ For every flag given, `PrintDefaults` will output the help string;

❺ Parse the flags and fill the variables.

After the flags have been parsed you can used them:

```
if *dnssec {        ← Dereference the dnssec flag variable
    // do something
}
```

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")        ⟵ Create a *cmd
err := cmd.Run()                            ⟵ Run() it
```

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:

```
import "os/exec"

cmd := exec.Command("/bin/ls", "-l")
buf, err := cmd.Output()                     ⟵ buf is a ([]byte)
```

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[a]:

```
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")     ⟵ Mandatory brackets
```

If there were no errors (returned in `e`), you can use `conn` to read and write. The primitives defined in the package *net* are:

```
// Read reads data from the connection.
Read(b []byte)(n int, err error)
```

This makes `conn` an **`io.Reader`**.

```
// Write writes data to the connection.
Write(b []byte)(n int, err error)
```

This makes `conn` also an **`io.Writer`**, in fact `conn` is an **`io.ReadWriter`**.[b]

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

```
package main
import ( "io/ioutil"; "http"; "fmt" ) ❶

func main() {
        r, err := http.Get("http://www.google.com/robots.txt") ❶
```

[a] In case you are wondering, 192.0.32.10 and 2620:0:2d0:200::10 are `www.example.org`.
[b] The variable `conn` also implements a `close` method, this really makes it an **`io.ReadWriteCloser`**.
[c] Exercise Q33 is about using these.

```
        if err != nil { fmt.Printf("%s\n", err.String()); return } ❷
        b, err := ioutil.ReadAll(r.Body)     ❸
        r.Body.Close()
        if err == nil { fmt.Printf("%s", string(b)) } ❹
}
```

❶ The imports needed;

❶ Use http's `Get` to retrieve the html;

❷ Error handling;

❸ Read the entire document into `b`;

❹ If everything was OK, print the document.

Exercises

Q29. (2) Processes

1. 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).

   Listing 8.5. Processes in Perl

   ```
   #!/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{$_}}]";
}
```

Q30. (0) Word and letter count

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

    1. Count the number of characters (including spaces);
    2. Count the number of words;
    3. 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.

Q31. (0) Uniq

1. 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 item which don't have the same successor:

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

    Listing 8.8 is a Perl implementation of the algorithm.

Listing 8.8. uniq(1) in Perl

```
#!/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 = $_; }
}
```

Q32. (2) Quine A *Quine* is a program that prints itself.

1. Write a Quine in Go.

Q33. (1) Echo server

1. 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.
2. Make the server concurrent so that every request is taken care of in a separate goroutine.

Q34. (2) 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

$$(8 * (75 + (8 * 6))) - (7/1) = 977$$

1. 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.
2. 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.

Q35. (1) Finger daemon

1. 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 [35] 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.

Answers

A29. (2) Processes

1. There is lots of stuff to do here. We can divide our program up in the following sections:

    1. Starting `ps` and capturing the output;
    2. Parsing the output and saving the child PIDs for each PPID;
    3. Sorting the PPID list;
    4. Printing the sorted list to the screen

    In the solution presented below, we've used a **`map[int][]int`**, i.e. a map indexed with integers, pointing to a slice of ints – which holds the PIDs. The builtin `append` is used to grow the integer slice.

    A possible program is:

    Listing 8.6. Processes in Go

```
package main
import ( "fmt"; "os/exec"; "sort"; "strconv"; "strings")

func main() {
        ps := exec.Command("ps", "-e", "-opid,ppid,comm")
        output, _ := ps.Output()
        child := make(map[int][]int)
        for i, s := range strings.Split(string(output), "\n") {
                if i == 0 { continue } // Kill first line
                if len(s) == 0 { continue } // Kill last line
                f := strings.Fields(s)
                fpp, _ := strconv.Atoi(f[1]) // Parent's pid
                fp, _ := strconv.Atoi(f[0])  // Child's pid
                child[fpp] = append(child[fpp], fp)
        }
        schild := make([]int, len(child))
        i := 0
        for k, _ := range child { schild[i] = k; i++ }
        sort.Ints(schild)
        for _, ppid := range schild {
                fmt.Printf("Pid %d has %d child", ppid, len(child[ppid
                    ]))
                if len(child[ppid]) == 1 {
                        fmt.Printf(": %v\n", child[ppid])
                        continue
                }
                fmt.Printf("ren: %v\n", child[ppid])
        }
}
```

A30. (0) Word and letter count

1. The following program is an implementation of `wc(1)`.

Listing 8.7. wc(1) in Go

```
package main

import (
        "os"
        "fmt"
        "bufio"
        "strings"
)

func main() {
        var chars, words, lines int
        r := bufio.NewReader(os.Stdin) ❶
        for {
                switch s, ok := r.ReadString('\n'); true { ❷
                case ok != nil: ❸
                        fmt.Printf("%d %d %d\n", chars, words, lines);
                        return
                default:
                        chars += len(s)
                        words += len(strings.Fields(s))
                        lines++
                }
        }
}
```

❶ Start a new reader that reads from standard input;

❷ Read a line from the input;

❸ If we received an error, we assume it was because of a EOF. So we print the current values;

❹ Otherwise we count the charaters, words and increment the lines.

A31. (0) Uniq

1. The following is a uniq implementation in Go.

Listing 8.9. uniq(1) in Go

```
package main

import "fmt"

func main() {
        list := []string{"a", "b", "a", "a", "c", "d", "e", "f"}
        first := list[0]
```

```
        fmt.Printf("%s ", first)
        for _, v := range list[1:] {
                if first != v {
                        fmt.Printf("%s ", v)
                        first = v
                }
        }

}
```

A32. (2) Quine

| This solution is from Russ Cox. It was posted to the Go Nuts mailing list.

Listing 8.10. A Go quine

```
1. /* Go quine */
   package main
   import "fmt"
   func main() {
    fmt.Printf("%s%c%s%c\n", q, 0x60, q, 0x60)
   }
   var q = `/* Go quine */
   package main
   import "fmt"
   func main() {
    fmt.Printf("%s%c%s%c\n", q, 0x60, q, 0x60)
   }
   var q = `
```

A33. (1) Echo server

1. A simple echo server might be:

Listing 8.11. A simple echo server

```
package main
import ( "net"; "fmt";"bufio" )

func main() {
        l, err := net.Listen("tcp", "127.0.0.1:8053")
        if err != nil {
                fmt.Printf("Failure to listen: %s\n", err.Error())
        }
        for {
                if c, err := l.Accept(); err == nil { Echo(c) }
        }
}

func Echo(c net.Conn) {
```

```
        defer c.Close()
        line, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
                fmt.Printf("Failure to read: %s\n", err.Error())
                return
        }
        _, err = c.Write([]byte(line))
        if err != nil {
                fmt.Printf("Failure to write: %s\n", err.Error())
                return
        }
}
```

When started you should see the following:

```
% nc 127.0.0.1 8053
Go is *awesome*
Go is *awesome*
```

2. To make the connection handling concurrent we *only need to change one line* in our echo server, the line:

```
if c, err := l.Accept(); err == nil { Echo(c) }
```

becomes:

```
if c, err := l.Accept(); err == nil { go Echo(c) }
```

A34. (2) Number cruncher

1. The following is one possibility. It uses recursion and backtracking to get an answer.

Listing 8.12. Number cruncher

```
package main

import ( "fmt"; "strconv"; "flag")

const (
        _ = 1000 * iota
        ADD
        SUB
        MUL
        DIV
        MAXPOS = 11
)

var mop = map[int]string{ADD: "+", SUB: "-", MUL: "*", DIV: "/"}
var (
        ok    bool
        value int
)
```

```
type Stack struct {
        i    int
        data [MAXPOS]int
}

func (s *Stack) Reset()     { s.i = 0 }
func (s *Stack) Len() int   { return s.i }
func (s *Stack) Push(k int) { s.data[s.i] = k; s.i++ }
func (s *Stack) Pop() int   { s.i--; return s.data[s.i] }

var found int
var stack = new(Stack)

func main() {
        flag.Parse()
        list := []int{1, 6, 7, 8, 8, 75, ADD, SUB, MUL, DIV}
        magic, ok := strconv.Atoi(flag.Arg(0))  // Arg0 is i
        if ok != nil { return }
        f := make([]int, MAXPOS)
        solve(f, list, 0, magic)
}

func solve(form, numberop []int, index, magic int) {
        var tmp int
        for i, v := range numberop {
                if v == 0 { goto NEXT }
                if v < ADD { // it's a number, save it
                        tmp = numberop[i]
                        numberop[i] = 0
                }
                form[index] = v
                value, ok = rpncalc(form[0 : index+1])

                if ok && value == magic {
                        if v < ADD {
                                numberop[i] = tmp // reset and go on
                        }
                        found++
                        fmt.Printf("%s = %d  #%d\n", rpnstr(form[0:
                             index+1]), value, found)
                }

                if index == MAXPOS-1 {
                        if v < ADD {
                                numberop[i] = tmp // reset and go on
                        }
                        goto NEXT
                }
                solve(form, numberop, index+1, magic)
                if v < ADD {
```

```
                        numberop[i] = tmp // reset and go on
                }
        NEXT:
        }
}

func rpnstr(r []int) (ret string) {     // Convert rpn to infix
    notation
        s := make([]string, 0) // Still memory intensive
        for k, t := range r {
                switch t {
                case ADD, SUB, MUL, DIV:
                        a, s := s[len(s)-1], s[:len(s)-1]
                        b, s := s[len(s)-1], s[:len(s)-1]
                        if k == len(r)-1 {
                                s = append(s, b+mop[t]+a)
                        } else {
                                s = append(s, "("+b+mop[t]+a+")")
                        }
                default:
                        s = append(s, strconv.Itoa(t))
                }
        }
        for _, v := range s { ret += v }
        return
}

func rpncalc(r []int) (int, bool) {
        stack.Reset()
        for _, t := range r {
                switch t {
                case ADD, SUB, MUL, DIV:
                        if stack.Len() < 2 { return 0, false }
                        a := stack.Pop()
                        b := stack.Pop()
                        if t == ADD { stack.Push(b + a) }
                        if t == SUB {
                                // disallow negative subresults
                                if b-a < 0 {
                                        return 0, false
                                }
                                stack.Push(b - a)
                        }
                        if t == MUL { stack.Push(b * a) }
                        if t == DIV {
                                if a == 0 {
                                        return 0, false
                                }
                                // disallow fractions
                                if b%a != 0 {
```

```
                                        return 0, false
                                }
                                stack.Push(b / a)
                        }
                default:
                        stack.Push(t)
                }
        }
        if stack.Len() == 1 { // there is only one!
                return stack.Pop(), true
        }
        return 0, false
}
```

2. When starting `permrec` we give 977 as the first argument:

```
% ./permrec 977
1+(((6+7)*75)+(8/8)) = 977  #1
...                         ...
((75+(8*6))*8)-7 = 977      #542
(((75+(8*6))*8)-7)*1 = 977  #543
(((75+(8*6))*8)-7)/1 = 977  #544
```

A35. (1) Finger daemon

This solution is from Fabian Becker.

Listing 8.13. A finger daemon

1.
```
package main

import (
        "bufio"
        "errors"
        "flag"
        "io/ioutil"
        "net"
        "os/user"
        "strconv"
)

func main() {
        flag.Parse()
        ln, err := net.Listen("tcp", ":79")
        if err != nil {
                panic(err)
        }
        for {
                conn, err := ln.Accept()
                if err != nil {
                        continue
                }
```

```
                go handleConnection(conn)
        }
}

func handleConnection(conn net.Conn) {
        defer conn.Close()
        reader := bufio.NewReader(conn)
        usr, _, _ := reader.ReadLine()

        if info, err := getUserInfo(string(usr)); err != nil {
                conn.Write([]byte(err.Error()))
        } else {
                conn.Write(info)
        }
}

func getUserInfo(usr string) ([]byte, error) {
        u, e := user.Lookup(usr)
        if e != nil {
                return nil, e
        }
        data, err := ioutil.ReadFile(u.HomeDir + ".plan")
        if err != nil {
                return data, errors.New("User doesn't have a .plan
                     file!\n")
        }
        return data, nil
}
```

A Colophon

This work was created with LaTeX. The main text is set in the Google Droid fonts. All typewriter text is typeset in DejaVu Mono.

Contributors

The following people have helped to make this book what it is today.

- Miek Gieben <miek@miek.nl>;
- JC van Winkel;
- Xing Xing, Chinese translation, 这里是中文译本: http://www.mikespook.com/learning-go/ .

Help with proof reading, checking exercises and text improvements (no particular order and either real name or an alias): *Adam J. Gray*, *Alex Sychev*, *Andrea Spadaccini*, *Andrey Mirtchovski*, *Anthony Magro*, *Babu Sreekanth*, *Ben Bullock*, *Bob Cunningham*, *Brian Fallik*, *Cecil New*, *Damian Gryski*, *Dan Kortschak*, *David Otton*, *Fabian Becker*, *Filip Zaludek*, *Hadi Amiri*, *Haiping Fan*, *Jaap Akkerhuis*, *JC van Winkel*, *Jeroen Bulten*, *Jinpu Hu*, *Jonathan Kans*, *Joshua Stein*, *Makoto Inoue*, *Mayuresh Kathe*, *Michael Stapelberg*, *Olexandr Shalakhin*, *Paulo Pinto*, *Peter Kleiweg*, *Philipp Schmidt*, *Robert Johnson*, *Russel Winder*, *Sonia Keys*, *Stefan Schroeder*, *Thomas Kapplet*, *T.J. Yang*, *"Cobold"*, *"Simoc"*, *"Uriel"*, *Xing Xing*.

The following people provided smaller improvements, like nits, typos and other tweaks: *Daniele Pala*.

Miek Gieben

Miek Gieben has a master's degree in Computer Science from the Radboud University Nijmegen (Netherlands). He is involved in the development and now the deployment of the DNSSEC protocol – the successor of the DNS and as such co-authored [9].

After playing with the language Erlang, Go was the first concurrent language that actually stuck with him.

He fills his spare time with coding in, and writing of Go. He is the maintainer of the Go DNS library: https://github.com/miekg/dns. He maintains a personal blog on http://www.miek.nl and tweets under the name @miekg. The postings and tweets may sometimes actually have to do something with Go.

License and copyright

B Index

C Bibliography

[1] Haskell Authors. Haskell. http://www.haskell.org/, 1990.

[2] Inferno Authors. Inferno. http://www.vitanuova.com/inferno/, 1995.

[3] Plan 9 Authors. Plan 9. http://plan9.bell-labs.com/plan9/index.html, 1992.

[4] Plan 9 Authors. Limbo. http://www.vitanuova.com/inferno/papers/limbo.html, 1995.

[5] D. Crockford. The application/json media type for javascript object notation (json). http://www.ietf.org/rfc/rfc4627.txt, 2006.

[6] Brian Kernighan Dennis Ritchie. The C programming language, 1975.

[7] Ericsson Cooperation. Erlang. http://www.erlang.se/, 1986.

[8] Larry Wall et al. Perl. http://perl.org/, 1987.

[9] Kolkman & Gieben. Dnssec operational practices. http://www.ietf.org/rfc/rfc4641.txt, 2006.

[10] Go Authors. Defer, panic, and recover. http://blog.golang.org/2010/08/defer-panic-and-recover.html, 2010.

[11] Go Authors. Effective Go. http://golang.org/doc/effective_go.html, 2010.

[12] Go Authors. Go faq. http://golang.org/doc/go_faq.html, 2010.

[13] Go Authors. Go language specification. http://golang.org/doc/go_spec.html, 2010.

[14] Go Authors. Go package documentation. http://golang.org/doc/pkg/, 2010.

[15] Go Authors. Go release history. http://golang.org/doc/devel/release.html, 2010.

[16] Go Authors. Go tutorial. http://golang.org/doc/go_tutorial.html, 2010.

[17] Go Authors. Go website. http://golang.org/, 2010.

[18] Go Authors. Getting Started. http://golang.org/doc/install/, 2012.

[19] Go Community. Function accepting a slice of interface types. http://groups.google.com/group/golang-nuts/browse_thread/thread/225fad3b5c6d0321, 2010.

[20] Go Community. Go issue 65: Compiler can't spot guaranteed return in if statement. http://code.google.com/p/go/issues/detail?id=65, 2010.

[21] James Gosling et al. Java. http://oracle.com/java/, 1995.

[22] LAMP Group at EPFL. Scala. http://www.scala-lang.org/, 2003.

[23] C. A. R. Hoare. Quicksort. http://en.wikipedia.org/wiki/Quicksort, 1960.

[24] C. A. R. Hoare. Communicating sequential processes (csp). http://www.usingcsp.com/cspbook.pdf, 1985.

[25] Rob Pike. Newsqueak: A language for communicating with mice. http://swtch.com/~rsc/thread/newsqueak.pdf, 1989.

[26] Rob Pike. The Go programming language, day 2. http://golang.org/doc/{G}oCourseDay2.pdf, 2010.

[27] Rob Pike. The Go programming language, day 3. http://golang.org/doc/{G}oCourseDay3.pdf, 2010.

[28] Bjarne Stroustrup. The C++ programming language, 1983.

[29] Ian Lance Taylor. Go interfaces. http://www.airs.com/blog/archives/277, 2010.

[30] Imran On Tech. Using fizzbuzz to find developers... http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/, 2010.

[31] Wikipedia. Bubble sort. http://en.wikipedia.org/wiki/Bubble_sort, 2010.

[32] Wikipedia. Communicating sequential processes. http://en.wikipedia.org/wiki/Communicating_sequential_processes, 2010.

[33] Wikipedia. Duck typing. http://en.wikipedia.org/wiki/Duck_typing, 2010.

[34] Wikipedia. Iota. http://en.wikipedia.org/wiki/Iota, 2010.

[35] D. Zimmerman. The finger user information protocol. http://www.ietf.org/rfc/rfc1196.txt, 1990.

This page is intentionally left blank.