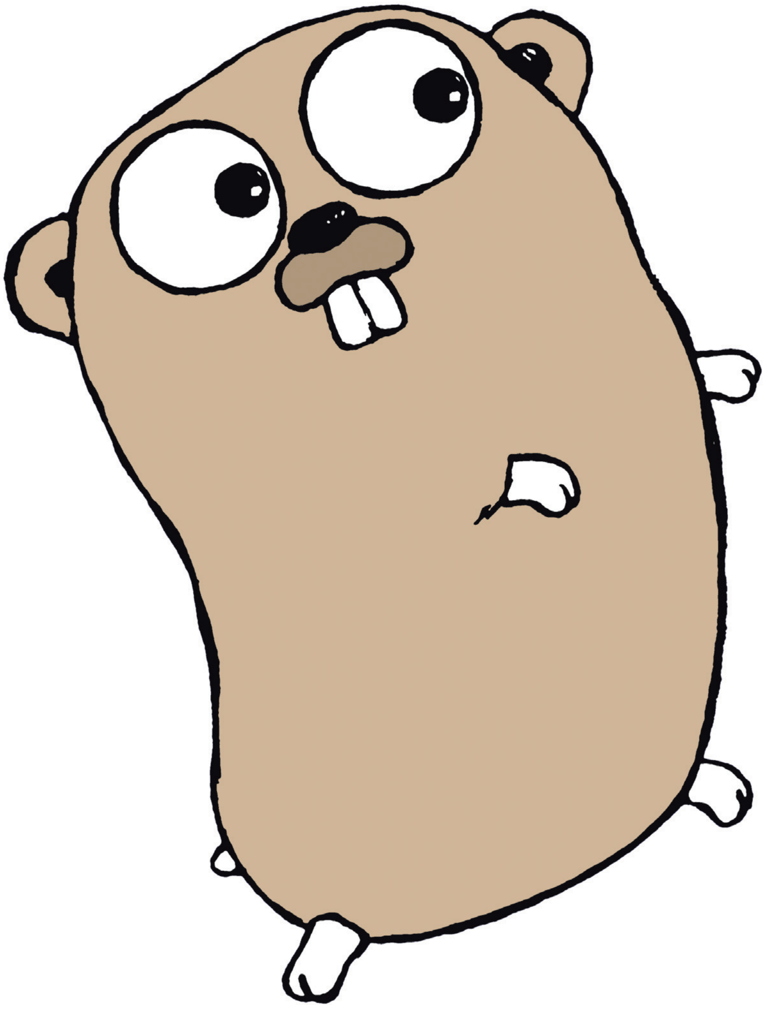
THE WAY TO GO

A Thorough Introduction to the Go Programming Language

IVO BALBAERT



THE WAY TO GO

THE WAY TO GO

A Thorough Introduction to the Go Programming Language

IVO BALBAERT

iUniverse, Inc.
Bloomington

The Way to Go
A Thorough Introduction to the Go Programming Language

iUniverse books may be ordered through booksellers or by contacting:

iUniverse
1663 Liberty Drive
Bloomington, IN 47403
www.iuniverse.com
1-800-Authors (1-800-288-4677)

ISBN: 978-1-4697-6916-5 (sc)
ISBN: 978-1-4697-6917-2 (ebk)

Printed in the United States of America

iUniverse rev. date: 03/05/2012

CONTENTS

LIST OF ILLUSTRATIONS

Preface

Code less, compile quicker, execute faster => *have more fun!*

This text presents the most comprehensive treatment of the Go programming language you can find. It draws on the whole spectrum of Go sources available: online documentation and blogs, books, articles, audio and video, and my own experience in software engineering and teaching programming languages and databases, organizing the concepts and techniques in a systematic way.

Several researchers and developers at Google experienced frustration with the software development processes within the company, particularly using C++ to write large server software. The binaries tended to be huge and took a long time to compile, and the language itself was quite old. A lot of the ideas and changes in hardware that have come about in the last couple of decades haven't had a chance to influence C++. So the researchers sat down with a clean sheet of paper and tried to design a language that would solve the problems they had:

1. *software needs to built quickly,*
2. *the language should run well on modern multi-core hardware,*
3. *the language should work well in a networked environment,*
4. *the language should be a pleasure to use.*

And so was born "Go", a language that has the feel of a dynamic language like Python or Ruby, but has the performance and safety of languages like C or Java.

Go seeks to reinvent programming in the most practical of ways: its not a fundamentally new language with strange syntax and arcane concepts; instead it builds and improves a lot on the existing C/Java/C#-style syntax. It proposes interfaces for object-oriented programming and goroutines / channels for concurrent and parallel programming.

This book is intended for developers who want to learn this new, fascinating and promising language. Some basic knowledge of programming and some experience with a programming language and environment is assumed, but a thorough knowledge of C or Java or the like is not needed.

For those of you who are familiar with C or the current object oriented languages, we will compare the concepts in Go with the corresponding concepts in these languages (throughout the book we will use the well known OO abrevation, to mean object-oriented).

This text explains everything from the basic concepts onwards, but at the same time we discuss advanced concepts and techniques such as a number of different patterns when applying goroutines and channels, how to use the google api from Go, how to apply memoization, how to use testing in Go and how to use templating in web applications.

In Part I we discuss the origins of the language (ch 1) and get you started with the installation of Go (ch 2) and a development environment (ch 3).

Part 2 then guides you through the core concepts of Go: the simple and composite types (ch 4, 7, 8), control structures (ch 5), functions (ch 6), structs with their methods (ch 10), and interfaces (ch 11). The functional and object-oriented aspects of Go are thoroughly discussed, as well as how Go code in larger projects is structured (ch 9).

Part 3 learns you how to work with files in different formats (ch 12) and how to leverage the error-handling mechanism in Go (ch 13). It also contains a thorough treatment of Go's crown jewel: goroutines and channels as basic technique for concurrent and multicore applications (ch 14). Then we discuss the networking techniques in Go and apply this to distributed and web applications (ch 15).

Part 4 shows you a number of Go language patterns and idioms (ch 16, 17), together with a collection of useful code snippets (ch 18). With all of the techniques which you have learned in the previous chapters, a complete Go project is built (ch 19) and you get an introduction in how to use Go in the cloud (Google App Engine) (ch 20). In the last chapter (ch 21) we discuss some real world uses of go in businesses and organizations all over the world. The text is concluded with quotes of users, listings, references to Go packages and tools, answers to questions and exercises, and a bibliography of all resources and references.

Go has very much a 'no nonsense' approach to it: extreme care has gone into making things easy and automatic; it adheres to the KISS principle from Agile programming: Keep It Short and Simple!

Solving or leaving out many of the 'open' features in C, C++ or Java makes the developer's life much easier! A few examples are: default initializations of variables; memory is allocated and freed automatically; fewer, but more powerful control constructs. As we will see Go also aims to prevent unnecessary typing, often Go code is shorter and easier to read than code from the classic object-oriented languages.

Go is simple enough to fit in your head, which can't be said from C++ or Java; the barrier to entry is low, compared to e.g. Scala (the Java concurrent language). Go is a modern day C.

Most of the code-examples and exercises provided interact with the console, which is not a surprise since Go is in essence a systems language. Providing a graphical user interface (GUI) framework which is platform-independent is a huge task. Work is under way in the form of a number of 3rd party projects, so somewhere in the near future there probable will be a usable Go GUI framework. But in this age the web and its protocols are all pervasive, so to provide a GUI in some examples and exercises we will use Go's powerful http and template packages.

We will always use and indicate what is called idiomatic Go-code, by which we mean code which is accepted as being best practice. We try to make sure that examples never use concepts or techniques which were not covered up until that point in the text. There are a few exceptions where it seemed better to group it with the discussion of the basic concept: in that case the advanced concept is referenced and the § can be safely skipped.

All concepts and techniques are thoroughly explained through 227 working code examples (on a grey background), printed out and commented in the text and available online for execution and experimenting.

The book is cross-referenced as much as possible, forward as well as backward. And of course this is what you must do: after setting up a Go environment with a decent editor, start experimenting with the code examples and try the exercises: mastering a new language and new concepts can only be achieved through exercising and experimenting, so the text contains 130 exercises, with downloadable solutions. We have used the famous Fibonacci algorithm in examples and exercises in 13 versions to illustrate different concepts and coding techniques in Go.

The book has an website (https://sites.google.com/site/thewaytogo2012/) from where the code examples can be downloaded and on which complementary material and updates are available.

For your convenience and further paving your path to become a Go master, special chapters are dedicated to the best practices and language patterns in Go, and another to the pitfalls for the Go beginner. Handy as a desktop-reference while coding is chapter 18, which is a collection of the most useful code snippets, with references to the explanations in the text.

And last but not least, a comprehensive index leads you quickly to the page you need at the moment. All code has been tested to work with the stable Go-release **Go 1.**

Here are the words of Bruce Eckel, a well known authority on C++, Java and Python:

"Coming from a background in C/C++, I find Go to be a real breath of fresh air. At this point, I think it would be a far better choice than C++ for doing systems programming because it will be much more productive and it solves problems that would be notably more difficult in C++. This is not to say that I think C++ was a mistake -- on the contrary, I think it was inevitable. At the time, we were deeply mired in the C mindset, slightly above assembly language and convinced that any language construct that generated significant code or overhead was impractical. Things like garbage collection or language support for parallelism were downright ridiculous and no one took them seriously. C++ took the first baby steps necessary to drag us into this larger world, and Stroustrup made the right choices in making C++ comprehensible to the C programmer, and able to compile C. We needed that at the time.

We've had many lessons since then. Things like garbage collection and exception handling and virtual machines, which used to be crazy talk, are now accepted without question. The complexity of C++ (even more complexity has been added in the new C++), and the resulting impact on productivity, is no longer justified. All the hoops that the C++ programmer had to jump through in order to use a C-compatible language make no sense anymore -- they're just a waste of time and effort. Now, Go makes much more sense for the class of problems that C++ was originally intended to solve."

I would like to express my sincere gratitude to the Go team for creating this superb language, especially "Commander" Rob Pike, Russ Cox and Andrew Gerrand for their beautiful and illustrative examples and explanations. I also thank Miek Gieben, Frank Müller, Ryanne Dolan and Satish V.J. for the insights they have given me, as well as countless other members of the Golang-nuts mailing list.

Welcome to the wonderful world of developing in Go!

PART 1

WHY LEARN GO—GETTING STARTED

Chapter 1—Origins, Context and Popularity of Go

1.1 Origins and evolution

Go's year of genesis was 2007, and the year of its public launch was 2009. The initial design on Go started on September 21, 2007 as a 20% part-time project at *Google Inc.* by three distinguished IT-engineers: *Robert Griesemer* (known for his work at the Java HotSpot Virtual Machine)**,** *Rob 'Commander' Pike* **(**member of the Unix team at Bell Labs, worked at the Plan 9 and Inferno operating systems and the Limbo programming language) and *Ken Thompson* (member of the Unix team at Bell Labs, one of the fathers of C, Unix and Plan 9 operating systems, co-developed UTF-8 with Rob Pike). By January 2008 Ken Thompson had started working on a compiler to explore the ideas of the design; it produced C as output.

This is a gold team of 'founding fathers' and experts in the field, who have a deep understanding of (systems) programming languages, operating systems and concurrency.

Fig 1.1: The designers of Go: Griesemer, Thompson and Pike

By mid 2008, the design was nearly finished, and full-time work started on the implementation of the compiler and the runtime. *Ian Lance Taylor* joins the team, and in May 2008 builds a gcc-frontend.

Russ Cox joins the team and continues the work on the development of the language and the libraries, called packages in Go. On October 30 2009 Rob Pike gave the first talk on Go as a Google Techtalk.

On November 10 2009, the Go-project was officially announced, with a BSD-style license (so fully open source) released for the Linux and Mac OS X platforms. A first Windows-port by Hector Chu was announced on November 22.

Being an *open-source project*, from then on a quickly growing community formed itself which greatly accelerated the development and use of the language. Since its release, more than 200 non-Google contributors have submitted over 1000 changes to the Go core; over the past 18 months 150 developers contributed new code. This is one of the largest open-source teams in the world, and is in the top 2% of all project teams on Ohloh (source: www.ohloh.net). Around April 2011 10 Google employees worked on Go full-time. Open-sourcing the language certainly contributed to its phenomenal growth. In 2010 *Andrew Gerrand* joins the team as a co-developer and Developer Advocate.

Go initiated a lot of stir when it was publicly released and on January 8, 2010 Go was pronounced 'language of the year 2009' by Tiobe (www.tiobe.com, well-known for its popularity ranking of programming languages). In this ranking it reached its maximum (for now) in Feb 2010, being at the 13th place, with a popularity of 1,778 %.

Year	**Winner**
2010	Python
2009	Go
2008	C
2007	Python
2006	Ruby
2005	Java
2004	PHP
2003	C++

Go Programming Language of the year 2009 at Tiobe

TIMELINE:

Initial design	Public release	Language of the year 2009	Used at Google	Go in Google App Engine
2007 Sep 21	**2009 Nov 10**	**2010 Jan 8**	**2010 May**	**2011 May 5**

Since May 2010 Go is used in production at Google for the back-end infrastructure, e.g. writing programs for administering complex environments. Applying the principle: 'Eat your own dog food': this proves that Google wants to invest in it, and that the language is production-worthy.

The principal website is http://golang.org/: this site runs in Google App Engine with godoc (a Go-tool) serving (as a web server) the content and a Python front-end. The home page of this site features beneath the title Check it out! the so called Go-playground, a sandbox which is a simple editor for Go-code, which can then be compiled and run, all in your browser without having installed Go on your computer. A few examples are also provided, starting with the canonical "Hello, World!".

Some more info can be found at http://code.google.com/p/go/, it hosts the issue-tracker for Go bugs and -wishes: http://code.google.com/p/go/issues/list

Go has the following logo which symbolizes its speed: =GO, and has a gopher as its mascot.

Fig 1.2: The logo's of Go

The Google discussion-group *Go Nuts* (http://groups.google.com/group/golang-nuts/) is very active, delivering tens of emails with user questions and discussions every day.

For Go on Google App Engine a separate group exists (https://groups.google.com/forum/#!forum/google-appengine-go) although the distinction is not always that clear. The community has a Go language resource site at http://go-lang.cat-v.org/ and **#go-nuts** on **irc.freenode.net** is the official Go IRC channel.

@go_nuts at Twitter (http://twitter.com/#!/go_nuts) is the Go project's official Twitter account, with #golang as the tag most used.

There is also a Linked-in group: http://www.linkedin.com/groups?gid=2524765&trk=myg_ugrp_ovr

The Wikipedia page is at http://en.wikipedia.org/wiki/Go_(programming_language)

A search engine page specifically for Go language code can be found at http://go-lang.cat-v.org/go-search

Go can be interactively tried out in your browser via the App Engine application Go Tour: http://go-tour.appspot.com/ (To install it on your local machine, use: go install go-tour.googlecode.com/hg/gotour).

1.2 Main characteristics, context and reasons for developing a new language

1.2.1 Languages that influenced Go

Go is a language designed from the ground up, as a 'C for the 21st century'.It belongs to the C-family, like C++, Java and C#, and is inspired by many languages created and used by its designers.

There was significant input from the Pascal / Modula / Oberon family (declarations, packages) For its concurrency mechanism it builds on experience gained with Limbo and Newsqueak, which themselves were inspired by Tony Hoare's CSP theory (Communicating Sequential Processes); this is essentially the same mechanism as used by the Erlang language.

It is a completely *open-source language*, distributed with a *BSD license*, so it can be used by everybody even for commercial purposes without a fee, and it can even be changed by others.

The resemblance with the C-syntax was maintained so as to be immediately familiar with a majority of developers, however comparing with C/C++ the syntax is greatly *simplified* and made more *concise and clean*. It also has characteristics of a dynamic language, so Python and Ruby programmers feel more comfortable with it.

The following figure shows some of the influences:

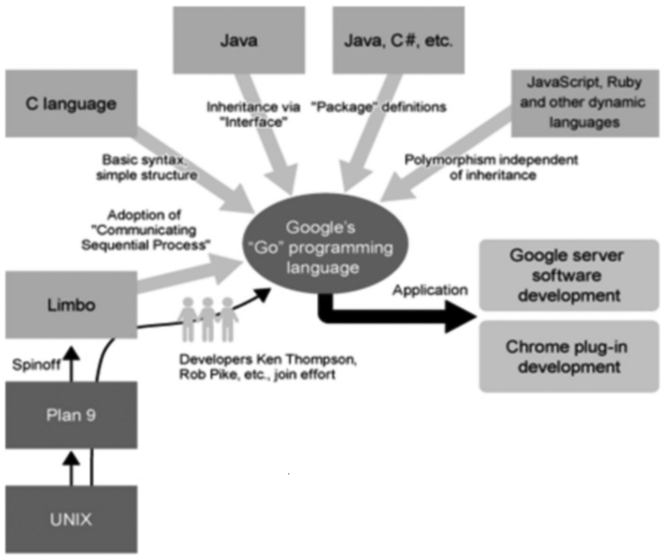

Java
Java, C#, etc.
C language
Inheritance via "Interface"
"Package" definitions
JavaScript, Ruby and other dynamic languages
Basic syntax, simple structure
Polymorphism independent of inheritance
Adoption of "Communicating Sequential Process"
Google's "Go" programming language
Application
Google server software development
Limbo
Chrome plug-in development
Spinoff
Developers Ken Thompson, Rob Pike, etc., join effort
Plan 9
UNIX

Fig 1.3: Influences on Go

1.2.2 Why a new language?

- C/C++ did not evolve with the computing landscape, no major systems language has emerged in over a decade: so there is a definite need for a new systems language, appropriate for needs of our computing era.
- In contrast to computing power, software development is not considerably faster or more successful (considering the number of failed projects) and applications still grow in size, so a new low-level language, but equipped with higher concepts, is needed.
- Before Go a developer had to choose between fast execution but slow and not efficient building (like C++), efficient compilation (but not so fast execution, like .NET or Java), or ease of programming (but slower execution, like the dynamic languages): Go is an attempt to combine all three wishes: efficient and thus fast compilation, fast execution, ease of programming.

1.2.3 Targets of the language

A main target was to combine the *efficacy, speed and safety* of a *strongly and statically compiled language* with the *ease of programming* of a *dynamic* language, so as to make programming more fun again.

So the language is *type-safe*, and it is also *memory-safe*: pointers are used in Go, but pointer-arithmetic is not possible.

Another target (and of course very important for internal use in Google) was that it should give excellent support for *networked-communication*, *concurrency* and *parallelization*, in order to get the most out of distributed and multicore machines. It is implemented through the concepts of goroutines, which are very lightweight-threads, and channels for communication between them. They are implemented as growing stacks (segmented stacks) and multiplexing of goroutines onto threads is done automatically.

This is certainly the great stronghold of Go, given the growing importance of multicore and multiprocessor computers, and the lack of support for that in existing programming languages.

Of the utmost importance was also the *building speed (compilation and linking to machine code)*, which had to be excellent (in the order of 100s of ms to a few s at most). This was born out of frustration with the build-times of C++-projects, heavily used in the Google infrastructure. This alone should give an enormous boost to *developer productivity* and give rise to a tighter test-code development cycle.

Dependency management is a big part of software development today but the "header files" of languages in the C tradition are causing considerable overhead leading to build times of hours for the great projects. A rigid and clean dependency analysis and fast compilation is needed. This is what Go provides with *its package model*: explicit dependencies to enable faster builds. The package model of Go provides for excellent scalability.

The entire Go standard library compiles in less than 20 seconds; typical projects compile in half a second: this lightning fast compiling process, even faster than C or Fortran, makes compilation a non-issue. Until now this was regarded as one of the great benefits of dynamic languages because the long compile/link step of C++ could be skipped, but with Go this is no longer an issue! Compilation times are negligible, so with Go we have the same productivity as in the development cycle of a scripting or dynamic language.

On the other hand, the *execution speed* of the native code should be comparable to C/C++.

Because memory-problems (so called memory-leaks) are a long time problem of C++, Go's designers decided that memory-management should not be the responsibility of the developer. So although Go executes native code, it runs in a kind of runtime, which takes care of an efficient and fast *garbage collection* (at this moment a simple mark- and sweep algorithm).

> Garbage collection, although difficult to implement for that kind of problems, was considered crucial for the development of the concurrent applications of the future.

It also has a built-in runtime *reflection* capability.

`go install` provides an *easy deployment system* for external packages.

Furthermore there is *support for legacy software*, notably C libraries can be used (see § 3.9).

1.2.4 Guiding design principles

Go tries to reduce typing, clutter and complexity in coding through a minimal amount of keywords (25). Together with the *clean, regular and concise syntax*, this enhances the compilation speed, because the keywords can be parsed without a symbol table.

These aspects reduce the number of code lines necessary, even compared with a language like Java.

Go has a minimalist approach: there tends to be only one or two ways of getting things done, so reading other people's code is generally pretty easy, and we all know *readability of code* is of the utmost importance in software engineering.

The design concepts of the language don't stand in each other's way, they don't add up complexity to one another: they are *orthogonal.*

Go is completely defined by an *explicit specification* that can be found at http://golang.org/doc/go_spec.html;

it is not defined by an implementation, as is Ruby for example. An explicit language specification was a requirement for implementing the two different compilers gc and gccgo (see § 2.1), and this in itself was a great help in clarifying the specification.

The Go grammar is LALR(1) (http://en.wikipedia.org/wiki/LALR_parser), this can be seen in src/cmd/gc/go.y); it can be parsed without a symbol table.

1.2.5 Characteristics of the language

It is essentially an *imperative* (procedural, structural) kind of language, built with *concurrency* in mind.

It is *not object-oriented* in the normal sense like Java and C++ because it doesn't have the concept of classes and inheritance. However it does have a concept of *interfaces*, with which much of polymorphism can be realized. Go has a clear and expressive *type-system*, but it is lightweight and without hierarchy. So in this respect it could be called a hybrid language.

> Object-orientation as in the predominant OO-languages was considered to be too 'heavy', leading to often cumbersome development constructing big type-hierarchies, and so not compliant with the speed goal of the language.

Functions are the basic building block in Go, and their use is very versatile. In chapter 6 we will see that Go also exhibits the fundamental aspects of a *functional language.*

It is *statically typed*, thus a *safe language*, and compiles to native code, so it has a very efficient execution.

It is *strongly typed*: implicit type conversions (also called castings or coercions) are not allowed; the principle is: keep things explicit!

It has certain characteristics of a *dynamically* typed language (through the *var* keyword).

> So it also appeals to programmers who left Java and the .Net world for Python, Ruby, PHP and Javascript.

Go has support for *cross-compilation*: e.g. developing on a Linux-machine for an application that will execute on Windows. It is the first programming language in which *UTF-8* can be used, not only in strings, but also in program code (Go source-files are UTF-8): Go is truly international!

1.2.6 Uses of the language

Go was originally conceived as a *systems programming language*, to be used in the heavy server-centric (Google) world of web servers, storage architecture and the like. For certain domains like *high performance distributed syste*ms Go has already proved to be a more productive language than most others. Go shines in and makes massive concurrency easy, so it should be a good fit for game server development.

Complex event processing (CEP, see http://en.wikipedia.org/wiki/Complex_event_processing), where one needs both mass concurrency, high level abstractions and performance, is also an excellent target for Go usage. As we move to an Internet of Things, CEP will come to the forefront.

But it turned out that it is also a *general* programming language, useful for solving text-processing problems, making frontends, or even scripting-like applications.

However Go is not suited for real-time software because of the garbage collection and automatic memory allocation.

Go is used internally in Google for more and more heavy duty distributed applications; e.g. a part of Google Maps runs on Go.

Real life examples of usage of Go in other organizations can be found at http://go-lang.cat-v.org/organizations-using-go. Not all uses of Go are mentioned there, because many companies consider this as private information. An application has been build inside Go for a large storage area network (SAN).(See Chapter 21 for a discussion of a sample of current use cases).

A Go compiler exists for *Native Client (NaCl)* in the Chrome-browser; it will probably be used for the execution of native code in web applications in the Chrome OS.

Go also runs on Intel as well as *ARM* processors (see chapter 2), so it runs under the Android OS, for example on Nexus smartphones.

Go on Google App Engine: on May 5 2011 a Go SDK appeared to use the language in the Cloud in web applications via the Google App Engine infrastructure, making it the first true compiled language that runs on App Engine, which until then only hosted Python and Java apps. This was mainly the work of *David Symonds* and *Nigel Tao*. The latest **stable** version is SDK 1.6.1 based on r60.3, released Dec 13 2011. **The current release is based on Go 1 (Beta).**

1.2.7 Missing features?

A number of features which can be found in most modern OO-languages are missing in Go, some of them might still be implemented in the future.

- No function or operator overloading: this is to simplify the design.
- No implicit conversions: this is by design, to avoid the many bugs and confusions arising from this in C/C++
- No classes and type inheritance: Go takes another approach to object-oriented design (chapter 10-11)
- No variant types: almost the same functionality is realized through interfaces (see chapter 11)
- No dynamic code loading
- No dynamic libraries
- No generics
- No exceptions (although recover / panic (see § 13.2-3) goes a lot in that direction)
- No assertions
- No immutable variables

A discussion around this from the Go-team themselves can be found in the Go FAQ: http://golang.org/doc/go_faq.html

1.2.8 Programming in Go

When coming to Go and having a background in other contemporary (mostly class or inheritance-oriented languages like Java, C#, Objective C, Python, Ruby) you can fall in the trap of trying to program in Go like you did in your X-language. Go is built on a different model, so trying to move code from X to Go usually produces non-idiomatic code and overall works poorly: you have to start over, thinking in Go.

If you take a higher point of view and start analyzing the problem from within the Go mindset, often a different approach suggests itself which leads to an elegant and idiomatic Go-solution.

1.2.9 Summary

Here are the killer features of Go:

- Emphasis on simplicity: *easy to learn*
- Memory managed and syntactically lightweight: *easy to use*
- Fast compilation: *enhances productivity (dev cycle of a scripting language)*
- Fast compiled code: *comparable to C*
- Concurrency support: *write simpler code*
- Static typing
- Consistent standard library
- Easy deployment system (go install)
- Self-documenting (and well-documented)
- Free and Open Source (BSD licensed)

Chapter 2—Installation and Runtime Environment

2.1 Platforms and architectures

The Go-team developed compilers for the following operating systems (OS):

- Linux
- FreeBSD
- OS X (also named Darwin)

There are 2 versions of compilers: the *gc Go-compilers* and *gccgo*; they both work on Unix-like systems. The gc compiler/runtime has been ported to Windows and is integrated in the main distribution. Both compilers work in a single-pass manner.

Go 1 is available in source and binary form on these platforms:

FreeBSD 7+: amd64, 386
Linux 2.6+: amd64, 386, arm
OS X (Snow Leopard + Lion): amd64, 386
Windows (2000 + later): amd64, 386

The portability of Go-code between OS's is excellent (assuming you use pure Go-code, no cgo, inotify or very low level packages): just copy the source code to the other OS and compile, but you can also cross compile Go sources (see § 2.2).

(1) The gc Go-compilers:

They are based on Ken Thompson's previous work on the C toolchain for the Plan 9 operating system.

The Go compilers and linkers are written in C and produce native code (there is no Go bootstrapping or self-hosting involved), so a different compiler (instruction set) is required for every combination of processor-architecture (32 bit and 64 bit, no others at this point) and OS.

They compile faster than gccgo and produce good native code; they are not linkable with gcc; they work non-generational, non-compacting and parallel.

Compilers exist for the following processor-architectures from Intel and AMD:

No of bits	***Processor name***	***Compiler***	***Linker***
64 bit implementation	**amd64** (also named **x86-64**)	**6g**	**6l**
32 bit implementation	**386** (also named **x86** or **x86-32**)	**8g**	**8l**
32 bit RISC implementation	**arm** (**ARM**)	**5g**	**5l**

The naming system is a bit strange at first sight (the names come from the Plan 9-project):

g = compiler: makes object code from source code (program text)
l = linker: transforms object code into executable binaries (machine code)

(The corresponding *C-compilers* are: 6c, 8c and 5c; and the *assemblers* are: 6a, 8a and 5a .)

The following OS-processor combinations are released:

OS	**ARCH**	**OS version**
linux	386 / amd64 / arm	>= Linux 2.6
darwin	386 / amd64	OS X (Snow Leopard + Lion)
freebsd	386 / amd64	>= FreeBSD 7
windows	386 / amd64	>= Windows 2000

The windows implementation (both 8g and 6g) is realized by external contributors and is 95% complete.

The Google-team is committed to the arm implementation, it can eventually be used in the Android OS in Google's smartphones: Go runs wherever Android will run.

Flags: these are options which are given on the command-line and that can influence the compilation/linking or give a special output.

The *compiler flags* are:

```
C:\Users\ivo>8g
gc: usage: 8g [flags] file.go...
flags:
  -I DIR search for packages in DIR
  -d print declarations
  -e no limit on number of errors printed
  -f print stack frame structure
  -h panic on an error
  -o file specify output file                   // see § 3.4
  -S print the generated assembly code
  -V print the compiler version                // see § 2.3 (7)
  -u disable package unsafe
  -w print the parse tree after typing
  -x print lex tokens
```

- **I** can be used to indicate the map in which the Go files to compile are (it can also contain the variable $GOPATH)

The *linker flags* are:

```
C:\Users\ivo>8l
usage: 8l [-options] [-E entry] [-H head] [-I interpreter] [-L dir] [-T text] [-R
rnd] [-r path] [-o out] main.8
```

- **L** can be used to indicate the map in which the Go files to link are (it can also contain the variable $GOPATH)

In depth-info: The sources of the compilers and linkers can be found under $GOROOT/src/cmd. Modifying the Go language itself is done in C code, not Go. The lexer/parser is generated with GNU bison. The grammar/parser is controlled by the yacc file go.y at $GOROOT/src/cmd/gc/go.y and the output is y.tab.{c,h} in the same directory. See the Makefile in that directory for more about the build process. An overview of the build process can be seen by examining $GOROOT/src/make.bash

Most of the directories contain a file doc.go, providing more information.

(2) The gccgo-compiler:

This is a more traditional compiler using the GCC back-end: the popular GNU compiler, which targets a very wide range of processor-architectures. Compilation is slower, but the generated native code is a bit faster. It also offers some interoperability with C.

From 25-3-2011 on with GCC Release 4.6.0 the Go-compiler was integrated in the family of supported languages (containing also Ada, C, C++, Fortran, Go and Java).

(for more information see: http://golang.org/doc/gccgo)

Since Go 1 both compilers (gc and gccgo) are on par: equivalent in their functionality.

(3) File extensions and packages:

The extension for Go source code files is not surprisingly **.go**

C files end in **.c** and assembly files in **.s**; Source code-files are organized in *packages*. Compressed files for packages containing executable code have the extension **.a** (AR archive)

The packages of the Go standard library (see § 9.1) are installed in this format.

Linker (object-) files can have the extension **.o**

An executable program has by default the extension **.out** on Unix and **.exe** on Windows.

Remark: When creating directories for working with Go or Go-tools, never use spaces in the directory name: replace them by _ for example.

2.2 Go Environment variables

The Go-environment works with a small number of OS environment variables. They are best defined before the installation starts; on Windows you will have to create the empty Go root map first, like c:/go. Here are the most important:

$GOROOT	mostly it has the value $HOME/go, but of course you can choose this: it is the root of the go tree (or installation)
$GOARCH	the processor-architecture of the target machine**,** one of the values of the 2nd column of fig 2.1: 386, amd64, arm.
$GOOS	the operating system of the target machine**,** one of the values of the 1st column of fig 2.1: darwin, freebsd, linux, windows
$GOBIN	the location where the binaries (compiler, linker, etc.) are installed, default is $GOROOT/bin

The target machine is the machine where you are going to run the Go-application.

The Go-compiler architecture enables *cross-compiling*: you can compile on a machine(= the host) which has other characteristics (OS, processor) than the target machine.

To differentiate between them you can use **$GOHOSTOS** and **$GOHOSTARCH**: these are the name of the host operating system and compilation architecture: set them when cross-compiling to another platform/architecture. (They default to the local system and normally take their values from $GOOS and $GOARCH).

	Host (H) developing on		Target (T) running on	
Examples	**GOHOSTARCH**	**GOHOSTOS**	**GOARCH**	**GOOS**
H: 64 bit Linux T: 32 bit Linux	amd64	linux	386	linux
H: 32 bit Linux T: 64 bit Linux	386	linux	amd64	linux
H: 32 bit Windows T: 64 bit Linux	386	windows	amd64	linux

The following variables can also be of use:

$GOPATH	defaults to GOROOT, it specifies a list of paths that contain Go source code package binaries (objects), and executables (command binaries); they are located inside the GOPATHs' src, pkg, and bin subdirectories respectively; this variable *must* be set in order to use the go tool.
$GOROOT_FINAL	defaults equal to $GOROOT, so doesn't need to be set explicitly. If after installation of Go, you want to move the installation to another location, this variable contains the final location.
$GOARM	for arm-architectures, possible values are 5, 6; default is 6
$GOMAXPROCS	specifies the number of cores or processors your application uses, see § 14.1.3

In the following sections we discuss the installation of Go on the operating systems where it is feature complete, that is: Linux, OS X and Windows. For FreeBSD this is also the case, but installation is very similar to that on Linux. Porting Go to other OS's like OpenBSD, DragonFlyBSD, NetBSD, Plan 9, Haiku and Solaris are in progress, the most recent info can be found on: http://go-lang.cat-v.org/os-ports

2.3 Installing Go on a Linux system

It is instructive to download and compile the full source-code of Go yourself. Full and up to date instructions can be found at http://golang.org/doc/install.html, but we will review and comment them here:

(1) Set Go-Environment variables:

On a Linux system, set these variables in your shell profile, which is the $HOME/.bashrc file or the $HOME/.profile or equivalent (use an editor like gedit or vi):

```
export GOROOT=$HOME/go
export GOBIN=$GOROOT/bin
export GOARCH=amd64
export GOOS=linux
```

To ensure that the binaries can be found from anywhere in the filesystem, also add this line:

```
export PATH=$GOBIN:$PATH
```

The go tool also uses the variable GOPATH:

```
         export GOPATH=$HOME/goprograms
or:      export GOPATH=$GOROOT
```

$HOME is /home/user1 if your are logged in as user1.

Adding the current directory.:

```
export GOPATH=$GOPATH:.
```

GOPATH may be set to a colon (:)-separated list of paths inside which Go source code, package objects, and executables may be found.

After the changes restart terminal or reload .bashrc with: `source .bashrc`

(test the values with `env | grep ^GO` or `echo $GOROOT` etc. in a terminal window)

(2) Install C-tools:

The Go tool-chain is written in C, so for the build-process you need these programs:

- GCC,
- the standard C libraries Libc6-dev,
- the parser generator Bison,
- make,
- gawk
- the text editor ed.

With the following command these tools are installed if necessary on Debian-based Linux systems, like Ubuntu:

```
sudo apt-get install bison ed gawk gcc libc6-dev make
```

On other Linux distributions RPM's can be used.

(3) Install Mercurial:

The Go source-code is maintained in the Mercurial revision control application. Type hg on the command-line to see if this application is installed.

If not, install it with: `sudo apt-get install mercurial`

If this produces an error, on Ubuntu/Debian systems you might have to do first:

```
apt-get install python-setuptools python-dev build-essential
```

If this fails, try installing manually from the Mercurial download page: http://mercurial.selenic.com/wiki/Download

> Mercurial versions 1.7.x and up require the configuration of Certification Authorities (CAs). Error messages of the form: warning: go.googlecode.com certificate with fingerprint b1:af: ... bc not verified (check hostfingerprints or web.cacerts config setting) when using Mercurial indicate that the CAs are missing. Check your Mercurial version (hg --version) and configure the CAs if necessary.

(4) Fetch the Go repository:

Go will install to a directory named go, indicated by the value of $GOROOT. This directory should not exist. Then check out the repository with the command:

```
hg clone -u release https://go.googlecode.com/hg/ $GOROOT
```

(5) *Build Go:*

```
cd $GOROOT/src
./all.bash
```

This building and testing takes some time (order of minutes) and then when successful the following text appears:

```
ALL TESTS PASSED

  ---

  Installed Go for linux/amd64 in /home/you/go.

  Installed commands in /home/you/go/bin.

  *** You need to add /home/you/go/bin to your $PATH. ***

The compiler is 6g.
```

adapted to the system you have installed Go on.

Remark 1: If you encounter an error in the make cycle, pull an update on the repository with `hg pull -u` and restart step (5) .

Remark 2: *the net test:*

One of the http tests (net) goes out and touches google.com.

An often reported problem is that the installation pauses indefinitely at:

```
'make[2]: Leaving directory `/localusr/go/src/pkg/net'
```

If you develop on a machine behind a firewall, it can be necessary to temporarily disable the firewall while building everything.

Another way to solve this is to use $DISABLE_NET_TESTS to bypass network tests: set in your shell profile .bashrc: export DISABLE_NET_TESTS=1

If this doesn't work you can disable the test of the net package by adding net to the NOTEST list in the Makefile in the map go/src/pkg .

If you don't want to run the tests, you can leave them out by running ./make.bash instead.

(6) *Testing the installation:*

Using your favorite text-editor, make a file with the following code, and save this as test.go:

Listing 2.1—hello_world1.go:

```
package main
func main() {
         println("Hello", "world")
}
```

Compile this first Go-program with:	6g test.go
This compiles to a file:	test.6
which is linked with the command:	6l test.6
This produces the executable named:	6.out
which executes with the command:	./6.out
and produces the output:	Hello, world

(This is on a 64 bit installation, use 8g / 8l for 32 bit, or 5g / 5l for arm.)

(7) *Verifying the release of the installation:*

The Go-releases are identified by their version number and a

- release-number for stable releases, like `release.r60 9481`
- release-date for weekly releases, like `release.2011-01-06 release 7053`

Verify the installed release with:

```
cd $GOROOT
hg identify
```

A quicker way is to use `go version` or the –V flag of the compiler: `6g –V` or `8g –V` which gives an output like: `8g version 7053 release.2011-01-06 release`

From within Go-code, the current release which is executing can be obtained with the function Version from the package runtime:

Listing 2.2—version.go:

```
package main
import (
    "fmt"
    "runtime"
)
func main() {
    fmt.Printf("%s", runtime.Version())
}
```

Output: 7053 release.2011-01-06 release

(8) <u>*Update the installation to a newer release:*</u>

```
cd $GOROOT
hg pull
hg update release
cd src
sudo ./all.bash
```

The latest releases can be viewed at: http://golang.org/doc/devel/release.html

<u>Remark about releases:</u> The *first stable Go-release* was **r56** (2011.03.16)

From 15-3-2011 onwards there are still weekly releases with the latest additions/updates, to be downloaded with hg update weekly. The latest release is **Go 1** and this is the first stable release which will be maintained and supported on a timescale of years.

Distinction is made between the following branches in the Go repository:

- *Go release:* these are the stable releases, best suited for most Go-development
- *Go weekly:* containing the last changes, roughly once a week
- *Go tip*: the latest release

The last 2 are normally only necessary when you urgently need the fixes or improvements; update through hg update weekly or hg update tip.

The gofix—tool can be used to update Go source-code (written in an older release) to the latest release.

<u>Documentation for the different releases:</u>

http://golang.org/pkg shows documentation for the latest stable release; it us updated when the next release happens. Documentation for the latest weekly is at: http://weekly.goneat.org/pkg/ Documentation for tip can be found at: http://tip.goneat.org/pkg/ (at some point 'goneat' in these url's will be replaced by 'golang': http://weekly.golang.org/pkg/—http://tip.golang.org/pkg/)

Installation from a package:

Information about packaged Go-installations can be found at: http://go-lang.cat-v.org/packages. A package for the Ubuntu installation can be downloaded from (23/2/2011): https://launchpad.net/~cz.nic-labs/+archive/golang

2.4 Installing Go on an OS X system

Your Mac-system must contain an Intel 64 bit-processor, PowerPC processors are not yet supported. Follow the instructions from § 2.3

(A port to PowerPC is in progress: https://codedr-go-ppc.googlecode.com/hg/)

Remarks:

Install C-tools: they are installed as part of Xcode, so this is necessary.

Install the Xcode.mpkg from the Optional Installs on my Snow Leopard DVD to get gcc-4.3 installed.Do mkdir $GOBIN before the compile-step *(5).*

A detailed instruction can be found at:

http://www.kelvinwong.ca/2009/11/12/installing-google-go-on-mac-os-x-leopard/

2.5 Installing Go on a Windows system

Go directly in Windows: http://code.google.com/p/go/wiki/WindowsPort

Based on previous work by Hector Chu, Joe Poirier takes care of the Windows release.

A zip-file with a binaries release for win32/win64 can be downloaded from: http://code.google.com/p/gomingw/downloads/list

Unzipping this file creates a go directory structure, with the map go/bin containing all the necessary binaries to run Go. Then you have to add the Go-variables manually:

Start, Computer, RMouse: Properties, in the Menu: choose Advanced System Settings; click the button: Environment Variables, System Variables, New:

Variable name	Variable value (for example)
GOROOT	c:\go
GOBIN	c:\go\bin
GOOS	windows
GOARCH	386

In the same dialog-window: Edit the PATH-variable: add the GOBIN map followed by ; in front, like this: %GOBIN%;…rest of PATH…

There is also a GO installer executable gowin32_releasedate_installer.exe which adds the Go-environment variables and adds GOBIN to the PATH.

Testing the installation: make the file `hello_world1.go` (see § 2.3 (6))

Compile this first Go-program with:	`8g test.go`
This compiles to a file:	`test.8`
which is linked with the command:	`8l test.8`
This produces the executable named:	`8.out.exe`
which executes with the command:	`8.out.exe`
and produces the output:	`hello, world`

Possible problem:

```
8g test.go      test.go:3: fatal error: can’t find import: fmt
```

The cause can be that the GO-variables are not set properly, or that the submap of pkg doesn’t have the right name, it should be: E:\Go\GoforWindows\gowin32_release\go\pkg\windows_386 for example, where release is substituted with the exact date or release number, e.g. 2011-01-06

You can set the GO-variables also on the command-line, like:

```
set GOROOT=E:\Go\GoforWindows\gowin32_release.r60\go
set GOBIN=$GOROOT\bin
set PATH=%PATH%;$GOBIN
```

Executing 8g then finds the import path to the packages.

(2) Go in a virtual machine running Linux in Windows:

In this option you run Go in e.g. Linux in a virtual machine in Windows.

A good choice is VMware: http:// www.vmware.com

Download the VMware player: http://www.vmware.com/products/player/

Search in Virtual Appliances for a Linux-platform, download and install it; then follow the instructions from § 2.3

(3) Go in Wubi: dual booting in Windows:

Another alternative is to install an Ubuntu-Linux side by side with Windows (using the Windows filesystem) through Wubi: http://www.wubi-installer.org. Then you have Linux without having to partition hard-drives. Install Go in this Linux by following the instructions from § 2.3

Of course a real partitioning of the hard-drive system of the machine can also be done, so that Linux is installed as a really separate OS, side by side with Windows.

Information on other OS-ports (FreeBSD, OpenBSD, DragonFlyBSD, NetBSD, Plan 9, Haiku, Solaris can be found here: http://go-lang.cat-v.org/os-ports

(4) Express-Go:

Express Go is a bytecode implementation of Go programming language for Windows made by Alexey Gokhberg and announced March 28 2011. The software is based on code of gc Go compiler and tools. The original code has been ported from C to C++ (Visual C++ 2008 Express Edition has been used as a development platform). The concepts, architecture, and code of the original gc implementation have been reused as much as possible, however, Express Go generates MSIL bytecode for the .NET virtual machine instead of the native machine code.

It includes an implementation of the virtual machine based on the just-in-time compilation (JIT) principle. The JIT virtual machine reads a bytecode executable file and compiles its content into the native machine code prior to execution. The virtual machine has been designed from scratch. The instruction set was intentionally made low-level and language-independent. High-level language-specific features are to be supported by the runtime libraries built into the interpreter. The original Go runtime library was partially ported to C++ and integrated into the interpreter of the virtual machine.Memory management is implemented using the Boehm-Demers-Weiser conservative garbage collector.

The binary distribution is located here:

http://www.unicorn-enterprises.com/download/express_go.zip

For more detailed information: see

http://www.unicorn-enterprises.com/express_go.html

(5) Building Go on Windows: To be able to do this, you first have to install the MINGW/ MINSYS environment. Note that in the MINGW environment you must use Unix like / instead of the Windows \ !

(A) Via MINGW zipfile:

With this method there is no restriction on where the MinGW folder can reside.

Download gowin-env.zip from https://bitbucket.org/akavel/gowin-env/downloads

An then unzip it into e.g. E:\Go\GoforWindows\MinGW

There are 2 batch files in the root of the MinGW folder: go-env.bat and run-mingw.bat

Set the Go environment variables in go-env.bat, for example:

```
set GOARCH=386
set GOOS=windows
set GOROOT=E:\Go\GoforWindows\gowin32_release.r59\go
set GOBIN=%GOROOT%\bin
set GOPATH=%GOROOT%
```

run-mingw.bat will import the environment variables from go-env.bat if it exists

Starting the MINGW-environment via command window:

open the command window: `cmd` in Start, Search box
go to MINGW installation map: e.g. `cd E:\Go\GoforWindows\MinGW`
`run-mingw.bat    (or run-mingw-mintty.bat)`
this starts a new command window

(B) Via Windows installation:

Download and save the latest version from:

http://sourceforge.net/projects/mingw/files/Automated%20MinGW%20Installer/mingw-get-inst/

Open and run the saved automated MinGW installer executable file, which is named mingw-get-inst-yyyymmdd.exe, where yyyymmdd is the version date stamp (e.g. mingw-get-inst-20110802.exe).

The MinGW Setup Wizard window will open with the title "Setup—MinGW-Get". Except for the following, accept the setup defaults, unless it's necessary to change them.

For Repository Catalogues: check the Download latest repository catalogues button.

For Select Components: the MinGW Compiler Suite, the C Compiler box is automatically checked. Scroll down to the bottom of the list and check the MinGW Developer ToolKit box, which includes the MSYS Basic System.

Before Ready to Install, review and verify the installation settings, which should look similar this:

```
Installing:
        mingw-get
        pkginfo
        C Compiler
        MSYS Basic System
        MinGW Developer Toolkit
    Downloading latest repository catalogues
    Destination location:
        C:\MinGW
```

When the installation settings are correct, install.

The installation loads the package installation catalogues and downloads and installs the files. The installation may take some time, largely depending on the download speed.

Add the Go-environment variable settings (see A)) in the beginning of msys.bat

The MSYS command window may be opened by opening and running the C:\MinGW\msys\1.0\msys.bat batch file or via the Start menu by choosing: MinGW / MinGW Shell.

<u>How to build Go on Windows:</u>

To download the Go-source you will need an hg client for Windows, download and install the binary package from http://mercurial.selenic.com/wiki/Download

Download the Go source by entering the following on the command line:

```
$ hg clone -r release https://go.googlecode.com/hg/ $GOROOT
```

Make a bin directory under `$GOROOT`.

cd to the src directory: `$ cd GOROOT/src` and then build Go with: `$ ./all.bash`

2.6 What is installed on your machine?

The Go tree as the installation is called has the following structure under the go-root map ($GOROOT):

README, AUTHORS, CONTRIBUTORS, LICENSE

\bin	all executables, like the compilers and the go-tools
\doc	tutorial programs, codewalks, local documention, talks, logo's, …
\include	C/C++ header files
\lib	templates for the documentation
\misc	configuration files for editors for working with Go, cgo examples, …
\pkg\os_arch	with os_arch e.g. linux_amd64, windows_386, …
	the object files (.a) of all the packages of the standard library
\src	bash-scripts and make command files
\cmd	scripts and source files (Go, C) of the compilers and commands

\lib9 \libbio \libmach : C-files

\pkg	Go source files of all packages in the standard library (it is open source!)

On Windows 386 release r59 comprises 3084 files in 355 maps, amounting to 129 Mb.

On Linux 64 bit release r60 comprises 3958 files, totaling 176 Mb.

2.7 The Go runtime

Although the compiler generates native executable code, this code executes within a *runtime* (the code of this tool is contained in the package runtime). This runtime is somewhat comparable with the virtual machines used by the Java and .NET-languages. It is responsible for handling memory allocation and garbage collection (see also § 10.8), stack handling, goroutines, channels, slices, maps, reflection, and more.

runtime is the "top level" package that is linked into every Go package, and it is mostly written in C. It can be found in $GOROOT/src/pkg/runtime/ (see the mgc0.c and other m* files in that Directory).

Garbage collector: 6g has a simple but effective mark-and-sweep collector. Work is underway to develop the ideas in IBM's Recycler garbage collector to build a very efficient, low-latency concurrent collector. Gccgo at the moment has no collector; the new collector is being developed for both compilers. Having a garbage-collected language doesn't mean you can ignore memory allocation issues: allocating and deallocating memory also uses cpu-resources.

Go executable files are much bigger in size than the source code files: this is precisely because the Go runtime is embedded in every executable. This could be a drawback when having to distribute the executable to a large number of machines. On the other hand deployment is much easier than with Java or Python, because with Go everything needed sits in 1 static binary, no other files are needed. There are no dependencies which can be forgotten or incorrected versioned.

2.8 A Go interpreter

Because of the fast compilation and (as we will see) the resemblance with dynamic languages the question easily arises whether Go could be implemented in a *read-eval-print loop* like in these languages. Such a Go interpreter has been implemented by Sebastien Binet and can be found at:

https://bitbucket.org/binet/igo

Chapter 3—Editors, IDE's and Other tools.

Because Go is still a young language work on (plugins for) IDE's is still in progress, however there is some very decent support in certain editors. Some environments are cross-platform (indicated by CP) and can be used on Linux, OS X and Windows.

Consult http://go-lang.cat-v.org/text-editors/ for the latest information.

3.1 Basic requirements for a decent Go development environment

What can you expect from an environment giving above what you could accomplish with a simple text-editor and the compiler/link tools on the command-line ? Here is an extensive wish-list:

(1) **Syntax highlighting:** this is of course crucial, and every environment cited provides configuration- or settings files for this purpose; preferably different color-schemes (also customizable) should be available.
(2) **Automatic saving** of code, at least before compiling.
(3) **Line numbering** of code must be possible.
(4) Good **overview and navigation** in the codebase must be possible; different source-files can be kept open.
Possibility to set **bookmarks** in code.
Bracket matching.
From a function call, or type use, **go to the definition** of the function or type.
(5) Excellent **find** and **replace** possibilities, the latter preferably with preview.
(6) Being able to **(un)comment** lines and selections of code.
(7) **Compiler errors**: double clicking the error-message should highlight the offending codeline.
(8) **Cross platform**: working on Linux, Mac OS X and Windows so that only 1 environment needs to be learned / installed.
(9) Preferably **free**, although some developers would be willing to pay for a qualitatively high environment, which distinguishes itself from the rest.
(10) Preferably **open-source**

(11) **Plugin-architecture**: so its relatively easy to extend or replace a functionality by a new plugin.
(12) **Easy to use**: an IDE is a complexer environment, but still it must have a lightweight feel.
(13) **Code snippets (templates):** quick insertion of of much used pieces of code can ease and accelerate coding.
(14) The concept of a **Go project,** with a view of its constituent files and packages, and where the Makefile typically plays the role of configuration file. Closely related is the concept of a **build system**: it must be easy to compile/link (= build), clean (remove binaries) and/or run a program or a project. Running a program should be possible in console view or inside the IDE.
(15) **Debugging** capabilities (breakpoints, inspection of values, stepping through executing code, being able to step over the standard library code).
(16) Easy acces to **recent files and projects.**
(17) **Code completion** (intellisense) capabilities: syntax-wise (keywords), packages, types and methods within packages, program types, variables, functions and methods; function signatures.
(18) **An AST-view** (abstract syntax tree) **of a project/package code**.
(19) **Built-in go tools**, such as: go fmt, go fix, go install, go test, …
(20) Convenient and integrated **browsing of go documentation.**
(21) Easy switching between **different Go-environments** (8g, 6g, different installation root, …)
(22) **Exporting code to different formats,** such as pdf, html or even **printing** of the code.
(23) **Project templates** for special kind of project (such as a web application, an App Engine project) to get you started quickly.
(24) **Refactoring** possibilities.
(25) **Integration with version control-systems** like hg or git.
(26) Integration with **Google App Engine.**

3.2 Editors and Integrated Development Environments

Syntax highlighting and other Go-utilities exist for the following editors: Emacs, Vim, Xcode3, KD Kate, TextWrangler, BBEdit, McEdit, TextMate, TextPad, JEdit, SciTE, Nano, Notepad++, Geany, SlickEdit, SublimeText2.

The **GEdit** Linux text-editor can be made into a nice Go environment (see http://gohelp.wordpress.com/).

SublimeText (http://www.sublimetext.com/dev) is an innovative cross platform text-editor (Linux, Mac OSX, Windows) with extensions for a lot of programming languages, in particular for Go a plugin GoSublime exists (https://github.com/DisposaBoy/GoSublime) which provides code completion and code snippets.

Here are the more elaborated environments for Go-programming; some of them are plugins for existing (Java) environments:

NetBeans: (commercial) Go For NetBeans plug-in, provides syntax highlighting and code templates http://www.winsoft.sk/go.htm ; a new free plugin is in the making: http://www.tunnelvisionlabs.com/downloads/go/

gogo: A basic environment for Linux and Mac: http://www.mikeparr.info/golinux.html

GoIde: (CP) is a plugin for the IntelliJ IDE: http://go-ide.com/
Download it from https://github.com/mtoader/google-go-lang-idea-plugin
Nice editing features and code-completion support,
cfr: http://plugins.intellij.net/plugin/?idea&id=5047

LiteIDE (golangide): (CP) is a nice environment for editing, compiling and running Go programs and -projects. Contains an abstract syntax tree (AST) viewer of the source code and a built-in make tool: http://code.google.com/p/golangide/downloads/list

GoClipse:(CP) is a plugin for the Eclipse IDE: http://goclipse.googlecode.com/svn/trunk/goclipse-update-site/; Interesting features are automatic make file creation, and a kind of code completion (Content Assist via GoCode)

If you are not familiar with the IDE's, go for LiteIDE, else GoClipse or GoIde are good choices.

Code completion is built in (e.g. in LiteIDE and GoClipse) through the *GoCode plugin*:
To install gocode:

```
$ git clone git://github.com/nsf/gocode.git
$ cd gocode
$ export GOROOT=$HOME/go
$ export PATH=$HOME/go/bin:$PATH
$ make install
```

Test gocode status:

```
$ gocode status
```

should give as output:

```
Server's GOMAXPROCS == 2
Package cache contains 1 entries
Listing these entries:
        name: unsafe (default alias: unsafe)
        imports 9 declarations and 0 packages
        this package stays in cache forever (built-in package)
```

In the following table we list the IDE requirements summed up in § 3.1 for the environments which are most advanced at this time, + means it works, ++ is better, blanc means it does not.

	Golang LiteIDE	**GoClipse**	**GoIde**
Syntax highlighting	++	+	+
Automatic saving before building	+		
Line numbering	+	+	
Bookmarks			
Bracket matching		+	
Find / Replace	+	++	
Go to definition			
(Un)Comment		+	
Compiler error double click	++	+	
Cross platform	+	+	
Free	+	+	
Open source	+	+	
Plugin-architecture	+	+	
Easy to use	++	+	
Code snippets			
Project concept	+	+	
Code Folding			
Build system	+	+	
Debugging	+	+	

Recent files and projects	+	+	
Code completion	+	++	
AST-view of code	++	+	
Built-in go tools	+	+	
Browsing of go documentation	+		
Easy switching different Go-environments	++	+	
Exporting code to different formats	++	+	
Project templates			
Integration with version control-systems			
Integration with Google App Engine		++	

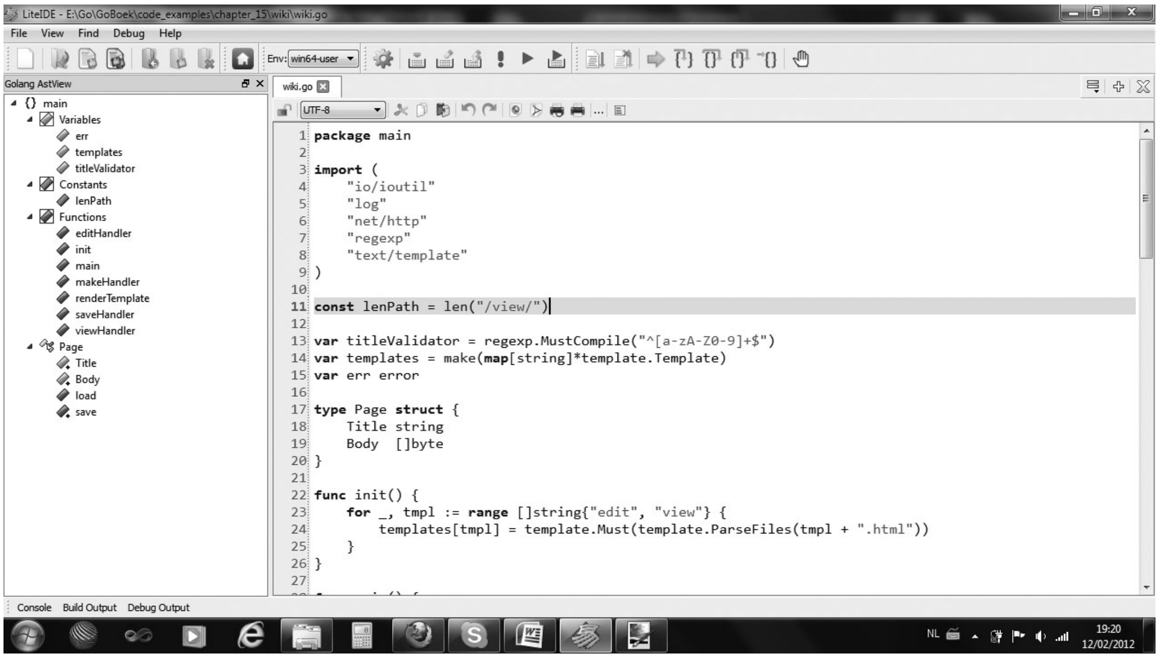
Here follows a more detailed discussion of LiteIDE and GoClipse.

3.2.1. Golang LiteIDE

The current version is X10; website: http://code.google.com/p/golangide/)

LiteIDE is a nice lightweight IDE with a plugin-architecture (based on QT, Kate and SciTE), containing all necessary features for comfortable Go cross-platform development, with very good editing, code-completion and debugging support. It has the concept of a Go project, tied to a Makefile when this is present in its root directory. It is possible to work side by side with different Go-environments because you can use different environment (.env)-files with their own Go-variables, for example for 32 bit / 64 bit or different releases (a stable releases versus a weekly release for instance).

Very nice also is the AST-view of the code, giving a nice overview of the constants, variables, functions and the different types, with their properties and methods.



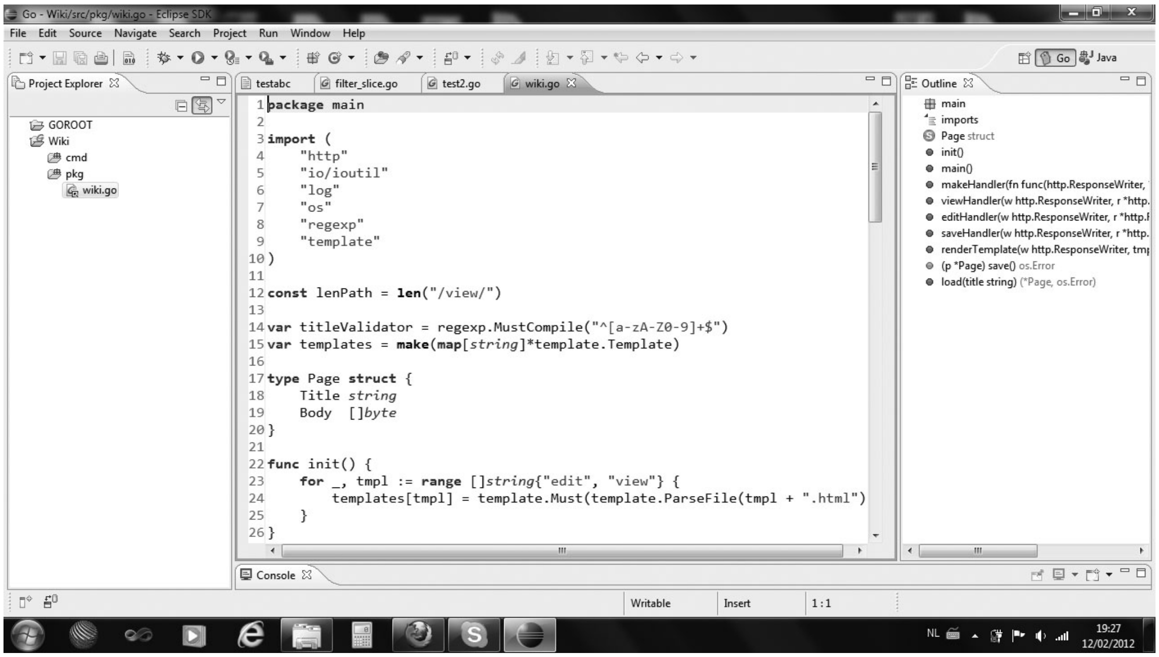
```go
package main

import (
    "io/ioutil"
    "log"
    "net/http"
    "regexp"
    "text/template"
)

const lenPath = len("/view/")

var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")
var templates = make(map[string]*template.Template)
var err error

type Page struct {
    Title string
    Body  []byte
}

func init() {
    for _, tmpl := range []string{"edit", "view"} {
        templates[tmpl] = template.Must(template.ParseFiles(tmpl + ".html"))
    }
}

```

Fig 3.1: LiteIDE and its AST-view

3.2.2. GoClipse

(Current version: 0.2.1; website: http://code.google.com/p/goclipse/)

This is a plugin for the well known Eclipse-environment, a big environment with starts rather slow and depends on a Java VM installed, but on the other hand it can use much of the built-in Eclipse functionality. A nice editor, code-completion and outline, project view and debugging support.


Go - Wiki/src/pkg/wiki.go - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Project Explorer
GOROOT
Wiki
cmd
pkg
wiki.go
testabc
filter_slice.go
test2.go
wiki.go
Outline
main
imports
Page struct
init()
main()
makeHandler(fn func(http.ResponseWriter,
viewHandler(w http.ResponseWriter, r *http.
editHandler(w http.ResponseWriter, r *http.
saveHandler(w http.ResponseWriter, r *http.
renderTemplate(w http.ResponseWriter, tm
(p *Page) save() os.Error
load(title string) (*Page, os.Error)
Console
Writable
Insert
1:1
Go
Java
NL
19:27
12/02/2012

```
package main

import (
    "http"
    "io/ioutil"
    "log"
    "os"
    "regexp"
    "template"
)

const lenPath = len("/view/")

var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")
var templates = make(map[string]*template.Template)

type Page struct {
    Title string
    Body  []byte
}

func init() {
    for _, tmpl := range []string{"edit", "view"} {
        templates[tmpl] = template.Must(template.ParseFile(tmpl + ".html")
    }
}
```

Fig 3.2: GoClipse and its outline code-view

3.3 Debuggers

Application programming needs good debugging support, and in this area still a lot of work needs to be done. A debugger (***Oogle***): support for gdb (at least version 7 is needed), the GNU debugger is being built in Go's gc linker (6l, 8l) by the Go-team from Google (not for Windows and ARM-platforms) (see also http://blog.golang.org/2010/11/debugging)

Support for gdb version 7.1 is build in in LiteIDE and Goclipse.

If you don't want to use a debugger, the following is useful in a simple debugging strategy:

1) use print-statements (with print / println and the fmt.Print functions) at well chosen places
2) in fmt.Printf functions use the following specifiers to obtain complete info about variables:
    %+v gives us a complete output of the instance with its fields
    %#v gives us a complete output of the instance with its fields and qualified type name
    %T gives us the complete type specification

3) use a panic-statement (see § 13.2) to obtain a stack trace (a list of all the called functions up until that moment)
4) use the defer keyword in tracing code execution (see § 6.4).

3.4 Building and running go-programs with command- and Makefiles

Messages from the compiler:

When a program has been written (applying common naming- and style-rules), subject it to *gofmt* (see §3.5 to format it correctly), and then build (compile / link) it; if the build-process (which is also called *compile-time*) doesn't give any errors, the message appears:

```
---- Build file exited with code 0 .
```

In most IDE's, building also saves the latest changes to the source-file.

If building produces (an) error(s), we get the following output:

```
---- Build file exited with code 1,
```

and the line number where the error occurs in the source-code is indicated, like:

```
hello_world.go:6: a declared and not used
```

In most IDE's, double-clicking the error line positions the editor on that code line where the error occurs.

Go does not distinguish between warnings and errors. The philosophy is: when it is worth warning for a misuse of something, it better be signaled as an error to always be on the safe side. If you try to execute a program which is not yet compiled, you get the following error:

```
---- Error run with code File not found, resource error
```

When executing a program you are in the *run-time* environment.

If the program executes correctly, after execution the following is output:

```
---- Program exited with code 0
```

The –o flag:

The compiler and linker also have the –o flag to give the executable a simpler name: if we have compiled a program test.go to test.6 and want our executable to have the name test instead of test.6, issue the following link-command:

```
6l -o test test.6
```

Command (batch) file in Windows:

A run.cmd or run.bat file containing the following commands (put the path to your Go root map in the 1st line):

```
set GOROOT=E:\Go\GoforWindows\gowin32_release.r60\go
set GOBIN=$GOROOT\bin
set PATH=%PATH%;$GOBIN
set GOARCH=386
set GOOS=windows
echo off
8g %1.go
8l -o %1.exe %1.8
%1
```

can be used to run test.go in a command-window (cmd) like: `run test`

Makefile in Linux and OS X:

You can build/link without having to know which of the 5-, 6- or 8- compilers/linkers you need to use by using makefiles. Produce a text file called Makefile in the map with your .go-files with the following content:

```
include $(GOROOT)/src/Make.inc
TARG=test
GOFILES=\
        test1.go\
test2.go\

include $(GOROOT)/src/Make.cmd
```

and make it executable with chmod 777 `./Makefile`

Fill in the variable GOFILES with all the go-files that need to be compiled and linked separated by a space. Leave a blank line before the include line. TARG contains the name of the executable. Then run `make` on the command-line: this compiles/links the source-files, but only when the source-code has changed since the last make run; `./test` executes the program.

If needed you can make your own variables in a Makefile like VAR1, and their value is substituted in the expression $(VAR1) .

From Go 1 onwards, the preferred way is to use the **`go`** tool:

`go build`	compiles and installs packages and dependencies
`go install`	install packages and dependencies

Instead of `make` you can use the tool **`gomake`**, which is included in the Go-distribution. It was the common build tool before Go 1(see §3.4, §9.5 for packages and § 13.7 for testing):

If $GOROOT is already set in the environment, running gomake is exactly the same as running make. Usage: gomake [target ...]

Common targets are:

all	(default) build the package or command, but do not install it.
install	build and install the package or command
test	run the tests (packages only)
bench	run benchmarks (packages only)
clean	remove object files from the current directory
nuke	make clean and remove the installed package or command

If it is convenient to use different Makefiles, like Makefile_Proj1, Makefile_Proj2, etc., you can invoke them separately by issuing the command with the –f flag: gomake –f Makefile_Proj1

Including code formatting:

If you append the following lines to Makefile:

```
format:
      gofmt -w $(GOFILES)
```

then gomake format will invoke gofmt on your source-files.

Makefile in Windows:

A Makefile and make can also be used in Windows by using MINGW (Minimalist GNU environment for Windows, see http://www.mingw.org/). In order to be able to run the Go toolchain (like gomake, gotest, etc.) the Windows-environment must be made more Unix-like. This can be done through MINGW, which even gives you the possibility of building Go on Windows (see §2.5 (5)).

A limited companion environment to make canonical Go-Makefiles work in a Windows environment including MSYS can be found at https://bitbucket.org/akavel/gowin-env .

Download gowin_env.zip and unzip it, this produces a map gowin_env, with submap msys and some bat-files. Open console.bat in Notepad or Wordpad and edit the line set GOROOT= to the root of your Go-installation, for example:

```
set GOROOT=E:\Go\GoforWindows\gowin32_release.r57.1\go
```

Do the same for go_env.bat. Then double click console.bat and change to the map where you have your Makefile ready, and invoke the make command to compile and link the programs indicated in the Makefile.

Compiling all go-files in a directory:

Here is the Linux-version of a useful script for the purpose of quick testing for compilation errors in a lot of files:

Listing 3.1—gocomp:

```
#!/bin/bash
FILES=~/goprograms/compiletest/*.go
for f in $FILES
do
  echo “Processing $f file...”
  # take action on each file. $f stores current file name
  # cat $f
  6g $f >> compout
done
```

You need to replace ~/goprograms/compiletest/ with your map name; the output of the compilation is then appended to the file compout.

A Windows-version is just as easy to write, e.g. a gocomp.bat file with this line:

```
FOR %%X in (*.go) DO 8g %%X >> compout
```

will compile all the go-files in the current directory and write the output to compout.

3.5 Formatting code: go fmt or gofmt

The Go-authors didn't want endless discussions about code-style for the Go-language, discussions which in the past have risen for so many programming languages and that were certainly to an extent a waste of precious development time. So they made a tool: **go fmt** (or formerly gofmt). It is a pretty-printer which imposes the officia*l, standard code formatting and styling* on the Go source-code. It is a syntax level rewriting tool, a simple form of refactoring. It is used rigorously in Go-development, and should be used by all Go-developers: use gofmt on your Go-program before compiling or checking in!

Although not without debate, the use of gofmt and as a consequence the syntax-liberty you have to give up certainly have big advantages in making Go-code uniform and better readable and thus facilitates the cognitive task of comprehending foreign Go-code. Most editors have it built in.

For *indentation* of different levels in code the rule is not strict: tabs or spaces can be used, a tab can be 4 or 8 spaces. In the real code-examples and exercises *1 tab takes the width of 4 spaces*, in the printed examples in this book for clarity *1 tab equals 8 spaces.* In writing code in an editor of IDE, use the tab, don't insert spaces.

On the command-line: `gofmt –w program.go` reformats program.go (without –w the changes are shown, but not saved). `gofmt –w *`.go works on all go-files. `gofmt map1` works recursively on all .go files in the map1 directory and its subdirectories.

It can also be used for simple changes (refactoring) in a codebase by specifying a rule with the –r flag and a rule between ‘ ; the rule has to be of the form: `pattern -> replacement`

Examples:

```
gofmt -r "(a) -> a" –w *.go
```

this will replace all unnecessary doubled (( )) with () in all go-files in the current directory.

```
gofmt -r "a[n:len(a)] -> a[n:]" –w *.go
```

this will replace all superfluous len(a) in such slice-declarations

```
gofmt –r ‘A.Func1(a,b) -> A.Func2(b,a)’ –w *.go
```

this will replace Func1 with Func2 and swap the function’s arguments

For more info see: http://golang.org/cmd/gofmt/

3.6 Documenting code: go doc or godoc

`go doc` extracts and generates documentation for Go programs and packages.

It extracts comments that appear before top-level declarations in the source-code, with no intervening newlines: together with the declaration they serve as explanatory text for the item.

It can also function as a web server delivering documentation; this is how the golang-website http://golang.org works.

General format:

`go doc package`

to get the ‘package comment’ documentation for package, ex.: `go doc fmt`
this comment will appear first on the output produced by godoc.

`go doc package/subpackage`

to get the documentation for subpackage in package, ex: `go doc container/list`

`go doc package function`

to get the documentation for function in package, ex.: `go doc fmt Printf`
provides explanations for the use of fmt.Printf()

These commands only work when the Go-source is located under the Go-root: ../go/src/pkg.

The command godoc itself has some more options.

Godoc can also be started as a local webserver: `godoc -http=:6060` on the command-line starts the server, then open a browser window with the url: http://localhost:6060 and you have a local documentation server, without the need for an internet-connection.

Godoc can also be used to generate documentation for self-written Go-programs: see §9.6.

For more info see: http://golang.org/cmd/godoc/

3.7 Other tools

The tools in the Go-toolset (http://golang.org/cmd/) are partly shell-scripts and also written in Go itself. From Go 1 onwards they are implemented as commands for the go tool:

`go install` (formerly **`goinstall`**) is the go-package install tool; much like rubygems for the Ruby language. It is meant for installing go packages outside of the standard library, and it works with the source-code format of the package, which then has to be locally compiled and linked (see also § 9.7).

`go fix` (formerly **`gofix`**) is a tool that you can use to update Go source-code (outside of the standard packages) from an older to the newest release: it tries to automatically 'fix' changes. It is meant to reduce the amount of effort it takes to update existing code, automating it as much as possible. gofix takes care of the easy, repetitive, tedious changes; if a change in API isn't simply a substitution of a new function and requires manual examination, gofix prints a warning giving the file name and line number of the use, so that a developer can examine and rewrite the code. The Go-team regularly updates the tool together with new API changes, and it is also used internally in Google to update Go source code. go fix works because Go has support in its standard libraries for parsing Go source files into (abstract) syntax trees (AST's) and also for printing those syntax trees back to Go source code.

`go fix .` tries to update all .go-files in the current directory when necessary; the changed filenames are printed on standard output.

Examples:—show differences with the installed Go-release: `go fix -diff .`
- update to changes: `go fix -w .`

`go test` (formerly **`gotest`**) is a lightweight test framework for unit-testing (see chapter 13)

3.8 Go's performance

According to the Go-team and measured in simple algorithmic programs, performance is *typically within 10-20% of C*. There are no official benchmarks, but regular experimental comparisons between languages show a very good performance track.

A more realistic statement is that *Go is 20% slower than C++* . This puts Go programs at conservatively twice as fast and requiring 70% less memory when compared to an equivalent Java or Scala application. In many cases that difference is irrelevant, but for a company like Google with thousands of servers the potential efficiency improvements are certainly worth the investment.

The current popular languages execute in a virtual machine: JVM for Java and Scala, .NET CLR for C#, etc..Even with the massive improvements in virtual machines, JIT-compilers and scripting language interpreters (Ruby, Python, Perl, Javascript) none can come close to C and C++ for performance.

If Go is 20% slower than C++, that's still 2 to 10 times faster than any language that's not statically typed and compiled, and it is far more memory efficient.

Comparing benchmarks of a program in 2 or more languages is very tricky: the programs each should do exactly the same things, utilizing the best possible concepts and techniques for that task from the language. For example when processing text, the language that processes this as raw bytes will almost certainly outperform the language that works with it as Unicode. The person performing the benchmarks often writes both programs, but often he/she is much more experienced in one language than in the other, and this can of course very much influence the results: each program should be written by a developer well versed in the language. Otherwise, like in statistics, it is not difficult to artificially influence the performance behavior of one language compared to another; it is not an exact science. The outcome can also depend upon the problem to solve: in most cases older languages have optimally tuned libraries for certain tasks, and it must be taken into account that some of the Go-libraries are still early versions. For a lot of results, see The Computer Language Benchmark Game—http://shootout.alioth.debian.org/ (ref 27).

Some benchmarking results:

(1) Comparing Go and Python on simple webserver applications, measured in transactions/s:

The native Go http package is 7-8 x faster than web.py, web.go slightly less performant with a ratio of 6-7 compared to web.py. The *tornado* asynchronous server/framework much used in Python web environments performs considerably better than web.py: Go outperforms tornado only with a factor of 1.2 to 1.5 (see ref 26).

(2) Go is on average 25 x faster than Python 3, uses 1/3 of the memory, with number of lines of code almost equal to double. (see ref 27).
(3) The article by Robert Hundt (Jun 2011, ref 28) comparing C++, Java, Go, and Scala, and the reaction of the Go team (ref 29): some conclusions taking into account the tuning from the Go-team:
    - The Scala and Go versions are significantly more compact (less code lines) than the verbose C++ and Java
    - Go compiles quicker than alle others, 5-6x comparing to Java and C++, 10x compared to Scala
    - Go has the largest binary size (every executable contains the runtime)

- Optimized Go code is as fast as C++, 2 to 3x faster than the Scala version and 5 to 10x faster than Java.
- Its memory consumption is also very much comparable to that of C++, nearly half of what Scala required, and it was nearly 4x less than Java

3.9 Interaction with other languages.

3.9.1. Interacting with C

The cgo program provides the mechanism for FFI-support (foreign function interface) to allow safe calling of C libraries from Go code: http://golang.org/cmd/cgo is the primary cgo documentation (see also ref. 23). cgo replaces the normal Go-compilers, it outputs Go and C files that can be combined into a single Go package. It is good practice to combine the calls to C in a separate package.

The following import is then necessary in your Go program: `import “C”`
(this needs to be on a line of its own)
and usually also: `import “unsafe”`

You can include C-libraries (or even valid C-code) by placing these statements as comments (with // or /* */) immediately above the import “C” line:

```
// #include <stdio.h>
// #include <stdlib.h>
import “C”
```

C is not a package from the standard library, it is simply a special name interpreted by cgo as a reference to C’s namespace. Within this namespace exist the C types denoted as C.uint, C.long, etc. and functions like C.random() from libc can be called.

Variables in the Go program have to be converted to the C type when used as parameter in C functions, and vice-versa, examples:

```
var i int
C.uint(i)          // from Go int to C unsigned int
int(C.random())    // from C (random() gives a long) to Go int
```

The following program contains 2 Go functions Random() and Seed(), which call the equivalent C functions C.random() and C.srandom():

Listing 3.2—c1.go:

```
package rand
// #include <stdlib.h>
import “C”
func Random() int {
   return int(C.random())
}
func Seed(i int) {
   C.srandom(C.uint(i))
}
```

Strings do not exist as explicit type in C: to convert a Go string s to its C equivalent use: C.CString(s). The reverse is done with the function C.GoString(cs), where cs is a C ‘string’.

Memory allocations made by C code are not known to Go’s memory manager.

It is up to the developer to free the memory of C variables with **C.free**, as follows:

```
defer C.free(unsafe.Pointer(Cvariable))
```

This line best follows the line where Cvariable was created, so that the release of memory is not forgotten. The following code contains a Go function Print() which prints a string to the console by using the C function fputs from stdio, explicitly freeing the the memory used:

Listing 3.3—c2.go:

```
package print

// #include <stdio.h>
// #include <stdlib.h>
import “C”
import “unsafe”

func Print(s string) {
   cs := C.CString(s)
   defer C.free(unsafe.Pointer(cs))
   C.fputs(cs, (*C.FILE)(C.stdout))
}
```

Building cgo packages:

A Makefile like in §9.5 (because we create a separate package) can be used here; apart from the GOFILES variable, a variable CGOFILES now lists the files to be compiled with cgo. For example Listing 3.2 could be compiled in a package rand with the following Makefile, issuing the command gomake or make:

```
include $(GOROOT)/src/Make.inc

TARG=rand
CGOFILES=\
   c1.go\

include $(GOROOT)/src/Make.pkg
```

3.9.2. Interacting with C++

SWIG (Simplified Wrapper and Interface Generator) support exists for calling C++ and C code from Go on Linux. Using SWIG is a bit more involved:

- Write the SWIG interface file for the library to be wrapped
- SWIG will generate the C stub functions
- These can then be called using the cgo machinery
- the Go files doing so are automatically generated as well

This interface handles overloading, multiple inheritance and allows to provide a Go implementation for a C++ abstract class.

A problem is that SWIG doesn't understand all of C++, e.g. it can't parse TObject.h

PART 2

CORE CONSTRUCTS AND TECHNIQUES OF THE LANGUAGE

Chapter 4—Basic constructs and elementary data types

4.1. Filenames—Keywords—Identifiers

Go source-code is stored in .go files, these *filenames* consist of lowercase-letters, like `scanner.go` If the name consist of multiple parts, there are separated by underscores _, like `scanner_test.go` Filenames may not contain spaces or any other special characters.

A source-file contains code lines, who's length have no intrinsic limit.

Nearly all things in Go-code have a name or an *identifier.* Go, like all languages in the C-family, is *case-sensitive*. Valid identifiers begin with a letter (a *letter* is every letter in Unicode UTF-8 or _), and followed by 0 or more letters or Unicode digits, like: `X56, group1, _x23, i, θε12`

The following are NOT valid identifiers:

> `1ab` (starts with digit), `case` (= keyword in Go), `a+b` (operators are not allowed)

The _ itself is a special identifier, called the *blank identifier.* It can be used in declarations or variable assignments like any other identifier (and any type can be assigned to it), but its value is discarded, so it cannot be used anymore in the code that follows.

Sometimes it is possible that variables, types or functions have no name because it is not really necessary at that point in the code and even enhances flexibility: these are called *anonymous.*

This is the set of 25 *keywords* or reserved words used in Go-code:

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**

It is kept deliberately small to simplify the code-parsing, the first step in the compilation process. A keyword cannot be used as an identifier.

Apart from the keywords Go has a set of 36 *predeclared identifiers*: these contain the names of elementary types and some basic built-in functions (see § 6.5); all these will be explained further in the next chapters:

append	**bool**	**byte**	**cap**	**close**	**complex**	**complex64**	**complex128**	**uint16**
copy	**false**	**float32**	**float64**	**imag**	**int**	**int8**	**int16**	**uint32**
int32	**int64**	**iota**	**len**	**make**	**new**	**nil**	**panic**	**uint64**
print	**println**	**real**	**recover**	**string**	**true**	**uint**	**uint8**	**uintptr**

Programs consist out of *keywords, constants, variables, operators, types and functions*.

The following delimiters are used: parentheses ( ), brackets [ ] and braces { }.

The following punctuation characters . , ; : and … are used.

Code is structured in *statements*. A statement doesn't need to end with a ; (like it is imposed in the C-family of languages). The Go compiler automatically inserts semicolons at the end of statements.

However if multiple statements are written on one line (a practice which is not encouraged for readability reasons), they must be separated by ;

4.2. Basic structure and components of a Go-program

Listing 4.1 hello_world.go:

```
package main

import "fmt"

func main() {
    fmt.Println("hello, world")
}
```

4.2.1 Packages, import and visibility

Packages are a way to structure code: a program is constructed as a "package" (often abbreviated as pkg), which may use facilities from other packages.

Every go-file belongs to one (and only one) *package* (like a library or namespace in other languages). Many different .go files can belong to one package, so the filename(s) and package name are generally not the same.

The package to which the code-file belongs must be indicated on the first line, e.g.: `package main`. A standalone executable belongs to package main. Each Go application contains one package called **main**.

An application can consist of different packages, but even if you use only package main, you don't have to stuff all code in 1 big file: you can make a number of smaller files each having package main as 1st codeline. If you compile a source file with a package name other than main, like pack1, the object file is stored in pack1.a; a package name is written in lowercase letters.

Standard library:

The Go installation contains a number of ready-to-use packages, which form the *standard library*. On Windows the directories of the standard library can be found in the subdirectory pkg\windows_386 of the Go-root map. On Linux the directories of the standard library can be found in the subdirectory pkg\linux_amd64 of the Go-root map (or linux_amd32 in case of a 32 bit installation). The general path (the global Go tree) where the standard library can be found is `$GOROOT/pkg/$GOOS_$GOARCH/` .

The standard library of Go contains a lot of packages (like fmt, os), but you can also create your own packages (see chapter 8).

To build a program, the packages, and the files within them, must be compiled in the correct order. Package dependencies determine the order in which to build packages.

Within a package, the source files must all be compiled together. The package is compiled as a unit, and by convention each directory contains one package.

If a package is changed and recompiled, all the client programs that use this package must be recompiled too!

The package model uses *explicit dependencies* to enable faster builds: the Go compiler pulls transitive dependency type info from the object file .o—but only what it needs.

If A.go depends on B.go depends on C.go:

- compile C.go, B.go, then A.go.
- to compile A.go, compiler reads B.o not C.o.

At a large scale, this can be a huge speedup.

Every piece of code is compiled only once.

A Go program is created by linking together a set of packages through the import keyword.

`import "fmt"` tells Go that this program needs (functions, or other elements, from) the package fmt, which implements functionality for formatted IO. The package names are enclosed within " ". Import loads the public declarations from the compiled package, it does not insert the source code.

If multiple packages are needed, they can each be imported by a separate statement:

```
        import "fmt"
        import "os"
or:     import "fmt"; import "os"
```

but a shorter and more elegant way (called *factoring the keyword,* also applicable to **const**, **var** and **type**) is available:

```
        import (
                "fmt"
                "os"
        )
```

(It can be even shorter: `import ("fmt"; "os")` but gofmt enforces the distributed version)

Only apply this when there is more than one entry; in that case it is also clearer to list the package names in alphabetical order.

If the name of a package does not start with . or /, like "fmt" or "container/list", Go looks for it in the global Go tree. If it starts with ./ the package is searched in the actual directory; starting with / (even on Windows) it is searched for in the (absolute) path indicated.

Packages contain all other code-objects.

Apart from _ , identifiers of code-objects have to be unique in a package: there can be *no naming conflicts*. But the same identifier can be used in different packages: the *package name qualifies it* to be different.

Packages expose their code-objects to code outside of the package according to the following rule, which is enforced by the compiler :

VISIBILITY RULE:

When the identifier (of a constant, variable, type, function, struct field, …) starts with an *uppercase letter*, like Group1, then the 'object' with this identifier *is visible in code outside the package* (thus available to client-programs, 'importers' of the package), it is said to be *exported* (like public in OO languages). Identifiers which start with a lowercase letter are not visible outside the package, but they are visible and usable in the whole package (like private).

(Capital letters can come from the entire Unicode-range,like Greek; not only ASCII letters are allowed.)

So importing a package gives (only) access to the exported objects in that package.

Suppose we have a thing (variable or function) called Thing (starts with T so it is exported) in a package pack1, then when pack1 is imported in the current package, Thing can be called with the usual *dot-notation* from OO-languages: pack1.Thing (The pack1. is necessary !)

So packages also serve as *namespaces* and can help to avoid name-clashes (name-conflicts)*:* variables with the same name in two packages are differentiated by their package name, like:

`pack1.Thing` and `pack2.Thing`

A package can, if this is useful (for shortening, name conflicts, …), also be given another name (an *alias)*, like: `import fm "fmt"`. The alias is then used in the following code:

Listing 4.2—alias.go:

```
package main
import fm “fmt” // alias3

func main() {
    fm.Println(“hello, world”)
}
```

Remark: importing a package which is not used in the rest of the code is a build-error (for example: imported and not used: os) . This follows the Go-motto: “no unnecessary code! ”

Package level declarations and initializations:

After the import statement 0 or more constants (const), variables (var), and types (type) can be declared; these are *global* (have *package scope)* and are known in all functions in the code (like c and v in gotemplate.go below), and they are followed by one or more functions (func).

4.2.2 Functions

The simplest function declaration has the format: `func functionName()`

Between the mandatory parentheses ( ) no, one, or more *parameters* (separated by ,) can be given as input to the function. After the name of each parameter variable must come its type.

A `main` function as starting is required (usually the first func), otherwise the build-error `undefined: main.main` occurs. `main` has no parameters and no return type (in contrary to the C-family), otherwise you get the build-error:

`func main must have no arguments and no return values` results.

When the program executes, after initializations the first function called (the entry-point of the application) will be main.main() (like in C). The program exits—immediately and successfully—when main.main returns.

The code in functions (the body) is enclosed between braces: { }

The first { must be on the same line as the func-declaration: this is imposed by the compiler and gofmt (build-error: `syntax error: unexpected semicolon or newline before {`).

(This is because the compiler then produces func main() ; which is an error.)

The last } is positioned after the function-code in the column beneath **f**unction; for small functions it is allowed that everything is written on one line, like for example: `func Sum(a, b int) int { return a + b` }

The same rule applies wherever { } are used (for example: if, etc.)

So schematically a general function looks like:

```
func functionName(parameter_list) (return_value_list) {
                …
 }

where parameter_list is of the form (param1 type1, param2 type2, …)
and return_value_list is of the form (ret1 type1, ret2 type2, …)
```

Function names only start with a capital letter when the function has to be used outside the package; then they follow PascalCasing, otherwise they follow camelCasing: every new wordin the name starts with a capital letter.

The line: `fmt.Println("hello, world")` calls the function Println from the package fmt, which prints the string-parameter to the console, followed by a newline-character \n .

The same result can be obtained with `fmt.Print("hello, world\n")`

These functions Print and Println can also be applied to variables, like in: `fmt.Println(arr)`; they use the default output-format for the variable arr.

Printing a string or a variable can be done even simpler with the predefined functions print and println: **`print`**`("ABC") or `**`println`**`("ABC") or (with a variable i): println(i)`

These are only to be used in the debugging phase; when deploying a program replace them with their fmt relatives.

The execution of a function is stopped when the closing } is reached or when a **`return`** statement is encountered, the execution of the program continues with the line following the call of the function.

The program exits normally with code 0 (`Program exited with code 0`); a program that terminates abnormally exits with another integer code like 1; this can be used to test succesfull execution of he program from a script.

4.2.3 Comments

Listing 4.3—hello_world2.go:

```
package main
import “fmt” // Package implementing formatted I/O.
func main() {
    fmt.Printf(“Καλημέρα κόσμε; or こんにちは 世界\n”)
}
```

This illustrates the international character by printing Κ α λ η μ┌ ρ α κ ┌ σ μ ε ; or こんにちは 世界, and also the characters used to indicate a *comment.*

Comments of course are not compiled, but they are used by godoc (see § 3.6)

A one-line comment starts with //, at the beginning or somewhere in a line; this is mostly used. A multi-line or block-comment starts with /* and ends with */, nesting is not allowed; this is used for making package documentation and commenting out code.

Every package should have a *package comment*, a block comment immediately preceding the package statement, introducing the package and provide information relevant to the package and its functionality as a whole. A package can be spread over many files, but the comment needs to be in only one of them. This comment is shown when a developer demands info of the package with `godoc`. Subsequent sentences and/or paragraphs can give more details. Sentences should be properly punctuated.

Example:

```
// Package superman implements methods for saving the world.
//
// Experience has shown that a small number of procedures can prove
// helpful when attempting to save the world.
package superman
```

Nearly every top-level type, const, var and func, and certainly every exported name in a program should have a comment. This comment (called a *doc comment)* appears on the preceding line, and for a function Abcd should start with: “Abcd …”.

Example:

```
// enterOrbit causes Superman to fly into low Earth orbit, a position
// that presents several possibilities for planet salvation.
func enterOrbit() error {
  ...
}
```

The godoc-tool (see §3.6) collects these comments to produce a technical documentation.

4.2.4 Types

Variables (like constants) contain data, and data can be of different *data types,* or *types* for short. A declaration of a variable with **`var`** automatically initializes it to the zero-value defined for its type.A type defines the set of values and the set of operations that can take place on those values.

Types can be *elementary (or primitive),* like int, float, bool, string,
or *structured (or composite),* like struct, array, slice, map, channel,
and *interfaces*, which only describe the behavior of a type.

A structured type which has no real value (yet) has the value **nil**, which is also the default value for these types (in Objective-C this is also called nil, in Java it is null, in C anc C++ it is NULL or 0). There is no type-hierarchy.

Functions can also be of a certain type, this is the type of the variable which is returned by the function. This type is written after the function name and its optional parameter-list, like:

```
func FunctionName (a typea, b typeb) typeFunc
```

The returned variable var of typeFunc appears somewhere in the function in the statement:

```
return var
```

A function can *return more than one variable*, then the return-types are indicated separated by comma's and surrounded by ( ), like: `func FunctionName (a typea, b typeb) (t1 type1, t2 type2)`

Example: the function Atoi (see § 4.7): `func Atoi(s string) (i int, err error)`

Then return takes the form: **`return var1, var2`**

This is often used when the success (true/false) of the execution of a function or the error-message is returned together with the return value (see multiple assignments below).

Use the keyword `type` for defining your own type. Then you probably want to define a struct-type (see Chapter 10), but it is also possible to define an *alias* for an existing type, like in:

```
type IZ int
```

and then we can declare variables like: `var a IZ = 5`

We say that a has int as *underlying type*, this makes conversion possible (see § 4.2.6).

If you have more than one type to define, you can use the *factored* keyword form, as in:

```
type (
       IZ int
       FZ float
       STR string
)
```

Every value must have a type after compilation (the compiler must be able to infer the types of all values):

Go is a *statically typed* language.

4.2.5 General structure of a Go-program

The following program compiles but does nothing useful, but is shows the preferred structure for a Go-program. This structure is not necessary, the compiler does not mind if main() or the variable declarations come last, but a uniform structure makes Go code better readable from top to bottom. All structures will be further explained in this and the coming chapters, but the general ideas are:

- After import: declare constants, variables and the types
- Then comes the init() function if there is any: this is a special function that every package can contain and that is executed first.
- Then comes the main() function (only in the package main)
- Then come the rest of the functions, the methods on the types first; or the functions in order as they are called from main() onwards; or the methods and functions alphabetically if the number of functions is high.

Listing 4.4—gotemplate.go:

```
package main

import (
    "fmt"
)

const c = "C"

var v int = 5

type T struct{}

func init() { // initialization of package
}

func main() {
        var a int
        Func1()
        // ...
        fmt.Println(a)
}

func (t T) Method1() {
        //...
}

func Func1() { // exported function Func1
        //...
}
```

The order of execution (program startup) of a Go application is as follows:

(1) all packages in package main are imported in the order as indicated,
in every package:
(2) if it imports packages, (1) is called for this package (recursively)
but a certain package is imported only once
(3) then for every package (in reverse order) all constants and variables are evaluated, and the init() if it contains this function.
(4) at last in package main the same happens, and then main() starts executing.

4.2.6 Conversions

If necessary and possible a value can be *converted (cast, coerced)* into a value of another type. Go never does *implicit* (automatic) conversion, it must be done *explicit* like so, with the syntax like a function call (a type is here used as a kind of function):

```
valueOfTypeB = typeB(valueOfTypeA)
```

Examples:
```
a := 5.0
b := int(a)
```

But this can only succeed in certain well defined cases, for example from a narrower type to a broader type (for example: int16 to int32). When converting from a broader type to a narrower type (for example: int32 to int16, or float32 to int) loss of value (*truncation)* can occur. When the conversion is impossible and the compiler detects this, a compile-error is given, otherwise a runtime-error occurs.

Variables with the same underlying type can be converted into one another:

```
var a IZ = 5
c := int(a)
d := IZ(c)
```

4.2.7 About naming things in Go

Clean, readable code and simplicity are a major goal for Go development. gofmt imposes the code-style. Names of things in Go should be short, concise, evocative. Long names with mixed caps and underscores which are often seen e.g. in Java or Python code often hinders readability. Names should not contain an indication of the package: the qualification with the package name is sufficient. A method or function which returns an object is named as a noun, no Get… is needed. To change an object, use SetName. If necessary, Go uses MixedCaps or mixedCaps rather than underscores to write multiword names.

4.3. Constants

A constant **`const`** contains data which does not change.

This data can only be of type boolean, number (integer, float or complex) or string.

It is defined as follows: `const identifier [type] = value`, for example: `const Pi = 3.14159`

The type specifier [type] is optional, the compiler can implicitly derive the type from the value.

Explicit typing example: `const b string = “abc”`

Implicit typing example: `const b = “abc”`

A value derived from an untyped constant becomes typed when it is used within a context that requires a typed value (otherwise formulated: an untyped constant takes the type needed by its context):

```
var n int
f(n + 5) // untyped numeric constant “5” becomes typed as int
```

Constants must be evaluated at compile time; a const can be defined as a calculation, but all the values necessary for the calculation must be available at compile time.

So this is ok: `const c1 = 2/3`

this is NOT: `const c2 = getNumber()// gives the build error: getNumber() used as value`

Numeric constants have no size or sign, can be of *arbitrary high precision* and do no overflow:

```
const Ln2= 0.693147180559945309417232121458\
             176568075500134360255254120680009
const Log2E= 1/Ln2                          // this is a precise reciprocal
const Billion = 1e9                         // float constant
const hardEight = (1 << 100) >> 97
```

As demonstrated \ can be used as a continuation character in a constant.

In contrast to numeric variables of different types, with constants you don’t have to worry about conversions: they are like ideal numbers.

Constants can overflow only when they are assigned to a numeric variable with too little precision to represent the value, this results in a compile error. Multiple assignment is allowed, like in:

```
const beef, two, c = “meat”, 2, “veg”
const Monday, Tuesday, Wednesday, Thursday, Friday, Saturday = 1, 2, 3, 4, 5, 6

const (
       Monday, Tuesday, Wednesday = 1, 2, 3
       Thursday, Friday, Saturday = 4, 5, 6
)
```

Constants can be used for *enumerations*:

```
const (
   Unknown = 0
   Female  = 1
   Male    = 2
)
```

Unknown, `Female, Male` are now aliases for 0, 1 and 2. They can in effect be used to test for these values, like in a switch / case construct (§ 5.3).

In such cases, the value **`iota`** can be used to enumerate the values:

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

The first use of iota gives 0, whenever iota is used again on a new line, its value is incremented by 1; so a=0, b=1, c=2. This can be shortened to:

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

iota can also be used in an expression, like iota + 50. A new const block or declaration initializes iota back to 0.

Of course, the value of a constant cannot change during the execution of the program; doing so is prevented by a compiler error: cannot assign to value, where value is the value of the constant.

An example from the time package: the names for the days of the week:

```
const (
    Sunday = iota
    Monday
    Tuesday
```

```
        Wednesday
        Thursday
        Friday
        Saturday
    )
```

You can give the enumeration a type name like in this example:

```
    type Color int
    const (
            RED Color = iota  // 0
            ORANGE           // 1
            YELLOW           // 2
            GREEN            // ..
            BLUE
            INDIGO
            VIOLET           // 6
    )
```

Remark: There is a convention to name constant identifiers with all uppercase letters, like: const INCHTOwCM = 2.54; this improves readability and can be used as long as it is not in conflict with the Visibility Rule of §4.2

4.4. Variables

4.4.1 Introduction

The general form for declaring a variable uses the keyword **`var`**`: var identifier type`

Important to note is that the type is written after the identifier of the variable, contrary to almost any other programming language. Why did the Go designers chose for this convention?

First it removes some ambiguity which can exist in C declarations, e.g. in writing `int* a, b`;

Only a is a pointer and b is not. In order to declare them both pointers, the asterisk must be repeated. (for a longer discussion on this topic, see: http://blog.golang.org/2010/07/gos-declaration-syntax.html)

However in Go, they can both be declared pointers as follows: `var a, b *int`

Secondly it reads well from left to right and so is easier to understand.

Some examples:

```
var a int
var b bool
var str string
```

which also can be written as:

```
var (
        a int
        b bool
        str string
)
```

This form is mainly used to declare variables globally.

When a variable is declared it contains automatically the default zero or null value for its type: 0 for int, 0.0 for float, false for bool, empty string (“”) for string, nil for pointer, zero-ed struct, etc.: *all memory in Go is initialized*.

The naming of identifiers for variables follows the camelCasing rules (start with a small letter, every new part of the word starts with a capital letter), like: `numShips, startDate`

But if the variable has to be exported, it must start with a capital letter (visibility rule §4.2).

A variable (constant, type, function) is only known in a certain range of the program, called the *scope*. Variables etc. declared outside of any function (in other words at the top level) have *global (or package) scope*: they are visible and available in all source files of the package.

Variables declared in a function have *local scope*: they are only known in that function, the same goes for parameters and return-variables. In chapter 5 we will encounter control constructs like if and for; a variable defined inside such a construct is only known within that construct (*construct scope)*. Mostly you can think of a scope as the codeblock (surrounded by { }) in which the variable is declared.

Although identifiers have to be unique, an identifier declared in a block may be redeclared in an inner block: in this block (but only there) the redeclared variable takes priority and *shadows* the outer variable with the same name; if used care must be taken to avoid subtle errors (see § 16.1).

Variables can get their value (which is called *assigning* and uses the assignment-operator =) at compile time, but of course a value can also be computed or changed during runtime.

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

In general a variable b can only be assigned to a variable a as in a = b, when a and b are of the same type.

Declaration and assignment(initialization) can of course be combined, in the general format:

```
var identifier [type] = value
```
Examples:
```
var a int  = 15
var i = 5
var b bool = false
var str string = "Go says hello to the world!"
```

But the Go-compiler is intelligent enough to derive the type of a variable from its value *(dynamically,* also called *automatic type inference,* somewhat like in the scripting languages Python and Ruby, but there it happens in run time*)*, so the following forms (omitting the type) are also correct:

```
var a   = 15
        var b   = false
        var str = "Go says hello to the world!"
```

or:
```
var (
    a   = 15
    b   = false
    str = "Go says hello to the world!"
    numShips = 50
    city string
)
```

It can still be useful to include the type information in the case where you want the variable to be typed something different than what would be inferred, such as in: `var n int64 = 2`

However an expression like `var a` is not correct, because the compiler has no clue about the type of a. Variables could also be expressions computed at runtime, like:

```
var (
    HOME = os.Getenv("HOME")
```

```
        USER = os.Getenv("USER")
        GOROOT = os.Getenv("GOROOT")
    )
```

The var syntax is mainly used at a global, package level, in functions it is replaced by the *short declaration syntax* := (see § 4.4).

Here is an example of a program which shows the operating system on which it runs. It has a local string variable getting its value by calling the Getenv function (which is used to obtain environment-variables) from the os-package.

Listing 4.5—goos.go:

```
package main
import (
    "fmt"
    "os"
)

func main() {
    var goos string = os.Getenv("GOOS")
    fmt.Printf("The operating system is: %s\n", goos)
    path := os.Getenv("PATH")
    fmt.Printf("Path is %s\n", path)
}
```

The output can for example be: The `operating system` is: windows, or The `operating system` is: linux, followed by the contents of the path variable.

Here `Printf` is used to format the ouput (see § 4.4.3).

4.4.2 Value types and reference types

Memory in a computer is used in programs as a enormous number of boxes (that's how we will draw them), called *words.* All words have the same length of 32 bits (4 bytes) or 64 bits (8 bytes), according to the processor and the operating system; all words are identified by their *memory address* (represented as a hexadecimal number).

All variables of elementary (primitive) types like int, float, bool, string, … are *value types*, they point directly to their value contained in memory:

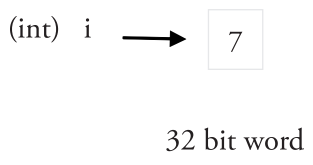


Fig 4.1: Value type

Also composite types like arrays (see chapter 7) and structs (see Chapter 10) are value types.

When assigning with = the value of a value type to another variable: `j = i`, a copy of the original value i is made in memory.

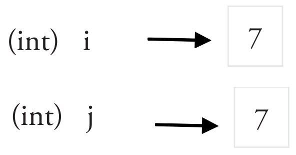


Fig 4.2: Assignment of value types

The memory address of the word where variable i is stored is given by &i (see § 4.9), e.g. this could be 0xf840000040. Variables of value type are cointained in *stack* memory.

The actual value of the address will differ from machine to machine and even on different executions of the same program as each machine could have a different memory layout and and also the location where it is allocated could be different.

More complex data which usually needs several words are treated as reference types.

A *reference type* variable r1 contains the address (a number) of the memory location where the value of r1 is stored (or at least the 1st word of it):

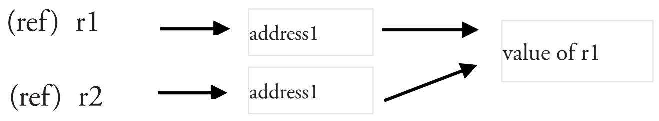


Fig 4.3: Reference types and assignment

This address which is called a *pointer* (as is clear from the drawing, see § 4.9 for more details) is also contained in a word.

The different words a reference type points to could be sequential memory addresses (the memory layout is said to be *contiguously*) which is the most efficient storage for computation; or the words could be spread around, each pointing to the next.

When assigning r2 = r1, only the reference (the address) is copied.

If the value of r1 is modified, all references of that value (like r1 and r2) then point to the modified content.

In Go pointers (see § 4.9) are reference types, as well as slices (ch 7), maps (ch 8) and channels (ch 13). The variables that are referenced are stored in the *heap*, which is garbage collected and which is a much larger memory space than the stack.

4.4.3 Printing

The function `Printf` is visible outside the fmt-package because it starts with a P, and is used to print output to the console. It generally uses a format-string as its first argument:

```
    func Printf(format string, list of variables to be printed)
```

In Listing 4.5 the format string was: `“The operating system is: %s\n”`

This format-string can contain one or more format-specifiers %.., where .. denotes the type of the value to be inserted, *e.g.* **%s** stands for a string-value. **%v** is the general default format specifier. The value(s) come in the same order from the variables summed up after the comma, and they are separated by comma's if there is more than 1. These % placeholders provide for very fine control over the formatting.

The function `fmt.Sprintf` behaves in exactly the same way as Printf, but simply returns the formatted string: so this is the way to make strings containing variable values in your programs (for an example, see `Listing 15.4—simple_tcp_server.go`).

The functions `fmt.Print` and `fmt.Println` perform fully automatic formatting of their arguments using the format-specifier %v, adding spaces between arguments and the latter a newline at the end. So `fmt.Print(“Hello:”, 23)` produces as `output: Hello: 23`

4.4.4 Short form with the := assignment operator

With the type omitted, the keyword var in the last statements of § 4.4.1 is pretty superfluous, so we may write in Go: `a := 50` or `b := false`

Again the types of a and b (int and bool) are inferred by the compiler.

This is the preferred form, but it *can only be used inside functions, not in package scope*. The := operator effectively makes a new variable; it is also called an *initializing declaration*.

Remark: If after the lines above in the same codeblock we declare a:= 20, this is not allowed : the compiler gives the error `“no new variables on left side of :=”` ; however a = 20 is ok because then the same variable only gets a new value.

A variable a which is used, but not declared, gives a compiler error: `undefined: a`

Declaring a *local* variable, but not using it, is a compiler error; like variable a in the following main() function:

```
func main() {
        var a string = “abc”
        fmt.Println(“hello, world”)
}
```

which gives the error: a declared and not used

Also setting the value of a is not enough, the value must be read in order to count as a use, so `fmt.Println(“hello, world”, a)` removes the error.

However for global variables this is allowed.

Other convenient shortening forms are:

Multiple declarations of variables of the same type on a single line, like: `var a, b, c int`

(this is an important reason why the type is written after the identifier(s))

Multiple assignments of variables on a single line, like: `a, b, c = 5, 7, “abc”`

This assumes that variables a, b and c where already declared, if not: a, b, c := 5, 7, “abc”

The values from the right-hand side are assigned to the variables on the left-hand side in the same order, so a has the value 5, b has the value 7, c has the value "abc".

This is called *parallel* or *simultaneous* assignment.

With two variables it can be used to perform a *swap* of the values: `a, b = b, a`

(This removes the need for making a swap function in Go)

The blank identifier _ can also be used to throw away values, like the value 5 in: `_, b = 5, 7`

_ is in effect a write-only variable, you cannot ask for its value. It exists because a declared variable in Go must also be used, and sometimes you don't need to use all return values from a function.

The multiple assignment is also used when a function returns more than 1 value, like here where val and an error err are returned from Func1: `val, err = Func1(var1)`

4.4.5 Init-functions

Apart from global declaration with initialization, variables can also be initialized in an init()-function. This is a special function with the name init() which cannot be called, but is executed automatically before the main() function in package main or at the start of the import of the package that contains it.

Every source file can contain only 1 init()-function. Initialization is always single-threaded and package dependency guarantees correct execution order.

A possible use is to verify or repair correctness of the program state before real execution begins.

Example: Listing 4.6—init.go:

```
package trans
import "math"

var Pi float64

func init() {
        Pi = 4 * math.Atan(1) // init() function computes Pi
}
```

In its init() the variable Pi is initialized by calculation.

The program in Listing 4.7 use_init.go imports the package trans (which is in the same directory) and uses Pi:

```
package main
import (
        "fmt"
        "./trans"
)

var twoPi = 2 * trans.Pi

func main() {
            fmt.Printf("2*Pi = %g\n", twoPi) // 2*Pi = 6.283185307179586
}
```

An init() function is also frequently used when (for example for a server application) a backend() goroutine is required from the start of the application, like in:

```
func init() {
   // setup preparations
   go backend()
}
```

EXERCISES: Deduce the output of the following programs and explain your answer, then compile and execute them.

Exercise 4.1: `local_scope.go:`

```
package main
var a = "G"

func main() {
       n()
       m()
       n()
}
```

```
func n() { print(a) }
func m() {
       a := "O"
       print(a)
}
```

Exercise 4.2: `global_scope.go:`

```
package main
var a = "G"
func main() {
       n()
       m()
       n()
}

func n() {
       print(a)
}

func m() {
       a = "O"
       print(a)
}
```

Exercise 4.3: `function_calls_function.go`

```
package main
var a string

func main() {
       a = "G"
       print(a)
       f1()
}
```

```
func f1() {
	a := "O"
	print(a)
	f2()
}

func f2() {
	print(a)
}
```

4.5. Elementary types and operators

In this paragraph, we discuss the boolean, numerical and character data types.

Values are combined together with *operators* into *expressions*, which are also values of a certain type. Every type has its own defined set of operators, which can work with values of that type. If an operator is used for a type for which it is not defined, a compiler error results.

A unary operator works on one value (postfix), a binary operator works on two values or operands (infix).

The two values for a binary operator must be of the same type. Go does not implicitly convert the type of a value, if necessary this must be done by an explicit conversion (see § 4.2): Go is *strongly typed.*There is *no operator overloading* as in C and Java. An expression is by default evaluated from left to right.

There is a built-in *precedence* amongst the operators (see § 4.5.3) telling us which operator in an expression has the highest priority, and so gets executed first. But the use of *parentheses* **( )** around expression(s) can alter this order: an expression within ( ) is always executed first.

4.5.1. Boolean type bool

An example: `var b bool = true`

The possible values of this type are the predefined constants **`true`** and **`false.`**

Two values of a certain type can be compared with each other with the *relational operators* == and != producing a boolean value:

Equality operator: `==`

This gives true if the values on both sides are the same (values), false otherwise. This supposes that they are of the same type.

Example: `var aVar = 10`

```
aVar == 5       ← false
aVar == 10      ← true
```

Not-equal operator: **!=**

This gives true if the values on both sides are different (values), false otherwise.

Example: `var aVar = 10`

```
aVar != 5       ← true
aVar != 10      ← false
```

Go is very strict about the values that can be compared: they have to be of the same type, or if they are interfaces (see Chapter 11) they must implement the same interface type. If one of them is a constant, it must be of a type compatible with the other. If these conditions are not satisfied, one of the values has first to be converted to the other's type.

Boolean constants and variables can also be combined with *logical* operators (not, and, or) to produce a boolean value. Such a logical statement is not a complete Go-statement on itself.

The resultant boolean value can then be tested against in conditional structures (see chapter 5). And, or and equals are binary operators; not is a unary operator. We will use T representing a true statement, and F for a false statement.

The following are the *logical operators:*

NOT operator: **!** !T ← false

!F ← true

It turns the boolean value into its opposite.

AND operator: **&&**

T && T ← true

T && F	← false
F && T	← false
F && F	← false

It only gives true if both operands are true.

OR operator: ||

T || T	← true
T || F	← true
F || T	← true
F || F	← false

It is true if any one of the operands is true, it only gives false if both operands are false.

The && and || operators behave in a *shortcut* way: when the value of the left side is known and it is sufficient to deduce the value of the whole expression (false with && and true with ||), then the right side is not computed anymore. For that reason: if one of the expressions involves a longlasting calculation, put this expression at the right side.

Like in all expressions, ( ) can be used to combine values and influence the result.

In format-strings %t is used as a format specifier for booleans.

Boolean values are most often used (as values or combined with their operators) for testing the conditions of if-, for- and switch-statements (see chapter 5).

A useful naming convention for important boolean values and functions is to let the name begin with *is* or *Is*, like isSorted, isFound, isFinished, isVisible, so code in if-statements reads as a normal sentence, e.g.: `unicode.IsDigit(ch)` (see § 4.5.5).

4.5.2. Numerical types

4.5.2.1 ints and floats

There are types for integers, floating point numbers and there is also native support for *complex numbers.* The bit representation is two's complement (for more info see http://en.wikipedia.org/wiki/Two's_complement).

Go has *architecture dependent types* such as `int, uint, uintptr`.

They have the appropriate length for the machine on which the program runs:

an `int` is the default signed type: it takes 32 bit (4 bytes) on a 32 bit machine and 64 bit(8 bytes) on a 64 bit machine; the same goes for the unsigned uint.

`uintptr` is an unsigned integer large enough to store a pointer value.

A `float` type does not exist.

The *architecture independent types* have a *fixed size* (in bits) indicated by their names:

For integers:	`int8 (-128 -> 127)`
	`int16 (-32768 -> 32767)`
	`int32 (– 2,147,483,648 -> 2,147,483,647)`
	`int64 (– 9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807)`
For unsigned integers:	`uint8 (with alias byte, 0 -> 255)`
	`uint16 (0 -> 65,535)`
	`uint32 (0 -> 4,294,967,295)`
	`uint64 (0 -> 18,446,744,073,709,551,615)`
For floats:	`float32` (+- 10^{-45} -> +- 3.4 * 10^{38})
`(IEEE-754)`	`float64` (+- 5 * 10^{-324} -> 1.7 * 10^{308})

int is the integer type which offers the fastest processing speeds.

The initial (default) value for integers is 0, and for floats this is 0.0

A float32 is reliably accurate to about 7 decimal places, a float64 to about 15 decimal places. Due to the fact that perfect accuracy is not possible for floats comparing them with == or != must be done very carefully; if needed a test on the difference being smaller than a very small number (the accuracy limit) must be made.

Use float64 whenever possible, because all the functions of the math package expect that type.

Numbers may be denoted in octal notation with a prefix of 0 (like 077), hexadecimal with a prefix of 0x (like 0xFF) or scientific notation with e, which represents the power of 10 (e.g.: 1e3 = 1000 or 6.022e23 = 6.022 x 10^{23}).

Note that you can make a number like this: `a := uint64(0)` which is in fact a conversion to type uint64.

Because Go is strongly typed, *mixing of types is not allowed*, as in the following program. But constants are considered typeless in this respect, so with constants mixing is allowed.

Listing 4.8—type_mixing.go (does not compile!):

```
package main
func main() {
    var a int
    var b int32
    a = 15
    b = a + a // compiler error
    b = b + 5 // ok: 5 is a constant
}
```

The compiler error is: `cannot use a + a (type int) as type int32 in assignment`

Likewise an int16 cannot be assigned to an int32, there is no implicit coercion.

In the following program `casting.go`, an explicit conversion is done to avoid this (see §4.2)

Listing 4.9—casting.go:

```
package main
import “fmt”

func main() {
        var n int16 = 34
        var m int32

// compiler error: cannot use n (type int16) as type int32 in assignment
        //m = n
        m = int32(n)
```

```


        fmt.Printf(“32 bit int is:  %d\n”, m)
        fmt.Printf(“16 bit int is:  %d\n”, n)
}
// the output is:
32 bit int is:  34
16 bit int is:  34
```

Format specifiers:

In format-strings %d is used as a format specifier for integers (**%x** or %X can be used for a hexadecimal representation), %g is used for float types (**%f** gives a floating point, **%e** gives a scientific notation), **%0nd** shows an integer with n digits, and leading 0 is necessary.

%n.mg represents the number with m digits after the decimal sign, and n before it, instead of g also e and f can be used, for example: the %5.2e formatting of the value 3.4 gives 3.40e+00

Conversions of numerical values:

In a conversion like `a32bitInt = int32(a32Float)` truncation of the decimal part occurs. In general information is lost when converting to a smaller type, therefore in order to avoid loss of accuracy always convert to the bigger numerical type. Or you could write suitable functions to perform safe downsizing conversions, like the following for converting an int to a uint8:

```
func Uint8FromInt(n int) (uint8, error) {
    if 0 <= n && n <= math.MaxUint8 {    // conversion is safe
        return uint8(n), nil
    }
    return 0, fmt.Errorf("%d is out of the uint8 range", n)
}
```

Or for safe converting of a float64 to an int:

```
func IntFromFloat64(x float64) int {
    if math.MinInt32 <= x && x <= math.MaxInt32 { // x lies in the integer range
        whole, fraction := math.Modf(x)
         if fraction >= 0.5 {
                whole++
        }
        return int(whole)
    }
    panic(fmt.Sprintf("%g is out of the int32 range", x))
}
```

In the case that x does not sit in the integer range, the program stops with a panic message (see § 13.2).

Question 4.1: Are int and int64 the same type ?

4.5.2.2 Complex numbers

For these data we have the following types:

```
complex64  (with a 32 bit real and imaginary part)
complex128 (with a 64 bit real and imaginary part)
```

A complex number is written in the form: re + im*i*, where re is the real part, and im is the imaginary part, and i is the √ -1 .

Example:

```
var c1 complex64 = 5 + 10i
fmt.Printf("The value is: %v", c1)
// this will print:  5 + 10i
```

If re and im are of type float32, a variable c of type complex64 can be made with the function complex: `c = complex(re, im)`

The functions `real(c)` and `imag(c)` give the real and imaginary part respectively.

In format-strings %v the default format specifier can be used for complex numbers; otherwise use %f for both constituent parts.

Complex numbers support all the normal arithmetic operations as other numbers. You can only compare them with == and !=, but again be aware of precision. The package cmath contains common functions for operating on complex numbers. When memory constraints are not too tight, use type complex128 because all cmath functions use it.

4.5.2.3 Bit operators

They work only on *integer* variables having bit-patterns of equal length:

%**b** is the format-string for bit-representations.

Binary: Bitwise and: **&**

bits in the same position are and-ed together, see AND-operator in §4.5.1, replacing T (true) by 1 and F (false) by 0:

1 **&** 1 ← 1
1 **&** 0 ← 0
0 **&** 1 ← 0
0 **&** 0 ← 0

Bitwise or: **|**

bits in the same position are or-ed together, see OR-operator in §4.5.1, replacing T (true) by 1 and F (false) by 0

```
1 | 1 ← 1
1 | 0 ← 1
0 | 1 ← 1
0 | 0 ← 0
```

Bitwise xor: ^

bits in the same position are taken together according to the rule:

```
1 ^ 1 ← 0
1 ^ 0 ← 1
0 ^ 1 ← 1
0 ^ 0 ← 0
```

Bit clear: &^ forces a specified bit to 0. (equivalent to *and not*)

Unary: Bitwise complement: ^

is defined with the xor operator: it is m ^ x with m = "all bits set to 1" for unsigned x and m = -1 for signed x

e.g.: ^2 = ^10 = -01 ^ 10 = -11

BitShift

Left Shift: `<<` , for example: bitP << n

the bits of bitP shift n positions to the left, the empty positions on the right are filled with 0's; if n is 2, the number is multiplied by 2, left shift by n effects to a multiplication by 2^n

```
So 1 << 10  // equals 1 KB   (kilobyte)
   1 << 20  // equals 1 MB   (megabyte)
   1 << 30  // equals 1 GB   (gigabyte)
```

Right Shift: `>>` , for example: bitP >> n

the bits of bitP shift n positions to the right, the empty positions on the left are filled with 0's; if n is 2, the number is divided by 2, right shift by n effects to a division by 2^n

When the result is assigned to the first operand, they can also be abbreviated like a <<= 2 or b `^= a & 0xffffffff`

<u>Commonly used constants in memory resource usage:</u>

Applying the << operator and the use of iota in constants, the following type definition neatly defines memory constants:

```
type ByteSize float64
const (
    _ = iota  // ignore first value by assigning to blank identifier
    KB ByteSize = 1<<(10*iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

<u>A type BitFlag for working with bits in communication:</u>

```
type BitFlag int
const (
    Active  BitFlag = 1 << iota     // 1 << 0 == 1
    Send                           // 1 << 1 == 2
    Receive                        // 1 << 2 == 4
)

flag := Active | Send            // == 3
```

4.5.2.4 Logical operators

Here we have the usual ==, != (see § 4.5.1) and <, <=, > and >=

They are called logical because the result value is of type bool: `b3:= 10 > 5  // b3 is true`

4.5.2.5 Arithmetic operators

The common binary operators +, -, * and / exist for both integers and floats.

(In contrast to the general rule, this could be called a form of operator overloading; moreover the + operator also exists for strings; but outside this Go does not allow operator overloading.)

/ for integers is integer division, for example: 9 / 4 -> 2.

The modulus operator % is only defined for integers: 9 % 4 -> 1

Integer division by 0 causes the program to crash, a *run-time panic* occurs (if it is obvious then the compiler can detect it); see Chapter 13 for how to test for this properly.

Division by 0.0 with floating point numbers gives an infinite result: `+Inf`

Exercise 4.4: Try this out: divby`0.go`

There are shortcuts for these operations: b = b + a can be shortened to b += a, and the same goes for -=, *=, /= and %=.

As unary operators for integers and floats we have ++ (increment) and -- (decrement), but only after the number (postfix):

```
i++ is short for i += 1 is short for i = i + 1
i-- is short for i -= 1 is short for i = i – 1
```

Moreover ++ and - - may only be used as statements, not expressions; so n = i++ is invalid, and subtler expressions like f(i++) or a[i]=b[i++], which are accepted in C, C++ and Java, cannot be used in Go.

No error is generated when an *overflow* occurs during an operation: high bits are simply discarded. Constants can be of help here, and if you need integers or rational numbers of unbounded size (that is only limited by the available memory) you can use the big package from the standard library, which provides the types big.Int and big.Rat (see § 9.4).

4.5.2.6 Random numbers

Some programs like games or statistical applications need random numbers. The package rand implements pseudo-random number generators.

For a simple example see Listing 4.10—random.go, which prints 10 random non-negative integers.

```
package main
import (
        "fmt"
        "rand"
        "time"
)

func main() {
        for i := 0; i < 10; i++ {
                      a := rand.Int()
                      fmt.Printf("%d / ", a)
        }
for i := 0; i < 5; i++ {
                r := rand.Intn(8)
                fmt.Printf("%d / ", r)
        }
        fmt.Println()
        timens := int64(time.Now().Nanosecond())
        rand.Seed(timens)
        for i := 0; i < 10; i++ {
                fmt.Printf("%2.2f / ", 100*rand.Float32())
        }
}
```

```
Output, for example: 816681689 / 1325201247 / 623951027 / 478285186 / 1654146165 /
1951252986 / 2029250107 / 762911244 / 1372544545 / 591415086 / / 3 / 0 / 6 / 4 / 2 /22.10
/ 65.77 / 65.89 / 16.85 / 75.56 / 46.90 / 55.24 / 55.95 / 25.58 / 70.61 /
```

The functions `rand.Float32` and `rand.Float64` return a pseudo-random number of that type in [0.0, 1.0),) here means the upper bound not included. The function `rand.Intn` takes an int n and returns a non-negative pseudo-random number in [0,n).

You can use the Seed(value)-function to provide a starting value for the pseudo-random generation. Often the current offset time in nanoseconds is used for that purpose (see § 4.8).

4.5.3. Operators and precedence

Some operators have higher priority (precedence) than others; binary operators of the same precedence associate from left to right. The following table lists all operators and their precedence, top to bottom (7 -> 1) is highest to lowest:

Precedence	Operator(s)
7	`^  !`
6	`*  /  %  <<  >>  &  &^`
5	`+  -  |  ^`
4	`==  !=  <  <=  >=  >`
3	`<-`
2	`&&`
1	`||`

It is of course allowed to clarify expressions by using ( ) to indicate priority in operations: expressions contained in ( ) are always computed first.

4.5.4. Aliasing types

When working with types, a type can also be given another name, so that this new name can be used in the code (for shortening names, or avoiding a name-clash).

In `type TZ int` TZ is declared as a new name for the int type (perhaps it represents time zones in a program), and can then be used to declare int-variables, like in the following program:

Listing 4.11—type.go :

```go
package main
import "fmt"

type TZ int

func main() {
    var a, b TZ = 3, 4
    c := a + b
    fmt.Printf("c has the value: %d", c) // prints: c has the value: 7
}
```

In fact this alias is a brand new type, which can have methods that the original type does not have (see Chapter 10); TZ can have a method to output the time zone-info in a clear or pretty way.

Exercise 4.5: Define an alias type Rope for string and declare a variable with it.

4.5.5. Character type

Strictly speaking this is not a type in Go: characters are a special case of integers. The byte type is an alias for uint8, and this is ok for the traditional ASCII-encoding for characters (1 byte): `var ch byte = 'A'` ; a character is surrounded by single quotes ' '.

In the ASCII-table the decimal value for A is 65, and the hexadecimal value is 41, so the following are also declarations for the character A:

```
    var ch byte = 65 or var ch byte = '\x41'
```

(\x is always followed by exactly 2 hexadecimal digits).

Another possible notation is a \ followed by exactly 3 octal digits, e.g. '\377'.

But there is also support for Unicode (UTF-8): *characters* are also called *Unicode code points* or *runes*, and a Unicode character is represented by an int in memory. In documentation they are commonly represented as U+hhhh, where h us a hexadecimal digit. In fact the type *rune* exists in Go and is an alias for type int32.

To write a Unicode-character in code preface the hexadecimal value with \u or \U.

Because they need at least 2 bytes we have to use the int16 or int type. If 4 bytes are needed for the character \U is used; \u is always followed by exactly 4 hexadecimal digits and \U by 8 .

The following code (see `Listing 4.12 char.go`)

```
    var ch int = '\u0041'
    var ch2 int = '\u03B2'
    var ch3 int = '\U00101234'
    fmt.Printf("%d - %d - %d\n", ch, ch2, ch3)   // integer
    fmt.Printf("%c - %c - %c\n", ch, ch2, ch3)   // character
    fmt.Printf("%X - %X - %X\n", ch, ch2, ch3)   // UTF-8 bytes
    fmt.Printf("%U - %U - %U", ch, ch2, ch3)     // UTF-8 code point
```

prints out:

```
65 - 946 - 1053236
A - β - ┌
41 - 3B2 - 101234
U+0041 - U+03B2 - U+101234
```

In format-strings %c is used as a format specifier for characters: the character is shown, format-specifiers %v or %d show the integer representing the character; %U outputs the U+hhhh notation (for another example: see § 5.4.4).

The package unicode has some useful functions for testing characters, like (ch is a character):

testing for a letter:	`unicode.IsLetter(ch)`
testing for a digit:	`unicode.IsDigit(ch)`
testing for a whitespace character:	`unicode.IsSpace(ch)`

They return a bool value. The `utf8` package further contains functions to work with runes.

4.6. Strings

Strings are a sequence of UTF-8 characters (the 1 byte ASCII-code is used when possible, a 2-4 byte UTF-8 code when necessary). UTF-8 is the most widely used encoding, the standard encoding for text files, XML files and JSON strings. While able to represent characters that need 4 bytes, ASCII-characters are still stored using only 1 byte. A Go string is thus a sequence of *variable-width* characters (each 1 to 4 bytes, see Ex. 4.6), contrary to strings in other languages as C++, Java or Python that are fixed-width (Java uses always 2 bytes). The advantages are that Go strings and text files occupy less memory/disk space, and since UTF-8 is the standard, Go doesn't need to encode and decode strings as other languages have to do.

Strings are *value* types and *immutable*: once created you cannot modify the contents of the string; formulated in another way: strings are immutable arrays of bytes.

2 kinds of string literals exist:

Interpreted strings: surrounded by ""(double quotes),
escape sequences are interpreted:
for example: \n represents a newline
\r represents a carriage return
\t represents a tab
\u or \U Unicode characters
the *escape character* \ can also be used to remove the special meaning of the following character, so \" simply prints a ", and \' is ', \\ prints a \

Raw strings: surrounded by ` ` (back quotes: AltGr + £), they are not interpreted; they can span multiple lines.
in `This is a raw string \n` \n is not interpreted but taken literally.

Strings are length-delimited and do not terminate by a special character as in C/C++

The initial (default) value of a string is the *empty string* “” .

The usual comparison operators (`== != < <= >= >`) work on strings by comparing byte by byte in memory. The length of a string str (the number of bytes) is given by the len() function: `len(str)`

The contents of a string (the ‘raw’ bytes) is accessible via *standard indexing methods*, the index between [ ], with the index starting from 0:

the first byte of a string str is given by:	`str[0]`
the i-th byte by:	`str[i]`
the last byte by:	`str[len(str)-1]`

However these translate only to real characters if only ASCII characters are used!

Note: Taking the address of a character in a string, like &str[i], is illegal.

Adding (concatenating) strings: `+`

Two strings s1 and s2 can be made into one string s with: `s := s1 + s2`

s2 is appended to s1 to form a new string s.

Multi-line strings can be constructed as follows:

```
str := “Beginning of the string “+
       “second part of the string”
```

The + has to be on the first line, due to the insertion of ; by the compiler.

The append shorthand += can also be used for strings:

```
s := “hel” + “lo,”
s += “world!”
fmt.Println(s) // prints out “hello, world!”
```

Concatenating strings in a loop using + is not the most efficient way, a better approach is to use `strings.Join()` (see § 4.7.10), even better is to use writing in a byte-buffer (§ 7.2.6).

In chapter 7 we will see that strings can be considered as slices of bytes (or ints), and that the slice-indexing operations thus also apply for strings. The for-loop from § 5.4.1 loops over the index and so only returns the raw bytes, to loop over the Unicode characters in the string we must use the for-range loop from § 5.4.4 (see also the example in § 7.6.1). In the next § we learn a number of useful methods for working with strings. And then there is also the function fmt.Sprint(x) from the fmt package, to produce a string out of data in the format you want (see § 4.4.3).

Exercise 4.6: `count_characters.go`

Create a program that counts the number of bytes and characters (runes) for this string:
"asSASA ddd dsjkdsjs dk"
Then do the same for this string: "asSASA ddd dsjkdsjsこん dk"
Explain the difference. (hint: use the unicode/utf8 package.)

4.7. The strings and strconv package

Strings are a basic data structure, and every language has a number of predefined functions for manipulating strings. In Go these are gathered in the package *strings.*

Some very useful functions are:

4.7.1—Prefixes and suffixes:

HasPrefix tests whether the string s begins with prefix:
`strings.HasPrefix(s, prefix string) bool`
HasSuffix tests whether the string s end with suffix:
`strings.HasSuffix(s, suffix string) bool`

Listing 4.13—presuffix.go:

```
package main

import (
"fmt"
"strings"
)

func main() {
    var str string = "This is an example of a string"
    fmt.Printf("T/F? Does the string \"%s\" have prefix %s? ", str, "Th")
    fmt.Printf("%t\n", strings.HasPrefix(str, "Th"))
}
```

Output: T/F? Does the string "This is an example of a string" have prefix Th? True

This illustrates also the use of the escape character \ to output a literal "with \", and the use of 2 substitutions in a format-string.

4.7.2—Testing whether a string contains a substring:

Contains returns true if substr is within s: `strings.Contains(s, substr string) bool`

4.7.3—Indicating at which position (index) a substring or character occurs in a string:

Index returns the index of the first instance of str in s, or -1 if str is not present in s:

```
strings.Index(s, str string) int
```

LastIndex returns the index of the last instance of str in s, or -1 if str is not present in s:

```
strings.LastIndex(s, str string) int
```

If ch is a non-ASCII character use `strings.IndexRune(s string, ch int) int.`

An example: Listing 4.14—index_in_string.go:

```
package main
import (
        "fmt"
        "strings"
)
func main() {
        var str string = "Hi, I'm Marc, Hi."
        fmt.Printf("The position of \"Marc\" is: ")
        fmt.Printf("%d\n", strings.Index(str, "Marc"))
        fmt.Printf("The position of the first instance of \"Hi\" is: ")
        fmt.Printf("%d\n", strings.Index(str, "Hi"))
        fmt.Printf("The position of the last instance of \"Hi\" is: ")
        fmt.Printf("%d\n", strings.LastIndex(str, "Hi"))
        fmt.Printf("The position of \"Burger\" is: ")
        fmt.Printf("%d\n", strings.Index(str, "Burger"))
}
```

Output:

```
The position of "Marc" is: 8
The position of the first instance of "Hi" is: 0
The position of the last instance of "Hi" is: 14
The position of "Burger" is: -1
```

4.7.4—Replacing a substring:

With `strings.Replace(str, old, new, n)` you can replace the first n occurrences of old in str by new. A copy of str is returned, and if n = -1 all occurrences are replaced.

4.7.5—Counting occurrences of a substring:

Count the number of non-overlapping instances of substring str in s with: `strings.Count(s, str string) int`

An example: Listing 4.15—count_substring.go:

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	var str string = "Hello, how is it going, Hugo?"
	var manyG = "gggggggggg"

	fmt.Printf("Number of H's in %s is: ", str)
	fmt.Printf("%d\n", strings.Count(str, "H"))

	fmt.Printf("Number of double g's in %s is: ", manyG)
	fmt.Printf("%d\n", strings.Count(manyG, "gg"))
}
```

Output:

```
Number of H's in Hello, how is it going, Hugo? is: 2
Number of double g's in gggggggggg is: 5
```

4.7.6—Repeating a string:

Repeat returns a new string consisting of count copies of the string s: `strings.Repeat(s, count int) string`

An example: Listing 4.16—repeat_string.go :

```
package main
import (
	"fmt"
```

```
        “strings”
)

func main() {
        var origS string = “Hi there!”
        var newS string

        newS = strings.Repeat(origS, 3)
        fmt.Printf(“The new repeated string is: %s\n”, newS)
}
```

Output: The new repeated string is: Hi there! Hi there! Hi there!

4.7.7—Changing the case of a string:

ToLower returns a copy of the string s with all Unicode letters mapped to their lower case:

```
strings.ToLower(s) string
```

All uppercase is obtained with: `strings.ToUpper(s) string`

An example: `Listing 4.17—toupper_lower.go`:

```
package main
import (
“fmt”
“strings”
)
func main() {
var orig string = “Hey, how are you George?”
var lower string
var upper string
fmt.Printf(“The original string is: %s\n”, orig)
lower = strings.ToLower(orig)
fmt.Printf(“The lowercase string is: %s\n”, lower)
upper = strings.ToUpper(orig)
fmt.Printf(“The uppercase string is: %s\n”, upper)
}
```

Output:

```
The original string is: Hey, how are you George?
The lowercase string is: hey, how are you george?
The uppercase string is: HEY, HOW ARE YOU GEORGE?
```

4.7.8—Trimming a string:

Here you can use `strings.TrimSpace(s)` to remove all leading and trailing whitespace; if you want to specify in a string cut which characters to remove, use `strings.Trim(s, "cut")`.

Example: `strings.Trim(s, "\r\n")` removes all leading and trailing \r and \n from the string s. The 2nd string-parameter can contain any characters, which are all removed from the left and right-side of s. If you want to remove only leading or only trailing characters or strings, use **`TrimLeft`** `or` **`TrimRight.`**

4.7.9—Splitting a string:

On whitespace: `strings.Fields(s)` splits the string s around each instance of one or more consecutive white space characters, returning a slice of substrings []string of s or an empty list if s contains only white space.

On a separator sep: `strings.Split(s, sep)` : works the same as Fields, but splits around a separator character or string sep (e.g.: ; or, or -).

Because both return a []string, they are often used within a for-range loop (see § 7.3)

4.7.10—Joining over a slice:

This results in a string with all the elements of the slice, separated by sep:

```
Strings.Join(sl []string, sep string)
```

These simple operations are illustred in `Listing 4.18—strings_splitjoin.go:`

```
package main
import (
    "fmt"
    "strings"
)
func main() {
    str := "The quick brown fox jumps over the lazy dog"
    sl := strings.Fields(str)
    fmt.Printf("Splitted in slice: %v\n", sl)
    for _, val := range sl {
        fmt.Printf("%s - ", val)
    }
    fmt.Println()
    str2 := "GO1|The ABC of Go|25"
    sl2 := strings.Split(str2, "|")
    fmt.Printf("Splitted in slice: %v\n", sl2)
    for _, val := range sl2 {
        fmt.Printf("%s - ", val)
    }
    fmt.Println()
    str3 := strings.Join(sl2,";")
    fmt.Printf("sl2 joined by ;: %s\n", str3)
}
/* Output:
Splitted in slice: [The quick brown fox jumps over the lazy dog]
The - quick - brown - fox - jumps - over - the - lazy - dog -
Splitted in slice: [GO1 The ABC of Go 25]
GO1 - The ABC of Go - 25 -
sl2 joined by ;: GO1;The ABC of Go;25
*/
```

Documentation for other functions in this package can be found at: http://golang.org/pkg/strings/

4.7.11—Reading from a string:

The package also has a strings.**NewReader**(str) function. This procuces a pointer to a Reader value, that provides amongst others the following functions to operate on str:

- `Read()` to read a []byte
- `ReadByte() and ReadRune()` : to read the next byte or rune from the string.

4.7.12—Conversion to and from a string:

This functionality is offered by the package **strconv**.

It contains a few variables to calculate the size in bits of an int of the platform on which the program runs: `strconv.IntSize`

To convert a variable of a certain type **T** to a string will always succeed.

For converting from numbers we have the following functions:

`strconv.`**`Itoa`**`(i int) string` : returns the decimal string representation of i

`strconv.`**`FormatFloat`**`(f float64, fmt byte, prec int, bitSize int) string`:
converts the 64-bit floating-point number f to a string, according to the format fmt (can be `'b', 'e', 'f' or 'g'`), precision prec and bitSize is 32 for float32 or 64 for float64.

Converting a string to another type **tp** will not always be possible, in that case a runtime error is thrown: `parsing "…": invalid argument`

For converting to numbers we have the following functions:

`strconv.`**`Atoi`**`(s string) (i int, err error)` : convert to an int

`strconv.`**`ParseFloat`**`(s string, bitSize int) (f float64, err error)` : convert to a 64bit floating-point number

As can be seen from the return-type these functions will return 2 values: the converted value (if possible) and the possible error. So when calling such a function the multiple assignment form will be used: `val, err = strconv.Atoi(s)`

For an example of use, see program: `string_conversion.go.`

In this program we disregard the possible conversion-error with the blank identifier _:

```
anInt, _ = strconv.Atoi(origStr)
```

<u>`Listing 4.19—string_conversion.go:`</u>

```
Package main
import (
    "fmt"
    "strconv"
)
func main() {
    var orig string = "666"
    var an int
    var newS string
    fmt.Printf("The size of ints is: %d\n", strconv.IntSize)
    an, _ = strconv.Atoi(orig)
    fmt.Printf("The integer is: %d\n", an)
    an = an + 5
    newS = strconv.Itoa(an)
    fmt.Printf("The new string is: %s\n", newS)
}
```

```
/* Output:
The size of ints is: 32
The integer is: 666
The new string is: 671 */
```

In § 5.1 discussing the if-statement we will see a way to test a possible error on return.

For more information see: http://golang.org/pkg/strconv/

4.8. Times and dates

The package **time** gives us a datatype **`time.Time`** (to be used as a value) and functionality for displaying and measuring time and dates.

The current time is given by **`time.Now()`**, and the parts of a time can then be obtained as t.Day(), t.Minute(), etc. ; you can make your own time-formats as in: `fmt.Printf("%02d.%02d.%4d\n", t.Day(), t.Month(), t.Year()) // e.g.: 21.07.2011`

The type **`Duration`** represents the elapsed time between two instants as an int64 nanosecond count. The type **Location** maps time instants to the zone in use at that time, UTC represents Universal Coordinated Time.

There is a predefined function func `(t Time) Format(layout string) string`, which formats a time t into a string according to a layout string, with some predefined formats like time.ANSIC or time.RFC822

The general layout defines the format by showing the representation of a standard time, which is then used to describe the time to be formatted; this seems strange, but an example makes this clear: `fmt.Println(t.Format("02 Jan 2006 15:04")) // outputs now: 21 Jul 2011 10:31`

(see program time.go, more info at: http://golang.org/pkg/time/)

Listing 4.20—time.go :

```
package main
import (
        "fmt"
        "time"
)
```

```
var week time.Duration

func main() {
        t := time.Now()
        fmt.Println(t)     // e.g. Wed Dec 21 09:52:14 +0100 RST 2011
        fmt.Printf("%02d.%02d.%4d\n", t.Day(), t.Month(), t.Year())
        // 21.12.2011
        t = time.Now().UTC()
        fmt.Println(t)     // Wed Dec 21 08:52:14 +0000 UTC 2011
        fmt.Println(time.Now()) // Wed Dec 21 09:52:14 +0100 RST 2011
  // calculating times:
        week = 60 * 60 * 24 * 7 * 1e9   // must be in nanosec
        week_from_now := t.Add(week)
        fmt.Println(week_from_now)   // Wed Dec 28 08:52:14 +0000 UTC 2011
// formatting times:
        fmt.Println(t.Format(time.RFC822))     // 21 Dec 11 0852 UTC
        fmt.Println(t.Format(time.ANSIC))      // Wed Dec 21 08:56:34 2011
        fmt.Println(t.Format("02 Jan 2006 15:04")) // 21 Dec 2011 08:52
        s := t.Format("20060102")
        fmt.Println(t, "=>", s)
               // Wed Dec 21 08:52:14 +0000 UTC 2011 => 20111221
}
```

The output is shown after the // in each line.

If you need to let something happen in an application after a certain amount of time or periodically (a special case of event-handling) time.**After** and time.**Ticker** are what you need: §14.5 discusses their interesting possibilities. There is also a function time.**Sleep**(Duration d), which pauses the current process (goroutine in fact, see § 14.1) for a Duration d.

4.9. Pointers

Unlike Java and .NET, Go gives the programmer control over which data structure is a pointer and which is not; however you cannot calculate with pointer values in programs. By giving the programmer control over basic memory layout, Go provides you the ability to control the total size of a given collection of data structures, the number of allocations, and the memory access patterns, all of which are important for building systems that perform well: pointers are important for performance and indispensable if you want to do systems programming, close to the operating system and network.

Because pointers are somewhat unknown to contemporary OO-programmers, we will explain them here and in the coming chapters in depth.

Programs store values in memory, and each memory block (or word) has an address, which is usually represented as a hexadecimal number, like 0x6b0820 or 0xf84001d7f0

Go has the *address-of* operator **&**, which when placed before a variable gives us the memory address of that variable.

The following code-snippet (see Listing 4.9 pointer.go) outputs for example: "An integer: 5, its location in memory: 0x6b0820" (this value will be different every time you run the program!)

```
var i1 = 5
fmt.Printf("An integer: %d, it's location in memory: %p\n", i1, &i1)
```

This address can be stored in a special data type called a *pointer,* in this case it is a pointer to an int, here i1: this is denoted by *int. If we call that pointer intP, we can declare it as

```
var intP *int
```

Then the following is true: `intP = &i1`, intP points to i1.
(because of its name a pointer is represented by **%p** in a format-string)
intP stores the memory address of i1; it points to the location of i1, it *references* the variable i1.

<u>A pointer variable contains the memory address of another value:</u> it points to that value in memory and it takes 4 bytes on 32 bit machines, and 8 bytes on 64 bit machines, regardless of the size of the value they point to. Of course pointers can be declared to a value of any type, be it primitive or structured; the * is placed before the type of the value (prefixing), so the * is here a *type modifier.* Using a pointer to refer to a value is called *indirection.*

A newly declared pointer which has not been assigned to a variable has the **`nil`** value.

A pointer variable is often abbreviated as ptr.

!! In an expression like var p *type always leave a space between the name of the pointer and the `* - var p*type` is syntactically correct, but in more complex expressions it can easily be mistaken for a multiplication !!

The same symbol * can be placed before a pointer like *intP, and then it gives the value which the pointer is pointing to; it is called the *dereference* (or *contents* or *indirection) operator*; another way to say it is that the pointer is flattened.

So for any variable var the following is true: `var == *(&var)`

Now we can understand the complete program pointer.go and its output:

Listing 4.21—pointer.go:

```
package main
import “fmt”

func main() {
        var i1 = 5
        fmt.Printf(“An integer: %d, its location in memory: %p\n”, i1, &i1)

        var intP *int
        intP = &i1
        fmt.Printf(“The value at memory location %p is %d\n”, intP, *intP)
}
```

Output:

```
An integer: 5, its location in memory: 0x24f0820
The value at memory location 0x24f0820 is 5
```

We could represent the memory usage as:

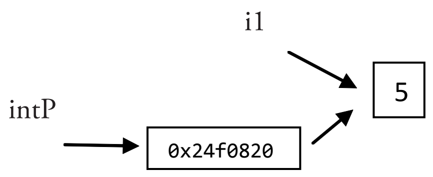


Fig 4.4: Pointers and memory usage

Program string_pointer.go gives us an example with strings.

It shows that assigning a new value to *p changes the value of the variable itself (here a string).

Listing 4.22—string_pointer.go:

```
package main
import "fmt"
func main() {
        s := "good bye"
        var p *string = &s
        *p = "ciao"
        fmt.Printf("Here is the pointer p: %p\n", p)  // prints address
        fmt.Printf("Here is the string *p: %s\n", *p) // prints string
        fmt.Printf("Here is the string  s: %s\n", s)  // prints same string
}
```

Output:

```
Here is the pointer p: 0x2540820
Here is the string *p: ciao
Here is the string  s: ciao
```

By changing the 'object' through giving *p another value, s is also changed.

Schematically in memory:

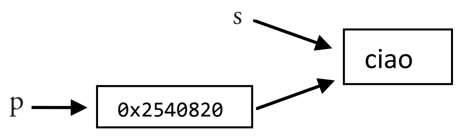


Fig 4.5: Pointers and memory usage, 2

Remark: you cannot take the address of a literal or a constant, as the following code snippet shows:

```
const i = 5
ptr := &i //error: cannot take the address of i
ptr2 := &10 //error: cannot take the address of 10
```

So Go, like most other low level (system) languages as C, C++ and D, has the concept of pointers. But calculations with pointers (so called *pointer arithmetic,* e.g. pointer + 2, to go through the bytes of a string or the positions in an array), which often lead to erroneous memory access in C and thus fatal crashes of programs, are *not* allowed in Go, making the language *memory-safe*. Go pointers resemble more the *references* from languages like Java, C# and VB.NET .

So: `c = *p++` is invalid Go code!!

One advantage of pointers is that you can pass a reference to a variable (for example as a parameter to a function), instead of passing a copy of the variable. Pointers are cheap to pass, only 4 or 8 bytes. When the program has to work with variables which occupy a lot of memory, or many variables, or both, working with pointers can reduce memory usage and increase efficiency. Pointed variables also persist in memory, for as long as there is at least 1 pointer pointing to them, so their lifetime is independent of the scope in which they were created.

On the other hand (but much less likely), because a pointer causes what is called an indirection (a shift in the processing to another address), prohibitive use of them could cause performance decrease.

Pointers can also point to other pointers, and this nesting can go arbitrarily deep, so you can have multiple levels of indirection, but in most cases this will not contribute to the clarity of your code.

As we will see, in many cases Go makes it easier for the programmer and will hide indirection like for example performing an automatic dereference.

A nil pointer dereference, like in the following 2 lines (see program testcrash.go), is illegal and makes a program crash:

Listing 4.23 testcrash.go:

```
package main

func main() {
        var p *int = nil
        *p = 0

}
// in Windows: stops only with: <exit code=”-1073741819” msg=“process crashed”/>
// runtime error: invalid memory address or nil pointer dereference
```

Question 4.2: Give all uses of the symbol * in Go

Chapter 5—Control structures

Until now we have seen that a Go program starts executing in main() and sequentially executes the statements in that function. But often we want to execute certain statements only if a condition is met: we want to make decisions in our code. For this Go provides the following *conditional* or *branching structures*:

> **`if else`** construct
> **`switch case`** construct
> **`select`** construct, for the switching between channels (see § 14.4)

Repeating one or more statements (a task) can be done with *the iterative or looping structure*:

> **`for`** (**`range`**) construct

Some other keywords like **`break`** and **`continue`** can alter the behavior of the loop.

There is also a **`return`** keyword to leave a body of statements and a **`goto`** keyword to jump the execution to a label in the code.

Go entirely omits the parentheses (and) around conditions in if, switch and for-loops, creating less visual clutter than in Java, C++ or C#

5.1—The if else construct

The if test a conditional (a boolean or logical) statement: if this evaluates to true the body of statements between { } after the if is executed, if it is false these statements are ignored and the statement following the if is executed.

```
if condition {
   // do something
}
```

In a 2nd variant an **`else`,**with a body of statements surrounded by { }, is appended, which is executed when the condition is false; we have then 2 exclusive branches (only one of them is executed):

```
if condition {
    // do something
} else {
    // do something else
}
```

In a 3rd variant another **`if`** condition can be placed after the else, so we have 3 exclusive branches:

```
if condition1 {
    // do something
} else if condition2 {
    // do something else
} else {
    // catch-all or default
}
```

The number of else if—branches is in principal not limited, but for readability reasons this should not be exaggerated. When using this form, place the condition which is most likely true first.

The { } are mandatory, also when there is only one statement in the body (some people do not like this, but on the other hand it is consistent and according to mainstream software engineering principles).

The { after the if and else must be on the same line. The else if and else keywords must be on the same line as the closing } of the previous part of the structure. Both of these rules are mandatory for the compiler.

This is invalid Go-code: if x {

```
}
else {                // INVALID
}
```

Note that every branch is indented with 4 (or 8) spaces or 1 tab, and that the closing } are vertically aligned with the if; this is enforced by applying gofmt.

While ( ) around the conditions are not needed, for complex conditions they may be used to make the code clearer. The condition can also be composite, using the logical operators &&, || and !, with the use of ( ) to enforce precedence or improve readability.

A possible application is the testing of different values of a variable and executing different statements in each case, but most often the **switch** statement from § 5.3 is better suited for this.

Listing 5.1—booleans.go:

```
package main

import “fmt”

func main() {
    bool1 := true
    if bool1 {
        fmt.Printf(“The value is true\n”)
    } else {
        fmt.Printf(“The value is false\n”)
    }
}
   // Output: The value is true
```

Note that it is not necessary to test: if bool1 == true, because bool1 is already a boolean value.

It is almost always better to test for true or positive conditions, but it is possible to test for the reverse with ! (not): `if !bool1` or `if !(condition)`. In the last case the ( ) around the condition are often necessary, for example: `if !(var1 == var2)`.

The idiom in Go-code is to omit the else-clause when the if ends in a break, continue, goto or return statement. (see also § 5.2). When returning different values x and y whether or not a condition is true use the following :

IDIOM

```
if condition {
    return x
}
return y
```

Remark: Don't use if / else with a return in both branches, this won't compile: `“function ends without a return statement”` (it's a compiler bug or feature, but it strengthens the idiom above).

Some useful examples:

(1) Checking if a string str is empty:

```
    if  str == “”     { … }
or: if  len(str) == 0  { … }
```

(2) Checking on what operating system the Go-program runs:

This can be done by testing the constant runtime.GOOS (see §2.2)

```
if runtime.GOOS == “windows” {
            …
  } else { // Unix-like
            …
  }
```

A good place to do this is in the init()-function. Here is a code-snippet which changes a prompt to contain the correct ‘end of input’:

```
var prompt = “Enter a digit, e.g. 3 “+  “or %s to quit.”

func init() {
  if runtime.GOOS == “windows” {
          prompt = fmt.Sprintf(prompt, “Ctrl+Z, Enter”)
  } else { // Unix-like
           prompt = fmt.Sprintf(prompt, “Ctrl+D”)
  }
}
```

(3) A function Abs to give the absolute value of an integer:

```
func Abs(x int) int {
  if x < 0 {
    return -x
  }
  return x
}
```

(4) A function isGreater to compare two integers:

```
func isGreater(x, y int) bool {
  if x > y {
    return true
  }
  return false
}
```

In a 4th variant the if can start with an initialization statement (in which a value is given to a variable). This takes the form (the ; after the initialization is mandatory):

```
if initialization; condition {
   // do something
}
```

For example instead of:

```
val := 10
 if val > max {
   // do something
}
```

you can write:

```
if val := 10; val > max {
   // do something
}
```

But pay attention that in the more concise form, the variable val initialized with := is only known within the if-statement (the scope is limited to the statements in { }, but if there is an else clause val is also known there): if a variable val existed in the code before the if, its value is hidden during the if-block. A simple solution to this is to not use := in the if initialization (see also example 3 in §5.2 for how this can be useful).

<u>Listing 5.2—ifelse.go:</u>

```
package main
import "fmt"
func main() {
    var first int = 10
    var cond int
             if first <= 0 {
                fmt.Printf("first is less than or equal to 0\n")
             } else if first > 0 && first < 5 {
                fmt.Printf("first is between 0 and 5\n")
             } else {
                fmt.Printf("first is 5 or greater\n")
             }

             if cond = 5; cond > 10 {
                fmt.Printf("cond is greater than 10\n")
```

```
            } else {
               fmt.Printf("cond is not greater than 10\n")
            }
}
```

Output: first is 5 or greater
cond is not greater than 10

The following code-snippet shows how the result of a function process( ) can be retrieved in the if, and action taken according to the value:

```
IDIOM           if value := process(data); value > max {
                …
                if value := process(data); value > max {
                …
                }
```

5.2—Testing for errors on functions with multiple return values

Often functions in Go are defined so that they return 2 values with successful execution: the value and true, and with unsuccessful execution: a 0 (or nil value) and false (see § 4.4). Instead of true and false, an error-variable can be returned: in the case of successful execution, the error is nil, otherwise it contains the error-information (an error in Go is a variable of type error: `var err error`, more on this in chapter 13). It is then obvious to test the execution with an if-statement; because of its notation this is often called the *comma, ok pattern.*

In the program `string_conversion.go` in § 4.7 the function strconv.Atoi converts a string to an integer. There we disregarded a possible error-condition with:

```
anInt, _ = strconv.Atoi(origStr)
```

If `origStr` cannot be converted to an integer, the function returns 0 for `anInt`, and the _ absorbs the error; the program continues to run.

This is not good: a program should test for every occurring error and behave accordingly, at least informing the user (world) of the error-condition and returning from possibly the function or even halting the program.

This is done in the second version of the code:

Example 1:

Listing 5.3—string_conversion2.go:

```
package main
import (
        "fmt"
        "strconv"
)

func main() {
        var orig string = "ABC"
        var an int
        var err error
        an, err = strconv.Atoi(orig)
        if err != nil {
            fmt.Printf("orig %s is not an integer - exiting with error\n", orig)
            return
        }
        fmt.Printf("The integer is %d\n", an)
// rest of the code
}
```

The idiom is to test if the error-variable err contains a real error (`if err != nil`), in that case an appropriate message is printed and the program execution leaves the executing function with **`return`.** We could also have used the form of return which returns a variable, like **`return err`**, so the calling function can in this case examine the error err.

```
IDIOM

value, err := pack1.Function1(param1)
if err != nil {
        fmt.Printf("An error occurred in pack1.Function1 with parameter %v",
        param1)
            return err
}
// normal case, continue execution:
```

In this case it was main() executing, so the program stops.

If we do want the program to stop in case of an error, we can use the function Exit from package os instead of return:

```
IDIOM

if err != nil {
          fmt.Printf("Program stopping with error %v", err)
          os.Exit(1)
}
```

(The integer which is given to Exit, here 1, can be tested upon in the script outside the program)

Sometimes this idiom is repeated a number of times in succession.

No else branch is written: if there is no error-condition, the code simply continues execution after the if { }.

Example 2: we try to open a file `name` for read-only with `os.Open`:

```
f, err := os.Open(name)
if err !=nil {
    return err
}
doSomething(f)  // In case of no error, the file f is passed to a function
doSomething
```

Exercise 5.1: Rewrite string_conversion2.go by using also := for the err variable, what can be changed?

Example 3: the possible production of an error can occur in the initialization of an if:

```
IDIOM

if err := file.Chmod(0664); err !=nil {
    fmt.Println(err)
    return err
}
```

(this is a Unix-only example, Chmod from package os attempts to change the mode of a file)

Example 4: the initialization can also contain a multiple assignment where ok is a bool return value, which is tested, like in this form:

```
IDIOM

if value, ok := readData(); ok {
…
}
```

Remark: If you accidentally forget a parameter left of the = sign of a multi-return function call, as in the following snippet:

```
func mySqrt(f float64) (v float64, ok bool) {
if f < 0 { return }  // error case
        return math.Sqrt(f),true
}

func main() {
        t := mySqrt(25.0)
        fmt.Println(t)
}
```

then you get the following compiler error: `multiple-value mySqrt() in single-value context`

It must be:

```
t, ok := mySqrt(25.0)
if ok { fmt.Println(t) }
```

Remark 2: When you are really sure that the things you are dealing with are integers and you don't want to test the return value at every conversion, you can wrap Atoi in a function with only returns the integer, like:

```
func atoi (s string) (n int) {
       n, _ = strconv.Atoi(s)
       return
}
```

In fact even the simple Print-functions of fmt (see § 4.4.3) also return 2 values:

```
count, err := fmt.Println(x)  // number of bytes printed, nil or 0, error
```

When printing to the console these are not checked, but when printing to files, network connections, etc. the error value should always be checked. (see also Exercise 6.1b)

5.3—The switch keyword

Compared to the C, Java—languages, switch in Go is considerably more flexible. It takes the general form:

```
switch var1 {
case val1:
      …
case val2:
      …
default:
      …
}
```

where var1 is a variable which can be of any type, and val1, val2, … are possible values of var1; they don't need to be constants or integers, but they must be of the same type, or expressions evaluating to that type. The opening { has to be on the same line as the switch.

More than one value can be tested in a case, the values are presented in a comma separated list like: `case val1, val2, val3:`

Each case-branch is exclusive; they are tried first to last; place the most probable values first.

The first branch that is correct is executed and then the switch-statement is complete: the break from C++, Java and C# happens but is implicit, certainly a big improvement!

So automatic fall-through is not the default behavior; if you want this, use the keyword **`fallthrough`** at the end of the branch.

So

```
switch i {
case 0: //empty case body, nothing is executed when i==0
case 1:
f() // f is not called when i==0!
}
```

And:

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

Fallthrough can also be used in a hierarchy of cases where at each level something has to be done in addition to the code already executed in the higher cases, and when also a default action has to be executed.

After the case … : multiple statements can follow without them being surrounded by { }, but braces are allowed. When there is only 1 statement: it can be placed on the same line as case.

The last statement of such a body can also be a **`return`** with or without an expression.

When the case-statements ends with a return statement, there also has to be a return statement after the } of the switch (see exercise 1).

The (optional) **`default`** branch is executed when no value is found to match var1 with, it resembles the else clause in if-else statements. It can appear anywhere in the switch (even as first branch), but it is best written as the last branch.

Listing 5.4—switch1.go:

```
package main
import "fmt"

func main() {
	var num1 int = 100

	switch num1 {
	case 98, 99:
		fmt.Println("It's equal to 98")
	case 100:
		fmt.Println("It's equal to 100")
	default:
		fmt.Println("It's not equal to 98 or 100")
	}
}
```

```
// Output: "It's equal to 100"
```

In § 12.1 we will use a switch-statement to read input from the keyboard (see Listing 12.2 switch.go). In a 2nd form of the switch-statement no variable is required (this is in fact a switch true) and the cases can test different conditions. The first condition that is true is executed. This looks very much like if-else chaining, and offers a more readable syntax if there are many branches.

```
switch {
  case condition1:
          …
  case condition2:
          …
  default:
          …
}
```

For example:

```
switch {
case i < 0:
    f1()
case i == 0:
    f2()
case i > 0:
    f3()
}
```

Any type that supports the equality comparison operator, such as ints, strings or pointers, can be used in these conditions.

Listing 5.5—switch2.go:

```
package main
import "fmt"

func main() {
         var num1 int = 7

         switch {
         case num1 < 0:
                   fmt.Println("Number is negative")
         case num1 > 0 && num1 < 10:
                   fmt.Println("Number is between 0 and 10")
         default:
                   fmt.Println("Number is 10 or greater")
         }
}
 // Output: Number is between 0 and 10
```

As a 3rd form and like the if, a switch can also contain an initialization statement:

```
    switch initialization {
    case val1:
           …
    case val2:
           …
    default:
           …
  }
```

This can blend nicely with case testing of conditions, like in:

```
switch result := calculate(); {
case result < 0:
  // …
case result > 0:
  // …
default:
  // 0
}
```

Or this snippet, where a and b are retrieved in the parallel initialization, and the cases are conditions:

```
switch a, b := x[i], y[j]; {
case a < b:  t = -1
case a == b: t = 0
case a > b:  t = 1
}
```

There is also a *type-switch* (see § 11.4) which tests the type of an interface variable.

Question 5.1: Give the output of the following code snippet:

```
k := 6
switch k {
case 4: fmt.Println("was <= 4"); fallthrough;
case 5: fmt.Println("was <= 5"); fallthrough;
case 6: fmt.Println("was <= 6"); fallthrough;
case 7: fmt.Println("was <= 7"); fallthrough;
case 8: fmt.Println("was <= 8"); fallthrough;
default: fmt.Println("default case")
}
```

Exercise 5.2–season.go: Write a function *Season* which has as input-parameter a month-number and which returns the name of the season to which this month belongs (disregard the day in the month).

5.4—The for construct

Only the **for** statement exists for repeating a set of statements a number of times; this is possible because it is more flexible than in other languages. One pass through the set is called an *iteration*.

Remark: There is no for-match for the do while-statement found in most other languages, probably because the use case for it was not that important.

5.4.1 Counter-controlled iteration

The simplest form is *counter-controlled iteration*, like in for1.go:

The general format is: `for init; condition; modif { }`

Listing 5.6—for1.go:

```
package main
import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        fmt.Printf("This is the %d iteration\n", i)
    }
}
```

Output:
```
This is the 0 iteration
This is the 1 iteration
This is the 2 iteration
This is the 3 iteration
This is the 4 iteration
```

The body { } of the for-loop is repeated a known number of times, this is counted by a variable (here i). The loop starts (so this is performed only once) with an *initialization* for i (i := 0); this is shorter than a declaration beforehand. This is followed by a *conditional check* on i (i < 10), which is performed before every iteration: when it is true, the iteration is done, the for-loop stops when the condition becomes false. Then comes a *modification* of i (i++), which is performed after every iteration, at which point the condition is checked again to see if the loop can continue. This modification could for example also be a decrement, or + or—using a step.

These are 3 separate statements which form the header of the loop, so they are separated by ; but there are no ( ) surrounding the header: `for (i = 0; i < 10; i++) { }` is invalid code!

Again the opening { has to be on the same line as the for. The counter-variable ceases to exist after the } of the for; always use short names for it like i, j, z or ix.

!! Never change the counter-variable in the for-loop itself, this is bad practice in all languages !!

Exercise 5.3: `i_undefined.go`: this program does not compile, why not ?

```
package main
import "fmt"
func main() {
        for i:=0; i<10; i++ {
                fmt.Printf("%v\n", i)
        }
        fmt.Printf("%v\n", i)  //<-- compile error:  undefined i
}
```

How could you make it work ?

More than 1 counter can also be used, as in: `for i, j := 0, N; i < j; i, j = i+1, j-1 {}`

which is often the preferred way in Go as we can use parallel assignment.

(see the example of reversing an array and exercise string_reverse.go in Chapter 7)

For-loops can be *nested,* like:

```
for i:=0; i<5; ji++ {
    for j:=0; j<10; j++ {
        println(j)
    }
}
```

What happens if we use this kind of for-loop for a general Unicode-string ?

Listing 5.7—for_string.go:

```
package main
import "fmt"
func main() {
        str := "Go is a beautiful language!"
        fmt.Printf("The length of str is: %d\n", len(str))
        for ix :=0; ix < len(str); ix++ {
                fmt.Printf("Character on position %d is: %c \n", ix, str[ix])
        }
        str2 := "日本語"
        fmt.Printf("The length of str2 is: %d\n", len(str2))
        for ix :=0; ix < len(str2); ix++ {
                fmt.Printf("Character on position %d is: %c \n", ix, str2[ix])
        }
}
/* Output:
The length of str is: 27
Character on position 0 is: G
Character on position 1 is: o
Character on position 2 is:
Character on position 3 is: i
…
Character on position 25 is: e
Character on position 26 is: !
The length of str2 is: 9
Character on position 0 is: æ
Character on position 1 is: —
Character on position 2 is:
Character on position 3 is: æ
Character on position 4 is: œ
Character on position 5 is: ¬
Character on position 6 is: è
Character on position 7 is: ª
Character on position 8 is: ┌
*/
```

If we print out the length len of strings str and str2, we get respectively 27 and 9.

We see that for normal ASCII-characters using 1 byte, an indexed character is the full character, whereas for non-ASCII characters (who need 2 to 4 bytes) the indexed character is no longer correct! The for-range from § 5.4.4 will solve this problem.

EXERCISES:

Exercise 5.4: `for_loop.go`

1. Create a simple loop with the for construct. Make it loop 15 times and print out the loop counter with the fmt package.
2. Rewrite this loop using goto. The keyword for may not be used now.

Exercise 5.5: `for_character.go`

Create a program that prints the following (up to 25 characters):

```
G
GG
GGG
GGGG
GGGGG
GGGGGG
GGGGGGG
...
```

1. - use 2 nested for loops
2. - use only one for loop and string concatenation

Exercise 5.6: `bitwise_complement.go` : Show the bitwise complement of the integers 0 till 10, use the bit-respresentation %b

Exercise 5.7: TheFizz-Buzz problem: `fizzbuzz.go`

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". (hint: use a switch with conditions)

Exercise 5.8: `rectangle_stars.go`: Print out a rectangle of width=20 and height=10 with the * character:

5.4.2 Condition-controlled iteration

The 2nd form contains no header and is used for *condition-controlled iteration* (the while-loop in other languages) with the general format: `for condition { }`

You could also argue that it is a for without init and modif section, so that the ; ; are superfluous.

Example: Listing 5.8—for2.go:

```
package main
import "fmt"

func main() {
        var i int = 5
        for i >= 0 {
            i = i - 1
            fmt.Printf("The variable i is now: %d\n", i)
        }
}
```

```
Output:       The variable i is now: 4
              The variable i is now: 3
              The variable i is now: 2
              The variable i is now: 1
              The variable i is now: 0
              The variable i is now: -1
```

5.4.3 Infinite loops

The condition can be absent: like in `for i:=0; ; i++ or for { } (or for ;; { }` but the ; ; is removed by gofmt): these are in fact *infinite loops.* The latter could also be written as: for true { }, but the normal format is: **`for { }`**

If a condition check is missing in a for-header, the condition for looping is and remains always true, so in the loop-body something has to happen in order for the loop to be exited after a number of iterations.

Always take care that the exit-condition will evaluate to true at a certain moment, in order to avoid an endless loop! This kind of loop is exited via a break statement (see § 5.5) or a return statement (see § 6.1).

But there is a difference: break only exits from the loop, while return exits from the function in which the loop is coded!

A typical use for an infinite loop is a server-function which is waiting for incoming requests.

For an example in which this all comes together, see the for loop in listing 12.17 (xml.go):

```
for t, err = p.Token(); err == nil; t, err = p.Token() {
  …
}
```

5.4.4 The for range construct

This is the iterator construct in Go and you will find it useful in a lot of contexts. It is a very useful and elegant variation, used to make a loop over every item in a collection (like arrays and maps, see chapters 7 and 8). It is similar to a foreach in other languages, but we still have the index at each iteration in the loop. The general format is: `for ix, val := range coll { }`

Be careful: val is here a copy of the value at that index in the collection, so it can be used only for read-purposes, the real value in the collection cannot be modified through val (try this out!). A string is a collection of Unicode-characters (or runes), so it can be applied for strings too. If str is a string, you can loop over it with:

```
for pos, char := range str {
…
}
```

Each rune char and its index pos are available for processing inside the loop. It breaks out individual Unicode characters by parsing the UTF-8 (erroneous encodings consume one byte and produce the replacement rune U+FFFD).

See it in action in Listing 5.9—range_string.go:

```
package main
import "fmt"

func main() {
	str := "Go is a beautiful language!"
	for pos, char := range str {
		fmt.Printf("Character on position %d is: %c \n", pos, char)
	}
fmt.Println()
	str2 := "Chinese: 日本語"
	for pos, char := range str2 {
		fmt.Printf("character %c starts at byte position %d\n", char,
		pos)
```

```
        }
        fmt.Println()
        fmt.Println(“index int(rune) rune    char bytes”)
        for index, rune := range str2 {
               fmt.Printf(“%-2d      %d      %U ‘%c’ % X\n”, index, rune, rune,
               rune, []byte(string(rune)))
        }
}
```

Output:

```
Character on position 0 is: G
Character on position 1 is: o
Character on position 2 is:
Character on position 3 is: i
              …
The length of str2 is: 18
character C starts at byte position 0
character h starts at byte position 1
character i starts at byte position 2
character n starts at byte position 3
character e starts at byte position 4
character s starts at byte position 5
character e starts at byte position 6
character : starts at byte position 7
character   starts at byte position 8
character 日 starts at byte position 9
character 本 starts at byte position 12
character 語 starts at byte position 15

index int(rune)  rune    char bytes
0       67       U+0043  ‘C’  43
1       104      U+0068  ‘h’  68
2       105      U+0069  ‘i’  69
3       110      U+006E  ‘n’  6E
4       101      U+0065  ‘e’  65
5       115      U+0073  ‘s’  73
6       101      U+0065  ‘e’  65
7       58       U+003A  ‘:’  3A
8       32       U+0020  ‘ ’  20
9       26085    U+65E5  ‘日’ E6 97 A5
12      26412    U+672C  ‘本’ E6 9C AC
15      35486    U+8A9E  ‘語’ E8 AA 9E
```

Compare this with the output of Listing 5.7 (for_string.go).

We see that the normal English characters are represented by 1 byte, and these Chinese characters by 3 bytes.

Exercise 5.9: What will this loop print out ?

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

Question 5.2: Describe the output of the following valid for-loops:

```
1)      for i := 0; ; i++ {
                fmt.Println("Value of i is now:", i)
        }

2)      for i := 0; i < 3; {
                fmt.Println("Value of i:", i)
        }

3)      s := ""
        for ; s != "aaaaa"; {
                fmt.Println("Value of s:", s)
                s = s + "a"
        }
4)      for i, j, s := 0, 5, "a"; i < 3 && j < 100 && s != "aaaaa"; i, j,
        s = i+1, j+1, s + "a" {
                        fmt.Println("Value of i, j, s:", i, j, s)
        }
```

5.5—Break / continue

Using break, the code of the for-loop in for2.go could then be rewritten (clearly less elegant) as:

Listing 5.10—for3.go:

```
for {
        i = i - 1
        fmt.Printf("The variable i is now: %d\n", i)
        if i < 0 {
               break
        }
    }
```

So in every iteration a condition (here i < 0) has to be checked to see whether the loop should stop. If the exit-condition becomes true, the loop is left through the **`break`** statement.

A break statement always breaks out of the innermost structure in which it occurs; it can be used in any kind of for-loop (counter, condition,etc.), but also in a switch, or a select-statement (see ch 13). Execution is continued after the ending } of that structure.

In the following example with a nested loop (for4.go) break exits the innermost loop:

`Listing 5.11—for4.go:`

```
package main
func main() {
         for i:=0; i<3; i++ {
                   for j:=0; j<10; j++ {
                            if j>5 {
                                break
                            }
                            print(j)
                   }
                   print(“ ”)
         }
}
```

```
// Output:     012345 012345 012345
```

The keyword `continue` skips the remaining part of the loop, but then continues with the next iteration of the loop after checking the condition, see for example Listing 5.12—for5.go:

```
package main
func main() {
        for i := 0; i < 10; i++ {
               if i == 5 {
                      continue
               }
               print(i)
               print(“ ”)
        }
}
```

```
Output:  0 1 2 3 4 6 7 8 9
5 is skipped
```

The keyword `continue` can only be used within a for-loop.

5.6—Use of labels with break and continue—goto

A code line which starts a for, switch, or select statement can be decorated with a *label* of the form identifier:

The first word ending with a colon : and preceding the code (gofmt puts it on the preceding line) is a label, like LABEL1: in `Listing 5.13—for6.go:`

(The name of a label is case-sensitive, it is put in uppercase by convention to increase readability.)

```
package main
import "fmt"

func main() {

LABEL1:
      for i := 0; i <= 5; i++ {
               for j := 0; j <= 5; j++ {
                        if j == 4 {
                                  continue LABEL1
                        }
                        fmt.Printf("i is: %d, and j is: %d\n", i, j)
               }
      }
}
```

Here continue points to LABEL1, the execution jumps to the label.

You see that cases j == 4 and j == 5 do not appear in the output: the label precedes the outer loop, which starts i at its next value, causing the j in the inner for loop to reset to 0 at its initialization. In the same way **`break`** LABEL can be used, not only from a for-loop, but also to break out of a switch. There is even a **`goto`** keyword, which has to be followed by a label name; for example it can be used to simulate a loop as in `Listing 5.14—goto.go:`

```
package main
func main() {
        i:=0
        HERE:
                 print(i)
                 i++
                 if i==5 {
                         return
                 }
                 goto HERE
}
```

which prints 01234.

But a backward goto quicky leads to unreadable 'spaghetti'-code and should not be used, there is always a better alternative.

!! The use of labels and certainly goto is discouraged: it can quickly lead to bad program design, the code can almost always be written more readable without using them. !!

An example where the use of goto is acceptable is in program `simple_tcp_server.go` from § 15.1 :there goto is used to jump out of an infinite read-loop and close the connection with a client when a read-error occurs on that connection.

Declaring a label and not using it is a compiler error (`label … defined and not used`)

If you really have to use goto, use it only with forward labels (a label appearing in the code a number of lines after the goto), and do not declare any new variables between the goto and the label, because this can lead to errors and unexpected results, like in `Listing 5.15—goto2.go` (does not compile!)

```
func main() {
         a := 1
         goto TARGET // compile error:
             // goto TARGET jumps over declaration of b at goto2.go:8
         b := 9
            TARGET:
         b += a
         fmt.Printf("a is %v *** b is %v", a, b)
}
```

Output: a is 1 *** b is 4241844

Question 5.3: Describe the output of the following valid for-loops:

1)

```
    i := 0
     for { //since there are no checks, this is an infinite loop
           if i >= 3 { break }
//break out of this for loop when this condition is met
            fmt.Println("Value of i is:", i)
        i++;
     }
     fmt.Println("A statement just after for loop.")
```

2)

```
for i := 0; i<7 ; i++ {
      if i%2 == 0 { continue }
          fmt.Println("Odd:", i)
  }
```

Chapter 6—Functions

Functions are the basic building blocks in Go code: they are very versatile so that even can be said that Go has a lot of characteristics of a functional language. In this chapter we elaborate on the elementary function-description in § 4.2.2

6.1 Introduction

Every program consist of a number of functions: it is the basic code block.

Because Go code is compiled the order in which the functions are written in the program does not matter; however for readability it is better to start with `main()` and write the functions in a logical order (for example the calling order).

Their main purpose is to break a large problem which requires many code lines into a number of smaller tasks (functions). Also the same task can be invoked several times, so a function promotes code reuse.

(In fact a good program honors the *DRY-principle*, Don't Repeat Yourself, meaning that the code which performs a certain task may only appear once in the program.)

In § 4.2 the main characteristics of functions were described, but now we have more material to build concrete and useful examples.

A function ends when it has executed its last statement (before }), or when it executes a **return** statement, which can be with or without argument(s); these arguments are the values that the functions returns from its computation (see § 6.2). A simple return can thus also be used to end an infinite for-loop, or to stop a goroutine.

There are 3 types of functions in Go:

- Normal functions with an identifier
- Anonymous or lambda functions (see § 6.8)
- Methods (see § 10.6)

Any of these can have parameters and return values. The definition of all the function parameters and return values, together with their types, is called the function *signature.*

And as a reminder, a syntax prerequisite:

This is invalid Go-code:

```
func g()
{               // INVALID
}
```

It must be:

```
func g() {      // VALID
}
```

A function is *called* or *invoked* in code, in a general format like:

```
pack1.Function(arg1,arg2, . . . ,argn)
```

where Function is a function in package pack1, and arg1, etc. are the *arguments*: the values which are passed into the *parameters* of the function (see § 6.2). When a function is invoked copies of the arguments are made and these are then passed to the called function. The invocation happens in the code of another function: the *calling* function. A function can call other functions as much as needed, and these in turn call other functions, and this can go on with theoretically no limit (or the stack upon which these function calls are placed is exhausted).

Here is the simplest example of a function calling another function:

Listing 6.1—greeting.go:

```
package main

func main() {
        println("In main before calling greeting")
        greeting()
        println("In main after calling greeting")
}

func greeting() {
        println("In greeting: Hi!!!!!")
}
```

Output:

```
In main before calling greeting
In greeting: Hi!!!!!
In main after calling greeting
```

A function call can have another function call as its argument, provided that this function has the same number and types of arguments in the correct order that the first function needs, e.g.:

Suppose f1 needs 3 arguments `f1(a, b, c int)`, and f2 returns 3 arguments:

`f2(a, b int) (int, int, int)`, then this can be a call to `f1: f1(f2(a, b))`

Function overloading, that is coding two or more functions in a program with the same function name but a different parameter list and/or a different return-type(s), is not allowed in Go. It gives the compiler error:

```
funcName redeclared in this block, previous declaration at lineno
```

The main reason is that overloading functions forces the runtime to do additional type matching which reduces performance; no overloading means only a simple function dispatch is needed. So you need to give your functions appropriate unique names, probably named according to their signature (but see also §11.12.5).

To declarer a function implemented outside Go, such as an assembly routine, you simply give the name and signature, and no body:

```
func flushICache(begin, end uintptr) // implemented externally
```

Functions can also be used in the form of a declaration, as a function type, like in:

```
type binOp func(int, int) int
```

In that case also the body { } is omitted.

Funtions are first-class values: they can be assigned to a variable, like in: `add := binOp`

The variable add gets a reference (points) to the function and it knows the signature of the function it refers to, so it is not possible to assign to it a function with a different signature.

Function values can be compared: they are equal if they refer to the same function or if both are nil. A function cannot be declared inside another function (no nesting), but this can be mimicked by using anonymous functions (see § 6.8).

Go has until now no concept of generics, which means for example defining 1 function which can be applied to a number of variable types. However most cases can be solved simply by using interfaces, especially the empty interface and a type switch (see § 11.12 and for examples ex. 11.10-11.12) and/or by using reflection (see § 11.10). Code complexity is increased using these techniques and performance lowered, so when performance is very important it is better and will produce more readable code to create the function explicitly for every type used.

6.2 Parameters and return values

A function can take parameters to use in its code, and it can return zero or more values (when more values are returned one often speaks of a *tuple* of values). This is also a great improvement compared to C, C++, Java and C#, and it is particularly handy when testing whether or not a function execution has resulted in an error (see § 5.2)

Returning (a) value(s) is done with the keyword return . In fact every function that returns at least 1 value must end with return or panic (see chapter 13).

Code after return in the same block is not executed anymore. If return is used, then every code-path in the function must end with a return statement.

Question 6.1: The following function does not compile, why not? Correct it.

```
func (st *Stack) Pop() int {
        v := 0
        for ix:= len(st)-1; ix>=0; ix-- {
                if v=st[ix]; v!=0 {
                        st[ix] = 0
                        return v
                }
        }
}
```

Parameters normally have names, but a function can be defined in which no parameter has a name, only a type, like: func `f(int, int, float64)`

A function with no parameters is called a *niladic* function, like `main.main()`.

6.2.1 Call by value / Call by reference

The default way in Go is *to pass a variable* as an argument to a function by *value*: a copy is made of that variable (and the data in it). The function works with and possibly changes the copy, the original value is not changed: `Function(arg1).`

If you want Function to be able to change the value of `arg1` itself ('in place'), you have to pass the memory address of that variable with &, this is *call (*pass) *by reference*: Function(`&arg1`); effectively a *pointer* is then passed to the function. If the variable that is passed is a pointer, then the pointer is copied, not the data that it points to; but through the pointer the function changes the original value.

Passing a pointer (a 32bit or 64bit value) is in almost all cases cheaper than making a copy of the object.

Reference types like slices (ch 7), maps (ch 8), interfaces (ch 10) and channels (ch 13) are pass by reference by default (even though the pointer is not directly visible in the code).

Some functions just perform a task, and do not return values: they perform what is called a *side-effect*, like printing to the console, sending a mail, logging an error, etc.

But most functions return values, which can be named or unnamed.

In the program simple_function.go the function takes 3 int parameters a,b and c and returns an int (the commented lines show more verbose alternative code, where a local variable is used):

<u>`Listing 6.2—simple_function.go:`</u>

```
package main
import “fmt”

func main() {
        fmt.Printf(“Multiply 2 * 5 * 6 = %d\n”, MultiPly3Nums(2, 5, 6))
        // var i1 int = MultiPly3Nums(2, 5, 6)
        // fmt.Printf(“Multiply 2 * 5 * 6 = %d\n”, i1)
}

func MultiPly3Nums(a int, b int, c int) int {
        // var product int = a * b * c
        // return product
        return a * b * c
}
```

<u>`Output:`</u>

```
Multiply 2 * 5 * 6 = 60
```

When it becomes necessary to return more than 4 or 5 values from a function, it is best to pass a slice (see chapter 7) is the values are of the same type (homogeneous), or to use a pointer to a struct (see chapter 10) if they are of different type (heterogeneous). Passing a pointer like that is cheap and allows to modify the data in place.

Question 6.2:

What difference if any is there between the following function calls:

```
(A)   func DoSomething(a *A) {
            b = a
      }
```

and:

```
(B)  func DoSomething(a A) {
            b = &a
      }
```

6.2.2 Named return variables

In the following program multiple_return.go the function takes one int parameter and returns 2 ints; the return values are filled in at the calling function in a parallel assignment.

The two functions `getX2AndX3 and getX2AndX3_2` show how unnamed—or *named return variables* are used. When there is more than 1 unnamed return variable, they must be enclosed within (), like `(int, int)`

Named variables used as *result parameters* are automatically initialized to their zero-value, and once they receive their value, a simple (empty) return statement is sufficient; furthermore even when there is only 1 named return variable, it has to be put inside ( ) (see § 6.6 `fibonacci.go`).

Listing 6.3—multiple_return.go:

```
package main
import "fmt"

var num int = 10
var numx2, numx3 int

func main() {
        numx2, numx3 = getX2AndX3(num)
        PrintValues()
        numx2, numx3 = getX2AndX3_2(num)
        PrintValues()
}

func PrintValues() {
fmt.Printf("num = %d, 2x num = %d, 3x num = %d\n", num, numx2, numx3)
}
func getX2AndX3(input int) (int, int) {
return 2 * input, 3 * input
}
func getX2AndX3_2(input int) (x2 int, x3 int) {
x2 = 2 * input
x3 = 3 * input
// return x2, x3
return
}
```

Output:
```
num = 10, 2x num = 20, 3x num = 30
num = 10, 2x num = 20, 3x num = 30
```

Caution: `return` or `return var` is ok,
but `return var = expression` gives a compiler error:
`syntax error: unexpected =, expecting semicolon or newline or }`

Even with named return variables you can still ignore the names and return explicit values.

When any of the result variables has been shadowed (not a good practice!) the return-statement must contain the result variable names.

!! Use named return variables: they make the code clearer, shorter and self-documenting !!

Exercise 6.1: `mult_returnval.go`

Write a function which accepts 2 integers and returns their sum, product and difference. Make a version with unnamed return variables, and a 2nd version with named return variables.

Exercise 6.2: `error_returnval.go`

Write a function MySqrt which calculates the square root of a float64, but returns an error if this number is negative; make a version with unnamed and a second one with named return variables.

6.2.3 Blank identifier

The blank identifier _ can be used to discard values, effectively assigning the right-hand-side value to nothing, as in `blank_identifier.go`.

The function ThreeValues has no parameters and 3 return-values, where only the first and the third return value are captured in i1 and f1.

Listing 6.4—blank_identifier.go:

```
package main
import "fmt"

func main() {
        var i1 int
        var f1 float32
        i1, _, f1 = ThreeValues()
        fmt.Printf("The int: %d, the float; %f\n", i1, f1)
}

func ThreeValues() (int, int, float32) {
        return 5, 6, 7.5
}
```

Output:
```
The int: 5, the float; 7.500000
```

Another illustration of a function with 2 parameters and 2 return-values which calculates the minimum and maximum value of the 2 parameters is presented in minmax.go.

Listing 6.5—minmax.go:

```
package main
import "fmt"

func main() {
        var min, max int
        min, max = MinMax(78, 65)
        fmt.Printf("Minimum is: %d, Maximum is: %d\n", min, max)
}

func MinMax(a int, b int) (min int, max int) {
        if a < b {
                min = a
                max = b
        } else {  // a = b or a < b
                min = b
                max = a
        }
        return
}
// Output:      Minimum is: 65, Maximum is: 78
```

6.2.4 Changing an outside variable

Passing a pointer to a function not only conserves memory because no copy of the value is made. It has also as side-effect that the variable or object can be changed inside the function, so that the changed object doesn't have to be returned back from the function. See this in the following little program, where reply, a pointer to an integer, is being changed in the function itself.

Listing 6.6—side_effect.go:

```
package main
import (
        “fmt”
)

// this function changes reply:
func Multiply(a, b int, reply *int) {
        *reply = a * b
}

func main() {
        n := 0
        reply := &n
        Multiply(10, 5, reply)
        fmt.Println(“Multiply:”, *reply) // Multiply: 50
}
```

This is only a didactic example, but it is much more useful to change a large object inside a function. However this technique obscures a bit what is going on, the programmer should be very much aware of this side-effect, and if necessary make it clear to users of the function through a comment.

6.3 Passing a variable number of parameters

If the last parameter of a function is followed by **…type**, this indicates that the function can deal with a variable number of parameters of that type, possibly also 0: a so called *variadic* function:

```
func myFunc(a, b, arg ...int) {}
```

The function receives in effect a slice of the type (see chapter 7), which can be looped through with the `for _, v := range` construct of § 5.4.4.

Given the function and call:

```
func Greeting(prefix string, who ...string)
Greeting("hello:", "Joe", "Anna", "Eileen")
```

within `Greeting`, who will have the value `[]string{"Joe", "Anna", "Eileen"}`

If the parameters are stored in an array arr, the function can be called with the parameter arr…

This is demonstrated in `Listing 6.7—varnumpar.go:`

```
package main
import "fmt"

func main() {
        x := Min(1, 3, 2, 0)
        fmt.Printf("The minimum is: %d\n", x)
        arr := []int{7,9,3,5,1}
        x = Min(arr...)
        fmt.Printf("The minimum in the array arr is: %d", x)
}

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

Output:
```
The minimum is: 0
The minimum in the array arr is: 1
```

Exercise 6.3: `varargs.go`

Make a function that has as arguments a variable number of ints and which prints each integer on a separate line.

A function with a variadic parameter can give this parameter to other functions, like in the following code snippet:

```
        function F1(s … string) {
                F2(s …)
                F3(s)
}

        func F2(s … string) {    }
        func F3(s []string)  {    }
```

The variable number parameter can be passed as such or as a slice of its type.

But what if the variable parameters are not of the same type? To have to write say 5 parameters of different type is not elegant. There are 2 possible solutions:

Use a struct: (see chapter 10)

Define a struct type, say Options, which gathers all possible parameters:

```
      type Options struct {
               par1 type1,
               par2 type2,
                      ...
      }
```

A function F1 can then be called with its normal parameters a and b and with no optional parameters as: `F1(a, b, Options {})`

If one or more of the optional parameters have values, call F1 as:

```
        F1(a, b, Options {par1:val1, par2:val2})
```

Use of the empty interface:

If in a variadic parameter the type is not specified, it defaults to the empty `interface { }`

which can stand for any other type (see § 11.9). This could also be used if not only the number of parameters is unknown, but also their type is not specified (perhaps they are each of a different type. The function which receives this parameter can then do a for-range loop over the values, and process them in a type-switch like so:

```
    func typecheck(..,..,values … interface{})  {
            for _, value := range values {
                    switch v := value.(type) {
                            case int:  …
                            case float: …
                            case string: …
                            case bool: …
                            default: …
                    }
            }
    }
```

6.4 Defer and tracing

The `defer` keyword allows us to postpone the execution of a statement or a function until the end of the enclosing (calling) function: it executes something (a function or an expression) when the enclosing function returns (*after* every return and even when an error occurred in the midst of executing the function, not only a return at the end of the function), but before the } (Why after? Because the return statement itself can be an expression which does something instead of only giving back 1 or more variables).

defer resembles the finally-block in OO-languages as Java and C#; in most cases it also serves to free up allocated resources.

This is illustrated in `Listing 6.8—defer.go :`

```
package main
import “fmt”

func main() {
        Function1()
}
```

```
func Function1() {
        fmt.Printf(“In Function1 at the top\n”)
        defer Function2()
        fmt.Printf(“In Function1 at the bottom!\n”)
}

func Function2() {
        fmt.Printf(“Function2: Deferred until the end of the calling function!”)
}
```

```
Output:         In Function1 at the top
                In Function1 at the bottom!
                Function2: Deferred until the end of the calling function!
                (compare the output when defer is removed)
```

If the defer has arguments they are evaluated at the line of the defer-statement; this is illustrated in the following snippet, where the defer will print 0:

```
func a() {
    i := 0
    defer fmt.Println(i)
    i++
    return
}
```

When many defer’s are issued in the code, they are executed at the end of the function in the inverse order (like a stack or LIFO): the last defer is first executed, and so on.

This is illustrated in the following contrived snippet:

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

Which prints out: 4 3 2 1 0

`defer` allows us to guarantee that certain clean-up tasks are performed before we return from a function, for example:

1) closing a file stream:
```
// open a file
defer file.Close() (see § 12.2)
```

2) unlocking a locked resource (a mutex):
```
mu.Lock()
defer mu.Unlock() (see § 9.3)
```

3) printing a footer in a report:
```
printHeader()
defer printFooter()
```

4) closing a database connection:
```
// open a database connection
defer disconnectFromDB()
```

It can be helpful to keep the code cleaner and so often shorter.

The following listing simulates case 4):

Listing 6.9—defer_dbconn.go :
```
package main
import "fmt"

func main() {
    doDBOperations()
}

func connectToDB () {
    fmt.Println( "ok, connected to db" )
}

func disconnectFromDB () {
    fmt.Println( "ok, disconnected from db" )
}

func doDBOperations() {
     connectToDB()
     fmt.Println("Defering the database disconnect.")
     defer disconnectFromDB() //function called here with defer
     fmt.Println("Doing some DB operations ...")
     fmt.Println("Oops! some crash or network error ...")
     fmt.Println("Returning from function here!")
     return //terminate the program
     // deferred function executed here just before actually returning, even if
         there is a return or abnormal termination before
}
/* Output:
ok, connected to db
Defering the database disconnect.
Doing some DB operations ...
Oops! some crash or network error ...
Returning from function here!
ok, disconnected from db
*/
```

Tracing with defer:

A primitive but sometimes effective way of tracing the execution of a program is printing a message when entering and leaving certain functions. This can be done with the following 2 functions:

```
func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }
```

where we will call untraced with the defer keyword, as in the following program

Listing 6.10— defer_tracing.go:

```
package main
import "fmt"

func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

func a() {
         trace("a")
         defer untrace("a")
         fmt.Println("in a")
}
```

```
func b() {
        trace("b")
        defer untrace("b")
        fmt.Println("in b")
        a()
}

func main() {
        b()
}
```

which outputs:

```
entering: b
in b
entering: a
win a
leaving: a
leaving: b
```

This can be done more succinctly: as in Listing 6.11—_defer_tracing2.go:

```
package main
import "fmt"

func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}
func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
        fmt.Println("in a")
}

func b() {
        defer un(trace("b"))
        fmt.Println("in b")
        a()
}
func main() {
        b()
}
```

Using defer to log parameter- and return values from within the function:

This is another possible use of defer which might come in handy while debugging:

Listing 6.12— defer_logvalues.go:

```
package main

import (
    “log”
    “io”
)
func func1(s string) (n int, err error) {
    defer func() {
        log.Printf(“func1(%q) = %d, %v”, s, n, err)
    }()
    return 7, io.EOF
}

func main() {
    func1(“Go”)
}
```

```
// Output: 2011/10/04 10:46:11 func1(“Go”) = 7, EOF
```

6.5 Built-in functions

These are predefined functions which can be used as such, without having to import a package to get access to them. They sometimes apply to different types, e.g. len, cap and append, or they have to operate near system level like panic. That’s why they need support from the compiler.

Here is a list, they will be further discussed in the following chapters.

close		Used in channel communication
len	**cap**	len gives the length of a number of types (strings, arrays, slices, maps, channels); cap is the capacity, the maximum storage (only applicable to slices and maps)
new	**make**	Both new and make are used for allocating memory: *new for value types and user-defined types like structs,* *make for built-in reference types (slices, maps, channels)*

They are used like functions with the type as its argument: new(type), make(type)

new(T) allocates zeroed storage for a new item of type T and returns its address, so it returns a pointer to the type T (see also § 10.1: Using new)
It can be used with primitive types as well:

```
        v := new(int) // v has type *int
```

make(T) returns an *initialized* variable of type T, so it does more work than new (see also §7.2.3/4, §8.1.1. and § 14.2.1)
!! new( ) is a function, don't forget its parentheses !!

copy	**append**	used resp. for copying and concatenating slices (see § 7.5.5)
panic	**recover**	both are used in a mechanism for handling errors (see § 13.2)
print	**println**	low level printing functions (see § 4.2), use the fmt package in production programs
complex	**real imag**	used for making and manipulating complex numbers (see § 4.5.2.2)

6.6 Recursive functions

A function that call itself in its body is called *recursive*. The proverbial example is the calculation of the numbers of the Fibonacci sequence, in which each number is the sum of its two preceding numbers.

The sequence starts with:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,
10946, …
```

This is done in the following program:

Listing 6.13—_ fibonacci.go:

```
package main
import "fmt"

func main() {
        result := 0
        for i:=0; i <= 10; i++ {
                result = fibonacci(i)
                fmt.Printf("fibonacci(%d) is: %d\n", i, result)
        }
}

func fibonacci(n int) (res int) {
        if n <= 1 {
            res = 1
        } else {
            res = fibonacci(n-1) + fibonacci(n-2)
        }
        return
}
```

Output:

```
fibonacci(0) is: 1
fibonacci(1) is: 1
fibonacci(2) is: 2
fibonacci(3) is: 3
fibonacci(4) is: 5
fibonacci(5) is: 8
fibonacci(6) is: 13
fibonacci(7) is: 21
fibonacci(8) is: 34
fibonacci(9) is: 55
fibonacci(10) is: 89
```

Many problems have an elegant recursive solution, like the famous Quicksort algorithm.

An important problem when using recursive functions is *stack overflow*: this can occur when a large number of recursive calls are needed and the programs runs out of allocated stack memory. This can be solved by using a technique called lazy evaluation, implemented in Go with a channel and a goroutine (see § 14.8). Exercise 14.12 solves the Fibonacci problem in this way.

Mutually recursive functions can also be used in Go: these are functions that call one another. Because of the Go compilation process, these functions may be declared in any order. Here is a simple example: even calls odd, and odd calls even.

Listing 6.14—_ mut_recurs.go:

```
package main
```

```
import (
    “fmt”
)

func main() {
    fmt.Printf(“%d is even: is %t\n”, 16, even(16)) // 16 is even: is true
    fmt.Printf(“%d is odd: is %t\n”, 17, odd(17))
  // 17 is odd: is true
    fmt.Printf(“%d is odd: is %t\n”, 18, odd(18))
  // 18 is odd: is false
}

func even(nr int) bool {
    if nr == 0 {return true}
    return odd(RevSign(nr)-1)
}

func odd(nr int) bool {
    if nr == 0 {return false}
    return even(RevSign(nr)-1)
}

func RevSign(nr int) int {
         if nr < 0 {return -nr}
         return nr
}
```

EXERCISES:

Exercise 6.4: `fibonacci2.go`
Rewrite the Fibonacci program above to return 2 named variables (see § 6.2), namely the value and the position of the Fibonacci-number, like 5 and 4 or 89 and 10.

Exercise 6.5: `10to1_recursive.go`
Print the numbers from 10 to 1 in that order using a recursive function printrec(i int)

Exercise 6.6: `factorial.go`
Write a program which prints the factorial (!) of the first 30 integers

The factorial n! of a number n is defined as: n! = n * (n-1)!, 0!=1

So this clearly a good candidate for a recursive function Factorial.

Make a 2nd version of Factorial with a named return variable.

Remark that when using type int the calculation is only correct up until 12!, this is of course because an int can only contain integers which fit in 32 bit. Go doesn't warn against this *overflow-error*! How can you gset more correct results ?

The best solution is to use the big package (see § 9.4).

6.7 Functions as parameters

Functions can be used as parameters in another function, the passed function can then be called within the body of that function, that is why it is commonly called a *callback*. To illustrate here is a simple example:

Listing 6.15—_ function_parameter.go:

```
package main
import (
       "fmt"
)

func main() {
       callback(1, Add)
}

func Add(a,b int) {
       fmt.Printf("The sum of %d and %d is: %d\n", a, b, a + b)
}

func callback(y int, f func(int, int)) {
       f(y, 2)  // this becomes Add(1, 2)
}
```

```
// Output:  The sum of 1 and 2 is: 3
```

A good example of the use of a function as a parameter is the `strings.IndexFunc()` function:

It has the signature `func IndexFunc(s string, f func(c int) bool) int` and returns the index into s of the first Unicode character for which f(c) is true, or -1 if none do.

For example `strings.IndexFunc(line, unicode.IsSpace)` will return the index of the 1st whitespace character in line. Of course you can make your own function f, e.g.:

```
func IsAscii(c int) bool {
        if c > 255 {
              return false
        }
        return true
}
```

In § 14.10.1 we will also discuss an example of a function which has another function as a parameter, in the context of writing a client-server program:

```
type binOp func(a, b int) int
func run(op binOp, req *Request) { … }
```

Exercise 6.7: `strings_map.go`

The Map function in the package strings is also a good example of the use of higher order functions, like strings.IndexFunc(). Look up its definition in the package documentation and make a little test program with a map function that replaces all non-ASCII characters from a string with a ? or a space. What do you have to do to delete these characters?

6.8 Closures (function literals)

Sometimes we do not want to give a function a name, then we make an *anonymous function* (also known under the names of a *lambda function*, a *function literal,* or a *closure)*, for example: `func(x, y int) int { return x + y }`

Such a function cannot stand on its own (the compiler gives the error: `non-declaration statement outside function body`) but it can be assigned to a variable which is a reference to that function:
`fplus := func(x, y int) int { return x + y }`

and then invoked as if fplus was the name of the function: `fplus(3,4)`

or it can be invoked directly: `func(x, y int) int { return x + y } (3, 4)`

Here is a call to a lambda function which calculates the sum of integer floats till 1 million, gofmt reformats a lambda function in this way:

```
func() {
    sum = 0.0
    for i := 1; i<= 1e6; i++ {
       sum += i
    }
}()
```

The first ( ) is the parameter-list and follows immediately after the keyword func because there is no function-name. The { } comprise the function body, and the last pair of ( ) represent the call of the function.

Here is an example of assigning it to a variable, g in the following snippet (`function_literal.go`) which is then of type function*:*

Listing 6.16—_ function_literal.go:

```
package main
import “fmt”

func main() {
        f()
}

func f() {
        for i := 0; i < 4; i++ {
                 g := func(i int) { fmt.Printf(“%d “, i) }
                 g(i)
                 fmt.Printf(“ - g is of type %T and has value %v\n”, g, g)
        }
}
```

Output:
```
0 - g is of type func(int) and has value 0x681a80
1 - g is of type func(int) and has value 0x681b00
2 - g is of type func(int) and has value 0x681ac0
3 - g is of type func(int) and has value 0x681400
```

We see that the type of g is func(int), its value is a memory address.

So in this assignment we have in fact a variable with a function value: *lambda functions can be assigned to variables and treated as values.*

Exercise 6.8: Write a program which in main() defines a lambda function which prints "Hello World". Assign it to a variable fv, call the function with as that value, and check the type of fv (`lambda_value.go`)

Anonymous functions like all functions can be with- or without parameters; in the following example there is a value v passed in the parameter u:

```
func (u string) {
        fmt.Println(u)
        …
}(v)
```

Consider this contrived example: what is the value of ret after return from f ?

Listing 6.17—_ return_defer.go:

```
package main
import "fmt"

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

func main() {
        fmt.Println(f())
}
```

The value is 2, because of the ret++, which is executed after return 1.

This can be convenient for modifying the error return value of a function.

defer and lambda functions:

The defer keyword (§ 6.4) is often used with a lambda function. It can then also change return values, provided that you are using named result parameters.

Lambda functions can also be launched as a goroutine with go (see chapter 14 and § 16.9).

Lambda functions are also called *closures* (a term from the theory of functional languages): they may refer to variables defined in a surrounding function. A closure is a function that captures some external state—for example, the state of the function in which it is created. Another way to formulate this is: a closure inherits the scope of the function in which it is declared. That state (those variables) is then shared between the surrounding function and the closure, and the variables survive as long as they are accessible; see the following examples in §6.9. Closures are often used as *wrapper* functions: they predefine 1 or more of the arguments for the wrapped function; this is also illustrated in the following § and in many examples to come.Another good application is using closures in performing clean error-checking (see § 16.10.2)

6.9 Applying closures: a function returning another function

In program `function_return.go` we see functions Add2 and Adder which return another lambda function `func(b int) int`:

```
func Add2() (func(b int) int)
func Adder(a int) (func(b int) int)
```

Add2 takes no parameters, but Adder takes an int as parameter.

We can make special cases of Adder and give them a name, like TwoAdder in listing 6.13.

Listing 6.18—_ function_return.go:

```
package main
import “fmt”

func main() {
        // make an Add2 function, give it a name p2, and call it:
        p2 := Add2()
        fmt.Printf(“Call Add2 for 3 gives: %v\n”, p2(3))
        // make a special Adder function, a gets value 3:
        TwoAdder := Adder(2)
        fmt.Printf(“The result is: %v\n”, TwoAdder(3))
}

func Add32() (func(b int) int) {
        return func(b int) int {
        return b + 2
        }
}
```

```
func Adder(a int) (func(b int) int) {
         return func(b int) int {
                return a + b
         }
}
/* Output:
Call Add2 for 3 gives: 5
The result is: 5
*/
```

Here is (nearly) the same function used in a slightly different way:

Listing 6.19—_ function_closure.go:

```
package main
import "fmt"

func main() {
         var f = Adder()
         fmt.Print(f(1)," - ")
         fmt.Print(f(20)," - ")
         fmt.Print(f(300))
}

func Adder() func(int) int {
         var x int
         return func(delta int) int {
                    x += delta
                    return x
         }
}
```

`Adder()` is now assigned to the variable f (which is then of type func(int) int)

The output is: 1 - 21 - 321

In the calls to f, delta in Adder() gets respectively the values 1, 20 and 300.

We see that between calls of f the value of x is retained, first it is 0 + 1 = 1, then it becomes 1 + 20 = 21, then 21 is added to 300 to give the result 321: the lambda function stores and accumulates the values of its variables: it still has access to the (local) variables defined in the current function.

These local variables can also be parameters, like `Adder(a int)` in listing 6.13.

This demonstrates clearly that Go literal functions are *closures*.

The variables that the lambda function uses or updates can also be declared outside of the lambda, like in this snippet:

```
var g int
go func(i int) {
        s := 0
        for j := 0; j < i; j++ { s += j }
        g = s
}(1000) // Passes argument 1000 to the function literal.
```

The lambda function can be applied to all elements of a collection, updating these variables. Afterwards these variables can be used to represent or calculate global values or averages.

Exercise 6.9: Write a non-recursive version of the Fibonacci program from § 6.6 using a function as a closure: (`fibonacci_closure.go`)

Exercise 6.10: Study and comprehend the working of the following program: (`compose.go`)

A function returning another function can be used as a *factory function*. This can be useful when you have to create a number of similar functions: write 1 factory function instead of writing them all individually. The following function illustrates this: it returns functions that add a suffix to a filename when this is not yet present:

```
func MakeAddSuffix(suffix string) func(string) string {
    return func(name string) string {
        if !strings.HasSuffix(name, suffix) {
            return name + suffix
         }
         return name
    }
}
```

Now we can make functions like:

```
addBmp := MakeAddSuffix(".bmp")
addJpeg := MakeAddSuffix(".jpeg")
```

and call them as:

```
addBmp("file")          // returns: file.bmp
addJpeg("file")    // returns: file.jpeg
```

A function which can return another function and a function which has another function as a parameter are called *higher order functions,* which is a hallmark of the category of languages called *functional languages*. We have seen in §6.7 that functions are also values, so it is clear that Go possesses some of the major characteristics of a functional language. Closures are used frequently in Go, often in combination with goroutines and channels (see chapter 14 §14.8-9). In §11.14 program cars.go we see the power of functions in Go in action in an example with concrete objects.

6.10 Debugging with closures

When analyzing and debugging complex programs with myriads of functions in different code-files calling one another, it can often be useful to know at certain points in the program the file which is executing and the line number in it. This can be done by using special functions from the packages runtime or log. In package runtime the function Caller() provides this information, so a closure where() could be defined which calls this, and then be invoked wherever it is needed:

```
where := func() {
    _, file, line, _ := runtime.Caller(1)
    log.Printf("%s:%d", file, line)
}
where()
// some code
where()
// some more code
where()
```

The same can be achieved by setting a flag in the log package:

```
log.SetFlags(log.Llongfile)
log.Print("")
```

or if you like the brevity of "where":

```
var where = log.Print

func func1() {
where()
... some code
where()
... some code
where()
}
```

6.11 Timing a function

Sometimes it is interesting to know how long a certain computation took, e.g. for comparing and benchmarking calculations. A simple way is to record the start-time before the calculation, and the end-time after the calculation by using the function `Now()` from the time package; and the difference between them can then be calculated with `Sub()`. In code, this goes like:

```
start := time.Now()
longCalculation()
end := time.Now()
delta := end.Sub(start)
fmt.Printf(“longCalculation took this amount of time: %s\n”, delta)
```

See it in action in the program `Listing 6.20—fibonacci.go.`

If you have optimized a piece of code always time the former version and the optimized version to see that there is a significant (enough) advantage; in the following § we see an optimization applied which is certainly worthwhile.

6.12 Using memoization for performance

When doing heavy computations one thing that can be done for increasing performance is not to repeat any calculation that has already been done and that must be reused. Instead cache the calculated value in memory, which is called *memoization*. A great example of this is the Fibonacci program (see § 6.6 and 6.11):

to calculate the n-th Fibonacci number, you need the 2 preceding ones, which normally have already been calculated. If you do not stock the preceding results, every higher Fibonacci number results in an ever greater avalanche of recalculations, which is precisely what the version from listing 6.11 (fibonnaci.go) does.

Simple stock the n-th Fibonacci number in an array at index n (see chapter 7), and before calculating a fibonnaci-number, first look in the array if it has not yet been calculated.

This principle is applied in Listing 6.17 (fibonacci_memoization.go). The performance gain is astounding, time both programs for the calculation up to the 40th Fibonnaci number:

```
normal (fibonacci.go): the calculation took this amount of time: 4.730270 s
with memoization: the calculation took this amount of time: 0.001000 s
```

In this algorithm memoization is obvious, but it can often be applied in other computations as well, perhaps using maps (see chapter 7) instead of arrays or slices.

Listing 6.21—_ fibonacci_memoization.go:

```
package main
import (
        “fmt”
        “time”
)

const LIM = 41
var fibs [LIM]uint64
func main() {
        var result uint64 = 0
        start := time.Now()
        for i:=0; i < LIM; i++ {
               result = fibonacci(i)
               fmt.Printf(“fibonacci(%d) is: %d\n”, i, result)
        }
        end := time.Now()
        delta := end.Sub(start)
        fmt.Printf(“longCalculation took this amount of time: %s\n”, delta)
}

func fibonacci(n int) (res uint64) {
        // memoization: check if fibonacci(n) is already known in array:
        if fibs[n] != 0 {
               res = fibs[n]
               return
        }
        if n <= 1 {
               res = 1
        } else {
               res = fibonacci(n-1) + fibonacci(n-2)
        }
        fibs[n] = res
        return
}
```

Memoization is useful for relatively expensive functions (not necessarily recursive as in the example) that are called lots of times with the same arguments. It can also only be applied to *pure functions*, these are functions that always produce the same result with the same arguments, and have no side-effects.

Chapter 7—Arrays and Slices

In this chapter we start with examining data-structures that contain a number of items, so called *collections*, such as arrays (slices) and maps. Here the Python influence is obvious.

The array-type, indicated by the `[ ]`, notation is well-known in almost every programming language as the basic workhorse in applications. The Go array is very much the same, but has a few peculiarities. It is not as dynamic as in C, but for that Go has the slice type. This is an abstraction built on top of Go's array type, and so to understand slices we must first understand arrays. Arrays have their place, but they are a bit inflexible, so you don't see them too often in Go code. Slices, though, are everywhere. They build on arrays to provide great power and convenience.

7.1 Declaration and initialization

7.1.1 Concept

An array is a numbered and fixed-length sequence of data items (elements) of the same *single type (*it is a *homogeneous data structure*); this type can be anything from primitive types like integers, strings to self-defined types. The length must be a constant expression, that must evaluate to a non-negative integer value. It is part of the type of the array, so arrays declared as [5]int and [10] int differ in type. Initialization of an array with values known at compile time is done with array literals (see below).

Remark: If we would like the item type can be any type by using the empty interface as type (see § 11.9). When using the values we would first have to do a type assertion (see § 11.3).

The items can be accessed (and changed) through their *index* (the position), the index starts from 0, so the 1st element has index 0, the 2nd index 1, etc. (arrays are *zero-based* as usual in the C-family of languages). The number of items, also called the *length* len or size of the array, is fixed and must be given when declaring the array (len has to be determined at compile time in order to allocate the memory); the maximum array length is 2 Gb.

The format of the declaration is: `var identifier [len]type`

For example: `var arr1 [5]int`

which can be visualized in memory as:

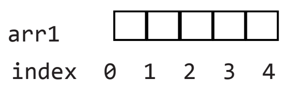


Fig 7.1: Array in memory

Each compartment contains an integer; when declaring an array, each item in it is automatically initialized with the default zero-value of the type, here all items default to 0.

The length of arr1 `len(arr1)` is 5, and the index ranges from 0 to `len(arr1)-1`.

The first element is given by `arr1[0]`, the 3rd element is given by `arr1[2]` ; in general the element at index i is given by `arr1[i]`. The last element is given by: `arr1[len(arr1)-1]`

Assigning a value to an array-item at index i is done by: `arr[i] = value`, so arrays are *mutable.*

Only valid indexes can be be used. When using an index equal to or greater than `len(arr1)`: if the compiler can detect this, the message index out of bounds is given; but otherwise the code compiles just fine and executing the program will give the panic (see chapter 13):

runtime error: index out of range.

Because of the index, a natural way to loop over an array is to use the for-construct:

- for initializing the items of the array
- for printing the values, or in general:
- for procession each item in succession.

A basic example is given in `Listing 7.1—for_arrays.go:`

```
package main
import “fmt”

func main() {
        var arr1 [5]int

        for i:=0; i < len(arr1); i++ {
                arr1[i] = i * 2
        }

        for i:=0; i < len(arr1); i++ {
                fmt.Printf("Array at index %d is %d\n", i, arr1[i])
        }
}
```

Output:

```
Array at index 0 is 0
Array at index 1 is 2
Array at index 2 is 4
Array at index 3 is 6
Array at index 4 is 8
```

Very important here is the condition in the for-loop: `i < len(arr1)`

`i <= len(arr1)` would give an index out of range-error.

IDIOM:

```
for i:=0; i < len(arr1); i++ {
arr1[i]= …
}
```

The for-range construct is also useful:

*I*DIOM:

```
for i:= range arr1 {
…
}
```

Here i is also the index in the array. Both for-constructs also work for slices (§ 7.2) of course.

Question 7.1: What is the output of the following code snippet?

```
a := [...]string{"a", "b", "c", "d"}
        for i := range a {
                fmt.Println("Array item", i, "is", a[i])
        }
```

An array in Go is a *value type* (not a pointer to the first element like in C/C++), so it can be created with **new**():

```
var arr1 = new([5]int)
```

What is the difference with: `var arr2 [5]int ?` arr1 is of type *[5]int, arr2 is of type [5]int .

The consequence is that when assigning an array to another, a distinct copy in memory of the array is made. For example when:

```
arr2 := arr1
arr2[2] = 100
```

then the arrays have different values; changing arr2 after the assignment does not change arr1.

So when an array is passed as an argument to a function like in `func1(arr1)`, a copy of the array is made, and func1 cannot change the original array arr1.

If you want this to be possible or you want a more efficient solution, then arr1 must be passed by reference with the &-operator, as in `func1(&arr1)`, like in the following example

Listing 7.2—pointer_array.go:

```go
package main
import "fmt"
func f(a [3]int)   { fmt.Println(a) }
func fp(a *[3]int) { fmt.Println(a) }

func main() {
         var ar [3]int
         f(ar)   // passes a copy of ar
         fp(&ar) // passes a pointer to ar
}
```

Output:

```
 [0 0 0]
&[0 0 0]
```

Another equivalent way is to make a slice of the array and pass that to the function (see Passing an array to a function in § 7.1.4)

EXERCISES:

Exercise 7.1: `array_value.go`: Prove that when assigning an array to another, a distinct copy in memory of the array is made.

Exercise 7.2: for_array.go: Write the loop that fills an array with the loop-counter (from 0 to 15) and then prints that array to the screen.

Exercise 7.3: fibonacci_array.go: In § 6.6 we saw a recursive solution for calculating Fibonacci numbers. But they can also be calculated in an imperative way, using a simple array. Do this for the first 50 Fibonacci numbers.

7.1.2 Array literals

When the values (or some of them) of the items are known beforehand, a simpler initialization exists using the { , , } notation called *array literals* (or *constructors)*, instead of initializing every item in the [ ]= way. (All composite types have a similar syntax for creating literal values).

This is illustrated in the following code Listing 7.3—array_literals.go:

```
package main
import "fmt"

func main() {
	var arrAge = [5]int{18, 20, 15, 22, 16}
	var arrLazy = [...]int{5, 6, 7, 8, 22}
	// var arrLazy = []int{5, 6, 7, 8, 22}
	var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
	//var arrKeyValue = []string{3: "Chris", 4: "Ron"}

	for i := 0; i < len(arrKeyValue); i++ {
		fmt.Printf("Person at %d is %s\n", i, arrKeyValue[i])
	}
}
```

1st variant: `var arrAge = [5]int{18, 20, 15, 22, 16}`
Note that the [5]int can be omitted from the left-hand side
`[10]int { 1, 2, 3 }` : this is an array of 10 elements with the 1st three different from 0.

2nd variant: `var arrLazy = [...]int{5, 6, 7, 8, 22}`
... indicates the compiler has to count the number of items to obtain the length of the array.
But [...]int is not a type, so this is illegal:
`var arrLazy [...]int = [...]int{5, 6, 7, 8, 22}`
The ... can also be omitted (technically speaking it then becomes a slice).

3rd variant: *key: value syntax*

```
var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
```

Only the items with indexes (keys) 3 and 4 get a real value, the others are set to empty strings, as is shown in the output:

```
Person at 0 is
Person at 1 is
Person at 2 is
Person at 3 is Chris
Person at 4 is Ron
```

Here also the length can be written as ... or even be omitted.

You can take the address of an array literal to get a pointer to a newly created instance, see `Listing 7.4—pointer_array2.go:`

```
package main
import "fmt"

func fp(a *[3]int) { fmt.Println(a) }

func main() {
        for i := 0; i < 3; i++ {
                fp(&[3]int{i, i * i, i * i * i})
        }
}
```

Output:
```
&[0 0 0]
&[1 1 1]
&[2 4 8]
```

Geometrical points (or mathematical *vectors*) are a classic example of the use of arrays. To clarify the code often an alias is used:

```
type Vector3D [3]float32
var vec Vector3D
```

7.1.3 Multidimensional arrays

Arrays are always 1-dimensional, but they may be *composed* to form multidimensional arrays, like:

```
[3][5]int
[2][2][2]float64
```

The inner arrays have always the same length. Go's multidimensional arrays are *rectangular* (the only exceptions can be arrays of slices, see § 7.2.5).

Here is a code snippet which uses such an array:

Listing 7.5—multidim_array.go:

```
package main

const (
        WIDTH =1920
        HEIGHT = 1080
)

type pixel int
var screen [WIDTH][HEIGHT]pixel

func main() {
         for y := 0; y < HEIGHT; y++ {
                    for x := 0; x < WIDTH; x++ {
                             screen[x][y] = 0
                    }
         }
}
```

7.1.4 Passing an array to a function

Passing big arrays to a function quickly uses up much memory.

There are 2 solutions to prevent this:

1- Pass a pointer to the array
2- Use a slice of the array

The following example `Listing 7.6—array_sum.go` illustrates the first solution:

```
package main
import "fmt"

func main() {
        array := [3]float64{7.0, 8.5, 9.1}
        x := Sum(&array) // Note the explicit address-of operator
        // to pass a pointer to the array
        fmt.Printf("The sum of the array is: %f", x)
}
```

```
func Sum(a *[3]float64) (sum float64) {
        for _, v := range a {   // dereferencing *a to get back to the array is
        not necessary!
               sum += v
        }
        return
}
// Output: The sum of the array is: 24.600000
```

But this is not idiomatic Go, solution 2 using slices is: see § 7.2.2

7.2 Slices

7.2.1 Concept

A slice is a reference to a contiguous segment(section) of an array (which we will call the underlying array, and which is usually anonymous), so a slice is a *reference type* (thus more akin to the array type in C/C++, or the list type in Python). This section can be the entire array, or a subset of the items indicated by a start- and an end index (the item at the end-index is not included in the slice). Slices provide a dynamic window onto the underlying array.

Slices are indexable and have a length given by the len()-function.

The slice-index of a given item can be less than the index of the same element in the underlying array. Unlike an array, the length of a slice can change during execution of the code, minimally 0 and maximally the length of the underlying array: a slice is *variable-length array*.

The built-in capacity function `cap()` of a slice is a measure of how long a slice can become: it is the length of the slice + the length of the array beyond the slice. If s is a slice `cap` is the size of the array from `s[0]` to the end of the array. A slice's length can never exceed its capacity, so the following statement is always true for a slice s: `0 <= len(s) <= cap(s)`

Multiple slices can share data if they represent pieces of the same array; multiple arrays can never share data. A slice therefore shares storage with its underlying array and with other slices of the same array, by contrast distinct arrays always represent distinct storage. Arrays are in fact building blocks for slices.

Advantage: Because slices are references, they don't use up additional memory and so are more efficient to use than arrays, so they are used much more than arrays in Go-code.

The format of the declaration is: `var identifier []type` no length is needed.

A slice that has not yet been initialized is set to nil by default, and has length 0.

Format of initialization of a slice: `var slice1 []type = arr1[start:end]`

This represents the subarray of arr1 (*slicing* the array, `start:end` is called a *slice-expression*) composed of the items from index `start` to index `end-1`.So `slice1[0] == arr1[start]` is a true statement. This can be defined even before the array arr1 is populated.

If one writes: `var slice1 []type = arr1[:]` then slice1 is equal to the complete array arr1 (so it is a shortcut for `arr1[0:len(arr1)]`). Another way to write this is: `slice1 = &arr1.`

`arr1[2:]` is the same as `arr1[2:len(arr1)]` so contains all the items of the array from the 2nd till the last.

`arr1[:3]` is the same as `arr1[0:3]`, containing the array-items from the 1st till (not including) the 3rd .

If you need to cut the last element from slice1, use: `slice1 = slice1[:len(slice1)-1]`

A slice of the array with elements 1,2 and 3 can be made as follows: `s := [3]int{1,2,3}[:` or `s := […]int{1,2,3}[:]` or even shorter `s := []int{1,2,3}`

`s2 := s[:]` is a slice made out of a slice, having identical elements, but still refers to the same underlying array.

A slice s can be expanded to its maximum size with: `s = s[:cap(s)]`, any larger gives a run time error (see listing 7.7).

For every slice (also for strings) the following is always true:

```
s == s[:i] + s[i:] // i is an int: 0 <= i <= len(s)
len(s) <= cap(s)
```

Slices can also be initialized like arrays: `var x = []int{2, 3, 5, 7, 11}`

What this does is create an array of length 5 and then create a slice to refer to it.

A slice in memory is in fact *a structure with 3 fields*: a pointer to the underlying array, the length of the slice, and the capacity of the slice. This is illustrated in the following figure, where the slice y is of length 2 and capacity 4.

`y[0] = 3` and `y[1] = 5`. The slice `y[0:4]` contains the elements 3, 5, 7 and 11.

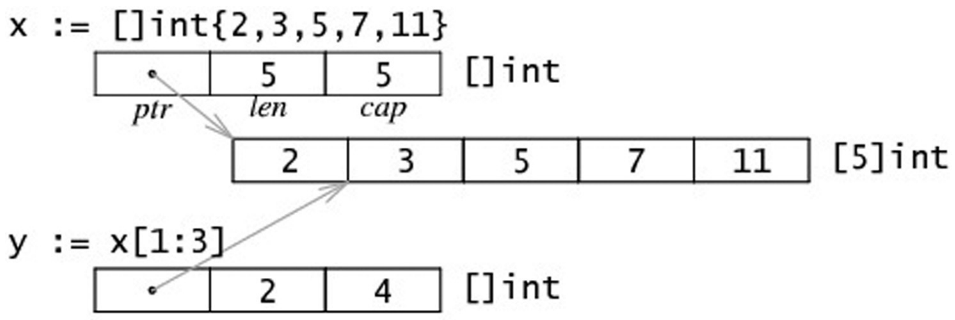


Fig 7.2: Slice in memory

<u>Listing 7.7—array_slices.go shows us the basic usage:</u>

```
package main
import "fmt"

func main() {
        var arr1 [6]int
        var slice1 []int = arr1[2:5] // item at index 5 not included!

        // load the array with integers: 0,1,2,3,4,5
        for i := 0; i < len(arr1); i++ {
               arr1[i] = i
        }

        // print the slice:
        for i := 0; i < len(slice1); i++ {
               fmt.Printf("Slice at %d is %d\n", i, slice1[i])
        }

        fmt.Printf("The length of arr1 is %d\n", len(arr1))
        fmt.Printf("The length of slice1 is %d\n", len(slice1))
        fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))

// grow the slice:

        slice1 = slice1[0:4]
        for i := 0; i < len(slice1); i++ {
               fmt.Printf("Slice at %d is %d\n", i, slice1[i])
        }
        fmt.Printf("The length of slice1 is %d\n", len(slice1))
        fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))

// grow the slice beyond capacity:
// slice1 = slice1[0:7 ] // panic: runtime error: slice bounds out of range
}
```

Output:

```
Slice at 0 is 2
Slice at 1 is 3
Slice at 2 is 4
The length of arr1 is 6
The length of slice1 is 3
The capacity of slice1 is 4
Slice at 0 is 2
Slice at 1 is 3
Slice at 2 is 4
Slice at 3 is 5
The length of slice1 is 4
The capacity of slice1 is 4
```

If s2 is a slice, then you can move the slice forward by one with `s2 = s2[1:]`, but the end is not moved. Slices can only move forward: `s2 = s2[-1:]` results in a compile error.

Slices cannot be re-sliced below zero to access earlier elements in the array.

!! Never use a pointer to a slice. A slice is already a reference type, so it is a pointer !!

Question 7.2: Given the slice of bytes b := `[]byte{'g', 'o', 'l', 'a', 'n', 'g'}`
What are: `b[1:4], b[:2], b[2:] and b[:] ?`

7.2.2 Passing a slice to a function

If you have a function which must operate on an array, you probably always want to declare the formal parameter to be a slice. When you call the function, slice the array to create (efficiently) a slice reference and pass that. For example, here is a function that sums all elements in an array:

```
func sum(a []int) int {
        s := 0
        for i := 0; i < len(a); i++ {
                s += a[i]
        }
        return s
}

func main {
        var arr = [5]int{0,1,2,3,4}
        sum(arr[:])
}
```

7.2.3 Creating a slice with make()

Often the underlying array is not yet defined, we can then make the slice together with the array using the `make( )` function: `var slice1 []type = make([]type, len)`
which can be shortened to: `slice1 := make([]type, len)`
where `len` is the length of the array and also the initial length of the slice.
So for the slice s2 made with: `s2 := make([]int, 10)`
the following is true: `cap(s2) == len(s2) == 10`

`make` takes 2 parameters: the type to be created, and the number of items in the slice.

If you want slice1 not to occupy the whole array (with length `cap`) from the start, but only a number len of items, use the form: `slice1 := make([]type, len, cap)`
make has the signature: `func make([]T, len, cap)` []T with optional parameter cap.
So the following statements result in the same slice:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

The following figure depicts the making of a slice in memory:

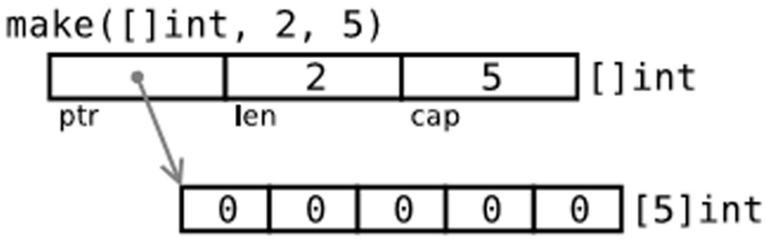


Fig 7.2: Slice in memory

Example: Listing 7.8—make_slice.go:

```
package main
import "fmt"

func main() {
         var slice1 []int = make([]int, 10)
         // load the array/slice:
         for i := 0; i < len(slice1); i++ {
         slice1[i] = 5 * i
         }
         // print the slice:
         for i := 0; i < len(slice1); i++ {
         fmt.Printf("Slice at %d is %d\n", i, slice1[i])
         }
         fmt.Printf("\nThe length of slice1 is %d\n", len(slice1))
         fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))
}
```

Output:

```
Slice at 0 is 0
Slice at 1 is 5
Slice at 2 is 10
Slice at 3 is 15
Slice at 4 is 20
Slice at 5 is 25
Slice at 6 is 30
Slice at 7 is 35
Slice at 8 is 40
Slice at 9 is 45
The length of slice1 is 10
```

The capacity of slice1 is 10

Because strings are in essence immutable arrays of bytes, they can be sliced too.

Exercise 7.4: `fibonacci_funcarray.go`: Starting from solution Ex 7.3, make a version in which main calls a function with parameter the number of terms in the series. The function returns a slice with the Fibonacci numbers up to that number.

7.2.4 Difference between new() and make()

This is often confusing at first sight: both allocate memory on the heap, but they do different things and apply to different types.

- new(T) allocates zeroed storage for a new item of type T and returns its address, a value of type *T: it returns a pointer to a newly allocated zero value of type T, ready for use; it applies to value types like arrays and structs (see Chapter 10); it is equivalent to &T{ }
- make(T) returns an initialized value of type T; it applies only to the 3 built-in reference types: slices, maps and channels (see chapters 8 and 13).

In other words, new allocates; make initializes; the following figure illustrates this difference:

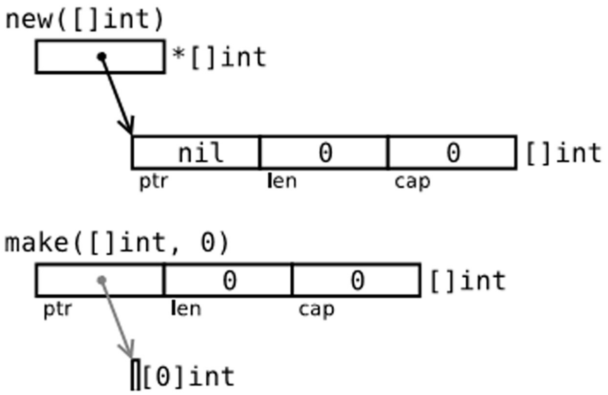


Fig 7.3: Difference between new() and make()

In the first drawing in fig 7.3:

```
        var p *[]int = new([]int)    // *p == nil; with len and cap 0
or      p := new([]int)
```

which is only rarely useful.

In the lower figure: `p := make([]int, 0)`
our slice is initialized, but here points to an empty array.

Both these statements aren't very useful, the following is:
`var v []int = make([]int, 10, 50)`
or `v := make([]int, 10, 50)`

This allocates an array of 50 ints and then creates a slice v with length 10 and capacity 50 pointing to the first 10 elements of the array.

Question 7.3: Given `s := make([]byte, 5)`, what is `len(s)` and `cap(s)` ?
`s = s[2:4]`, what is now `len(s)` and `cap(s)` ?

Question 7.4: Suppose `s1 := []byte{'p', 'o', 'e', 'm'} and s2 := d[2:]`
What is the value of s2 ?
We do: `s2[1] == 't'`, what is now the value of s1 and s2 ?

7.2.5 Multidimensional slices

Like arrays, slices are always 1-dimensional but may be composed to construct higher-dimensional objects. With slices of slices (or arrays of slices), the lengths may vary dynamically, so Go's multidimensional slices can be *jagged*. Moreover, the inner slices must be allocated individually (with `make`).

7.2.6 The bytes package

Slices of bytes are so common that Go has a package `bytes` with manipulation functions for that kind of type.

It is very analogous to the strings package (see. § 4.7). Moreover it contains a very handy type Buffer:

```
import "bytes"
type Buffer struct {
   ...
}
```

which is a variable-sized buffer of bytes with Read and Write methods, because reading and writing an unknown number of bytes is best done buffered.

A Buffer can be created as a value as in: `var buffer bytes.Buffer`
or as a pointer with new as in: `var r *bytes.Buffer = new(bytes.Buffer)`
or created with the function: `func NewBuffer(buf []byte) *Buffer`
that creates and initializes a new Buffer using buf as its initial contents; it is best to use `NewBuffer` only to read from buf.

Concatenating strings by using a buffer:

This works analogous to Java's StringBuilder class.

Make a buffer, append each string s in it with the `buffer.WriteString(s)` method, and convert at the end back to a string with `buffer.String()`, as in the following code snippet:

```
var buffer bytes.Buffer
for {
    if s, ok := getNextString(); ok { //method getNextString() not shown here
        buffer.WriteString(s)
    } else {
        break
    }
}
fmt.Print(buffer.String(), "\n")
```

This method is much more memory and CPU-efficient than +=, especially if the number of strings to concatenate is large.

EXERCISES:

Exercise 7.5: Given a slice sl we want to append a []byte data.
Write a function `Append(slice, data[]byte) []byte` which lets sl grow if it is not big enough to accommodate data.

Exercise 7.6: Split a buffer buf into 2 slices: the header is the first n bytes, the tail is the rest; use 1 line of code.

7.3 For range construct

This construct can be applied to arrays and slices:

```
for ix, value := range slice1 {
        …
}
```

The first return value ix is the index in the array or slice, the second is the value at that index; they are local variables only known in the body of the for-statement, so value is a copy of the slice item at that index and cannot be used to modify it!

Listing 7.9—slices_forrange.go:

```
package main
import "fmt"

func main() {
         slice1 := make([]int, 4)

         slice1[0] = 1
         slice1[1] = 2
         slice1[2] = 3
         slice1[3] = 4

         for ix, value := range slice1 {
                     fmt.Printf("Slice at %d is: %d\n", ix, value)
         }
}
```

`Listing 7.10—slices_forrange2.go` presents an example with strings, only the important code is shown here:

```
seasons := []string{"Spring","Summer","Autumn","Winter"}

        for ix, season := range seasons {
                 fmt.Printf("Season %d is: %s\n", ix, season)
        }

        var season string
        for _, season = range seasons {
                 fmt.Printf("%s\n", season)
}
```

_ can be used to discard the index.

If you only need the index, you can omit the 2nd variable, like in:

```
for ix := range seasons {
        fmt.Printf("%d ", ix)
```

```
    }
    // Output: 0 1 2 3
```

Use this version if you want to modify `seasons[ix]`

for range with multidimensional slices:

It can be convenient to write the nested for-loops of §7.1.3 as simply counting the rows and numbers of a matrix value, like in (taking the example of Listing 7.5—multidim_array.go):

```
    for row := range screen {
            for column := range screen[0] {
                screen[row][column] = 1
            }
    }
```

Question 7.5: Suppose we have the following slice:
`items := […]int{10, 20, 30, 40, 50}`

a) If we code the following for-loop, what will be the value of items after the loop ?
```
 for _, item := range items {
       item *= 2
 }
```
Try it out if you are not sure.

b) If a) does not work, make a for-loop in which the values are doubled.

Question 7.6: Sum up the contexts in the Go syntax where the ellipsis operator **…** is used.

EXERCISES:

Exercise 7.7: `sum_array.go`:

a) Write a function Sum which has as parameter an array arrF of 4 floating-point numbers, and which returns the sum of all the numbers in the array `sum_array.go`

How would the code have to be modified to use a slice instead of an array ?

The slice-form of the function works for arrays of different lengths!

b) Write a function SumAndAverage which returns these two as unnamed variables of type int and float32.

Exercise 7.8: `min_max.go`:

Write a minSlice function which takes a slice of ints and returns the minimum, and a maxSlice function which takes a slice of ints and returns the maximum.

7.4 Reslicing

We saw that a slice is often made initially smaller than the underlying array, like this:

```
      slice1 := make([]type, start_length, capacity)
```

with `start_length` of the slice and `capacity` the length of the underlying array.

This is useful because now our slice can grow till `capacity`.

Changing the length of the slice is called *reslicing,* it is done e.g. like: `slice1 = slice1[0:end]` where `end` is another end-index (length) than before.

Resizing a slice by 1 can be done as follows: `sl = sl[0:len(sl)+1] // extend length by 1`

A slice can be resized until it occupies the whole underlying array.

This is illustrated in program Listing 7.11—reslicing.go:

```
package main
import "fmt"

func main() {
        slice1 := make([]int, 0, 10)
        // load the slice, cap(slice1) is 10:
        for i := 0; i < cap(slice1); i++ {
               slice1 = slice1[0:i+1] // reslice
               slice1[i] = i
               fmt.Printf("The length of slice is %d\n", len(slice1))
        }
        // print the slice:
        for i := 0; i < len(slice1); i++ {
               fmt.Printf("Slice at %d is %d\n", i, slice1[i])
        }
}
```

Output:

```
The length of slice is 1
The length of slice is 2
The length of slice is 3
The length of slice is 4
The length of slice is 5
The length of slice is 6
The length of slice is 7
The length of slice is 8
The length of slice is 9
The length of slice is 10
Slice at 0 is 0
Slice at 1 is 1
Slice at 2 is 2
Slice at 3 is 3
Slice at 4 is 4
Slice at 5 is 5
Slice at 6 is 6
Slice at 7 is 7
Slice at 8 is 8
Slice at 9 is 9
```

Another example:

```
    var ar = [10]int{0,1,2,3,4,5,6,7,8,9}
    var a = ar[5:7] // reference to subarray {5,6} - len(a) is 2 and cap(a) is 5
```

reslicing a: `a = a[0:4]`

```
    // ref of subarray {5,6,7,8} - len(a) is now 4 but cap(a) is still 5.
```

Question 7.7: 1) If s is a slice, what is the length of this reslice ? `s[n:n]`
2) And what is the length of `s[n:n+1]`?

7.5 Copying and appending slices

To increase the capacity of a slice one must create a new, larger slice and copy the contents of the original slice into it. The following code illustrates the functions copy for copying slices, and `append` for appending new values to a slice.

Listing 7.12—copy_append_slice.go:

```
package main
import "fmt"

func main() {
        sl_from := []int{1,2,3}
        sl_to := make([]int,10)

        n := copy(sl_to, sl_from)
        fmt.Println(sl_to)  // output: [1 2 3 0 0 0 0 0 0 0]
fmt.Printf("Copied %d elements\n", n) // n == 3

        sl3 := []int{1,2,3}
        sl3 = append(sl3, 4, 5, 6)
        fmt.Println(sl3)  // output: [1 2 3 4 5 6]
}
```

```
func append(s[]T, x ...T) []T
```

The function **append** appends zero or more values to a slice s and returns the resulting slice, with the same type as s; the values of course have to be of same type as the element-type T of 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. The append always succeeds, unless the computer runs out of memory.

If you want to append a slice y to a slice x, use the following form to expand the second argument to a list of arguments: `x = append(x, y...)`

Remark: append is good for most purposes, however if you want complete control over the process, you could use a function AppendByte like this:

```
func AppendByte(slice []byte, data ...byte) []byte {
        m := len(slice)
        n := m + len(data)
        if n > cap(slice) { // if necessary, reallocate
               // allocate double what's needed, for future growth.
               newSlice := make([]byte, (n+1)*2)
               copy(newSlice, slice)
               slice = newSlice
```

```
            }
            slice = slice[0:n]
            copy(slice[m:n], data)
            return slice
    }
```

```
func copy(dst, src []T) int
```

The function copy copies slice elements of type T from a source src to a destination dst, overwriting the corresponding elements in dst, and returns the number of elements copied. Source and destination may overlap. The number of arguments copied is the minimum of len(src) and len(dst). When src is a string the element type is byte. If you want to continue working with the variable src, put: `src = dst` after the copy.

Exercise 7.9: `magnify_slice.go`: Given a slice s []int and a factor of type int, enlarge s so that its new length is len(s) * factor.

Exercise 7.10: `filter_slice.go`:
Using a higher order function for filtering a collection:
s is a slice of the first 10 integers. Make a function Filter which accepts s as 1st parameter and a fn func(int) bool as 2nd parameter and returns the slice of the elements of s which fulfil the function fn (make it true). Test this out with fn testing if the integer is even.

Exercise 7.11: `insert_slice.go`:
Make a function InsertStringSlice that inserts a slice into another slice at a certain index.

Exercise 7.12: `remove_slice.go`:
Make a function RemoveStringSlice that removes items from index start to end in a slice.

7.6 Applying strings, arrays and slices

7.6.1 Making a slice of bytes from a string

If s is a string (so in fact an array of bytes) a slice of bytes c can immediately be made with `c:=[]byte(s)`. This can also be done with the copy-function: `copy(dst []byte, src string)`

For-range can also be applied, example Listing 7.13—for_string.go:

```
package main
import "fmt"

func main() {
        s := "\u00ff\u754c"
        for i, c := range s {
                fmt.Printf("%d:%c ", i, c)
        }
}
```

```
Output: 0:ÿ 2:界
```

We see that Unicode-characters take 2 bytes; some characters can even take 3 or 4 bytes. If erroneous UTF-8 is encountered, the character is set to U+FFFD and the index advances by one byte. In the same way the conversion `c:=[]int(s)` is allowed, then each int contains a Unicode code point: every character from the string corresponds to one integer; similarly the conversion to runes can be done with: `r:=[]rune(s)`

The number of characters in a string s is given by `len([]int(s))`, but `utf8.RuneCountInString(s)` is faster (see Ex 4.6).

A string may be appended to a byte slice, like this:

```
var b []byte
var s string
b = append(b, s...)
```

7.6.2 Making a substring of a string

`substr := str[start:end]` takes the substring from str from the byte at index start to the byte at index end—1. Also `str[start:]` is the substring starting from index `start to len(str) - 1`, `str[:end]` is the substring starting from index 0 to `end - 1`.

7.6.3 Memory representation of a string and a slice

A string in memory is in fact a 2 word-structure consisting of a pointer to the string data and the length (see Fig 7.4); the pointer is completely invisible in Go-code; so for all practical purposes we can continue to see a string as a value type, namely its underlying array of characters. A substring (slice) `t = s[2:3] of the string s = "hello"` can be represented in memory as:

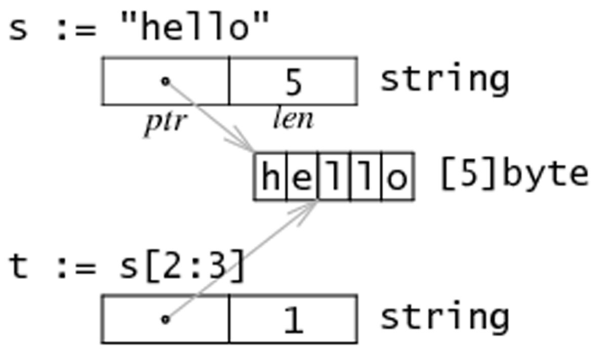


Fig 7.4: String and slice in memory

7.6.4 Changing a character in a string

Strings are immutable. This means when str denotes a string that str[index] cannot be the left side of an assignment: `str[i] = 'D'` where i is a valid index gives the error `cannot assign to str[i]`

To do this you first have to convert the string to an array of bytes, then an array-item of a certain index can be changed, and then the array must be converted back to a new string.

For example, change `"hello" to "cello"`:

```
s:="hello"
c:=[]byte(s)
c[0]='c'
s2:= string(c)      // s2 == "cello"
```

So it is clear that string-extractions are very easy with the slice-syntax.

7.6.5 Comparison function for byte arrays

The following function Compare returns an integer comparing 2 byte arrays lexicographically.

The result is : 0 if a ==b, -1 if a < b, 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
}
```

7.6.6 Searching and sorting slices and arrays

Searching and sorting are very common operations and the standard library provides for these in the package sort. To sort a slice of ints, import the package "sort" and simply call the function `func Ints(a []int) as in sort.Ints(arri)`, where `arri` is the array or slice to be sorted in ascending order. To test if an array is sorted, use `func IntsAreSorted(a []int) bool`, which returns true or false whether or not the array is sorted.

Similarly for float64 elements, you use the function `func Float64s(a []float64)` and for strings there is the function `func Strings(a []string)`.

To search an item in an array or slice, the array must first be sorted (the reason is that the search-functions are implemented with the binary-search algorithm). Then you can use the function func `SearchInts(a []int, n int) int`, which searches for n in the slice a, and returns its index. And of course the equivalent functions for float64s and strings exist also:

```
func SearchFloat64s(a []float64, x float64) int
func SearchStrings(a []string, x string) int
```

Further details can be found in the official information: http://golang.org/pkg/sort/

This is how to use the sort-package functions. In §11.6 we discover the theory behind it, and we implement ourselves the sort-functionality, much like in the package.

7.6.7 Simulating operations with append

The append method from § 7.5 is very versatile and can be used for all kinds of manipulations:

1) Append a slice b to an existing slice a:	`a = append(a, b...)`

2) Copy a slice a to a new slice b:	`b = make([]T, len(a))`
	`copy(b, a)`

3) Delete item at index i:	`a = append(a[:i], a[i+1:]...)`
4) Cut from index i till j out of slice a:	`a = append(a[:i], a[j:]...)`
5) Extend slice a with a new slice of length j:	`a = append(a, make([]T, j)...)`
6) Insert item x at index i:	`a = append(a[:i], append([]T{x}, a[i:]...)...)`
7) Insert a new slice of length j at index i:	`a = append(a[:i], append(make([]T, j), a[i:]...)...)`
8) Insert an existing slice b at index i:	`a = append(a[:i], append(b, a[i:]...)...)`
9) Pop highest element from stack:	`x, a = a[len(a)-1], a[:len(a)-1]`
10) Push an element x on a stack:	`a = append(a, x)`

So to represent a resizable sequence of elements use a slice and the append-operation.

A slice is often called a *vector* in a more mathematical context. If this makes code clearer you can define a vector alias for the kind of slice you need, Listing 10.11 (`method2.go`) shows a simple example.

If you need something more elaborated go and have a look at the following packages written by Eleanor McHugh: http://github.com/feyeleanor/slices, http://github.com/feyeleanor/chain, and http://github.com/feyeleanor/lists.

7.6.8 Slices and garbage collection

A slice points to the underlying array; this array could potentially be very much bigger than the slice as in the following example. As long as the slice is referred to, the full array will be kept in memory until it is no longer referenced. Occasionally this can cause the program to hold all the data in memory when only a small piece of it is needed.

Example: this `FindDigits` function loads a file into memory and searches it for the first group of consecutive numeric digits, returning them as a new slice.

```
var digitRegexp = regexp.MustCompile("[0-9]+")

func FindDigits(filename string) []byte {
        b, _ := ioutil.ReadFile(filename)
        return digitRegexp.Find(b)
}
```

This code works as described, but the returned [ ]byte points into an array containing the entire file. Since the slice references the original array, as long as the slice is kept around the garbage collector can't release the array; the few useful bytes of the file keep the entire contents in memory.

To fix this problem one can copy the interesting data to a new slice before returning it:

```
func FindDigits(filename string) []byte {
        b, _ := ioutil.ReadFile(filename)
        b = digitRegexp.Find(b)
        c := make([]byte, len(b))
        copy(c, b)
        return c
}
```

<u>EXERCISES</u>:

<u>Exercise 7.12</u>: `string_split.go`:
Write a function Split with parameters a string to split and the position to split, which returns the two substrings.

<u>Exercise 7.13</u>: `string_split2.go`:
If `str` is a string, what then is `str[len(str)/2:] + str[:len(str)/2]`? Test it

<u>Exercise 7.14</u>: `string_reverse.go`:
Write a program that reverses a string, so "Google" is printed as " elgooG".
(Hint: use a slice of bytes and conversions.)
If you coded a solution with two slices, try a variant which uses only one (Hint: use swapping)
If you want to be able to reverse Unicode-strings: use [ ]int !

Exercise 7.15: `uniq.go`
Write a program that traverses an array of characters, and copies them to another array only if the character is different from that which precedes it.

Exercise 7.16: `bubblesort.go`
Sort a slice of ints through a function which implements the Bubblesort-algorithm (look up the definition of this algorithm at http://en.wikipedia.org/wiki/Bubble_sort)

Exercise 7.17: `map_function.go`
In functional languages a map-function is a function which takes a function and a list as arguments, and its result is a list where the argument-function is applied to each element of the list, formally:
map (F(), (e1,e2, . . . ,en)) = (F(e1), F(e2), … F(en))
Write such a function `mapFunc` which takes as arguments:

- a (lambda) function that multiplies an int by 10
- a list of ints

and returns the list of all ints multiplied by 10.

Chapter 8—Maps

Maps are a special kind of data structure: an unordered collection of pairs of items, where one element of the pair is the *key*, and the other element, associated with the key, is the data or the *value*, hence they are also called *associative arrays* or *dictionaries.* They are ideal for looking up values fast: given the key, the corresponding value can be retrieved very quickly.

This structure exists in many other programming languages under other names such as Dictionary (dict in Python), hash, HashTable, etc.

8.1 Declaration, initialization and make

8.1.1 Concept

A map is a reference type and declared in general as: `var map1 map[keytype]valuetype`
e.g.: `var map1 map[string]int`

(A space is allowed between [keytype] valuetype, but gofmt removes this)

The length of the map doesn't have to be known at declaration: a map can grow dynamically.

The value of an uninitialized map is nil.

The *key* type can be any type for which the operations == and != are defined, like string, int, float. So arrays, slices and structs cannot be used as key type, but pointers and interface types can. One way to use structs as a key is to provide them with a Key() or Hash() method, so that a unique numeric or string key can be calculated from the struct's fields.

The *value* type can be any type; by using the empty interface as type (see § 11.9), we could store any value, but when using that value we would first have to do a type assertion (see § 11.3).

Maps are cheap to pass to a function: 4 bytes on a 32 bit machine, 8 bytes on a 64 bit machine, no matter how much data they hold. Looking up a value in a map by key is fast, much faster than a linear search, but still around 100x slower than direct indexing in an array or slice; so if performance is very important try to solve the problem with slices.

A map with function values can also be used as a branching structure (see chapter 5): the keys are used to select the branch which is a function that is executed.

If key1 is a key value of map map1, then map1[key1] is the value associated with key1, just like the array-index notation (an array could be considered as a simple form of a map, where the keys are integers starting from 0).

The value associated with key1 can be set to (or if already present changed to) val1 through the assignment: `map1[key1] = val1`

The assignment `v:= map1[key1]` stores in v the value corresponding to key1; if `key1` is not present in the map, then v becomes the zero-value for the value type of `map1`.

As usual `len(map1)` gives the number of pairs in the map, which can grow or diminish because map-pairs may be added or removed during runtime.

In `Listing 8.1—make_maps.go` a number of maps are shown:

```
package main
import "fmt"

func main() {
        var mapLit map[string]int
        //var mapCreated map[string]float32
        var mapAssigned map[string]int

        mapLit = map[string]int{"one": 1, "two": 2}
        mapCreated := make(map[string]float32)
        mapAssigned = mapLit

        mapCreated["key1"] = 4.5
        mapCreated["key2"] = 3.14159
        mapAssigned["two"] = 3

        fmt.Printf("Map literal at \"one\" is: %d\n", mapLit["one"])
        fmt.Printf("Map created at \"key2\" is: %f\n", mapCreated["key2"])
        fmt.Printf("Map assigned at \"two\" is: %d\n", mapLit["two"])
        fmt.Printf("Map literal at \"ten\" is: %d\n", mapLit["ten"])
}
```

Output:

```
Map literal at "one" is: 1
Map created at "key2" is: 3.141590
Map assigned at "two" is: 3
Map literal at "ten" is: 0
```

`mapLit` illustrates the use of *map literals*: a map can be initialized with the `{key1: val1, key2: val2}` notation, just like arrays and structs.

Maps are *reference types:* memory is allocated with the **`make`**-function:

Initialization of a map:	`var map1[keytype]valuetype = make(map[keytype]valuetype)`
or shorter with:	`map1 := make(map[keytype]valuetype)`
`mapCreated` is made in this way:	`mapCreated := make(map[string]float)`
which is equivalent to:	`mapCreated := map[string]float{}`

Also `mapAssigned` is a reference to the same value as mapLit, changing mapAssigned also changes `mapLit` as the program output shows.
!! Don't use new, always use make with maps !!

<u>Remark</u>: If you by mistake allocate a reference object with new(), you receive a pointer to a nil reference, equivalent to declaring an uninitialized variable and taking its address:

```
mapCreated := new(map[string]float)
```

Then we get in the statement: `mapCreated["key1"] = 4.5`
the compiler error: `invalid operation: mapCreated["key1"] (index of type *map[string]float).`

To demonstrate that the value can be any type, here is an example of a map which has a func() int as its value: <u>`Listing 8.2—map_func.go`</u>:

```
package main
import "fmt"

f unc main() {
        mf := map[int]func() int{
               1: func() int { return 10 },
               2: func() int { return 20 },
               5: func() int { return 50 },
        }
        fmt.Println(mf)
}
```

The output is: map[1:0x10903be0 5:0x10903ba0 2:0x10903bc0]: the integers are mapped to addresses of functions.

8.1.2 Map capacity

Unlike arrays, maps dynamically grow to accommodate new key-values that are added, they have no fixed or maximum size. But you can optionally indicate an initial *capacity* cap for the map, as

in `make(map[keytype]valuetype, cap)`

e.g.: `map2 := make(map[string]float, 100)`

When the map has grown to its capacity, and a new key-value is added, then the size of the map will automatically increase by 1. So for large maps or maps that grow a lot, it is better for performance to specify an initial capacity, even if this is only known approximately.

Here is a more concrete example of a map: map the name of a musical note to its frequency in Hz (measuring frequencies in Hz for equal-tempered scale (A4 = 440Hz)):

```
noteFrequency := map[string]float32{
  “C0”: 16.35, “D0”: 18.35, “E0”: 20.60, “F0”: 21.83,
  “G0”: 24.50, “A0”: 27.50, “B0”: 30.87, “A4”: 440}
```

8.1.3 Slices as map values

When one key has only one associated value, the value can be a primitive type. What if a key corresponds to many values ? For example, when we have to work with all the processes on a Unix-machine, where a parent process as key (process-id pid is an int value) can have many child processes (represented as a slice of ints with their pid’s as items). This can be elegantly solved by defining the value type as a [ ]int or a slice of another type.

Here are some examples defining such maps:

```
mp1 := make(map[int][]int)
mp2 := make(map[int]*[]int)
```

8.2 Testing if a key-value item exists in a map—Deleting an element

Testing the existence of key1 in map1:

We saw in § 8.1 that `val1 = map1[key1]` returns us the value val1 associated with key1. If key1 does not exist in the map, val1 becomes the zero-value for the value’s type.

But this is ambiguous: now we can’t distinguish between this case, or the case where key1 does exist and its value is the zero-value!

In order to test this, we can use the following *comma ok* form: `val1, isPresent = map1[key1]`

`isPresent` returns a Boolean value: if key1 exists in map1, val1 will contain the value for key1, and `isPresent` will be true; if key1 does not exist in map1, val1 will contain the zero-value for its type, and `isPresent` will be false.

If you just want to check for the presence of a key and don't care about its value, you could write:

```
_, ok := map1[key1] // ok == true if key1 is present, false otherwise
```

Or combined with an if:

```
if _, ok := map1[key1]; ok {
        // …
}
```

Deleting an element with key1 from map1:

This is done with: `delete(map1, key1)`

When key1 does not exist this statement doesn't produce an error.

Both techniques are illustrated in the Listing 8.4—map_testelement.go:

```
package main
import "fmt"

func main() {
        var value int
        var isPresent bool

        map1 := make(map[string]int)
        map1["New Delhi"] = 55
        map1["Bejing"] = 20
        map1["Washington"] = 25
        value, isPresent = map1["Bejing"]
        if isPresent {
               fmt.Printf("The value of \"Bejing\" in map1 is: %d\n", value)
        } else {
               fmt.Println("map1 does not contain Bejing")
        }

        value, isPresent = map1["Paris"]
        fmt.Printf("Is \"Paris\" in map1 ?: %t\n", isPresent)
        fmt.Printf("Value is: %d\n", value)

        // delete an item:
        delete(map1, "Washington")
        value, isPresent = map1["Washington"]
        if isPresent {
               fmt.Printf("The value of \"Washington\" in map1 is: %d\n", value)
        } else {
        fmt.Println("map1 does not contain Washington")
        }
}
```

Output:
```
The value of "Bejing" in map1 is: 20
Is "Paris" in map1 ?: false
Value is: 0
map1 does not contain Washington
```

8.3 The for range construct

This construct can also be applied to maps:

```
for key, value := range map1 {
        …
}
```

The first return value key is the key of the map, thwe second is the value for that key; they are local variables only known in the body of the for-statement. The first element in a map iteration is chosen at random.If you are only interested in the values,
use the form:

```
for _, value := range map1 {
               …
}
```

To get only the keys, you can use:

```
for key := range map1 {
       fmt.Printf("key is: %d\n", key)
}
```

Example: Listing 8.5—maps_forrange.go:

```
package main
import "fmt"

func main() {
        map1 := make(map[int]float32)
        map1[1] = 1.0
        map1[2] = 2.0
        map1[3] = 3.0
        map1[4] = 4.0
for key, value := range map1 {
fmt.Printf("key is: %d - value is: %f\n", key, value)
        }
}
```

Output:

```
key is: 3 - value is: 3.000000
key is: 1 - value is: 1.000000
key is: 4 - value is: 4.000000
key is: 2 - value is: 2.000000
```

We see that a map is *not key-ordered*, neither is it sorted on the values.

Question 8.1: What is the output of the following code snippet:

```
capitals  :=  map[string]  string  {"France":"Paris",  "Italy":"Rome",
"Japan":"Tokyo" }
for key := range capitals {
        fmt.Println("Map item: Capital of", key, "is", capitals[key])
}
```

Exercise 8.1: `map_days.go`

Make a map to hold together the number of the day in the week (1 -> 7) with its name.

Print them out and test for the presence of tuesday and hollyday.

8.4 A slice of maps

Suppose we want to make a slice of maps, then we must use make() 2 times, first for the slice, then for each of the map-elements of the slice, like in Listing 8.3

But take care to use the map-items in the slice by index, as in version A. The item value in version B is a copy of the map-value, so the real map-variables don't get initialized.

Listing 8.3—slice_maps.go:

```
package main
import (
        "fmt"
)

func main() {
// Version A:
         items := make([]map[int]int, 5)
         for i := range items {
                items[i] = make(map[int]int, 1)
                items[i][1] = 2
         }
         fmt.Printf("Version A: Value of items: %v\n", items)
// Version B: NOT GOOD!
         items2 := make([]map[int]int, 5)
         for _, item := range items2 {
         item = make(map[int]int, 1)
                // item is only a copy of the slice element.
         item[1] = 2
                // This 'item' will be lost on the next iteration.
         }
         fmt.Printf("Version B: Value of items: %v\n", items2)
}
/* Output:
Version A: Value of items: [map[1:2] map[1:2] map[1:2] map[1:2] map[1:2]]
Version B: Value of items: [map[] map[] map[] map[] map[]]
*/
```

8.5 Sorting a map

By default a map isw not sorted, not even on the value of its keys (see listing 8.5)

If you want a sorted map, copy the keys (or values) to a slice, sort the slice (using the sort package, see § 7.6.6), and then print out the keys and/or values using a for-range on the slice.

This is illustrated in the following program:

Listing 8.6—sort_map.go:

```
// the telephone alphabet:
package main
import (
        "fmt"
        "sort"
)

var (
        barVal = map[string]int{"alpha": 34, "bravo": 56, "charlie": 23,
        "delta": 87, "echo": 56, "foxtrot": 12, "golf": 34, "hotel": 16,
        "indio": 87, "juliet": 65, "kilo": 43, "lima": 98}
)

func main() {
        fmt.Println("unsorted:")
        for k, v := range barVal {
               fmt.Printf("Key: %v, Value: %v / ", k, v)
        }
        keys := make([]string, len(barVal))
        i := 0
        for k, _ := range barVal {
               keys[i] = k
               i++
        }
        sort.Strings(keys)
        fmt.Println()
        fmt.Println("sorted:")
        for _, k := range keys {
               fmt.Printf("Key: %v, Value: %v / ", k, barVal[k])
        }
}
```

```
/* Output:

unsorted:

Key: indio, Value: 87 / Key: echo, Value: 56 / Key: juliet, Value: 65 / Key: charlie,
Value: 23 / Key: hotel, Value: 16 / Key: lima, Value: 98 / Key: bravo, Value: 56 / Key:
alpha, Value: 34 / Key: kilo, Value: 43 / Key: delta, Value: 87 / Key: golf, Value: 34 /
Key: foxtrot, Value: 12 /

sorted:

Key: alpha, Value: 34 / Key: bravo, Value: 56 / Key: charlie, Value: 23 / Key: delta,
Value: 87 / Key: echo, Value: 56 / Key: foxtrot, Value: 12 / Key: golf, Value: 34 / Key:
```

```
hotel, Value: 16 / Key: indio, Value: 87 / Key: juliet, Value: 65 / Key: kilo, Value: 43
/ Key: lima, Value: 98 /
*/
```

But if what you want is a sorted list you better use a slice of structs, which is more efficient:

```
type struct {
    key string
    value int
}
```

8.6 Inverting a map

By this we mean switching the values and keys. If the value type of a map is acceptable as a key type, and the map values are unique, this can be done easily:

Listing 8.7—invert_map.go:

```
package main
import (
        “fmt”
)

var (
        barVal = map[string]int{“alpha”: 34, “bravo”: 56, “charlie”: 23,
        “delta”: 87, “echo”: 56, “foxtrot”: 12, “golf”: 34, “hotel”: 16,
        “indio”: 87, “juliet”: 65, “kilo”: 43, “lima”: 98}
)

func main() {
        invMap := make(map[int]string, len(barVal))
        for k, v := range barVal {
               invMap[v] = k
        }
        fmt.Println(“inverted:”)
        for k, v := range invMap {
               fmt.Printf(“Key: %v, Value: %v / “, k, v)
        }
}
/* Output:
inverted:
```

```
Key: 12, Value: foxtrot / Key: 16, Value: hotel / Key: 87, Value: delta / Key: 23,
Value: charlie /
Key: 65, Value: juliet / Key: 43, Value: kilo / Key: 56, Value: bravo / Key: 98,
Value: lima /
Key: 34, Value: golf /
*/
```

This goes wrong of course when the original value items are not unique; in that case no error occurs, but the processing of the inverted map is simply stopped when a nonunique key is encountered, and it will most probably not contain all pairs from the original map! A solution is to carefully test for the uniqueness and making use of a multi-valued map, in this case of type `map[int][]string`

Exercise 8.2: `map_drinks.go`

Construct a collection which maps English names of drinks to the French (or your native language) translations; print first only the drinks available, and then print both (the name and the translation). Then produce the same output, but this time the English names of the drinks must be sorted.

Chapter 9—Packages

A The standard library

9.1 Overview of the standard library.

The Go-distribution contains over 150 standard built-in packages for common functionality, like fmt, os, . . . , as a whole designated as the *standard library*, for the most part (except some low level routines) built in Go itself. They are documented at: http://golang.org/pkg/

In the examples and exercises throughout the book, we use the packages of the standard library; for a quick index and practical examples, see: References in the text to Go—packages p. 350 . Here we will discuss their general purpose of a number of them grouped by function, we will not go into details about their inner structure.

`unsafe:` contains commands to step out of the Go type-safety, not needed in normal programs; can be useful when interfacing with C/C++

`syscall—os—os/exec:`

`os:`	gives us a platform-independent interface to operating-system functionality; its design is Unix-like; it hides the differences between various operating systems to give a consistent view of files and other OS-objects.
`os/exec:`	gives you the possibility to run external OS commands and programs.
`syscall:`	is the low-level, external package, which provides a primitive interface to the underlying OS 's calls.

As an example of its power, here is a Go-program to make a Linux-machine reboot (start with `sudo ./6.out`):

Listing 9.1 reboot.go:

```
package main
import (
   “syscall”
)
const LINUX_REBOOT_MAGIC1 uintptr = 0xfee1dead
const LINUX_REBOOT_MAGIC2 uintptr = 672274793
const LINUX_REBOOT_CMD_RESTART uintptr = 0x1234567

func main() {
  syscall.Syscall(syscall.SYS_REBOOT,
    LINUX_REBOOT_MAGIC1,
    LINUX_REBOOT_MAGIC2,
    LINUX_REBOOT_CMD_RESTART)
}
```

`archive/tar and /zip–compress:`	functionality for (de)compressing files.

`fmt–io–bufio–path/filepath–flag:`

`fmt`	contains functionality for formatted input-output.
`io:`	provides basic input-output functionality, mostly as a wrapper around os-functions.
`bufio:`	wraps around io to give buffered input-output functionality.
`path/filepath:`	routines for manipulating filename paths targeted at the OS used.
`flag:`	functionality to work with command-line arguments

`strings–strconv–unicode–regexp–bytes:`

`strings:`	for working with strings.
`strconv:`	converting them to basic data types.
`unicode:`	special functions for Unicode characters.
`regexp:`	provides pattern-search functionality in complex strings.
`bytes:`	contains functions for the manipulation of byte slices.
`index/suffixarray:`	for very fast searching in strings.

`math–math/cmath–math/big–math/rand–sort:`

`math:`	basic mathematical constands and functions.
`math/cmath:`	manipulations with complx numbers.
`math/rand:`	contains pseudo-random number generators.
`sort`	functionality for sorting arrays and user-defined collections.
`math/big:`	multiprecision arithmetic for working with arbitrarily large integers and rational numbers.

`container: /list–ring–heap:` implement containers for manipulating collections:
`list:` to work with doubly linked lists.
For example, to iterate over a list(where l is a *List):

```
for e := l.Front(); e != nil; e = e.Next() {
      // do something with e.Value
}
```

`ring:` to work with circular lists.

`time–log:`
`time:` basic functionality for working with times and dates.
`log:` functionality for logging information in a running program; we'll use it throughout examples in the following chapters.

`encoding/json–encoding/xml–text/template:`
`encoding/json:` implements the functions for reading/decoding as well as writing/encoding data in JSON forwmat
`encoding/xml:` simple XML 1.0 parser; for examples of json and xml: see § 12.9/10
`text/template:` to make data-driven templates which can generate textual output mixed with data, like HTML (see § 15.7)

`net–net/http–html`: (see chapter 15)
`net:` basic functions for working with network-data
`http:` functionality for parsing HTTP requests/replies, provides an extensible HTTP server and a basic client.
`html:` parser for HTML5.

`crypto–encoding–hash–…:` a multitude of packages for en- and decrypting data.

`runtime–reflect:`

`runtime:` operations for interacting with the Go-runtime, such as the garbage collection and goroutines.

`reflect:` implements runtime introspection, allowing a program to manipulate variables with arbitrary types.

The package `exp` contains 'experimental' packages, that is new packages being build. When they have matured enough they become independent packages at the next stable release. If a previous version existed, then this is moved to package old, in fact a recycle bin for deprecated packages. However the Go 1 release does not contain the old and `exp` packages.

`Exercise 9.1:` `dlinked_list.go`

Use the package container/list to construct a double linked list, put the values 101,102,103 in it and make a printout of the list.

`Exercise 9.2:` `size_int.go`

Use a function from the unsafe package to test the size of an int variable on your computer.

9.2 The regexp package.

For info about regular expressions and the syntax it uses, see: http://en.wikipedia.org/wiki/Regular_expression

In the following program we want to search a string pattern pat in a string searchIn.

Testing if the pattern occurs is easy, just use Match: `ok, _ := regexp.Match(pat, []byte(searchIn))`

where ok will be true or false, or use MatchString: `ok, _ := regexp.MatchString(pat, searchIn)`

For more functionality, you must first make a (pointer to a) Regexp object from the pattern; this is done through the Compile function. Then we have at our disposal a whole number of Match-, Find- and Replace-functions.

`Listing 9.2—pattern.go:`

```
Package main
import (
        "fmt"
        "regexp"
        "strconv"
)
func main() {
        // string to search
        searchIn := "John: 2578.34 William: 4567.23 Steve: 5632.18"
        pat := "[0-9]+.[0-9]+"  // pattern to search for in searchIn

        f := func (s string) string {
               v, _ := strconv.ParseFloat(s, 32)
```

```
                return strconv.FormatFloat(v * 2, ‘f’, 2, 32)
        }

        if ok, _ := regexp.Match(pat, []byte(searchIn)); ok {
                fmt.Println(“Match found!”)
        }

        re, _ := regexp.Compile(pat)
        // replace pat with “##.#”
        str := re.ReplaceAllString(searchIn, “##.#”)
        fmt.Println(str)
        // using a function :
        str2 := re.ReplaceAllStringFunc(searchIn, f)
        fmt.Println(str2)
}
/* Output:
Match found!
John: ##.# William: ##.# Steve: ##.#
John: 5156.68 William: 9134.46 Steve: 11264.36
*/
```

The Compile function also returns an error, which we have safely ignored here because we have entered the pattern ourselves and know that it is a valid regular expression. Should the expression be entered by the user or taken from a data source, then it is necessary to check this parsing error. In this example we could also have used the function **`MustCompile`** which is like Compile but panics (stopping the program with an error message, see § 13.2) when the pattern is not a valid regular expression.

9.3 Locking and the sync package.

In more complex programs different parts of the application may execute simultaneously or *concurrently* as this is technically called, usually by executing each part of the program on a different *thread* of the operating system. When these different parts share and work with the same variables, most likely problems occur: the order in which these shared variables are updated cannot be predicted and hence also their values are unpredictable! (this is commonly called a *race condition:* the threads race for the updates of the variables). This is of course intolerable in a correct program, so how do we solve this issue?

The classic approach is to let only one thread at a time change the shared variable: the code in which the variable is changed (called the *critical section*) is locked when a thread starts executing, so no other thread can start with it. Only when the executing thread has finished the section an unlock occurs, so another thread can access it.

In particular the map type which we have studied in this chapter does not contain any internal locking to achieve this effect (this is left out for performance reasons); it is said that the map type is not *thread-safe*. So when concurrent accesses happen to a shared map datastructure, corrupted, that means not correct, map data can result.

In Go this kind of locking is realized with the Mutex variable of the sync package . sync comes from synchronized, here meaning the threads are synchronized to update the variable(s) in an orderly fashion.

A `sync.Mutex` is a *mutual exclusion lock*: it serves to guard the entrance to the critical section of code so that only one thread can enter the critical section at one time.

Suppose Info is a shared memory variable which must be guarded, then a typical technique is to include a mutex in it, like:

```
import  “sync”
type Info struct {
        mu    sync.Mutex
        // … other fields, e.g.:
        Str    string
}
```

A function which has to change this variable could then be written like:

```
func Update(info *Info) {
        info.mu.Lock()
        // critical section:
        info.Str = // new value
        // end critical section
        info.mu.Unlock()
}
```

An example of its usefulness is a shared buffer, which has to be locked before updating: the SyncedBuffer:

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

The sync package also has a **`RWMutex`**: a lock which allows many reader threads by using **`RLock()`**, but only one writer thread. If **`Lock()`** is used, the section is locked for writing as with the normal Mutex. It also contains a handy function `once.Do(call)` where once is a variabole of type **Once**, which guarantees that the function call will only be invoked 1 time, regardless of how many once. Do(call) ‘s there are.

For relatively simple situations using locking through the sync package so that only one thread at a time can access the variable or the map will remedy this problem. If this slows done the program too much or causes other problems, the solution must be rethought with goroutines and channels in mind: this is the technology proposed by Go for writing concurrent applications. We will go deeply into this in chapter 14, in § 14.7 we will also compare the two approaches.

9.4 Accurate computations and the big package.

We know that programmatically performed computations are sometimes not accurate. If you use Go’s `float64` type in floating point numbers computation, the results are accurate to about 15 decimal digits, enough for most tasks. When computing with very big whole numbers the range of the types int64 or uint64 might also be too small. In that case `float32` or `float64` can be used if accuracy is not a concern, but if it is we cannot use floating-point numbers because they are only represented by approximation in memory.

For performing perfectly accurate computations with integer values Go provides the `big` package, contained in the `math` package: `big.Int` for integers and `big.Rat` for rational numbers (these are numbers than can be represented by a fraction like 2/5 or 3.1416, but not irrational numbers like e or π) . These types can hold an arbitrary number of digits, only limited by the machine’s available memory. The downside is the bigger memory usage and the processing overhead: they are a lot slower to process than built-in integers.

A big integer is constructed with the function `big.NewInt(n)`, where n is an int64; a big rational number with `big.NewRat(n, d)` where both n (the numerator) and d (the denominator) are of type int64. Because Go does not support operator overloading all the methods of the big types have names, like Add() and Mul(). They are methods (see § 10.6) acting on the integer or rational as receiver, and in most cases they modify their receiver and also return the receiver as result, so that the operations can be chained and memory saved, because no temporary big.Int variables have to be created to hold intermediate results.

We see this in action in listing 9.3:

Listing 9.3—big.go:

```
package main
import (
       "fmt"
       "math"
       "math/big"
)

func main() {
// Here are some calculations with bigInts:
       im := big.NewInt(math.MaxInt64)
       in := im
       io := big.NewInt(1956)
       ip := big.NewInt(1)
       ip.Mul(im, in).Add(ip, im).Div(ip, io)
       fmt.Printf("Big Int: %v\n", ip)
// Here are some calculations with bigInts:
       rm := big.NewRat(math.MaxInt64, 1956)
       rn := big.NewRat(-1956, math.MaxInt64)
       ro := big.NewRat(19, 56)
       rp := big.NewRat(1111, 2222)
       rq := big.NewRat(1, 1)
       rq.Mul(rm, rn).Add(rq, ro).Mul(rq, rp)
       fmt.Printf("Big Rat: %v\n", rq)
}
/* Output:
Big Int: 43492122561469640008497075573153004
Big Rat: -37/112
*/
```

B Custom and external packages: use, build, test, document, install

9.5 Custom packages and visibility

Packages are the primary means in Go of organizing and compiling code. A lot of basic information about them has already been given in chapter 4 § 4.2, most notably the Visibility rule. Now we will see concrete examples of the use of packages that you write yourself. In the next sections we will review some packages of the standard library. By custom packages we mean self-written packages or otherwise external to the standard library.

When writing your own packages, use short, single-word, lowercase names without _ for the filename(s). Here is a simple example as to how packages can find each other and how the Visibility rule works:

Our current directory (code examples\chapter 9) contains the program `package_test.go`. It uses code from a program `pack1.go` which is contained in the custom package `pack1`. This program (together with its compiled-and- linked archived form `pack1.a`) resides in a subdirectory pack1 of our current directory; so the linker links the object code of the package(s) together with the object code of the main program.

`Listing 9.4—pack1.go:`

```
package pack1

var Pack1Int int = 42
var pack1Float = 3.14
func ReturnStr() string {
        return "Hello main!"
}
```

It exports an int variable `Pack1Int` and a function `ReturnStr` which returns a string. This program does not do anything when it is run, because it does not contain a main()—function.

In the main program `package_test.go` the package is imported via the statement

```
import "./pack1/pack1"
```

The general format of the import is:

`import "path or url to the package"` like `import "github.com/org1/pack1"`

If it is a path it is relative to the directory of the current package.

`Listing 9.5—package_test.go:`

```
package main
import (
        "fmt"
        "./pack1/pack1"
)
```

```
func main() {
        var test1 string
        test1 = pack1.ReturnStr()
        fmt.Printf("ReturnStr from package1: %s\n", test1)
        fmt.Printf("Integer from package1: %d\n", pack1.Pack1Int)
        // fmt.Printf("Float from package1: %f\n", pack1.pack1Float)
}
```

Output:
```
ReturnStr from package1: Hello main!
Integer from package1: 42
```

In case the package pack1 would be in the same map as our current program, it could be imported with: `import "./pack1"`, but this is not considered a good practice.

The line `fmt.Printf("Float from package1: %f\n", pack1.pack1Float)`, trying to access an unexported variable or function, does not even compile. It gives the error:

```
cannot refer to unexported name pack1.pack1Float
```

The packages which are utilized by the main-program must be built before the compilation of the main program. Every exported pack1-Item that is used in the main program must be qualified by the package name: `pack1.Item`. For another example: see Listings 4.6 and 4.7 .

So by convention there is a close relationship between subdirectories and packages: *each package (all the go-files belonging to it) resides in its own subdirectory, which has the same name as the package.* For clarity different packages reside in different directories.

Import with **.**: `import . "./pack1"`

When using . as an alias, one can use the items from the package without qualifying with their package names, as for example in: `test1 = ReturnStr()`.

It imports `pack1` in the current namespace, generally considered only good for testing purposes.

Import with **_**: `import _ "./pack1/pack1"`

The package pack1 is imported for its side-effects only, that is: its init functions are executed and global variables initialized.

Importing of installed external packages:

If you need one or more external packages in your application, you will first have to install them locally on your machine with the `go install` command (see § 9.7).

Suppose you want to use a (fictional) package which resides at http://codesite.ext/author/goExample/goex

where codesite could be googlecode, github, bitbucket, launchpad or others; ext = the extension, like .com or .org, and goex is the package name (often these start with go, by no means a necessary convention)

You install this with the command: `go install codesite.ext/author/goExample/goex`

This installs the code in the map `codesite.ext/author/goExample/goex` under $GOROOT/src/

Once installed, to import it in your code, use:

```
import goex "codesite.ext/author/goExample/goex"
```

So the import path will be the web-accessible URL for your project's root followed by the subdirectory.

The documention for go install at http://golang.org/cmd/goinstall/ lists the import paths for a number of widely used code repositories on the web.

Initialization of a package:

Program execution begins by importing the packages, initializing the main package and then invoking the function `main()`.

A package with no imports is initialized by assigning initial values to all its package-level variables and then calling any package-level `init()` function defined in its source. A package may contain multiple `init()` functions, even within a single source file; they execute in unspecified order. It is best practice if the determination of a package's values only depend on other values or functions found in the same package.

`init()` functions cannot be called.

Imported packages are initialized before the initialization of the package itself (with the exception of main), but initialization of a package occurs only once in the execution of a program.

Building and installing a package(see also § 9.7):

In Linux /OSX this can be done with an analogous Makefile script like in § 3.4(2):

```
include $(GOROOT)/src/Make.inc
TARG=pack1
GOFILES=\
  pack1.go\
  pack1b.go\

include $(GOROOT)/src/Make.pkg
```

and make it executable with chmod 777 `./Makefile.`

The include statements pull in functionality that automatically detects the machine's architecture and uses the correct compiler and linker.

Then `run` make on the command-line or the `gomake` tool: both make a map _obj containing the static library `pack1.a.`

The command `go install` (see § 9.7, the preferred way since Go 1) also copies `pack1.a` to the official local packages map $GOROOT/pkg, in a submap with the name of the operating system (e.g. linux).When this is done the package can be imported in a program simply by its name, like `import "pack1"` instead of `import "path to pack1".`

If this is not desirable or allowed, use the -I option while building with 6/8g:

```
6g—I map_pack1 package_test.go # where map_pack1 is the map which contains
pack1.a
```

(the flag—I lets the compiler search for packages in the map-name which follows the option.)

and the—L option while linking with 6/8l :

```
6l—L map_pack1 package_test.6
```

We will come back to the topic of making and building your own package when we have encountered the test-tool `go test` (chapter 13).

EXERCISES:

Question 9.1:

a) Can a package be divided over multiple source files ?
b) Can a single source file contain multiple packages ?

Exercise 9.3: Make a program `main_greetings.go` which can greet the user with "Good Day", or "Good Night". The different greetings should be in a separate package `greetings.`

In the same package, make a function IsAM() which returns a bool to indicate whether the current time is AM (before 12h) or PM; also make functions IsAfternoon() and IsEvening().

Use this in `main_greetings` to adapt your greeting.

(Hint: use the time package)

Exercise 9.4: Make a program `main_oddeven.go` which tests for the first 100 integers whether they are even or not. The function which does the test is contained in a package `even`.

Exercise 9.5: Using the Fibonacci-program from § 6.6:

(1) Place the Fibonacci-function in its own package fibo and call it from a main program which stores the last input-value to the function in a global variable
(2) Expand the fibo-package so that the operation is also a variable, which is passed through when calling Fibonacci. Experiment with + and *

```
main_fibo.go / fibonacci.go
```

9.6 Using godoc for your custom packages.

The godoc tool (§ 3.6) works also very nice for showing the comments in your own package: the comments must start with // and precede the declarations (package, types, functions …) with no blank line in between. godoc will produce will produce a series of html-pages, one for each go file.

For example:

- in the map doc_example we have the go-files sort and sortmain from §11.7 with some comments in the sort file (the files need not be compiled);
- navigate on the command-line to this map and start the command:

  ```
  godoc -http=:6060 -path=”.”
  ```

  (. is the current map, the -path flag can be of the form /path/to/my/package1 where the map package1 contains your sources, or accept a list of colon (:)-separated paths; unrooted paths are relative to the current working directory.
- open a browser at the address http://localhost:6060

You then see the local godoc-page (see § 3.6) with left from the Packages link a link to your map doc_example:

doc_example | Packages | Commands | Specification

Beneath it is an ordered overview of links to the source and all objects within it (so it is nice for navigating and looking up in the source code), together with the documentation/commentary:

Package sort

func Float64sAreSorted
func IntsAreSortedfunc IsSortedfunc Sort
func SortFloat64s
func SortInts
func SortStrings
func StringsAreSorted
type Float64Array
func (Float64Array) Len
func (Float64Array) Less
func (Float64Array) Swap
type IntArray
func (IntArray) Len
func (IntArray) Less
func (IntArray) Swap
type Interface
type StringArray
func (StringArray) Len
func (StringArray) Less
func (StringArray) Swap
Other packages

```
import “doc_example”
```

Sorting using `a general interface:`

Package files

sort.go

func Float64sAreSorted[**Top**]

```
func Float64sAreSorted(a []float64) bool
```

func IntsAreSorted[**Top**]

```
func IntsAreSorted(a []int) bool
```

func IsSorted[**Top**]

```
func IsSorted(data Interface) bool
Test if data is sorted
```

func Sort[**Top**]

```
func Sort(data Interface)
General sort function
```

func SortInts[**Top**]

```
func SortInts(a []int)
Convenience wrappers for common cases
```

type IntArray[**Top**]

```
Convenience types for common cases: IntArray
type IntArray []int
```

Fig 9.1: Package documentation with godoc

If you work in a team and the source tree is stored on a network disk, you can start the godoc process in it to give a continuous document support to all team members. With the—sync and—sync_minutes= n, you can even make it automatically update your documentation every n minutes!

9.7 Using go install for installing custom packages.

go install is Go's automatic package installation tool: it installs packages, downloading them from remote repositories over the internet if needed and installing them on the local machine: checkout, compile and install in one go.

It installs each of the packages given on the command line; it installs a package's prerequisites before trying to install the package itself and handles the dependencies automatically. The dependencies of the packages residing in submaps are also installed, but documentation or examples are not: they are browsable on the site.

The list of all installed packages can be found in $GOROOT/goinstall.log

It uses the variable GOPATH (see § 2.2).

<u>Remote packages (see § 9.5):</u>

Suppose we want to install the interesting package *tideland* (this contains a number of helpful routines, see <u>http://code.google.com/p/tideland-cgl/</u>).

Because we need create directory rights on some submaps of the Go-installation, we need to issue the command as root or su.

Make sure that the Go-environment variables are properly set in the .bashrc file of root.

Install with the command: `go install tideland-cgl.googlecode.com/hg`

This puts the executable file hg.a in the map $GOROOT/pkg/linux_amd64/tideland-cgl.googlecode.com, and puts the Go source-files in $GOROOT/src/tideland-cgl.googlecode.com/hg, and also hg.a in a submap _obj.

From then on the functionality of the package can be used in Go code, using for example as package name `cgl`, by importing:

```
import   cgl    "tideland-cgl.googlecode.com/hg"
```

From Go 1 onwards go install will expect Google Code import paths to be of the form:

```
"code.google.com/p/ tideland-cgl".
```

<u>Updating to a new Go release:</u>

After updating to a new Go release all package binaries of locally installed packages are deleted. When invoking `go install–a` the tool reinstalls all previously installed packages, reading the list from $GOROOT/goinstall.log. If you want to update, recompile, and reinstall all goinstalled packages, use: `go install –a -u -clean or go install -a -u -nuke`

Because Go releases were frequent, care should be taken to verify the release against which the package is build; after Go 1 it is enough to know it was build against it.

`go install` can also be used to compile/link and locally install your own packages: see § 9.8.2

More info can be found at http://golang.org/cmd/go/ and http://golang.org/cmd/goinstall/

9.8 Custom packages: map structure, go install and go test

For the purposes of demonstration, we take a simple package uc which has a function UpperCase to turn a string into uppercase letters. This certainly is not worth while to make as your own package, it wraps the same functionality from package "strings", but the same techniques can be applied to more complex packages.

9.8.1 Map-structure for custom packages

The following structure gives you a good start (where uc stands for a general package name; the names in bold are maps, italicized is the executable):

```
/home/user/goprograms

     ucmain.go       (main program for using package uc)
     Makefile (2—makefile for ucmain)
     ucmain

     src/uc   (contains go code for package uc)
              uc.go
              uc_test.go
              Makefile (1—makefile for package)
              uc.a
              _obj
                     uc.a
              _test
                            uc.a
     bin             (contains the final executable files)
                     ucmain

     pkg/linux_amd64
                     uc.a   (object file of package)
```

Put your projects somewhere in a map goprograms (you can create an environment variable GOPATH for this, see §2.2/3: put the line `export GOPATH=/home/user/goprograms` in .profile and .bashrc), and your packages as submaps of src. The functionality is implemented in uc.go, belonging to the package uc:

Listing 9.6—uc.go:

```
package uc
import "strings"

func UpperCase(str string) string {
        return strings.ToUpper(str)
}
```

The package must always be accompanied by one or more testfiles, here we made uc_test.go, along the lines explained and demonstrated in § 9.8

Listing 9.7—uc_test.go:

```
package uc
import "testing"

type ucTest struct {
        in, out string
}
var ucTests = []ucTest {
        ucTest{"abc", "ABC"},
        ucTest{"cvo-az", "CVO-AZ"},
        ucTest{"Antwerp", "ANTWERP"},
}

func TestUC(t *testing.T) {
        for _, ut := range ucTests {
                uc := UpperCase(ut.in)
                if uc != ut.out {
                        t.Errorf("UpperCase(%s) = %s, must be %s.", ut.in, uc,
                        ut.out)
                }
        }
}
```

Build and install the package locally with the command: `go install src/uc` this copies uc.a to pkg/linux_amd64.

Alternatively, using make: put in the map src/uc a Makefile (1) for the package with the following content:

```
include $(GOROOT)/src/Make.inc

TARG=uc
GOFILES=\
        uc.go\

include $(GOROOT)/src/Make.pkg
```

On the command-line in this map invoke: `gomake`

This makes map _obj and puts the compiled package archive `uc.a in it.`

The package can then be tested with: `go test`

This makes the map _test with uc.a in it; the output gives: PASS, so the tests are OK.

In § 13.8 we give another example of gotest and delve a little deeper.

Remark: It is possible that your current account doesn't have enough rights to run go install (`permission denied` error). In that case, switch to the root user su. Make sure that the Go-environment variables and the path to the Go-binaries are also set for su, as for your normal account (see § 2.3)

Then we make our main starting program as `ucmain.go`:

Listing 9.8—ucmain.go:

```
package main
import (
        "fmt"
        "./uc/uc"
)

func main() {
        str1 := "USING package uc!"
        fmt.Println(uc.UpperCase(str1))
}
```

Then simply issue the command `go install` in this map.

Alternatively, copy uc.a in map uc and put a Makefile (2) alongside it with the text:

include $(GOROOT)/src/Make.inc

```
TARG=ucmain
GOFILES=\
        ucmain.go\

include $(GOROOT)/src/Make.cmd
```

Issuing `gomake` compiles ucmain.go to ucmain.

Running `./ucmain gives:` USING package uc!

9.8.2 Locally installing the package

Local packages in user map:

Using the given map-structure, the following commands can be used to install local packages from source:

```
go install /home/user/goprograms/src/uc # build and install uc
cd /home/user/goprograms/uc
go install ./uc # build and install uc (= does same as previous command)
cd ..
go install .    # build and install ucmain
```

Installing under $GOROOT:

If we want the package *to be used from any Go-program on the system*, it must be installed under $GOROOT.

To do this, set GOPATH = $GOROOT in .profile and .bashrc; then `go install` uc wil:

1) copy the source code to `$GOROOT/src/pkg/linux_amd64/uc`
2) copy the package archive `to $GOROOT/pkg/linux_amd64/uc`

It can then be imported in any Go-source as: `import uc`

9.8.3 OS dependent code

It is very rare that your program should code differently according to the operating system on which it is going to run: in the vast majority of cases the language and standard library handle most portability issues.

You could have a good reason to write platform-specific code, such as assembly-language support. In that case it is reasonable to follow this convention:

```
prog1.go
prog1_linux.go
prog1_darwin.go
prog1_windows.go
```

`prog1.go` defines the common code interface above different operating systems, and put the OS-specific code in its own Go-file named `prog1_os.go`

For the go tool you can then specify: `prog1_$GOOS.go or prog1_$GOARCH.go`
or in the platform Makefile: `prog1_$(GOOS).go\ or prog1_$(GOARCH).go\`

Exercise 9.6: `package strev`

Apply all the techniques from the previous paragraph (§ 9.7) to the package `strev` from Exercise 9.2

9.9 Using git for distribution and installation.

9.9.1 Installing to github

All this is fine for a local package, but how do we distribute it to the programmer community ? Then we need a source version-control system in the cloud, like the popular **git**.

git is default installed on a Linx or OS X machine, for Windows you must first install it, see: http://help.github.com/win-set-up-git/

I will lead you through creating a git-repository for the package uc from § 9.8:

Go to the package directory uc and create a git repository in it: `git init .`

The message appears: `Initialized empty git repository in …/uc`

Every git project needs a README file with a description of the package. So open your text editor (gedit, notepad, LiteIde) and put some comments there.

Then add all the files to the repository with: `git add README uc.go uc_test.go Makefile`
and mark it as the first version: `git commit -m "initial revision"`

Now go to the github-website: https://github.com where you must login.

But probably you don't have a login yet, so go to https://github.com/plans where you can create a free account for open source projects. Choose a username and password, give a valid email-address and go further Create an Account. Then you will get a list with the git commands; we have already done the commands for the local repository. An excellent help system http://help.github.com/ will guide you if you encounter any problems.

For creating a new repository uc in the cloud; issue the instructions (substitute NNNN with your username):

```
git remote add origin git@github.com:NNNN/uc.git
git push -u origin master
```

Then you're done: go check the github page of your package: https://github.com/NNNN/uc

9.9.2 Installing from github

If somebody wants to install your cloud-project to a local machine, open a terminal session with su and execute: `go install github.com/NNNN/uc`
where NNNN is your username on github.

This copies the:

- ➔ package uc.a in the map $GOROOT/pkg/linux_amd64/github.com
- ➔ source in $GOROOT/src/pkg/github.com/NNNN/uc

so the package is now available on that machine for any other Go-application with the import path: `"github.com/NNNN/uc" instead of "./uc/uc"`

This can be shortened to: `import uc "github.com/NNNN/uc"`

Then adapt your Makefile: replace TARG=uc with `TARG=github.com/NNNN/uc`

Gomake (and go install) will now work with the local version under $GOROOT.

Hosting sites and version control systems: other possibilities

Here are the major code hosting sites (between () the version control systems it uses) are:

bitbucket (hg),
github (git),
googlecode (hg/git/svn),
and launchpad (bzr).

Choose whichever version control system you are familiar with or in which your code is versioned locally on your machine. Mercurial (hg) is the version control system used by the central Go repository, so it is the closest to a guarantee as you can get that a developer wanting to use your project will have the right software. Git is also very popular, and so is often available. If you have never used version control before, these websites have some nice HOWTOs and you can find many great tutorials by searching Google for "{name} tutorial" where {name} is the name of the version control system you would like to learn.

9.10 Go external packages and projects.

We now know how to use Go and its standard library, but the Go-ecosystem is bigger than this. When embarking on our own Go-projects, it is best to search first if we cannot use some existing 3rd party Go package(s) or project(s). Most of these can be installed with the `go install tool.`

The first place to look is the tab *Projects* on the Package Dashboard at the Go-website (hosted in Google App Engine): http://godashboard.appspot.com/project; this is the manually curated list.

Classified by categories, like Build Tools, Compression, Data Structures, Databases and Storage, Development Tools, etc., this contains a wealth of more than 500 projects, giving for each its name, a short description, and a download link. These can be found on the following code repository sites, with the source control systems used mentioned between ( ):

- on Google Code, e.g. http://code.google.com/p/goprotobuf/, (Mercurial(hg) or Subversion)
- on Github: e.g. https://github.com/kr/pretty.go, (Git)
- on BitBucket, e.g. https://bitbucket.org/binet/igo/ (Mercurial(hg))
- on Launchpad, e.g. http:// launchpad.net/mgo (Bazaar)

or on other popular code sharing sites, or the website of the author(s).

In the repositories you can also submit your own project after moderation by an administrator.

If you want to see the actual Project-activity look at the tab *Packages* on Package Dashboard at the Go-website http://godashboard.appspot.com/package

This gives an overview of the "Most Installed Packages" (this week and all time) and the Recently Installed Packages, with in the column count the number of installations by external developers. If column build has status ok, then this indicates that the package is goinstallable with the latest release of Go.

The "Go Projects" and "Go Packages" pages aren't related. If a package appears on one it may not necessarily appear on the other.

Other collections (partly overlapping with the Package Dashboard) are:

http://go-lang.cat-v.org/dev-utils (Developer-oriented)
http://go-lang.cat-v.org/go-code (Programs and applications)
http://go-lang.cat-v.org/library-bindings (Library bindings)
http://go-lang.cat-v.org/pure-go-libs (Pure Go libraries)

There are already many great external libraries, such as for:

- MySQL (*GoMySQL*), PostgreSQL(*go-pgsql*), MongoDB (*mgo, gomongo*), CouchDB (*couch-go*), ODBC (*godbcl*), Redis (*redis.go*) and SQLite3 (*gosqlite*) database drivers,
- SDL bindings,
- Google's Protocol Buffers (*goprotobuf*),
- XML-RPC (*go-xmlrpc*),
- Twitter (*twitterstream*),
- OAuth libraries, (*GOAuth*)
  and much more.

9.11 Using an external library in a Go program.

(This § builds a web application and its Google App Engine version, which are explained respectively in chapters 19 and 21. You can come back to this example after having worked through these chapters.)

When starting a new project or adding new functionalities to an existing project you could save development time by incorporating an already existing Go-library into your application. In order to do that you have to understand the API (Application Programming Interface) of the library, that is: which methods you can call in this library and how to call them. It could indeed be possible that you do not have the source of this library, but the makers will then surely have documented the API and detailed how to use it.

As an example we will write a small program using the *urlshortener* from Google APIs: you can try it out at http://goo.gl/. Enter a URL like http://www.destandaard.be and you will see a shorter URL returned like http://goo.gl/O9SUO, which is much easier to embed, say, in a service like twitter. Documentation for the Google urlshortener service can be found at http://code.google.com/apis/urlshortener/. (In chapter 19 we will develop our own version of an urlshortener.)

Google makes this technology available to other developers as an API that we can call from our own applications (free up to a specified limit). They have also generated a Go client-library to make it even easier.

Remark: Google has made life easier for Go developers to use their services by providing them with Google API Go clients. The Go-client programs where automatically generated from the JSON-description of the Google libraries. You can read more about it at http://code.google.com/p/google-api-go-client/.

Downloading and installing the Go client library:

This will be accomplished with the `go install`-tool (see § 9.7). But first verify that the GOPATH variable is set in your environment, because the external source code will be downloaded into the directory $GOPATH/src and the packages also will be installed under this directory `$GOPATH/pkg/"machine_arch"/`.

Then we install the API by invoking the following command in the console:

```
    go install google-api-go-client.googlecode.com/hg/urlshortener/v1
```

`go install` will download the source code, compile it and install the package.

(In Linux Ubuntu with 6g r60 9481 installation works fine, installed package is beneath pkg/linux_amd64.)

A web program to use the urlshortener service:

We can now use the installed package in our programs by importing it and giving it an alias (to type less): `import urlshortener "google-api-go-client.googlecode.com/hg/urlshortener/v1"`

We now write a web application (see chapter 15 § 15.4-8) that shows a web form where you can give in a long url and ask for a short url, and the inverse. For this we use the templating package and we write 3 handler functions: the root handler shows the form by executing the form template, the short handler (for urls of the form /short) add to this by taking a long url and returns the short url, and the long handler (urls of the form /long) does the inverse.

To invoke the urlshortener API first you have to make a service- instance `urlshortenerSvc` with:

```
urlshortenerSvc, _ := urlshortener.New(http.DefaultClient)
```

by using the default client that is available in the http package.

To get the short url we fill in the Url data structure with the given long url and call the service's Url.Insert method by calling Do():

```
url, _ := urlshortenerSvc.Url.Insert(&urlshortener.Url{LongUrl: longUrl,}).Do()
```

The Id of the url returned is then the short url which we asked for.

To get the long url we fill in the Url data structure with the short url and call the service's Url.Get method by calling Do() `url, err := urlshortenerSvc.Url.Get(shortUrl).Do()`

The LongUrl of the url returned is then the original url.

Here is the program:

<u>Listing 9.9—use_urlshortener.go:</u>

```
package main
import (
        "fmt"
        "net/http"
        "text/template"
         urlshortener "google-api-go-client.googlecode.com/hg/urlshortener/v1"
)

func main() {
        http.HandleFunc("/", root)
        http.HandleFunc("/short", short)
        http.HandleFunc("/long", long)
        http.ListenAndServe("localhost:8080", nil)
}

// the template used to show the forms and the results web page to the user
var rootHtmlTmpl = template.Must(template.New("rootHtml").Parse(`
<html><body>
<h1>URL SHORTENER</h1>
{{if .}}{{.}}<br /><br />{{end}}
<form action="/short" type="POST">
Shorten this: <input type="text" name="longUrl" />
<input type="submit" value="Give me the short URL" />
</form>
<br />
<form action="/long" type="POST">
Expand this: http://goo.gl/<input type="text" name="shortUrl" />
<input type="submit" value="Give me the long URL" />
</form>
</body></html>
`))

func root(w http.ResponseWriter, r *http.Request) {
        rootHtmlTmpl.Execute(w, nil)
}
func short(w http.ResponseWriter, r *http.Request) {
        longUrl := r.FormValue("longUrl")
        urlshortenerSvc, _ := urlshortener.New(http.DefaultClient)
        url, _ := urlshortenerSvc.Url.Insert(&urlshortener.Url{LongUrl:
        longUrl,}).Do()
        rootHtmlTmpl.Execute(w, fmt.Sprintf("Shortened version of %s is : %s",
        longUrl, url.Id))
}

func long(w http.ResponseWriter, r *http.Request) {
        shortUrl := "http://goo.gl/" + r.FormValue("shortUrl")
        urlshortenerSvc, _ := urlshortener.New(http.DefaultClient)
        url, err := urlshortenerSvc.Url.Get(shortUrl).Do()
        if err != nil {
               fmt.Println("error: %v", err)
               return
        }
```

```
            rootHtmlTmpl.Execute(w, fmt.Sprintf("Longer version of %s is : %s",
            shortUrl, url.LongUrl))
    }
```

Compile this code with: `6g -I $GOPATH/pkg/linux_amd64 urlshortener.go`
and link it with: `6l -L $GOPATH/pkg/linux_amd64 urlshortener.6`
To execute it: `./6.out`

(ensure that you do not have any other applications running on http://localhost:8080 or this step will fail). To test it navigate to the web page: `http://localhost:8080/`

For brevity of code the error returns were not checked, but this should be done for a real production application!

Turning it into a Google App-Engine application:
The only things in the code in the previous listing which have to change are:

package main → package urlshort
func main() → func init()

Create a map with the same name as the package: urlshort

Copy into this map the source code from the 2 following installed maps:

```
google-api-go-client.googlecode.com/hg/urlshortener
google-api-go-client.googlecode.com/hg/google-api
```

Furthermore you need a configuration file app.yaml, with content like:

```
application: urlshort
version: 0-1-test
runtime: go
api_version: 3

handlers:
- url: /.*
  script: _go_app
```

Now go to your project map and run in the console: `dev_appserver.py urlshort`
and start a web client for your application in the browser: `http://localhost:8080`

Chapter 10—Structs and Methods

Go supports *user-defined or custom types* in the form of alias types or structs. A struct tries to represent a real-world entity with its properties. Structs are *composite types,* to use when you want to define a type which consist of a number of properties, each having their own type and value, grouping pieces of data together. Then one can access that data as if it were part of a single entity. They are also *value types* and so are constructed with the **new** function.

The component pieces of data that constitute the struct type are called *fields.* A field has a type and a name; field names within a struct must be unique.

The concept was called ADT (Abstract Data Type) in older texts on software engineering, it was called a *record* in older languages like Cobol, and it also exists under the same name of *struct* in the C-family of languages, in the OO languages as a lightweight-class without methods. However because Go does not have the concept of a class, the struct type has a much more important place in Go.

10.1 Definition of a struct

The general format of the definition of a *struct* is as follows:

```
type identifier struct {
        field1 type1
        field2 type2
        …
}
```

Also type `T struct { a, b int }` is legal syntax, and more suited for simple structs.

The fields in this struct have *names*, like field1, field2, etc. If the field is never used in code, it can be named _.

These fields can be of any type, even structs themselves (see § 10.5), functions or interfaces (see Chapter 11). Because a struct is a value, we can declare a variable of the struct type, and give its fields values, like:

```
var s T
s.a = 5
s.b = 8
```

An array could be seen as a sort of struct but with indexed rather than named fields.

Using new:

Memory for a new struct variable is allocated with the **`new`** function, which returns a pointer to the allocated storage: `var t *T = new(T)`, which can be put on different lines if needed (e.g. when the declaration has to be package scope, but the allocation is not needed at the start):

```
var t *T
t = new(T)
```

The idiom to write this shorter is: `t := new(T)`, the variable t is a pointer to T: at this point the fields contain the zero-values according to their types.

However declaring `var t T` also allocates and zero-initializes memory for t, but now t is of type T. In both cases t is commonly called an *instance* or *object* of the type T.

A very simple example is given in `Listing 10.1—structs_fields.go:`

```
package main
import "fmt"

type struct1 struct {
        i1   int
        f1   float32
        str  string
}

func main() {
        ms := new(struct1)
        ms.i1 = 10
        ms.f1 = 15.5
        ms.str = "Chris"
        fmt.Printf("The int is: %d\n", ms.i1)
        fmt.Printf("The float is: %f\n", ms.f1)
        fmt.Printf("The string is: %s\n", ms.str)
        fmt.Println(ms)
}
```

Output:

```
The int is: 10
The float is: 15.500000
The string is: Chris
&{10 15.5 Chris}
```

The default (so %v) printout of a struct with fmt.Println() nicely shows its content.

The fields can be given a different value by using the dot-notation, as is custom in OO-languages: **`structname.fieldname = value`**

The values of the struct-fields can be retrieved with the same notation: **`structname.fieldname`**

This is called a *selector* in Go. In order to access the fields of a struct, whether the variable is of the struct type or a pointer to the struct type, we use the same selector-notation:

```
type myStruct struct { i int }
var v myStruct         // v has struct type
var p *myStruct        // p is a pointer to a struct
v.i
p.i
```

An even shorter notation and the idiomatic way to initialize a struct instance (a *struct-literal*) is the following:

```
ms := &struct1{10, 15.5, "Chris"}
// this means that ms is of type *struct1
```

or:

```
var mt struct1
mt = struct1{10, 15.5, "Chris"}
```

The composite literal syntax `&struct1{a, b, c}` is a shorthand; under the covers it still calls new(); the values must be given in field-order. In the following example we see that you can also initialize values by preceding them with the fieldnames. So `new(Type)` and `&Type{}` are equivalent expressions.

A typical example of a struct is a time-interval (with start- and end time expressed here in seconds):

```
type Interval struct {
        start int
        end   int
}
```

And here are some initializations:

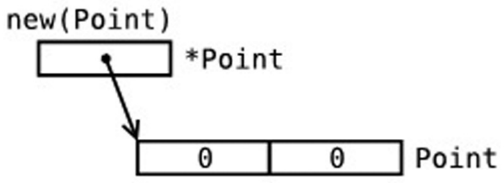
```
inter := Interval{0,3}                (A)
inter2 := Interval{end:5, start:1}   (B)
inter3 := Interval{end:5}            (C)
```

In case (A) the values given in the literal must be exactly in the same order as the fields are defined in the struct, the & is not mandatory. Case (B) shows another possibility where the fieldnames with a : precede the value; in that case their sequence must not be the same, and fields could also be omitted, like in case (C).

The naming of the struct type and its fields adheres to the Visibility-rule (§ 4.2); it is possible that an exported struct type has a mix of fields: some exported, others not.

The following figure clarifies the memory layout of a struct value and a pointer to a struct for the following struct type: `type Point struct { x, y int }`

Initialized with new:



Initialized as a struct literal:

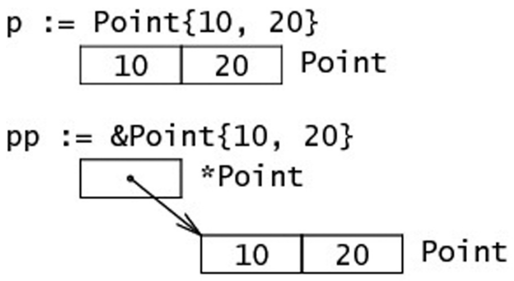


Fig 10.1: Memory layout of a struct

A type struct1 must be unique in the package pack1 in which it is defined, its complete type name is: `pack1.struct1`

The following example Listing 10.2—person.go shows a struct Person, a method upPerson which has a parameter of type *Person (so that the object itself can be changed!) and 3 different ways of calling this method:

```
package main
import (
        "fmt"
        "strings"
)

type Person struct {
        firstName     string
        lastName              string
}

func upPerson (p *Person) {
        p.firstName = strings.ToUpper(p.firstName)
        p.lastName = strings.ToUpper(p.lastName)
}

func main() {
// 1- struct as a value type:
        var pers1 Person
        pers1.firstName = "Chris"
        pers1.lastName = "Woodward"
        upPerson(&pers1)
        fmt.Printf("The name of the person is %s %s\n", pers1.firstName, pers1.
        lastName)

// 2—struct as a pointer:
        pers2 := new(Person)
        pers2.firstName = "Chris"
        pers2.lastName = "Woodward"
        (*pers2).lastName = "Woodward"      // this is also valid
        upPerson(pers2)
        fmt.Printf("The name of the person is %s %s\n", pers2.firstName, pers2.
        lastName)
// 3—struct as a literal:
        pers3 := &Person{"Chris","Woodward"}
        upPerson(pers3)
        fmt.Printf("The name of the person is %s %s\n", pers3.firstName, pers3.
        lastName)
}
/* Output:
The name of the person is CHRIS WOODWARD
The name of the person is CHRIS WOODWARD
The name of the person is CHRIS WOODWARD
*/
```

In case 2 we see that setting a value through a pointer like in `pers2.lastName = "Woodward"` works, there is no -> operator necessary like in C++: Go does the conversion automatically.

Note that we can also set the value by dereferencing the pointer:

```
(*pers2).lastName = "Woodward"
```

Structs and memory layout:

Structs in Go and the data they contain, even when a struct contains other structs, form a continuous block in memory: this gives a huge performance benefit. This is unlike in Java with its reference types, where an object and its contained objects can be in different parts of memory; in Go this is also the case with pointers.This is clearly illustrated in the following example:

```
type Rect1 struct { Min, Max Point }
type Rect2 struct { Min, Max *Point }
```

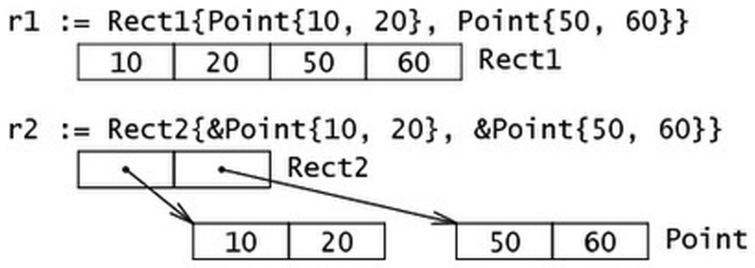

r1 := Rect1{Point{10, 20}, Point{50, 60}}
10
20
50
60
Rect1
r2 := Rect2{&Point{10, 20}, &Point{50, 60}}
Rect2
10
20
50
60
Point

Fig 10.2: Memory layout of a struct of structs

Recursive structs:

A struct type can be defined in terms of itself. This is particularly useful when the struct variable is an element of a linked list or a binary tree, commonly called a *node*. In that case the node contains links (the addresses) to the neighbouring nodes; su for a list and le, ri for a tree are pointers to another Node-variable.

Linked list:

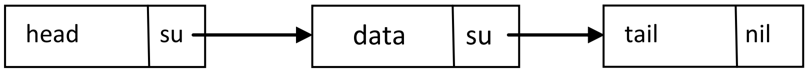


Fig. 10.3: Linked list as recursive struct

where the the data field contains the useful information (for example a float64), and su points to the successor node;

in Go-code:
```
type Node struct {
        data  float64
        su    *Node
}
```

The first element of the list is called the *head*, it points to the 2nd element; the last element is called the *tail*, it doesn't point to any successor, so its su field has value nil. Of course in a real list we would have many data-nodes, the list can grow or shrink dynamically.

In the same way you could define a doubly linked list with a predecessor node field pr and a successor field su.

```
type Node struct {
         pr    *Node
         data  float64
         su    *Node
}
```

Binary tree:

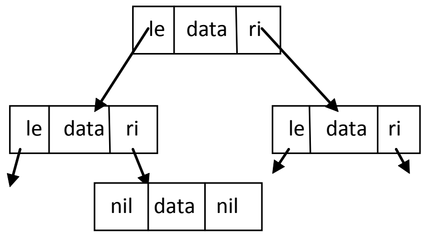


Fig 10.4: Binary tree as recursive struct

Here every node can at most have links to two other nodes: the left (le) and the right (ri); both of them can propagate this further. The top element of the tree is called the *root*; the bottom layer of nodes which have no more nodes beneath them are called the *leaves;* a leave node has nil-values for the le- and ri pointers. Such a node is a tree in itself, so we could write:

```
in Go-code:     type Tree struct {
                        le    *Tree
                        data  float64
                        ri    *Tree
                }
```

Conversion of structs:

As we have already seen conversion in Go follows strict rules. When we have a struct type and define an alias type for it, both types have the same underlying type and can be converted into one another as illustrated in Listing 10.3, but also note the compile-error cases which denote impossible assignments or conversions:

Listing 10.3—struct_conversions.go:

```
package main
import "fmt"

type number struct {
        f float32
}

type nr number   // alias type

func main() {
        a := number{5.0}
        b := nr{5.0}
        // var i float32 = b   // compile-error: cannot use b (type nr) as type
        float32 in assignment
        // var i = float32(b)  // compile-error: cannot convert b (type nr) to
        type float32
        // var c number = b    // compile-error: cannot use b (type nr) as type
        number in assignment
        // needs a conversion:
        var c = number(b)
        fmt.Println(a, b, c)
}
```

```
// output: {5} {5} {5}
```

EXERCISES:

Exercise 10.1: vcard.go: Define a struct Address and a struct VCard. The latter contains a person's name, a number of addresses, a birth date, a photo. Try to find the right data types. Make your own vcard and print its contents.

Hint: VCard must contain addresses, will they be included as values or as pointers ?

The 2nd choice is better, consuming less memory. So an Address struct with a name and two pointers to addresses could be printed out with %v as:

```
{Kersschot 0x126d2b80 0x126d2be0}
```

Exercise 10.2: Make a version of `personex1.go` where the parameter of upPerson is not a pointer. Explain the difference in behavior.

Exercise 10.3 `point.go:`

Define a 2 dimensional Point with coordinates X and Y as a struct. Do the same for a 3 dimensional point, and a Polar point defined with its polar coordinates. Implement a function Abs() that calculate the length of the vector represented by a Point, and a function Scale that multiplies the coordinates of a point with a scale factor(hint: use function Sqrt from package math).

Exercise 10.4: `rectangle.go:`

Define a struct Rectangle with int properties length and width. Give this type methods Area() and Perimeter() and test it out.

10.2 Creating a struct variable with a Factory method

10.2.1 A factory for structs

Go doesn't support constructors as in the OO-languages, but constructor-like factory functions are easy to implement. Often a *factory* is defined for the type for convenience; by convention its name starts with new or New. Suppose we define a File struct type:

```
type File struct {
        fd   int      // file descriptor number
        name string    // file name
}
```

Then the factory, which returns a pointer to the struct type, would be:

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

Often a Go constructor can be written succinctly using initializers within the factory function.

An example of calling it: `f := NewFile(10, “./test.txt”)`

If File is defined as a struct type, the expressions `new(File)` and `&File{}` are equivalent.

Compare this with the clumsy initializations in most OO languages: `File f = new File( …)`

In general we say that the factory *instantiates an object of the defined type*, just like in the class-based OO languages.

If you have a struct type T and you quickly want to see how many bytes an instance occupies in memory, use: `size := unsafe.Sizeof(T{})`

How to force using the factory method:

By applying the Visibility rule (§ 4.2.1, § 9.5) we can force the use of the factory method and forbid using new, effectively making our type private as it is called in OO-languages.

```
package matrix

type matrix struct {
        …
}

function NewMatrix(params) *matrix {
        m := new(matrix)
        // m is initialized
        return m
}
```

Because of the m of `matrix` we need to use the factory method in another package:

```
package main
import "matrix"
…
wrong := new(matrix.matrix)    // will NOT compile (matrix is private)
right := matrix.NewMatrix(…)   // the ONLY way to instantiate a matrix
```

10.2.2 new() and make() revisited for maps and structs:

The difference between these two built-in functions was clearly defined in § 7.2.4 with an example for slices.

By now we have seen 2 of the 3 types for which make() can be used:

slices / maps / channels (see chapter 14).

To illustrate the difference in behavior for maps and the possible errors, experiment with the following program:

Listing 10.4—new_make.go (does not compile!):

```
package main

type Foo map[string]string
type Bar struct {
        thingOne string
        thingTwo int
}

func main() {
        // OK:
        y := new(Bar)
        (*y).thingOne = "hello"
        (*y).thingTwo = 1
        // not OK:
        z := make(Bar) // compile error: cannot make type Bar
        z.thingOne = "hello"
```

```
         z.thingTwo = 1
         // OK:
         x := make(Foo)
         x["x"] = "goodbye"
         x["y"] = "world"
         // not OK:
         u := new(Foo)
         (*u)["x"] = "goodbye" // !! panic !!: runtime error: assignment to entry
         in nil map
         (*u)["y"] = "world"
}
```

To try to make() a struct variable is not so bad, the compiler gets the error; but newing a map and trying to fill it with data gives a runtime error! new(Foo) is a pointer to a nil, not yet allocated, map: so be very cautious with this!

10.3 Custom package using structs

Here is an example where in `main.go` a struct is used from a package `structPack` in submap struct_pack.

Listing 10.5—structPack.go:

```
package structPack

type ExpStruct struct {
         Mi1 int
         Mf1 float
}
```

Listing 10.6—main.go:

```
package main
import (
         "fmt"
         "./struct_pack/structPack"
)

func main() {
         struct1 := new(structPack.ExpStruct)
         struct1.Mi1 = 10
```

```
        struct1.Mf1 = 16.
        fmt.Printf("Mi1 = %d\n", struct1.Mi1)
        fmt.Printf("Mf1 = %f\n", struct1.Mf1)
}
```

Output:

```
Mi1 = 10
Mf1 = 16.000000
```

10.4 Structs with tags

A field in a struct can, apart from a name and a type, also optionally have a *tag*: this is a string attached to the field, which could be documentation or some other important label. The tag-content cannot be used in normal programming, only the package `reflect` can access it. This package which we explore deeper in the next chapter (§ 11.10), can investigate types, their properties and methods in runtime, for example: `reflect.TypeOf()` on a variable gives the right type; if this is a struct type, it can be indexed by Field, and then the Tag property can be used.

The program Listing 10.7—struct_tag.go shows how this works:

```
package main
import (
        "fmt"
        "reflect"
)

type TagType struct {  //  tags
        field1 bool   "An important answer"
        field2 string "The name of the thing"
        field3 int     "How much there are"
}

func main() {
        tt := TagType{true, "Barak Obama", 1}
        for i:= 0; i < 3; i++ {
                        refTag(tt, i)
        }
}

func refTag(tt TagType, ix int) {
        ttType := reflect.TypeOf(tt)
        ixField := ttType.Field(ix)
        fmt.Printf("%v\n", ixField.Tag)
}
/* Output:
An important answer
The name of the thing
How much there are */
```

10.5 Anonymous fields and embedded structs

10.5.1 Definition

Sometimes it can be useful to have structs which contain one or more *anonymous (or embedded) fields,* that is *fields with no explicit name.* Only the type of such a field is mandatory and the type is then also its name. Such an anonymous field can also be itself a struct: *structs can contain embedded structs*.

This compares vaguely to the concept of inheritance in the OO-languages, and as we will see it can be used to simulate a behavior very much like inheritance. This is obtained by embedding or composition, so we can say that in Go composition is of favoured over inheritance.

Consider the following program Listing 10.8—structs_anonymous_fields.go:

```
package main
import "fmt"

type innerS struct {
        in1  int
        in2  int
}

type outerS struct {
        b   int
        c   float32
        int    // anonymous field
        innerS // anonymous field
}

func main() {
        outer := new(outerS)
        outer.b = 6
        outer.c = 7.5
```

```
         outer.int = 60
         outer.in1 = 5
         outer.in2 = 10

         fmt.Printf("outer.b is: %d\n", outer.b)
         fmt.Printf("outer.c is: %f\n", outer.c)
         fmt.Printf("outer.int is: %d\n", outer.int)
         fmt.Printf("outer.in1 is: %d\n", outer.in1)
         fmt.Printf("outer.in2 is: %d\n", outer.in2)

// with a struct-literal:
         outer2 := outerS{6, 7.5, 60, innerS{5, 10}}
         fmt.Println("outer2 is: ", outer2)

}
```

Output:

```
              outer.b is: 6
              outer.c is: 7.500000
              outer.int is: 60
              outer.in1 is: 5
              outer.in2 is: 10
              outer2 is: {6 7.5 60 {5 10}}
```

To store data in an anonymous field or get access to the data we use the name of the data type: outer.int; a consequence is that we can only have one anonymous field of each data type in a struct.

10.5.2 Embedded structs

As a struct is also a data type, it can be used as an anonymous field; see the example above. The outer struct can directly access the fields of the inner struct with `outer.in1`; this is even the case when the embedded struct comes from another package. The inner struct is simply inserted or "embedded" into the outer. This simple 'inheritance' mechanism provides a way to derive some or all of your implementation from another type or types.

Here is another example Listing 10.9—embedd_struct.go:

```
package main
import "fmt"

type A struct {
         ax, ay int
}

type B struct {
        A
        bx, by float32
}

func main() {
        b := B{A{1, 2}, 3.0, 4.0}
        fmt.Println(b.ax, b.ay, b.bx, b.by)
        fmt.Println(b.A)
}
```

Output:

```
1 2 3 4
{1 2}
```

Exercise 10.5: `anonymous_struct.go`
Make a struct with 1 named float field, and 2 anonymous fields of type int and string. Create an instance of the struct with a literal expression and print the content.

10.5.3 Conflicting names

What are the rules when there are two fields with the same name (possibly a type-derived name)?

1) An outer name hides an inner name. This provides a way to override a field or method.
2) If the same name appears twice at the same level, it is an error if the name is used by the program. (If it's not used, it doesn't matter.) There are no rules to resolve the ambiguity; it must be fixed.

Examples:

```
type A struct { a int }
type B struct { a, b int }

type C struct { A; B }
var c C;
```

rule (2) When we use c.a it is an error, what is meant: c.A.a or c.B.a?
the compiler error is: `ambiguous DOT reference c.a`
disambiguate with either c.A.a or c.B.a

```
type D struct { B; b float32 }
var d D;
```

rule (1) Using d.b is ok: it is the float32, not the b from B; if we want the inner b we can get at it by d.B.b.

10.6 Methods

10.6.1 What is a method?

Structs look like a simple form of classes, so an OO programmer might ask: where are the methods of the class? Again Go has a concept with the same name and roughly the same meaning: a Go *method is a function that acts on variable of a certain type*, called the *receiver.* So a method is a special kind of function.

The receiver type can be (almost) anything, not only a struct type: any type can have methods, even a function type or alias types for int, bool, string or array. The receiver also cannot be an interface type (see Chapter 11), since an interface is the abstract definition and a method is the implementation; trying to do so generates the compiler error: `invalid receiver type…`

Lastly it cannot be a pointer type, but it can be a pointer to any of the allowed types.

The combination of a (struct) type and its methods is the Go equivalent of a class in OO. One important difference is that the code for the type and the methods binding to it are not grouped together; they can exist in different source files, the only requirement is that they have to be in the same package.

The collection of all the methods on a given type T (or *T) is called the *method set* of T (or *T).

Methods are functions, so again there is *no method overloading:* for a given type, there is only one method with a given name. But based on the receiver type, there is overloading: a method with the same name can exist on 2 of more different receiver types,e.g. this is allowed in the same package:

```
func (a *denseMatrix) Add(b Matrix) Matrix
func (a *sparseMatrix) Add(b Matrix) Matrix
```

Also an alias of a certain type doesn't have the methods defined on that type.

The general format of a method is:

```
func (recv receiver_type) methodName(parameter_list) (return_value_list) { … }
```

The receiver is specified in ( ) before the method name after the func keyword.

If recv is the receiver instance and Method1 the method name, then the call or invocation of the method follows the traditional object.method selector notation: **`recv.Method1()`**

In this expression if recv is a pointer, then it is automatically dereferenced.

If the method does not need to use the value recv, you can discard it by subsituting a _, as in:

```
func (_ receiver_type) methodName(parameter_list) (return_value_list) { … }
```

recv is like the this- or self existing in OO-languages, but in Go there is no specified keyword for it; if you like you can use self or this as name for the receiver variable, but you can choose freely. Here is a simple example of methods on a struct in `Listing 10.10—method.go:`

```
package main
import “fmt”

type TwoInts struct {
        a int
        b int
}

func main() {
        two1 := new(TwoInts)
        two1.a = 12
        two1.b = 10

        fmt.Printf(“The sum is: %d\n”, two1.AddThem())
        fmt.Printf(“Add them to the param: %d\n”, two1.AddToParam(20))

        two2 := TwoInts{3, 4}
        fmt.Printf(“The sum is: %d\n”, two2.AddThem())
}

func (tn *TwoInts) AddThem() int {
        return tn.a + tn.b
}

func (tn *TwoInts) AddToParam(param int) int {
        return tn.a + tn.b + param
}
```

Output:
```
The sum is: 22
Add them to the param: 42
The sum is: 7
```

And here is an example of methods on a non struct type `Listing 10.11—method2.go:`

```
package main
import "fmt"

type IntVector []int

func (v IntVector) Sum() (s int) {
        for _, x := range v {
               s += x
        }
        return
}

func main() {
        fmt.Println(IntVector{1, 2, 3}.Sum())      // Output: 6
}
```

Exercise 10.6: `employee_salary.go`

Define a struct employee with a field salary, and make a method giveRaise for this type to increase the salary with a certain percentage.

Exercise 10.7: `iteration_list.go`

What's wrong with the following code ?

```
package main
import "container/list"
```

```
func (p *list.List) Iter() {
        // …
}

func main() {
        lst := new(list.List)
        for _ = range lst.Iter() {
        }
}
```

A method and the type on which it acts must be defined in the same package, that's why you cannot define methods on type int, float or the like. Trying to define a method on an int type gives the compiler error:

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

For example if you want to define the following method on time.Time:

```
func (t time.Time) first3Chars() string {
        return time.LocalTime().String()[0:3]
}
```

You get the same error for a type defined in another, thus also non-local, package.

But there is a way around: you can define an alias for that type (int, float, …), and then define a method for that type. Or embed the type as an anonymous type in a new struct like in the following example. Of course this method is then only valid for the alias type.

Listing 10.12—method_on_time.go:

```
package main
import (
    "fmt"
    "time"
)

type myTime struct {
    time.Time //anonymous field
}

func (t myTime) first3Chars() string {
    return t.Time.String()[0:3]
}

func main() {
       m := myTime{time.Now()}
       //calling existing String method on anonymous Time field
       fmt.Println("Full time now:", m.String())
       //calling myTime.first3Chars
       fmt.Println("First 3 chars:", m.first3Chars())}
/* Output:
Full time now: Mon Oct 24 15:34:54 Romance Daylight Time 2011
First 3 chars: Mon
*/
```

10.6.2 Difference between a function and a method

A function has the variable as a parameter:	Function1(recv)
A method is called on the variable:	recv.Method1()

A method can change the values (or the state) of the receiver variable provided this is a pointer, just as is the case with functions (a function can also change the state of its parameter when this is passed as a pointer: call by reference).

!! Don't forget the ( ) after Method1, or you get the compiler error: `method recv.Method1 is not an expression, must be called` !!

The receiver must have an explicit name, and this name must be used in the method.

receiver_type is called the *(receiver) base type*, this type must be declared within the same package as all of its methods.

In Go the methods attached to a (receiver) type are not written inside of the structure, as is the case with classes; the coupling is much more loose: the association between method and type is established by the receiver.

Methods are not mixed with the data definition (the structs): they are orthogonal to types; representation (data) and behavior (methods) are independent.

10.6.3 Pointer or value as receiver

recv is most *often a pointer to the* `receiver_type` for performance reasons (because we don't make a copy of the instance, as would be the case with call by value), this is especially true when the receiver type is a struct.

Define the method on a pointer type if you need the method to modify the data the receiver points to. Otherwise, it is often cleaner to define the method on a normal value type.

This is illustrated in the following example `pointer_value.go`: change() receives a pointer to B, and changes its internal field; write() only outputs the contents of the B variable and receives its value by copy. Notice in main() that Go does plumbing work for us, we ourselves do not have to figure out whether to call the methods on a pointer or not, Go does that for us. b1 is a value and b2 is a pointer, but the methods calls work just fine.

`Listing 10.13—pointer_value.go:`

```
package main
import (
        "fmt"
)

type B struct {
        thing  int
}

func (b *B) change() {  b.thing = 1 }

func (b B) write() string { return fmt.Sprint(b) }

func main() {
        var b1 B  // b1 is a value
        b1.change()
        fmt.Println(b1.write())

        b2 := new(B)  // b2 is a pointer
        b2.change()
        fmt.Println(b2.write())
}
/* Output:
{1}
```

```
{1}
*/
```

Try to make write() change its receiver value b: you will see that it compiles fine, but the original b is not changed!

We saw that a method does not require a pointer as a receiver, as in the following example, where we only need the values of Point3 to compute something:

```
type Point3 struct { x, y, z float }
// A method on Point3:
func (p Point3) Abs() float {
        return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
}
```

This is a bit expensive, because Point3 will always be passed to the method by value and so copied, but it is valid Go. In this case the method can also be invoked on a pointer to the type (there is automatic dereferencing).

Suppose `p3` is defined as a pointer: `p3 := &Point3{ 3, 4, 5 }`
Then you can write `p3.Abs() instead of (*p3).Abs()`

And a method with a receiver type `*TwoInts` like `AddThem ()` in listing 10.11 `(method1.go)` can be invoked on an addressable value of type TwoInts; there is automatic indirection.

So `two2.AddThem()` can be used instead of `(&two2).AddThem()`

Calling methods on values and pointers:

There can be methods attached to the type, and other methods attached to a pointer to the type.

*But it does not matter: if for a type T a method Meth() exists on *T and t is a variable of type T, then `t.Meth()` is automatically translated to `(&t).Meth()`*

Pointer and value methods can both be called on pointer or non-pointer values, this is illustrated in the following program, where the type List has a method Len() on the value and a method Append() on a pointer to List, but we see that both methods can be called on variables of both types.

Listing 10.14—methodset1.go:

```
package main
import (
        "fmt"
)

type List []int
func (l List) Len() int { return len(l) }
func (l *List) Append(val int) { *l = append(*l, val) }

func main() {
        // A bare value
        var lst List
        lst.Append(1)
        fmt.Printf("%v (len: %d)\n", lst, lst.Len()) // [1] (len: 1)

        // A pointer value
        plst := new(List)
        plst.Append(2)
        fmt.Printf("%v (len: %d)\n", plst, lst.Len()) // &[2] (len: 1)
}
```

10.6.4 Methods and not-exported fields

Consider the beginning of a package person in `person2.go`: the type Person is clearly exported, but its fields are not! For example the statement p.firstname in `use_person2.go` is an error. How can we change, or even read the name of a Person object in another program ?

This is accomplished by a well known technique from OO-languages: provide getter- and setter-methods FirstName and SetFirstName. For the setter-method we use the prefix Set, for the getter-method we only use the fieldname.

Listing 10.15—person2.go:

```
package person

type Person struct {
        firstName     string
        lastName      string
}
```

```
func (p *Person) FirstName() string {
        return p.firstName
}
func (p *Person) SetFirstName(newName string) {
        p.firstName = newName
}
```

Listing 10.16—use_person2.go:

```
package main
import (
        “fmt”
        “./person”    // package in same map
)

func main() {
        p := new(person.Person)
        // error: p.firstName undefined
// (cannot refer to unexported field or method firstName)
        // p.firstName = “Eric”
        p.SetFirstName(“Eric”)
        fmt.Println(p.FirstName()) // Output: Eric
}
```

Concurrent access to objects:

It should not be possible that the fields (properties) of an object can be changed by 2 or more different threads at the same time. If this can occur in your application then in order to make concurrent acces safe you can use the methods of the package sync (see § 9.3). In § 14.17 we explore an alternative way by using goroutines and channels.

10.6.5 Methods on embedded types and inheritance

When an anonymous type is embedded in a struct, the visible methods of that type are embedded as well—in effect, the outer type *inherits* the methods: *to subtype something, you put the parent type within the subtype.* This mechanism offers a simple way to emulate some of the effects of subclassing and inheritance found in classic OO-languages; it is also very analogous to the *mixins* of Ruby.

Here is an illustrative example (which you can work out further in Exercise 10.8): suppose we have an interface type Engine, and a struct type Car that contains an anonymous field of type Engine:

```
type Engine interface {
        Start()
        Stop()
}

type Car struct {
        Engine
}
```

We could then construct the following code:

```
func (c *Car) GoToWorkIn {
        // get in car
        c.Start();
        // drive to work
        c.Stop();
        // get out of car
}
```

In the following complete example `method3.go` it is shown that a method on an embedded struct can be called directly on an instance of the embedding type.

Listing 10.17—method3.go:

```
package main
import (
        “fmt”
        “math”
)

type Point struct {
        x, y float64
}

func (p *Point) Abs() float64 {
        return math.Sqrt(p.x*p.x + p.y*p.y)
}

type NamedPoint struct {
        Point
        name string
}

func main() {
        n := &NamedPoint{Point{3, 4}, "Pythagoras"}
        fmt.Println(n.Abs()) // prints 5
}
```

Embedding injects fields and methods of an existing type into another type: methods associated with the anonymous field are promoted to become methods of the enclosing type. Of course a type can have methods which act only on variables of that type, not on variables of the embedded 'parent' type.

Also *overriding* (just as with fields) is implemented for methods: a method in the embedding type with the same name as a method in an embedded type overrides this. In `Listing 10.18—method4.go` `we have just added:`

```
func (n *NamedPoint) Abs() float64 {
    return n.Point.Abs() * 100.
}
```

And now the line: `fmt.Println(n.Abs())` prints 500.

Because a struct can embed multiple anonymous types, we have in effect a simple version of *multiple inheritance*, like in: `type Child struct { Father; Mother }`

This is further explored in § 10.6.7

Structs embedding structs from the same package have full access to one another's fields and methods.

Exercise 10.8: `inheritance_car.go`

Make a working example for the Car and Engine code above; also give the Car type a field wheelCount and a method numberOfWheels() to retrieve this.

Make a type Mercedes which embeds Car, an object of type Mercedes and use the methods.

Then construct a method sayHiToMerkel() only on type Mercedes and invoke it.

10.6.6 How to embed functionality in a type

There are basically 2 ways for doing this:

- A *Aggregation* (or *composition*): include a named field of the type of the wanted functionality
- B *Embedding*: Embed (anonymously) the type of the wanted functionality, like demonstrated in the previous § 10.6.5

To make it concrete suppose we have a type Customer and we want to include a Logging functionality with a type Log, which simply contains an accumulated message (of course this could be elaborated). If you want to equip all your domain types with a logging capability, you implement such a Log and add it as a field to your type as well as a method Log() returning a reference to this log.

Way A could be implemented as follows (we use the String() functionality from § 10.7):

Listing 10.19—embed_func1.go:

```
package main
import (
        "fmt"
)

type Log struct {
        msg string
}

type Customer struct {
    Name string
    log  *Log
}

func main() {
        c := new(Customer)
        c.Name = "Barak Obama"
        c.log = new(Log)
        c.log.msg = "1 - Yes we can!"
        // shorter:
        c := &Customer{"Barak Obama", &Log{"1 - Yes we can!"}}
        // fmt.Println(c)     // &{Barak Obama 1 - Yes we can!}
        c.Log().Add("2 - After me the world will be a better place!")
        //fmt.Println(c.log)
        fmt.Println(c.Log())
}

func (l *Log) Add(s string) {
        l.msg += "\n" + s
}

func (l *Log) String() string {
        return l.msg
}

func (c *Customer) Log() *Log {
        return c.log
}
/* Output of the log:
1 - Yes we can!
2 - After me the world will be a better place!
*/
```

Way B on the contrary would be like:

Listing 10.20—embed_func2.go:

```
package main
import (
        "fmt"
)

type Log struct {
        msg string
}

type Customer struct {
    Name string
    Log
}

func main() {
        c := &Customer{"Barak Obama", Log{"1 - Yes we can!"}}
        c.Add("2 - After me the world will be a better place!")
        fmt.Println(c)
}

func (l *Log) Add(s string) {
        l.msg += "\n" + s
}

func (c *Customer) String() string {
        return c.Name + "\nLog:" + fmt.Sprintln(c.Log)
}

func (l *Log) String() string {
  return l.msg
}
/* Output:
Barak Obama
Log:{1 - Yes we can!
2 - After me the world will be a better place!}
*/
```

The embedded type does not need to a pointer, Customer does not need an Add method anymore because it uses the method from Log, Customer can have its own String-method and using in it the log String() method.

This also works if the embedded type itself embeds another type, and the methods of the embedded type can be used directly by the embedding type.

So a good strategy is to build small reusable types as a toolbox for the composition of the own domain types.

10.6.7 Multiple inheritance

Multiple inheritance is the ability for a type to obtain the behaviors of more than one parent class. In classic OO languages it is usually not implemented (exceptions are C++ and Python), because in class-based hierarchies it introduces additional complexities for the compiler. But in Go it can be implemented simply by embedding all the necessary 'parent' types in the type under construction.

As an example suppose you want to have a type CameraPhone, with which you can Call() and with which you can TakeAPicture(), but the first method belongs to type Phone, and the second to type Camera.

Embedding both types solves the problem, as is illustrated in the following program:

Listing 10.21—mult_inheritance.go:

```
package main
import “fmt”

type Camera struct { }

func (c *Camera) TakeAPicture() string {
        return “Click”
}

type Phone struct { }

func (p *Phone ) Call() string {
        return “Ring Ring”
}

// multiple inheritance
type CameraPhone struct {
        Camera
        Phone
}

func main() {
        cp := new(CameraPhone)
        fmt.Println(“Our new CameraPhone exhibits multiple behaviors ...”)
        fmt.Println(“It exhibits behavior of a Camera: “, cp.TakeAPicture())
        fmt.Println(“It works like a Phone too: “, cp.Call())
}
/* Output:
Our new CameraPhone exhibits multiple behaviors ...
It exhibits behavior of a Camera: Click
It works like a Phone too: Ring Ring
*/
```

EXERCISES:

Exercise 10.9: `point_methods.go:`

Start from point.go (exercise from § 10.1): now implement the functions Abs() and Scale() as methods with the receiver type Point. Also implement Abs() for Point3 and Polar as methods. Do the same things as in point.go, but now use the methods.

Exercise 10.10: `inherit_methods.go`

Define a struct type Base which contains an id-field, and methods Id() to return the id and SetId() to change the id. The struct type Person contains Base, and fields FirstName and LastName. The struct type Employee contains a Person and a salary.

Make an employee instance and show its id.

Exercise 10.11: `magic.go`

Predict the outcome and then try out the following program:

```
package main
import "fmt"

type Base struct{}

func (Base) Magic() { fmt.Print("base magic ") }

func (self Base) MoreMagic() {
       self.Magic()
       self.Magic()
}

type Voodoo struct {
       Base
}

func (Voodoo) Magic() { fmt.Println("voodoo magic") }
func main() {
	v := new(Voodoo)
	v.Magic()
	v.MoreMagic()
}
```

10.6.8 Universal methods and method naming

In programming a number of basic operations appear over and over again, like opening, closing, reading, writing, sorting, etc. Moreover they have a general meaning to them: opening can be applied for a file, a network connection, a database connection, etc. The implementation details are in each case very different, but the general idea is the same. In Go this is applied extensively in the standard library through the use of interfaces (see Chapter 11), wherein such generic methods get canonical names as Open(), Read(), Write(), etc. If you want to write idiomatic Go, you should follow this convention, giving your methods where appropriate the same names and signatures as those 'canonical' methods. This makes Go software more consistent and readable. For example: if you need a convert-to-string method, name it String() and not ToString() (see § 10.7).

10.6.9 Comparison between Go types and methods and other object-oriented languages.

Methods in OO languages like C++, Java, C# or Ruby are defined in the context of classes and inheritance: when a method is invoked on an object, the runtime sees whether its class ore any of its superclasses have a definition for that method, otherwise an exception results.

In Go such an inheritance hierarchy is not at all needed: if the method is defined for that type it can be invoked, independent of whether or not the method exists for other types; so in that sense there is a greater flexibility.

This is nicely illustrated in the following schema:

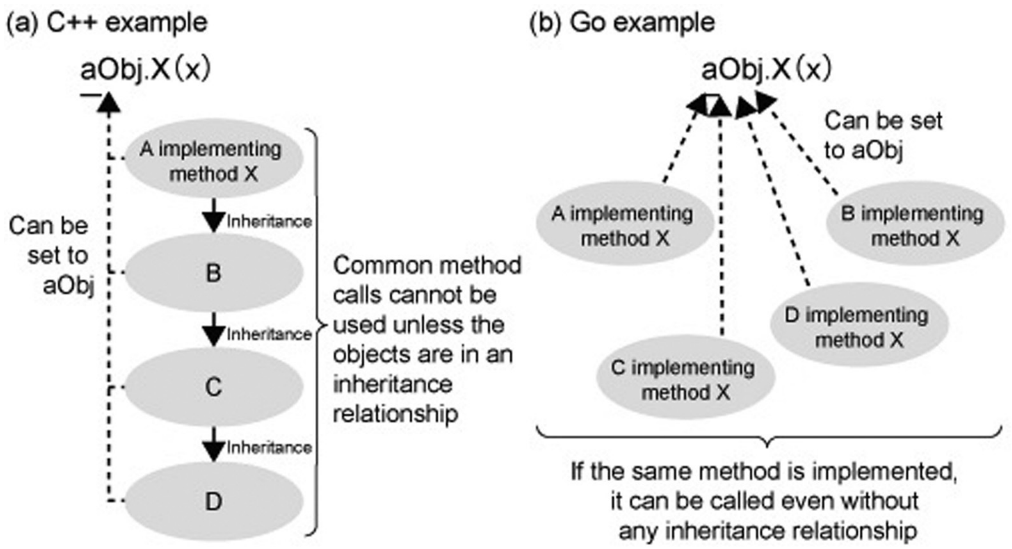


Fig 10.4: Methods in Go and OO-languages

Go doesn't require an explicit class definition as Java, C++, C#, etc do. Instead, a "class" is implicitly defined by providing a set of methods which operate on a common type. This type may be a struct or any other user-defined type.

For example: say we would like to define our own Integer type to add some possible methods for working with integers, like a tostring-conversion. In Go we would define this as:

```
type Integer int
func (i *Integer) String() string {
        return strconv.Itoa(i)
}
```

In Java or C# the type and the func would be placed together in a class Integer; in Ruby you could just write the method on the basic type int.

Summarized: in Go types are basically classes (data and associated methods). Go doesn't know inheritance like class oriented OO languages. Inheritance has two main benefits: code reuse and polymorphism.

Code reuse in Go is achieved through composition and delegation, and polymorphism through the use of interfaces: it implements what is sometimes called *component programming*.

Many developers say that Go's interfaces provide a more powerful and yet simpler polymorphic behaviour than class inheritance.

Remark: If you really need more OO capabilities, take a look at the **goop**—package ("Go Object-Oriented Programming") from Scott Pakin (https://github.com/losalamos/goop): it provides Go with JavaScript-style objects (prototype-based objects) but supports multiple inheritance and type-dependent dispatch so you can probably implement most of your favorite constructs from other programming languages.

Question 10.1: We call a method on a variable of a certain type with the dot-notation: `variable.method( )`; where have we encountered that OO dot notation before in Go ?

Question 10.2: a) Suppose we define: `type Integer int`
Fill in the body of the get() function: func `(p Integer) get() int { … }`

b) Defined are: `func f(i int) { }`
`var v Integer`
How would you call f with as parameter v ?

c) Suppose Integer is define as : `type Integer struct { n int }`
Now fill in the body of the get() function: func `(p Integer) get() int { … }`

d) Same question as in b) for the Integer struct type.

10.7 The String()-method and format specifiers for a type

When you define a type with a lot of methods, chances are you will want to make a customized string-output for it with a String( ) method, in other words: a human-readable and printable output. This is because if String( ) is defined for a certain type, then this method will be used in fmt.Printf() to produce the default output: the one which is produced with the format specifier %v. Also fmt.Print() and fmt.Println() will automatically use the String( ) method.

We test this out with the type of the program in § 10.4, Listing 10.22—method_string.go:

```
package main
import (
        "fmt"
        "strconv"
)

type TwoInts struct {
         a int
         b int
}

func main() {
         two1 := new(TwoInts)
         two1.a = 12
         two1.b = 10
         fmt.Printf("two1 is: %v\n", two1)   // output: two1 is: (12 / 10)
         fmt.Println("two1 is:", two1)       // output: two1 is: (12 / 10)
         fmt.Printf("two1 is: %T\n", two1)
                // output: two1 is: *main.TwoInts
         fmt.Printf("two1 is: %#v\n", two1)
                // output: &main.TwoInts{a:12, b:10}
}

func (tn *TwoInts) String() string {
         return "("+ strconv.Itoa(tn.a) +" / "+ strconv.Itoa(tn.b) + ")"
}
```

So whenever you will be using extensively a certain type that you made yourself, it is convenient to make a String( )-method for it. We also see that the format specifier **%T** gives us the complete type specification and **%#v** gives us a complete output of the instance with its fields (it can also be useful in generating Go code programmatically).

Remark: Don't make the mistake of defining String() in terms of itself, like in the following snippet. Then the program does an infinite recursion (TT.String() calls fmt.Sprintf which calls TT.String() …) and quickly gives an out of memory error:

```
type TT float64

func (t TT) String() string {
       return fmt.Sprintf("%v", s)
}
t.String()
```

EXERCISES:

Exercise 10.12: `type_string.go:`

Given a struct type T:
```
type T struct {
        a int
        b float32
        c string
}
```
and a value t: `t := &T{ 7, -2.35, "abc\tdef" }`
make a String() method for T so that for `fmt.Printf("%v\n", t)`
the following format is send to the output: `7 / -2.350000 / "abc\tdef"`

Exercise 10.13: `celsius.go`

Make an alias type Celsius for float64 and define a String() method for it which prints out the temperature with 1 decimal and °C.

Exercise 10.14: `days.go:`

Make an alias type Day for int. Define an array of strings with the daynames.
Define a String() method for type Day which shows the dayname.
Make an enum const type with iota for all the days of the week (MO, TU, …)

Exercise 10.15: `timezones.go:`

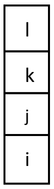
Make an alias type TZ for int. Define a few constants which define timezones like UTC and a map which maps these abbreviations to the full name, like:

UTC -> "Universal Greenwich time"

Define a String() method for type TZ which shows the full name of the timezone.

Exercise 10.16: `stack_arr.go / stack_struct.go`

Implement the *stack* datastructure:



It has cells to contain data, for example integers i, j, k, l, etc. the cells are indexed from the bottom (index 0) to the top (index n). Let's assume n=3 for this exercise, so we have 4 places.

A new stack contains 0 in all cells.

A new value is put in the highest cell which is empty (contains 0), on top: this is called *push.*

To get a value from the stack, take the highest value which is not 0: this is called *pop.* So we can understand why a stack is called a Last In First Out (LIFO) structure.

Define a new type Stack for this datastructure. Make 2 methods Push and Pop. Make a String() method (for debugging purposes) which shows the content of the stack as: [0:i] [1:j] [2:k] [3:l]

(1) take as underlying data structure an array of 4 ints: `stack_arr.go`
(2) take as underlying data structure a struct containing an index and an array of int, the index contains the first free position: `stack_struct.go`
(3) generalize both implementations by making the number of elements 4 a constant LIMIT.

10.8 Garbage collection and SetFinalizer

The Go developer doesn't have to code the release of memory for variables and structures which are not used anymore in the program. A separate process in the Go runtime, the garbage collector, takes care of that. It starts now and then, searches for variables which are not listed anymore and frees that memory. Functionality regarding this process can be accessed via the runtime package.

Garbage collection can be called explicitly by invoking the function `runtime.GC()`, but this is only useful in rare cases, e.g. when memory resources are scarce, a great chunk of memory could immediately be freed at that point in the execution, and the program can take a momentary decrease in performance (because of the garbage collection-process).

If you want to know the current memory status, use:

```
	fmt.Printf("%d\n", runtime.MemStats.Alloc/1024)
```

This will give you the amount of allocated memory by the program in Kb. For further measurements: see http://golang.org/pkg/runtime/#MemStatsType

Suppose special action needs to be taken right before an object obj is removed from memory, like writing to a log-file. This can be achieved by calling the function:

```
runtime.SetFinalizer(obj, func(obj *typeObj))
```

where func(`obj *typeObj`) is a function which takes a pointer-parameter of the type of obj which performs the additional action. func could also be an anonymous function.

`SetFinalizer` does not execute when the program comes to an normal end or when an error occurs, before the object was chosen by the garbage collection process to be removed.

Exercise 10.17: Starting from exercise 10.13 (the struct implementation of a stack datastructure), create a separate package stack for the stack implementation (`stack_struct.go`) and call it from a main program `main_stack.go`

Chapter 11—Interfaces and reflection

11.1 What is an interface?

Go is not a 'classic' OO language: it doesn't know the concept of classes and inheritance.

However it does contain the very flexible concept of *interfaces*, with which a lot of aspects of object-orientation can be made available. Interfaces in Go provide a way to *specify the behavior* of an object: if something can do *this*, then it can be used *here*.

An interface defines a set of methods (the *method set*), but these methods do not contain code: they are not implemented *(* they are *abstract)*. Also an interface cannot contain variables.

An interface is declared in the format:

```
type Namer interface {
        Method1(param_list) return_type
        Method2(param_list) return_type
        …
}
```

`where Namer` is an *interface type.*

The name of an interface is formed by the method name plus the [e]r suffix, such as Printer, Reader, Writer, Logger, Converter, etc., thereby giving an active noun as a name. A less used alternative (when ..er is not so appropriate) is to end it with able like in Recoverable, or to start it with an I (more like in .NET or Java) .

Interfaces in Go are short, they usually have from 0—max 3 methods.

Unlike in most OO languages, in Go interfaces can have values, a variable of the interface type or an *interface value:* `var ai Namer` ai is a multiword data structure with an uninitialized value of nil. Allthough not completely the same thing, it is in essence a pointer. So pointers to interface values are illegal; they would be completely useless and give rise to errors in code.

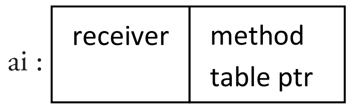


Fig 11.1: Interface value in memory

Its table of method pointers is build through the runtime reflection capability.

Types (like structs) can have the method set of the interface implemented; the implementation contains for each method real code how to act on a variable of that type: *they implement the interface*, the method set forms the interface of that type. A variable of a type that implements the interface can be assigned to ai (the receiver value), the method table then has pointers to the implemented interface methods. Both of these of course change when a variable of another type (that also implements the interface) is assigned to ai.

A type doesn't have to state explicitly that it implements an interface: interfaces are satisfied implicitly. Multiple types can implement the same interface.

A type that implements an interface can also have other functions.

A type can implement many interfaces.

An interface type can contain a reference to an instance of any of the types that implement the interface (an interface has what is called a dynamic type)

Even if the interface was defined later than the type, in a different package, compiled separately: if the object implements the methods named in the interface, then it implements the interface.

All these properties allow for a lot of flexibility.

As a first example, look at `Listing 11.1—interfaces.go:`

```
package main
import "fmt"


type Shaper interface {
        Area() float32
}
type Square struct {
        side float32
}

func (sq *Square) Area() float32 {
        return sq.side * sq.side
}

func main() {
        sq1 := new(Square)
        sq1.side = 5

        // var areaIntf Shaper
        // areaIntf = sq1
        // shorter, without separate declaration:
        // areaIntf := Shaper(sq1)
        // or even:
        areaIntf := sq1
        fmt.Printf("The square has area: %f\n", areaIntf.Area())
}
```

Output: The square has area: 25.000000

The program defines a struct Square and an interface `Shaper`, with one method Area().

In main() an instance of Square is constructed. Outside of main we have an Area() method with a receiver type of Square where the area of a square is calculated: the struct Square implements the interface Shaper.

Because of this we can assign a variable of type Square to a variable of the interface type:
`areaIntf = sq1`

Now the interface variable contains a reference to the Square variable and through it we can call the method Area() on Square. Of course you could call the method immediately on the Square instance sq1.Area(), but the novel thing is that we can call it on the interface instance, thereby generalizing the call. The interface variable both contains the value of the receiver instance and a pointer to the appropriate method in a method table.

This is Go's version of *polymorphism*, a well known concept in OO software: the right method is chosen according to the current type, or put otherwise: a type seems to exhibit different behaviors when linked to different instances.

If Square would not have an implementation of Area(), we would receive the very clear compiler error:

```
cannot use sq1 (type *Square) as type Shaper in assignment:
*Square does not implement Shaper (missing Area method)
```

The same error would occur if Shaper had another method Perimeter(), and Square would not have an implementation for that, even if Perimeter() was not called on a Square instance.

We know expand the example with a type Rectangle which also implements Shaper. We can now make an array with elements of type Shaper, and show polymorphism in action by using a for range on it and calling Area() on each item:

Listing 11.2—interfaces_poly.go:

```
package main
import “fmt”

type Shaper interface {
        Area() float32
}

type Square struct {
        side float32
}

func (sq *Square) Area() float32 {
        return sq.side * sq.side
}

type Rectangle struct {
   length, width float32
}

func (r Rectangle) Area() float32 {
   return r.length * r.width
}
```

```
func main() {
   r := Rectangle{5, 3} // Area() of Rectangle needs a value
   q := &Square{5      // Area() of Square needs a pointer
   // shapes := []Shaper{Shaper(r), Shaper(q)}
   // or shorter:
   shapes := []Shaper{r, q, c}
   fmt.Println("Looping through shapes for area ...")
   for n, _ := range shapes {
       fmt.Println("Shape details: ", shapesArr[n])
       fmt.Println("Area of this shape is: ", shapes[n].Area())
   }
}
/* Output:
Looping through shapes for area ...
Shape details:  {5 3}
Area of this shape is:  15
Shape details:  &{5}
Area of this shape is:  25
*/
```

At the point of calling `shapes[n].Area())` we only know that this is a Shaper object, under the cover it 'morphs' into a Square or a Rectangle and behaves accordingly.

Perhaps you can now begin to see how interfaces can produce cleaner, simpler, and more scalable code. In § 11.12.3 we will see how easy it is to add new interfaces for our types later on in the development.

Here is a another more concrete example: we have 2 types stockPosition and car, both have a method getValue(); realizing that we can define an interface `valuable` with this method. And the we can define methods that take a parameter of type valuable and which are usable by all types that implement this interface, like showValue():

Listing 11.3—valuable.go:

```
package main
import "fmt"

type stockPosition struct {
     ticker string
     sharePrice float32
     count float32
}

/* method to determine the value of a stock position */
func (s stockPosition) getValue() float32 {
     return s.sharePrice * s.count
}

type car struct {
     make string
     model string
     price float32
}

/* method to determine the value of a car */
func (c car) getValue() float32 {
     return c.price
}
/* contract that defines different things that have value */
type valuable interface {
     getValue() float32
}

/* anything that satisfies the "valuable" interface is accepted */
func showValue(asset valuable) {
     fmt.Printf("Value of the asset is %f\n", asset.getValue())
}

func main() {
     var o valuable = stockPosition{ "GOOG", 577.20, 4 }
     showValue(o)
     o = car{ "BMW", "M3", 66500 }
     showValue(o)
}
```

```
/* Output:
Value of the asset is 2308.800049
Value of the asset is 66500.000000
*/
```

An example from the standard library:

The io package contains an interface type Reader:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

If we define a variable r as: `var r io.Reader`
then the following is correct code:

```
r = os.Stdin                    // see § 12.1
r = bufio.NewReader(r)
r = new(bytes.Buffer)
f, _ := os.Open("test.txt")
r = bufio.NewReader(f)
```

because the right-hand side objects each implement a Read() method with the exact same signature. The static type of r is `io.Reader`.

Remark:
Sometimes the word interface is also used in a slightly different way: seen from the standpoint of a certain type, the interface of that type is the set of exported methods defined for that type, without there having to be an explicit interface type defined with these methods.

Exercise 11.1: `simple_interface.go`
Define an interface Simpler with methods Get() which returns an integer, and Set() which has an integer as parameter. Make a struct type Simple which implements this interface.

Then define a function which takes a parameter of the type Simpler and calls both methods upon it. Call this function from main to see if it all works correctly.

Exercise 11.2-3: `interfaces_poly2.go`

a) Expand the example interfaces_poly.go to a type Circle: interfaces_poly2.go
b) Now we will implement the same functionality by using an 'abstract' type Shape (abstract because it has no fields) which implements Shaper, and embedding his type in the other types. Now demonstrate that overriding is used as explained in § 10.6.5: `interfaces_poly3.go`

11.2 Interface embedding interface(s)

An interface can contain the name of one (or more) other interface(s), which is equivalent to explicitly enumerating the methods of the embedded interface in the containing interface.

For example interface File contains all the methods of ReadWrite and Lock, in addition to a Close() method.

```
type ReadWrite interface {
        Read(b Buffer) bool
        Write(b Buffer) bool
}

type Lock interface {
        Lock()
        Unlock()
}

type File interface {
        ReadWrite
        Lock
        Close()
}
```

11.3 How to detect and convert the type of an interface variable: type assertions

An interface type variable varI can contain a value of any type; we must have a means to detect this *dynamic* type, that is the actual type of the value stored in the variable at run time. The dynamic type may vary during execution, but is always assignable to the type of the interface variable itself. In general we can test if `varI` contains at a certain moment a variable of type T with the *type assertion* test:

```
v := varI.(T)                  // unchecked type assertion
```

varI must be an interface variable, if not the compiler signals the error: `invalid type assertion: varI.(T) (non-interface type (type of varI) on left)`

A type assertion may not be valid; the compiler does its utmost best to see if the conversion is valid, but cannot foresee all possible cases. If this conversion fails while running the program a runtime error occurs! A safer way is to use the following form:

```
if v, ok := varI.(T); ok {     // checked type assertion
      Process(v)
    return
}
// varI is not of type T
```

If this conversion is valid, v will contain the value of varI converted to type T and ok will be true, otherwise v is the zero value for T and ok is false, so no runtime error occurs!

!! Always use the comma, ok form for type assertions !!

In most cases you would want to test the value of ok in an if; then it is most convenient to use the form:

```
if v, ok := varI.(T); ok {
        // …
}
```

In this form shadowing the variable varI by giving varI and v the same name is sometimes done.

An example can be seen in Listing 11.4—type_interfaces.go:

```
package main
import (
        "fmt"
        "math"
)

type Square struct {
        side float32
}

type Circle struct {
        radius float32
}

type Shaper interface {
        Area() float32
}
```

```
func main() {
        var areaIntf Shaper
        sq1 := new(Square)
        sq1.side = 5

        areaIntf = sq1
        // Is Square the type of areaIntf ?
        if t, ok := areaIntf.(*Square); ok {
               fmt.Printf("The type of areaIntf is: %T\n", t)
        }
        if u, ok := areaIntf.(*Circle); ok {
               fmt.Printf("The type of areaIntf is: %T\n", u)
        } else {
               fmt.Println("areaIntf does not contain a variable of type
               Circle")
        }
}

func (sq *Square) Area() float32 {
        return sq.side * sq.side
}

func (ci *Circle) Area() float32 {
        return ci.radius * ci.radius * math.Pi
}
Output:  The type of areaIntf is: *main.Square
         areaIntf does not contain a variable of type Circle
         Type Square *main.Square with value &{5}
```

A new type Circle is defined, which also implements Shaper. In the line `t, ok := areaIntf.(*Square)` we test that areaIntf contains a variable of type Square, this is the case; then we test that it does not contain a variable of type Circle.

Remark: if we omit the * in `areaIntf.(*Square)` we get the compiler-error:

```
impossible type assertion: areaIntf (type Shaper) cannot have dynamic type
Square (missing Area method)
```

11.4 The type switch

The type of an interface variable can also be tested with a special kind of switch: the *type-switch* (this is the 2nd part of Listing 11.2):

```
switch t := areaIntf.(type) {
case *Square:
fmt.Printf("Type Square %T with value %v\n", t, t)
case *Circle:
fmt.Printf("Type Circle %T with value %v\n", t, t)
case float32:
fmt.Printf("Type float32 with value %v\n", t)
case nil:
fmt.Println("nil value: nothing to check?")
default:
fmt.Printf("Unexpected type %T", t)
}
```

```
Output:  Type Square *main.Square with value &{5}
```

The variable t receives both value and type from areaIntf. All of the listed types (except nil) have to implement the interface (Shaper in this case); if the current type is none of the case-types, the default clause is executed.

Fallthrough is not permitted. With a type-switch a runtime type analysis can be done.

Of course all the built-in types as int, bool and string can also be tested in a type switch.

If you only need to test the type of the variable in the switch and don't need the value, the assignment can be left out, like:

```
switch areaIntf.(type) {
case *Square:
        fmt.Printf("Type Square %T with value %v\n", t, t)
case *Circle:
        fmt.Printf("Type Circle %T with value %v\n", t, t)
        …
default:
        fmt.Printf("Unexpected type %T", t)
}
```

In the following code snippet a *type classifier* function is shown which accepts an array with a variable number of arguments of any type, and that executes something according to the determined type:

```
func classifier(items ...interface{}) {
    for i, x := range items {
        switch x.(type) {
        case bool:       fmt.Printf("param #%d is a bool\n", i)
        case float64:    fmt.Printf("param #%d is a float64\n", i)
        case int, int64: fmt.Printf("param #%d is an int\n", i)
        case nil:        fmt.Printf("param #%d is nil\n", i)
        case string:     fmt.Printf("param #%d is a string\n", i)
        default:         fmt.Printf("param #%d's type is unknown\n", i)
        }
    }
}
```

This function could be called e.g. as `classifier(13, -14.3, "BELGIUM", complex(1, 2),nil, false).`

When dealing with data of unknown type from external sources type testing and conversion to Go data types can be very useful, e.g. parsing data that are JSON- or XML-encoded.

In listing 12.17 (xml.go) we use a type switch while parsing an XML-document.

Exercise 11.4: `simple_interface2.go`

Continuing with exercise 11.1, make a second type RSimple which also implements the interface Simpler.

Expand the function `fI` so that it can distinguish between variables of both types.

11.5 Testing if a value implements an interface

This is a special case of the type assertion from § 11.3: suppose v is a value and we want to test whether it implements the Stringer interface, this can be done as follows:

```
type Stringer interface { String() string }
if sv, ok := v.(Stringer); ok {
        fmt.Printf("v implements String(): %s\n", sv.String()); // note: sv, not v
}
```

This is how the Print functions check if the type can print itself.

An interface is a kind of contract which the implementing type(s) must fulfill. Interfaces describe the behavior of types, what they can do. They completely separate the definition of what an object can do from how it does it, allowing distinct implementations to be represented at different times by the same interface variable, which is what polymorphism essentially is.

Writing functions so that they accept an interface variable as a parameter makes them more general.

!!   Use interfaces to make your code more generally applicable   !!

This is also ubiquitously applied in the code of the standard library. It is impossible to understand how it is build without a good grasp of the interface-concept.

In the following paragraphs we discuss 2 important examples. Try to understand them deep enough so that you can apply this principle as well.

11.6 Using method sets with interfaces

In § 10.6.3 and example program `methodset1.go` we saw that methods on variables in fact do not distinguish between values or pointers. When storing a value in an interface type it is slightly more complicated because a concrete value stored in an interface is not addressable, but luckily the compiler flags an error on improper use. Consider the following program:

Listing 11.5—methodset2.go:

```
package main
import (
        “fmt”
)

type List []int
func (l List) Len() int { return len(l) }
func (l *List) Append(val int) { *l = append(*l, val) }

type Appender interface {
        Append(int)
}

func CountInto(a Appender, start, end int) {
```

```
         for i := start; i <= end; i++ {
                a.Append(i)
         }
}

type Lener interface {
         Len() int
}

func LongEnough(l Lener) bool {
         return l.Len()*10 > 42
}

func main() {
// A bare value
         var lst List
// compiler error:
// cannot use lst (type List) as type Appender in function argument:
// List does not implement Appender (Append method requires pointer       // receiver)
         // CountInto(lst, 1, 10)
         if LongEnough(lst) {  // VALID: Identical receiver type
                fmt.Printf("- lst is long enough")
         }

         // A pointer value
         plst := new(List)
         CountInto(plst, 1, 10) // VALID: Identical receiver type
         if LongEnough(plst) {
         // VALID: a *List can be dereferenced for the receiver
         fmt.Printf("- plst is long enough")
         // - plst2 is long enoug
         }
}
```

<u>Discussion:</u>

CountInto called with the value lst gives a compiler error because CountInto takes an Appender, and Append() is only defined for a pointer. LongEnough on value lst works because Len() is defined on a value.

CountInto called with the pointer plst works because CountInto takes an Appender, and Append() is defined for a pointer. LongEnough on pointer plst works because a pointer can be dereferenced for the receiver.

Summarized: when you call a method on an interface, it must either have an identical receiver type or it must be directly discernible from the concrete type: P

- Pointer methods can be called with pointers.
- Value methods can be called with values.
- Value-receiver methods can be called with pointer values because they can be dereferenced first.
- Pointer-receiver methods cannot be called with values, however, because the value stored inside an interface has no address.

When assigning a value to an interface, the compiler ensures that all possible interface methods can actually be called on that value, and thus trying to make an improper assignment will fail on compilation.

11.7 1st example: sorting with the Sorter interface

A very good example comes from the Go-library itself, namely the package sort. To sort a collection of numbers and strings, you only need the number of elements `Len()`, a way to compare items i and j `Less(i, j)` and a method to swap items with indexes i and j `Swap(i, j)`

The Sort-function in sort has an algorithm that only uses these methods on a collection data (to implement it we use here a bubble sort (see Ex. 7.12) but any sort-algorithm could be used):

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

so Sort can accept a general parameter of an interface type `Sorter` which declares these methods:

```
type Sorter interface {
        Len() int
        Less(i, j int) bool
        Swap(i, j int)
}
```

The type int in Sorter does not mean that the collection data must contain ints, i and j are integer indices, and the length is also an integer.

Now if we want to be able to sort an array of ints, all we must do is to define a type and implement the methods of `Interface`:

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

Here is the code to call the sort-functionality in a concrete case:

```
data := []int{74, 59, 238, -784, 9845, 959, 905, 0, 0, 42, 7586, -5467984, 7586}
a := sort.IntArray(data) //conversion to type IntArray from package sort
sort.Sort(a)
```

The complete working code can be found in `sort.go and sortmain.go.`

To illustrate the power of the concept, the same principle is applied for an array of floats, of strings, and an array of structs `dayArray` representing the days of the week.

Listing 11.6—sort.go:

```
package sort

type Sorter interface {
        Len() int
        Less(i, j int) bool
        Swap(i, j int)
}

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

```
                }
        }
}
func IsSorted(data Sorter) bool {
        n := data.Len()
        for i := n - 1; i > 0; i-- {
                if data.Less(i, i-1) {
                        return false
                }
        }
        return true
}

// Convenience types for common cases
type IntArray []int
func (p IntArray) Len() int           { return len(p) }
func (p IntArray) Less(i, j int) bool { return p[i] < p[j] }
func (p IntArray) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }

type StringArray []string
func (p StringArray) Len() int              { return len(p) }
func (p StringArray) Less(i, j int) bool    { return p[i] < p[j] }
func (p StringArray) Swap(i, j int)         { p[i], p[j] = p[j], p[i] }
// Convenience wrappers for common cases
func SortInts(a []int          { Sort(IntArray(a)) }
func SortStrings(a []string)   { Sort(StringArray(a)) }

func IntsAreSorted(a []int) bool            { return IsSorted(IntArray(a)) }
func StringsAreSorted(a []string) bool      { return IsSorted(StringArray(a)) }
```

Listing 11.7—sortmain.go:

```
package main
import (
        “fmt”
        “./sort”
)

func ints() {
data := []int{74, 59, 238, -784, 9845, 959, 905, 0, 0, 42, 7586, -5467984, 7586}
        a := sort.IntArray(data) //conversion to type IntArray
        sort.Sort(a)
        if !sort.IsSorted(a) {
               panic("fail")
        }
        fmt.Printf("The sorted array is: %v\n", a)
}

func strings() {
        data := []string{"monday", "friday", "tuesday", "wednesday", "sunday",
        "thursday", "", "saturday"}
        a := sort.StringArray(data)
        sort.Sort(a)
        if !sort.IsSorted(a) {
               panic("fail")
        }
        fmt.Printf("The sorted array is: %v\n", a)
}
type day struct {
        num            int
        shortName      string
        longName       string
}

type dayArray struct {
        data []*day
}

func (p *dayArray) Len() int            { return len(p.data) }
func (p *dayArray) Less(i, j int) bool  { return p.data[i].num < p.data[j].num }
func (p *dayArray) Swap(i, j int)       { p.data[i], p.data[j] = p.data[j],
p.data[i] }

func days() {
        Sunday :=    day{0, "SUN", "Sunday"}
        Monday :=    day{1, "MON", "Monday"}
        Tuesday :=   day{2, "TUE", "Tuesday"}
        Wednesday := day{3, "WED", "Wednesday"}
        Thursday :=  day{4, "THU", "Thursday"}
        Friday :=    day{5, "FRI", "Friday"}
        Saturday :=  day{6, "SAT", "Saturday"}
        data := []*day{&Tuesday, &Thursday, &Wednesday, &Sunday, &Monday,
        &Friday, &Saturday}
        a := dayArray{data}
        sort.Sort(&a)
        if !sort.IsSorted(&a) {
               panic(“fail”)
        }
        for _, d := range data {
               fmt.Printf(“%s “, d.longName)
        }
        fmt.Printf(“\n”)
}

func main() {
    ints()
    strings()
    days()
}
```

```
Output:
The sorted array is: [-5467984 -784 0 0 42 59 74 238 905 959 7586 7586 9845]
The sorted array is: [ friday monday saturday sunday thursday tuesday wednesday]
Sunday Monday Tuesday Wednesday Thursday Friday Saturday
```

Remark: `panic(“fail”)` is a way to stop the program in a situation which could not occur in common circumstances (see chapter 13 for details of its use); we could also have printed a message and then used os.Exit(1).

This example has given us a better insight in the significance and use of interfaces. For sorting the primitive types we know that we don't have to write this code ourselves, the standard library provides for this (§ 7.6.6). For general sorting the sort package defines a

```
type Interface interface {
        Len() int
        Less(i, j int) bool
        Swap(i, j int)
}
```

which sums up the abstract methods needed for sorting, and a func `Sort(data Interface)` that can act on such a type. These can be used in implementing sorting for other kinds of data. In fact that is exactly what we did in the example above, using this for ints and strings, but also for a user defined type dayArray, to sort an internal array of strings.

<u>Exercise 11.5:</u> `interfaces_ext.go:` a) Expanding the same program, define a type Triangle, and let it implement AreaInterface. Test this by calculation the area of a specific triangle (area of a triangle = 0.5 * (base * height))

b) Define a new interface PeriInterface which defines a method Perimeter(). Let type Square also implement this interface and test it with our square instance.

<u>Exercise 11.6:</u> `point_interfaces.go:` Continue working on the exercise point_methods.go of §10.3. Define an interface Magnitude with a function Abs( ). With the methods from §10.3, Point, Point3 and Polar implement that interface. Do the same things as in point.go, but now use the methods through a variable of the interface type.

<u>Exercise 11.7:</u> `float_sort.go / float_sortmain.go`

Analogous as in § 11.7 and Listings 11.3/4 define a package float64 and in it a type Float64Array, and let this type implement the Sorter interface to sort an array of float64.

Also provide the following methods:

- a NewFloat64Array-method for making a variable with 25 elements (see § 10.2)
- a List()-method returning a string for pretty-printing the array, wrap this in a String()-method so that you don't have to call List() explicitly (see § 10.7)
- a Fill() method for creating such an array with 10 random floats (see § 4.5.2.6)

In the main-program create a variable of that type, sort it and test that.

<u>Exercise 11.8:</u> `sort.go / sort_persons.go`

Define a struct Person with firstName and LastName, and a type Persons as a [ ]Person.

Implement the Sorter interface for Persons and test it.

11.8 2nd example: Reading and Writing

Reading and writing are universal activities in software: reading and writing files comes to mind first, reading and writing to buffers (e.g. to slices of bytes or to strings), and to the standard input, output, and error streams, network connections, pipes, etc.—or to our own custom types. To make the codebase as generic as possible, Go takes a consistent approach to reading and writing data.

The package io provides us with the interfaces for reading an writing, io.Reader and io.Writer:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}

type Writer interface {
        Write(p []byte) (n int, err error)
  }
```

You can read from and write to any type as long as the type provides the methods Read() and Write(), necessary to satisfy the reading and writing interfaces. For an object to be readable it must satisfy the io.Reader interface. This interface specifies a single method with signature, Read([]byte) (int, error). The Read() method reads data from the object it is called on and puts the data read into the given byte slice. It returns the number of bytes read and an error object which will be nil if no error occurred, or io.EOF ("end of file") if no error occurred and the end of the input was reached, or some other non-nil value if an error occurred. Similarly, for an object to be writable it must satisfy the io.Writer interface. This interface specifies a single method with signature, Write([]byte) (int, error). The Write() method writes data from the given byte slice into the object the method was called on, and returns the number of bytes written and an error object (which will be nil if no error occurred).

io Readers and Writers are unbuffered; the package bufio provides for the corresponding buffered operations and so are especially useful for reading and writing UTF-8 encoded text files. We see this in action in the many examples of chapter 12.

Through using as much as possible these interfaces in the signature of methods, they become as generic as possible: every type that implements these interfaces can use these methods.

For example the function which implements a JPEG decoder takes a Reader as a parameter, and thus can decode from disk, network connection, gzipped http, etc.

11.9 Empty Interface

11.9.1 Concept

The *empty* or *minimal interface* has no methods and so doesn't make any demands at all.

```
type Any interface{}
```

So any variable, any type implements it (not only reference types as Object in Java/C#), and any or Any is really a good name as alias and abbreviation!

(It is analogous to the class Object in Java and C#, the base class of all classes, so Obj also fits.)

A variable of that interface type `var val interface{}` can through assignment receive a variable of any type. This is illustrated in the program Listing 11.8—empty_interface.go:

```
package main
import "fmt"

var i = 5
var str = "ABC"

type Person struct {
        name string
        age    int
}

type Any interface{}

func main() {
        var val Any
        val = 5
        fmt.Printf("val has the value: %v\n", val)
        val = str
        fmt.Printf("val has the value: %v\n", val)
        pers1 := new(Person)
        pers1.name = "Rob Pike"
        pers1.age = 55
        val = pers1
        fmt.Printf(“val has the value: %v\n”, val)
        switch t := val.(type) {
        case int:
               fmt.Printf(“Type int %T\n”, t)
        case string:
               fmt.Printf(“Type string %T\n”, t)
        case bool:
               fmt.Printf(“Type boolean %T\n”, t)
        case *Person:
               fmt.Printf(“Type pointer to Person %T\n”, *t)
        default:
               fmt.Printf(“Unexpected type %T”, t)
  }
}
Output:  val has the value: 5
         val has the value: ABC
         val has the value: &{Rob Pike 55}
         Type pointer to Person main.Person
```

An int, a string and a Person instance are assigned to an interface variable val. A type-switch tests for its type. Each `interface{}` variable takes up 2 words in memory: one word for the type of what is contained, the other word for either the contained data or a pointer to it.

Program `emptyint_switch.go` shows an example of usage of the empty interface in a type switch (see § 11.4) combined with a lambda function:

Listing 11.9—emptyint_switch.go:

```
package main
import “fmt”

type specialString string
var whatIsThis specialString = “hello”

func TypeSwitch() {
        testFunc := func(any interface{}) {
                switch v := any.(type) {
                case bool:
                fmt.Printf(“any %v is a bool type”, v)
                case int:
                fmt.Printf(“any %v is an int type”, v)
                case float32:
                fmt.Printf("any %v is a float32 type", v)
                case string:
                fmt.Printf("any %v is a string type", v)
                case specialString:
                fmt.Printf("any %v is a special String!", v)
                default:
                fmt.Println("unknown type!")
                }
        }
        testFunc(whatIsThis)
}

func main() {
        TypeSwitch()
}
// Output:      any hello is a special String!
```

Exercise 11.9: `simple_interface3.go`

Continuing on Exercise 11.2, add a function gI which, instead of an Simpler type, accepts a empty interface parameter. The function then tests with a type assertion if the parameter fulfills the Simpler type. Now call this gI function instead of the fI function. Make your code as safe as possible.

11.9.2 Constructing an array of a general type or with variables of different types

In § 7.6.6 we saw how arrays of ints, floats and strings can be searched and sorted. But what about arrays of other types, do we have to program that for ourselves ?

If we would like to we now know that this is possible by using the empty interface, let us give it the alias Element: `type Element interface {}`

Then define a container struct Vector, which contains a slice of Element-items:

```
type Vector struct {
        a []Element
}
```

Vectors can contain anything because any type implements the empty interface; in fact every element could be of different type. We can define a method At() that returns the ith element:

```
func (p *Vector) At(i int) Element {
        return p.a[i]
}
```

And a function Set() that sets the ith element:

```
func (p *Vector) Set(i int, Element e) {
        p.a[i] = e
}
```

Everything in the vector is stored as an Element, to get the original type back (unboxing) we need to use type-assertions. The compiler rejects assertions guaranteed to fail, but type assertions always execute at run time and so can produce run-time errors!

<u>Exercise 11.10:</u> `min_interface.go / minmain.go`

Analogous to the Sorter interface we developed in § 11.7, make a Miner interface with the necessary operations, and a function Min which has as a parameter a variable which is a collection of type Miner and which calculates and returns the minimum element in that collection.

11.9.3 Copying a data-slice in a slice of interface{}

Suppose you have a slice of data of myType and you want to put them in a slice of empty interface, like in:

```
var dataSlice []myType = FuncReturnSlice()
var interfaceSlice []interface{} = dataSlice
```

This doesn't work, the compiler gives you the error: `cannot use dataSlice (type []myType) as type []interface { } in assignment`

The reason is that the memory layout of both variables is not the same (try to reason this yourself or see <u>http://code.google.com/p/go-wiki/wiki/InterfaceSlice</u>) .

The copy must be done explicitly with a for-range statement, like in:

```
var dataSlice []myType = FuncReturnSlice()
var interfaceSlice []interface{} = make([]interface{}, len(dataSlice))
for ix, d := range dataSlice {
    interfaceSlice[i] = d
}
```

11.9.4 Node structures of general or different types

In §10.1 we encountered data-structures like lists and trees, using a recursive struct type called a node. The nodes contained a data field of a certain type. Now with the empty interface at our disposal, data can be of that type, and we can write generic code. Here is some starting code for a binary tree structure: the general definition, a method NewNode for creating such an empty structure, and a method SetData for giving the data a value:

Listing 11.10—node_structures.go:

```
package main
import "fmt"

type Node struct {
        le    *Node
        data  interface{}
        ri    *Node
}

func NewNode(left, right *Node) *Node {
        return &Node{left, nil, right}
}
func (n *Node) SetData(data interface{}) {
        n.data = data
}

func main() {
        root := NewNode(nil,nil)
        root.SetData("root node")
        // make child (leaf) nodes:
        a := NewNode(nil,nil)
        a.SetData("left node")
        b := NewNode(nil,nil)
```

```
         b.SetData("right node")
         root.le = a
         root.ri = b
         fmt.Printf("%v\n", root) // Output: &{0x125275f0 root node 0x125275e0}
}
```

11.9.5 Interface to interface

An interface value can also be assigned to another interface value, as long as the underlying value implements the necessary methods. This conversion is checked at runtime, and when it fails a runtime error occurs: this is one of the dynamic aspects of Go, comparable to dynamic languages like Ruby and Python.

Suppose:

```
var ai AbsInterface  // declares method Abs()
type SqrInterface interface { Sqr() float }
var si SqrInterface
pp := new(Point)  // say *Point implements Abs, Sqr
var empty interface{}
```

Then the following statements and type assertions are valid:

```
empty = pp;  // everything satisfies empty
ai = empty.(AbsInterface);  // underlying value pp implements Abs()
                        // (runtime failure otherwise)
si = ai.(SqrInterface);  // *Point has Sqr() even though AbsInterface doesn't
empty = si;  // *Point implements empty set
         // Note: statically checkable so type assertion not necessary.
```

Here is an example with a function call:

```
type myPrintInterface interface {
  print()
}

func f3(x myInterface) {
         x.(myPrintInterface).print()  // type assertion to myPrintInterface
}
```

The conversion to myPrintInterface is entirely dynamic: it will work as long as the underlying type of x (the *dynamic type*) defines a print method.

11.10 The reflect package

11.10.1 Methods and types in reflect

In §10.4 we saw how reflect can be used to analyse a struct. Here we elaborate further on its powerful possibilities. Reflection in computing is the ability of a program to examine its own structure, particularly through the types; it's a form of *metaprogramming*. reflect can be used to investigate types and variables at runtime, e.g. its size, its methods, and it can also call these methods 'dynamically'. It can also be useful to work with types from packages of which you do not have the source. It's a powerful tool that should be used with care and avoided unless strictly necessary.

Basic information of a variable is its type and its value: there are represented in the reflection package by the types *Type,* which represents a general Go type, and *Value* which is the reflection interface to a Go value.

Two simple functions, *reflect.TypeOf* and *reflect.ValueOf*, retrieve Type and Value pieces out of any value. For example if x is defined as: `var x float64 = 3.4`

then `reflect.TypeOf(x)` gives `float64 and reflect.ValueOf(x)` returns `<float64 Value>.`

In fact reflection works by examing an interface value, the variable is first converted to the empty interface. This becomes apparent if you look at the signatures of both of these functions:

```
	func TypeOf(i interface{}) Type
	func ValueOf(i interface{}) Value
```

The interface value then contains a type and value pair.

Reflection goes from interface values to reflection objects and back again as we will see.

Both reflect.Type and reflect.Value have lots of methods to let us examine and manipulate them. One important example is that Value has a *Type* method that returns the Type of a reflect.Value. Another is that both Type and Value have a *Kind* method that returns a constant indicating what sort of item is stored: Uint, Float64, Slice, and so on. Also methods on Value with names like Int and Float let us grab values (as int64 and float64) stored inside. The different Kinds of Type are defined as constants: type Kind uint8

```
const (
   Invalid Kind = iota
   Bool
   Int
   Int8
   Int16
   Int32
   Int64
   Uint
   Uint8
   Uint16
   Uint32
   Uint64
   Uintptr
   Float32
   Float64
   Complex64
   Complex128
   Array
   Chan
   Func
   Interface
   Map
   Ptr
   Slice
   String
   Struct
   UnsafePointer
)
```

For our variable x if `v:= reflect.ValueOf(x)` then `v.Kind()` is `float64`, so the following is true: `v.Kind() == reflect.Float64`

The Kind is always the underlying type: if you define:

```
type MyInt int
var m MyInt = 5
v := reflect.ValueOf(m)
```

then `v.Kind()` returns `reflect.Int`

The `Interface()` method on a Value recovers the (interface) value, so to print a Value v do this:
`fmt.Println(v.Interface())`

Experiment with these possibilities with the following code:

Listing 11.11—reflect1.go:

```
package main
import (
        "fmt"
        "reflect"
)

func main() {
        var x float64 = 3.4
        fmt.Println("type:", reflect.TypeOf(x))
        v := reflect.ValueOf(x)
        fmt.Println("value:", v)
        fmt.Println("type:", v.Type())
        fmt.Println("kind:", v.Kind())
        fmt.Println("value:", v.Float())
        fmt.Println(v.Interface())
        fmt.Printf("value is %5.2e\n", v.Interface())
        y := v.Interface().(float64)
        fmt.Println(y)
}
/* output:
type: float64
value: <float64 Value>
type: float64
kind: float64
value: 3.4
3.4
value is 3.40e+00
3.4
*/
```

Knowing x is a `float64` value, `reflect.`**`ValueOf(x)`**`.Float()` returns its value as a float64; the same works with `Int()`, `Bool()`, `Complex()` and `String()`.

11.10.2 Modifying (setting) a value through reflection

Continuing with the previous example (see Listing 11.9 reflect2.go), suppose we want to modify the value of x to say 3.1415. Value has a number of Set methods to do this, but here we must be careful: `v.SetFloat(3.1415)`

produces an Error: `will panic: reflect.Value.SetFloat using unaddressable value`

Why is this? The problem is that v is not settable (not that the value 7.1 is not addressable). Settability is a property of a reflection Value, and not all reflection Values have it: it can be tested with the CanSet() method.

In our case we see that this is false: `settability of v: false`

When v was created with `v := reflect.ValueOf(x)` a copy of x was passed to the function, so it is logical that you can't change the original x through v.

In order to change x through v we need to pass the address of x: `v = reflect.ValueOf(&x)`

Through Type() we see that v is now of type *float64 and still not settable.

To make it settable we need to let the Elem() function work on it which indirects through the pointer: `v = v.Elem()`

Now `v.CanSet()` gives true and `v.SetFloat(3.1415)` works!

Listing 11.12—reflect2.go:

```
package main
import (
       "fmt"
       "reflect"
)

func main() {
        var x float64 = 3.4
        v := reflect.ValueOf(x)
        // setting a value:
// Error: will panic: reflect.Value.SetFloat using unaddressable value
        // v.SetFloat(3.1415)
        fmt.Println("settability of v:", v.CanSet())
        v = reflect.ValueOf(&x) // Note: take the address of x.
        fmt.Println("type of v:", v.Type())
        fmt.Println("settability of v:", v.CanSet())
        v = v.Elem()
        fmt.Println("The Elem of v is: ", v)
        fmt.Println("settability of v:", v.CanSet())
        v.SetFloat(3.1415) // this works!
        fmt.Println(v.Interface())
        fmt.Println(v)
}
/* Output:
settability of v: false
type of v: *float64
settability of v: false
The Elem of v is: <float64 Value>
settability of v: true
3.1415
<float64 Value>
*/
```

Reflection Values need the address of something in order to modify what they represent.

11.10.3 Reflection on structs

Some of the possibilities for structs are demonstrated in listing 11.10. We see that `NumField()` gives us the number of fields in the struct; with a for-loop we can go through each of them indexed by I with `Field(i)`.

We can also call its methods, e.g. method n where n is its index with: `Method(n).Call(nil)`

Listing 11.13—reflect_struct.go

```
package main
import (
       "fmt"
       "reflect"
)

type NotknownType struct {
       s1, s2, s3      string
}
```

```
func (n NotknownType) String() string {
       return n.s1 + "-" + n.s2 + "-" + n.s3
}

// variable to investigate:
var secret interface {} = NotknownType{"Ada", "Go", "Oberon"}

func main() {
       value := reflect.ValueOf(secret)  // <main.NotknownType Value>
       typ := reflect.TypeOf(secret)   // main.NotknownType
       // alternative:
       // typ := value.Type()  // main.NotknownType
       fmt.Println(typ)
       knd := value.Kind()  // struct
       fmt.Println(knd)

       // iterate through the fields of the struct:
       for i:= 0; i < value.NumField(); i++ {
                fmt.Printf("Field %d: %v\n", i, value.Field(i))
                    //value.Field(i).SetString("C#")
      }
      // call the first method, which is String():
      results := value.Method(0).Call(nil)
      fmt.Println(results)  // [Ada - Go - Oberon]
}
/* Output:
main.NotknownType
struct
Field 0: Ada
Field 1: Go
Field 2: Oberon
[Ada - Go - Oberon]
*/
```

But if we try to change a value, we get a runtime error:

```
panic: reflect.Value.SetString using value obtained using unexported field
```

So we see that only exported fields (starting with a capital letter) of a struct are settable; this is demonstrated in listing 11.11:

Listing 11.14—reflect_struct2.go:

```
package main
import (
        “fmt”
        “reflect”
)

type T struct {
        A int
        B string
}

func main() {
        t := T{23, “skidoo”}
        s := reflect.ValueOf(&t).Elem()
        typeOfT := s.Type()
        for i := 0; i < s.NumField(); i++ {
               f := s.Field(i)
               fmt.Printf(“%d: %s %s = %v\n”, i,
               typeOfT.Field(i).Name, f.Type(), f.Interface())
        }
        s.Field(0).SetInt(77)
        s.Field(1).SetString(“Sunset Strip”)
        fmt.Println(“t is now”, t)
}
/* Output:
0: A int = 23
1: B string = skidoo
t is now {77 Sunset Strip}
*/
```

Ref. 37 presents an insightful overview of these laws of reflection.

11.11 Printf and reflection.

The capabilities of the reflection package discussed in the previous section are heavily used in the standard library, for example the function Printf etc. uses it to unpack its … arguments, Printf is declared as:

```
func Printf(format string, args ... interface{}) (n int, err error)
```

The ... argument inside Printf (or anywhere else) has type interface{}, and Printf uses the reflection package to unpack it and discover the argument list. As a result, Printf knows the actual types of their arguments. Because they know if the argument is unsigned or long, there is no %u or %ld, only %d. This is also how Print and Println can print the arguments nicely without a format string.

To make this more concrete, we implement a simplified version of such a generic print-function in the following example, which uses a type-switch to deduce the type, and according to this prints the variable out (we used the code from exercises 10.7b en 10.8).

Listing 11.15—print.go:

```
package main
import (
        "os"
        "strconv"
)

type Stringer interface {
        String() string
}

type Celsius float64

func (c Celsius) String() string {
        return strconv.FormatFloat(float64(c),'f', 1, 64) + "°C"
}

type Day int

var dayName = []string{"Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"}

func (day Day) String() string {
        return dayName[day]
}

func print(args ...interface{}) {
        for i, arg := range args {
                if i > 0 {os.Stdout.WriteString(" ")}
                switch a := arg.(type) { // type switch
                case Stringer: os.Stdout.WriteString(a.String())
                case int:             os.Stdout.WriteString(strconv.Itoa(a))
                case string:   os.Stdout.WriteString(a)
                // more types
                default:              os.Stdout.WriteString("???")
                }
        }
}

func main() {
        print(Day(1), "was", Celsius(18.36))  // Tuesday was 18.4 °C
}
```

In § 12.8 we explain how the same interfacing principle works for fmt.Fprintf() .

11.12 Interfaces and dynamic typing

11.12.1 Dynamic typing in Go

In classic OO languages (like C++, Java and C#) data and the methods which act upon that data are united in the *class*-concept: a class contains them both, they cannot be separated.

In Go there are no classes: data (structures, or more general types) and methods are treated *orthogonally*, they are much more loosely coupled.

Interfaces in Go are similar to their Java/C# counterparts: both specify a minimum set of methods that an implementer of an interface must provide. But they are also more fluid and generic: any type that provides code for the methods of an interface *implicitly* implements that interface, without having to say explicitly that it does.

Compared to other languages Go is the only one which combines interface values, static type checking (does a type implement the interface?), dynamic runtime conversion and no requirement for explicitly declaring that a type satisfies an interface. This property also allows interfaces to be defined and used without having to modify existing code.

A function which has a (one or more) parameter of an interface type can be called with a variable whose type implements that interface. *Types implementing an interface can be passed to any function which takes that interface as an argument.*

This resembles much more the *duck typing* in dynamic languages like Python and Ruby; this can be defined to mean that objects can be handled (e.g., passed to functions), based on the methods they provide, regardless of their actual types: what they are is less important than what they can.

This is illustrated in the program `duck_dance.go`, where the function DuckDance takes a variable of interface type IDuck. The program only compiles when DuckDance is called on a variable of a type which implements IDuck.

Listing 11.16—duck_dance.go:

```go
package main
import “fmt”
type IDuck interface {
        Quack()
        Walk()
}

func DuckDance(duck IDuck) {
        for i := 1; i <= 3; i++ {
                duck.Quack()
                duck.Walk()
        }
}

type Bird struct {
        // ...
}

func (b *Bird) Quack() {
        fmt.Println(“I am quacking!”)
}

func (b *Bird) Walk()  {
        fmt.Println(“I am walking!”)
}

func main() {
        b := new(Bird)
        DuckDance(b)
}
```

```
Output: I am quacking!
```

```
I am walking!
I am quacking!
I am walking!
I am quacking!
I am walking!
```

If type Bird however does not implement Walk() (comment it out), then we get a compiler error:

```
cannot use b (type *Bird) as type IDuck in function argument:
*Bird does not implement IDuck (missing Walk method)
```

If we call the function DuckDance() for a cat, Go gives a compiler error, whereas Python or Ruby end in a runtime error!

11.12.2 Dynamic method invocation

However in Python, Ruby and the like duck-typing is performed as late binding (during runtime): methods are simply called on those arguments and variables and resolved at runtime (they mostly have methods like responds_to to check whether the object knows this method, but this amounts to more coding and testing).

On the contrary in Go the implementation requirements (most often) gets *statically checked by the compiler*: it checks if a type implements all functions of an interface when there is an assignment of a variable to a variable of that interface. If the method call is on a general type like interface{ }, you can check whether the variable implements the interface by doing a *type assertion* (see § 11.3).

As an example suppose you have different entities represented as types which have to be written out as XML streams. Then we define an interface for XML writing with the one method you are looking for (tt even can be defined as a private interface):

```
type xmlWriter interface {
 WriteXML(w io.Writer) error
}
```

Now we can make a function StreamXML for the streaming of any variable, testing with a type assertion if the variable passed implements the interface; if not we call its own function encodeToXML to do the job:

```
// Exported XML streaming function.
func StreamXML(v interface{}, w io.Writer) error {
    if xw, ok := v.(xmlWriter); ok {
        // It’s an  xmlWriter, use method of asserted type.
        return xw.WriteXML(w)
    }
      // No implementation, so we have to use our own function (with perhaps
reflection):
    return encodeToXML(v, w)
}

// Internal XML encoding function.
func encodeToXML(v interface{}, w io.Writer) error {
    // ...
}
```

Go uses the same mechanism in their `gob` package: here they have defined the two interfaces `GobEncoder` and `GobDecoder`. These allow types to define an own way to encode and decode their data to and from byte streams; otherwise the standard way with reflection is used. So Go provides the advantages of a dynamic language, but not their disadvantages of run-time errors!

This alleviates for a part the need for unit-testing, which is very important in dynamic languages but also presents a considerable amount of effort.

Go interfaces promotes separation of concerns, improves code re-use, and makes it easier to build on patterns that emerge as the code develops. With Go-interfaces the *Injection Dependency* pattern can also be implemented.

11.12.3 Extraction of an interface

A refactoring pattern that is very useful is *extracting interfaces*, reducing thereby the number of types and methods needed, without having the need to manage a whole class-hierarchy as in more traditional class-based OO-languages.

The way interfaces behave in Go allows developers to discover their programs’ types as they write them. If there are several objects that all have the same behavior, and a developer wishes to abstract that behavior, they can create an interface and then use that. Expanding the example of § 11.1 `Listing 11.2—interfaces_poly.go`, suppose we find we need a new interface TopologicalGenus, which gives the rank of a shape (here simply implemented as returning an int). All we have to do is give the types we want this interface to implement the method Rank():

Listing 11.17—multi_interfaces_poly.go:

```
package main
import "fmt"

type Shaper interface {
        Area() float32
}

type TopologicalGenus interface {
        Rank() int
}

type Square struct {
        side float32
}

func (sq *Square) Area() float32 {
        return sq.side * sq.side
}

func (sq *Square) Rank() int {
        return 1
}

type Rectangle struct {
        length, width float32
}

func (r Rectangle) Area() float32 {
        return r.length * r.width
}

func (r Rectangle) Rank() int {
        return 2
}

func main() {
        r := Rectangle{5, 3} // Area() of Rectangle needs a value
        q := &Square{5}      // Area() of Square needs a pointer
        shapes := []Shaper{r, q}
        fmt.Println("Looping through shapes for area ...")
        for n, _ := range shapes {
               fmt.Println("Shape details: ", shapes[n])
               fmt.Println("Area of this shape is: ", shapes[n].Area())
        }
        topgen := []TopologicalGenus{r, q}
        fmt.Println("Looping through topgen for rank ...")
        for n, _ := range topgen {
               fmt.Println("Shape details: ", topgen[n])
               fmt.Println("Topological Genus of this shape is: ", topgen[n].
               Rank())
        }
}
/* Output:
Looping through shapes for area ...
Shape details:  {5 3}
Area of this shape is:  15
Shape details:  &{5}
Area of this shape is:  25
Looping through topgen for rank ...
Shape details:  {5 3}
Topological Genus of this shape is:  2
Shape details:  &{5}
Topological Genus of this shape is:  1
*/
```

So you don't have to work all your interfaces out ahead of time; *the whole design can evolve without invalidating early decisions*. If a type must implement a new interface, the type itself doesn't have to be changed, you must only make the new method(s) on the type.

11.12.4 Explicitly indicating that a type implements an interface

If you wish that the types of an interface explicitly declare that they implement it, you can add a method with a descriptive name to the interface's method set. For example:

```
type Fooer interface {
        Foo()
        ImplementsFooer()
}
```

A type Bar must then implement the ImplementsFooer method to be a Fooer, clearly documenting the fact.

```
type Bar struct{}
func (b Bar) ImplementsFooer() {}
func (b Bar) Foo() {}
```

Most code doesn't make use of such constraints, since they limit the utility of the interface idea. Sometimes though, they can be necessary to resolve ambiguities among similar interfaces.

11.12.5 Empty interface and function overloading

In § 6.1 we saw that function overloading is not allowed. In Go this can be accomplished by using a variable number of parameters with …T as last parameter (see § 6.3). If we take T to be the empty interface and we know that a variable of any type satisfies T, then this allows us to pass any number of parameters of any type to that function, which is what overloading means.

This is applied in the definition of the function

```
fmt.Printf(format string, a ...interface{}) (n int, errno error)
```

this function iterates over the slice a discovering the type of its arguments dynamically. For each type it looks if a method String() is implemented; if so, this is used for producing the output. We come back to it in § 11.10.

11.12.6 Inheritance of interfaces

When a type includes (embeds) another type (which implements one or more interfaces) as a pointer, then the type can use all of the interfaces-methods.

Example:
```
type Task struct {
        Command string
        *log.Logger
}
```

A factory for this type could be:

```
func NewTask(command string, logger *log.Logger) *Task {
        return &Task{command, logger}
}
```

When log.Logger implements a Log() method, then an instance task of Task can call it: `task.Log()`

A type can also inherit from multiple interfaces providing something like *multiple inheritance*:

```
type ReaderWriter struct {
        *io.Reader
        *io.Writer
}
```

The principles outlined above are applied throughout all Go-packages, thus maximing the possibility of using polymorphism and minimizing the amount of code (see e.g. § 12.8). This is considered an important best practice in Go-programming.

Useful interfaces can be detected when the development is already under way. It is easy to add new interfaces, because existing types don't have to change (they only have to implement their methods). Existing functions can then be generalized from having a parameter(s) of a constrained type to a parameter of the interface type: often only the signature of the function needs to be changed. Contrast this to class-based OO-languages where in such a case the design of the whole class-hierarchy has to be adapted.

Exercise 11.11: `map_function_interface.go`

In Ex. 7.13 we defined a map function to apply to a slice of ints (`map_function.go`).

With the help of the empty interface and the type switch we can now write a *generic* map function, which can be applied to many types. Construct a map-function mapFunc for ints and strings, which doubles the ints and concatenates the string with itself.

Hint: For readability define an alias for interface { }, like: `type obj       interface{}`

Exercise 11.12: `map_function_interface_var.go`

Make a slight variation to Ex. 11.9 to allow for mapFunc to receive a variable number of items.

Exercise 11.13: `main_stack.go—stack/stack_general.go`

In exercises 10.10 and 10.11 we developed some Stack struct-types. However they were limited to a certain fixed internal type. Now develop a general stack type using a slice with as element type the interface{ }.

Implement the following Stack-methods: Len() int, IsEmpty() bool, Push(x interface{}) and Pop() (x interface{}, error). Pop() changes the stack while returning the topmost element; write also a method Top() which only returns this element.

In the main program construct a stack with a number of elements of different types, Pop all of them and print them.

11.13 Summary: the object-orientedness of Go

Let us summarize what we have seen about this: Go has no classes, but instead loosely coupled types and their methods, implementing interfaces.

The 3 important aspects of OO-languages are encapsulation, inheritance and polymorphism, how are they envisioned in Go?

i) Encapsulation (data hiding): in contrast to other OO languages where there are 4 or more access-levels, Go simplifies this to only 2 (see the Visibility Rule in § 4.2):
   1) *package scope*: 'object' is only known in its own package, how? it starts with a lowercase letter
   2) *exported:* 'object' is visible outside of its package, how? it starts with an uppercase letter

   A type can only have methods defined in its own package.

ii) Inheritance: how? composition: embedding of 1 (or more) type(s) with the desired behavior (fields and methods); multiple inheritance is possible through embedding multiple types

iii) Polymorphism: how? interfaces: a variable of a type can be assigned to a variable of any interface it implements. Types and interfaces are loosely coupled, again multiple inheritance is possible through implementing multiple interfaces. Go's interfaces aren't a variant on Java or C# interfaces, they're much more: they are independent and are key to large-scale programming and adaptable, evolutionary design.

11.14 Structs, collections and higher order functions

Often when you have a struct in your application you also need a collection of (pointers to) objects of that struct, like:

```
type Any interface{}
type Car struct {
        Model          string
        Manufacturer   string
        BuildYear     int
        // ...
}

type Cars []*Car
```

We can then use higher order functions using the fact that functions can be arguments in defining the needed functionality, e.g.:

1) when defining a general Process() function, which itself takes a function f which operates on every car:

```
// Process all cars with the given function f:
func (cs Cars) Process(f func(car *Car)) {
        for _, c := range cs {
               f(c)
        }
}
```

2) building upon this, make Find-functions to obtain subsets, and callin Process() with a closure (so it knows the local slice cars):

```
// Find all cars matching a given criteria.
func (cs Cars) FindAll(f func(car *Car) bool) Cars {
        cars := make([]*Car, 0)
        cs.Process(func(c *Car) {
               if f(c) {
                      append(cars,c)
               }
        )
        return cars
}
```

3) and make a Map-functionality, producing something out of every car object:

```
// Process cars and create new data.
func (cs Cars) Map(f func(car *Car) Any) []Any {
        result := make([]Any, 0)
        ix := 0
        cs.Process(func(c *Car) {
                result[ix] = f(c)
                ix++
        })
        return result
}
Now we can define concrete queries like:
allNewBMWs := allCars.FindAll(func(car *Car) bool {
        return (car.Manufacturer == “BMW”) && (car.BuildYear > 2010)
})
```

We can also return functions based on arguments. Maybe we would like to append cars to collections based on the manufacturers, but those may be varying. So we define a function to create a special append function as well as a map of collections:

4)
```
func MakeSortedAppender(manufacturers []string) (func(car *Car), map[string]Cars)
{
        // Prepare maps of sorted cars.
        sortedCars := make(map[string]Cars)

        for _, m := range manufacturers {
                sortedCars[m] = make([]*Car, 0)
        }
        sortedCars[“Default”] = make([]*Car, 0)

        // Prepare appender function:
        appender := func(c *Car) {
                if _, ok := sortedCars[c.Manufacturer]; ok {
                        sortedCars[c.Manufacturer] = append(sortedCars[c.
                        Manufacturer], c)
                } else {
                        sortedCars[“Default”] = append(sortedCars[“Default”], c)
                }
        }
        return appender, sortedCars
}
```

We now can use it to sort our cars into individual collections, like in:

```
manufacturers := []string{“Ford”, “Aston Martin”, “Land Rover”, “BMW”, “Jaguar”}
sortedAppender, sortedCars := MakeSortedAppender(manufacturers)
```

```
    allUnsortedCars.Process(sortedAppender)
    BMWCount := len(sortedCars[“BMW”])
```

We make this code work in the following program `cars.go` (here we we show only the code in main(), the rest of the code has already been shown above:

Listing 11.18—cars.go:

```
    // .. types and functions

    func main() {
            // make some cars:
            ford := &Car{“Fiesta”,“Ford”, 2008}
            bmw := &Car{“XL 450”, “BMW”, 2011}
            merc := &Car{“D600”, “Mercedes”, 2009}
            bmw2 := &Car{“X 800”, “BMW”, 2008}
            // query:
            allCars := Cars([]*Car{ford, bmw, merc, bmw2})
            allNewBMWs := allCars.FindAll(func(car *Car) bool {
                   return (car.Manufacturer == “BMW”) && (car.BuildYear > 2010)
            })
            fmt.Println(“AllCars: “, allCars)
            fmt.Println(“New BMWs: “, allNewBMWs)
            //
            manufacturers := []string{“Ford”, “Aston Martin”, “Land Rover”, “BMW”,
            Jaguar”}
            sortedAppender, sortedCars := MakeSortedAppender(manufacturers)
            allCars.Process(sortedAppender)
            fmt.Println(“Map sortedCars: “, sortedCars)
            BMWCount := len(sortedCars[“BMW”])
            fmt.Println(“We have “, BMWCount, “BMWs”)
    }
```

which produces the following output:

```
AllCars: [0xf8400038a0 0xf840003bd0 0xf840003ba0 0xf840003b70]
New BMWs: [0xf840003bd0]

Map sortedCars: map[Default:[0xf840003ba0] Jaguar:[] Land Rover:[] BMW:[0xf840003bd0
0xf840003b70] Aston Martin:[] Ford:[0xf8400038a0]]
We have 2 BMWs
```

PART 3

ADVANCED GO

Chapter 12—Reading and writing

Apart from the packages fmt and os, we will also need to import and use the package bufio for buffered input and output.

12.1 Reading input from the user

How can we read user input from the keyboard (console)? Input is read from the keyboard or standard input, which is os.Stdin. The simplest way is to use the Scan- and Sscan-family of functions of the package fmt, as illustrated in this program:

Listing 12.1—readinput1.go:

```
// read input from the console:
package main
import "fmt"

var (
        firstName, lastName, s string
        i int
        f float32
        input = "56.12 / 5212 / Go"
        format = "%f / %d / %s"
)

func main() {
        fmt.Println("Please enter your full name: ")
        fmt.Scanln(&firstName, &lastName)
        // fmt.Scanf("%s %s", &firstName, &lastName)
        fmt.Printf("Hi %s %s!\n", firstName, lastName) // Hi Chris Naegels
        fmt.Sscanf(input, format, &f, &i, &s)
        fmt.Println("From the string we read: ", f, i, s)
// ouwtput: From the string we read: 56.12 5212 Go
}
```

Scanln scans text read from standard input, storing successive space-separated values into successive arguments; it stops scanning at a newline. Scanf does the same, but is uses the its first parameter as format string to read the values in the variables. Sscan and friends work the same way but read from an input string as instead from standard input. You can check on the number of items read in and the error when something goes wrong.

But it can also be done with a buffered reader from the package bufio, as is demonstrated in the following program:

Listing 12.2—readinput2.go:

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

var inputReader *bufio.Reader
        var input string
        var err error

func main() {

        inputReader = bufio.NewReader(os.Stdin)
        fmt.Println("Please enter some input: ")
        input, err = inputReader.ReadString('\n')

        if err == nil {
               fmt.Printf("The input was: %s\n", input)
        }
}
```

`inputReader` is a pointer to a `Reader` in bufio. This reader is created and linked to standard input in the line: `inputReader := bufio.NewReader(os.Stdin).`

The `bufio.NewReader()` constructor has the signature: `func NewReader(rd io.Reader) *Reader`

It takes as argument any object that satisfies the io.Reader interface (any object that has a suitable Read() method, see § 11.8) and returns a new buffered io.Reader that reads from the given reader, `os.Stdin` satisfies this requirement.

The reader has a method `ReadString(delim byte)`, in which reads the input until delim is found, `delim` is included; what is read is put into a buffer.

`ReadString` returns the read string and nil for the error; when it reads until the end of a file, the string read is returned and io.`EOF`; if `delim` is not found an error `err != nil` is returned.

In our case input from the keyboard is read until the ENTER key (which contains '\n') is tapped.

Standard output os.Stdout is the screen; `os.Stderr` is where error-info can be written to, mostly equal to `os.Stdout`.

In normal Go-code the var-declarations are omitted and the variables are declared with :=, like:

```
inputReader := bufio.NewReader(os.Stdin)
input, err := inputReader.ReadString('\n')
```

We will apply this idiom from now on.

The second example reads input from the keyboard with a few different switch-versions:

Listing 12.3—switch_input.go:

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

func main() {
        inputReader := bufio.NewReader(os.Stdin)
        fmt.Println("Please enter your name:")
        input, err := inputReader.ReadString('\n')

        if err != nil {
               fmt.Println("There were errors reading, exiting program.")
               return
        }

        fmt.Printf("Your name is %s", input)
         // For Unix: test with delimiter "\n", for Windows: test with "\r\n"
        switch input {
        case “Philip\r\n”:    fmt.Println(“Welcome Philip!”)
        case “Chris\r\n”:     fmt.Println(“Welcome Chris!”)
        case “Ivo\r\n”:       fmt.Println(“Welcome Ivo!”)
        default: fmt.Printf(“You are not welcome here! Goodbye!”)
        }

         // version 2:
         switch input {
         case “Philip\r\n”:   fallthrough
         case “Ivo\r\n”:      fallthrough
         case “Chris\r\n”:    fmt.Printf(“Welcome %s\n”, input)
         default: fmt.Printf(“You are not welcome here! Goodbye!\n”)
         }

        // version 3:
         switch input {
         case “Philip\r\n”, “Ivo\r\n”:      fmt.Printf(“Welcome %s\n”, input)
         default: fmt.Printf(“You are not welcome here! Goodbye!\n”)
         }
}
```

Note how the line-endings for Unix and Windows are different!

EXERCISES:

Exercise 12.1: `word_letter_count.go`

Write a program which reads text from the keybord. When the user enters ‘S’ in order to signal the end of the input, the program shows 3 numbers:

i) the number of characters including spaces (but excluding ‘\r’ and ‘\n’)
ii) the number of words
iii) the number of lines

Exercise 12.2: `calculator.go`

Make a simple (reverse polish notation) calculator. This program accepts input from the user in the form of integers (maximum 999999) and operators (+, -, *, /).

The input is like this: number1 ENTER number2 ENTER operator ENTER → result is displayed.

The programs stops if the user inputs "q". Use the package `stack` you developed in Ex. 11.3

12.2 Reading from and writing to a file

12.2.1 Reading from a file

Files in Go are represented by pointers to objects of type `os.File`, also called filehandles. The standard input os.Stdin and output os.Stdout we used in the previous section are both of type `*os.File`. This is used in the following program:

Listing 12.4—fileinput.go:

```
package main
import (
        "bufio"
        "fmt"
        "io"
        "os"
)

func main() {
        inputFile, inputError := os.Open("input.dat")
        if inputError != nil {
            fmt.Printf("An error occurred on opening the inputfile\n" +
               "Does the file exist?\n" +
        "Have you got acces to it?\n")
            return // exit the function on error
        }
        defer inputFile.Close()

        inputReader := bufio.NewReader(inputFile)
        for {
            inputString, readerError := inputReader.ReadString('\n')
            if readerError == io.EOF {
               return
        }
            fmt.Printf("The input was: %s", inputString)
        }
}
```

The variable `inputFile` is of type `*os.File`: this is a struct which represents an open file descriptor (a file handle). Then the file needs to be opened with the Open function from os: this accepts a parameter filename of type string, here `input.dat`, and opens the file in read-only mode.

This can of course result in an error when the file does not exist or the program has not sufficient rights to open the file: `inputFile, inputError = os.Open("input.dat")` . If we have a file, we assure with `defer.Close()` that the file is closed at the end, and then we apply `bufio.NewReader` to get a reader variable.

Using bufio's reader (and the same goes for writer) as we have done here means that we can work with convenient high level string objects, completely insulated from the raw bytes which represent the text on disk.

Then we read each line of the file (delimited by '\n') in an infinite for-loop with the method `ReadString('\n') or ReadBytes('\n').`

Remark: In a previous example we saw text-files in Unix end on \n, but in Windows this is \r\n. By using the method `ReadString` or `ReadBytes` with \n as a delimiter you don't have to worry about this.The use of the `ReadLine()` method could also be a good alternative.

When we have read the file past the end then `readerError != nil` (actually **io.EOF** is true), and the for-loop is left through the return statement.

Some alternatives:

1) Reading the contents of an entire file in a string:
   If this is sufficient for you needs, you can use the ioutil.ReadFile() method from the package io/ioutil, which returns a []byte containing the bytes read and nil or a possible error (see Listing 12.5). Analogously the WriteFile() function writes a []byte to a file.

Listing 12.5—read_write_file1.go:

```
package main
import (
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
    inputFile := "products.txt"
    outputFile := "products_copy.txt"
    buf, err := ioutil.ReadFile(inputFile)
    if err != nil {
    fmt.Fprintf(os.Stderr, "File Error: %s\n", err)
```

```
            // panic(err.Error())
        }
        fmt.Printf("%s\n", string(buf))
        err = ioutil.WriteFile(outputFile, buf, 0x644)
        if err != nil {
            panic(err. Error())
        }
}
```

2) Buffered Read:
Instead of using ReadString(), in the more general case of a file not divided in lines or a binary file, we could have used the Read() method on the bufio.Reader, with as input parameter a:

```
buf := make([]byte, 1024)
…
n, err := inputReader.Read(buf)
if (n == 0) { break}
```

where n is the number of bytes read, as applied in § 12.5 .

3) Reading columns of data from a file:
If the data columns are separated by a space, you can use the FScan-function series from the "fmt" package. This is applied in the following program, which reads in data from 3 columns into the variables v1, v2 and v3; they are then appended to column slices.

Listing 12.6—read_file2.go:

```
package main
import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Open("products2.txt")
    if err != nil {
            panic(err)
    }
    defer file.Close()

    var col1, col2, col3 []string
    for {
            var v1, v2, v3 string
            _, err := fmt.Fscanln(file, &v1, &v2, &v3)
                    // scans until newline
            if err != nil {
                    break
            }
            col1 = append(col1, v1)
            col2 = append(col2, v2)
            col3 = append(col3, v3)
    }

    fmt.Println(col1)
    fmt.Println(col2)
    fmt.Println(col3)
}
/* Output:
[ABC FUNC GO]
[40 56 45]
[150 280 356]
*/
```

Remark: The subpackage `filepath` of package `path` provides functions for manipulating filenames and paths that work across OS platforms; for example the function `Base()` returns the last element of a path without trailing separator: `import "path/filepath"`

```
                                        filename := filepath.Base(path)
```

Exercise 12.3: `read_csv.go`
We are given the file products.txt with the content:

```
"The ABC of Go";25.5;1500
"Functional Programming with Go";56;280
"Go for It";45.9;356
"The Go Way";55;500
```

The 1st field of each line is a title, the 2nd a price, the 3rd a quantity.

Almost the same file as in Listing 12.3c, but now the field are separated by a ";".

Again read in the data, but make a struct type to gather all the data of one line and use a slice of structs to print the data.

For more functionality in parsing csv-files, see the package `encoding/csv` (http://golang.org/pkg/encoding/csv/)

12.2.2 The package *compress*: reading from a zipped file

The package compress from the standard library provides facilities for reading compressed files in the following formats: bzip2, flate, gzip, lzw and zlib.

The following program illustrates the reading of a gzip file:

Listing 12.7—gzipped.go:

```
package main
import (
        "fmt"
        "bufio"
        "os"
        "compress/gzip"
)

func main() {
        fName := "MyFile.gz"
        var r *bufio.Reader
        fi, err := os.Open(fName)
        if err != nil {
        fmt.Fprintf(os.Stderr, "%v, Can't open %s: error: %s\n", os.Args[0],
        fName, err)
        os.Exit(1)
        }
        fz, err := gzip.NewReader(fi)
        if err != nil {
               r = bufio.NewReader(fi)
        } else {
               r = bufio.NewReader(fz)
        }

        for {
               line, err := r.ReadString('\n')
               if err != nil {
```

```
                        fmt.Println("Done reading file")
                        os.Exit(0)
                }
                fmt.Println(line)
        }
}
```

12.2.3 Writing to a file

This is demonstrated in the following program:

Listing 12.8—fileoutput.go:

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

func main () {
        outputFile, outputError := os.OpenFile("output.dat",
        os.O_WRONLY|os.O_CREATE, 0666)
        if outputError != nil {
                fmt.Printf("An error occurred with file creation\n")
                return
        }
        defer outputFile.Close()
        outputWriter:= bufio.NewWriter(outputFile)
        outputString := "hello world!\n"

        for i:=0; i<10; i++ {
                outputWriter.WriteString(outputString)
        }
        outputWriter.Flush()
}
```

Apart from a file handle, we now need a `Writer`from bufio. We open a file `output.dat` for write-only; the file is created when it does not exist with:

```
        outputFile, outputError := os.OpenFile("output.dat", os.O_WRONLY|os.O_
        CREATE, 0666)
```

We see that the `OpenFile` function takes a filename, one or more flags (logically OR-d together using the | bitwise OR operator if more than one), and the file permissions to use.

The following flags are commonly used:

`os.O_RDONLY` : the read flag for read-only access
`os.WRONLY` : the write flag for write-only access
`os.O_CREATE` : the create flag: create the file if it doesn't exist
`os.O_TRUNC` : the truncate flag: truncate to size 0 if the file already exists

When reading, the file permissions are ignored so we can use a value of 0. When writing we use the standard Unix file permissions of 0666 (even on Windows).

Then we make the writer-object (the buffer) with:

```
outputWriter := bufio.NewWriter(outputFile)
```

The for-loop repeats 10 times a write to the buffer: `outputWriter.WriteString(outputString)`

The buffer is then written completely to the file with: `outputWriter.Flush()`

In simple write tasks this can be done more efficient with: `fmt.Fprintf(outputFile, "Some test data.\n")` using the F version of the fmt Print functions that can write to any io.Writer, including a file (see also § 12.8).

The program `filewrite.go` illustrates an alternative to fmt.FPrintf:

Listing 12.9—filewrite.go:

```
package main
import "os"

func main() {
        os.Stdout.WriteString("hello, world\n")
        f, _ := os.OpenFile("test", os.O_CREATE|os.O_WRONLY, 0)
        defer f.Close()
        f.WriteString("hello, world in a file\n")
}
```

With `os.Stdout.WriteString("hello, world\n")` we can also write to the screen.

In f, `_ := os.OpenFile("test", os.O_CREATE|os.O_WRONLY, 0)` we create or open for write-only a file "test". A possible error is disregarded with `_` .

We don't make use of a buffer and write immediately to the file with: `f.WriteString( )`

Exercise 12. 4: `wiki_part1.go`

(this exercise stands on its own but is also a preparation for § 15.4)

The datastructure of our program is a struct containing the following fields:

```
type Page struct {
        Title string
        Body  []byte
}
```

Make a save-method on this struct to write a text-file with Title as name and Body as content.

Make a load function, which given the title string, reads the corresponding text-file. Use *Page as arguments, because the structs could be quite large and we don't want to make copies of them in memory. Use the ioutil functions from § 12.2.1

12.3 Copying files

How do you copy a file to another file ? The simplest way is using Copy from the package io:

Listing 12.10—filecopy.go:

```
package main

import (
        "fmt"
        "io"
        "os"
)

func main() {
        CopyFile("target.txt", "source.txt")
        fmt.Println("Copy done!")
}

func CopyFile(dstName, srcName string) (written int64, err error) {
        src, err := os.Open(srcName)
        if err != nil {
               return
        }
        defer src.Close()

        dst, err := os.OpenFile(dstName, os.O_WRONLY|os.O_CREATE, 0644)
        if err != nil {
               return
        }
        defer dst.Close()

        return io.Copy(dst, src)
}
```

Notice the use of defer: when the opening of the destination file would produce an error and return, then still the defer and so src.Close() is executed. If this would not be done, the source file would remain opened, using resources.

12.4 Reading arguments from the command-line

12.4.1 With the os-package

The package os also has a variable `os.Args` of type slice of strings and which can be used for elementary command-line argument processing, that is reading arguments which are given on the command-line when the program is started. Look at the following greetings-program:

Listing 12.11—os_args.go:

```
package main

import (
  "fmt"
  "os"
  "strings"
)
```

```
func main() {
 who := "Alice"
 if len(os.Args) > 1 {
        who += strings.Join(os.Args[1:], " ")
 }
 fmt.Println("Good Morning", who)
}
```

When we run this program from our editor or IDE the output is: `Good Morning Alice`
the same output we get when we run it on the command-line as: `os_args or ./os_args.`
But when we give arguments on the command-line like: `os_args John Bill Marc Luke`
we get as output: `Good Morning Alice John Bill Marc Luke`

When there is at least one command-line argument the slice `os.Args[]` takes the arguments (separated by a space) in, starting from index 1 (`os.Args[0]` contains the name of the program, `os_args` in this case). The `strings.Join` function glues them all together with a space in between.

Exercise 12. 5: `hello_who.go`

A variation on the "Hello World"-program: person names can be given after the program name on the command line, like: `hello_who Evan Michael Laura`, so that the output is:
`Hello Evan Michael Laura !`

12.4.2 With the flag-package

The package `flag` has an extended functionality for parsing of command-line options. But it is also often used to replace ordinary constants, e.g. when in certain cases we want to give a different value for our constant on the command-line. (see the project in chapter 19).

A Flag is defined in the package `flag` as a struct with the following fields:

```
type Flag struct {
        Name          string // name as it appears on command line
        Usage         string // help message
        Value         Value // value as set
        DefValue      string // default value (as text); for usage message
}
```

This is used in the following program `echo.go`, which simulates the Unix echo-utility:

Listing 12.12—echo.go:

```
package main
import (
        “os”
        “flag” // command line option parser
)

var NewLine = flag.Bool(“n”, false, “print on newline”)
        // echo -n flag, of type *bool

const (
        Space = “ ”
        Newline = “\n”
)

func main() {
        flag.PrintDefaults()
        flag.Parse()
        var s string = “”
        for i := 0; i < flag.NArg(); i++ {
               if i > 0 {
                      s += Space
               }
               s += flag.Arg(i)
        }

        if *NewLine { // -n is parsed, flag becomes true
               s += Newline
        }
        os.Stdout.WriteString(s)
}
```

`flag.Parse()` scans the argument list (or list of constants) and sets up flags, `flag.Arg(i)` is the ith argument. All `flag.Arg(i)` are available after Parse(), `flag.Arg(0)` is the first real flag, not the name of the program in contrast to `os.Args(0)`

`flag.NArg()` is the number of arguments. After parsing the flags or constants can be used.

`flag.Bool()` defines a flag, with default value false: when its first argument (here "n") appears on the command-line the flag is set to true (`NewLine` is of type *bool). We dereference the flag in if `*NewLine`, so when this is true a newline is added.

`flag.PrintDefaults()` prints out the usage information of the defined flag(s), for this case:

```
-n=false: print newline.
```

`flag.VisitAll(fn func(*Flag))` is another useful function: it visits the set flags in lexicographical order, calling fn for each (for an example see § 15.8)

When the command-line (in Windows) is: `echo.exe A B C`, the program outputs: A B C;
With `echo.exe -n A B C`, the program outputs:

```
A
B
C
```

so the newline character is printed. Each time the usage message is also printed before the data: `-n=false: print newline`

With flag.Bool you can make boolean flags, which can be tested against in your code, for example define a flag `processedFlag` which:

```
var processedFlag = flag.Bool("proc", false, "nothing processed yet")
```

Test it later in the code by dereferencing it:

```
if *processedFlag { // found flag -proc
        r = process()
}
```

To define flags of other data types, use flag.Int(), flag.Float64, flag.String().

You will find a concrete example in in § 15.8

12.5 Reading files with a buffer

We combine the techniques of buffered reading of a file and command-line flag parsing in the following example. If there are no arguments what is typed in is shown again in the output.

Arguments are treated as filenames and when the files exist, their content is shown.

Test it out with: `cat test`

Listing 12.13—cat.go:

```
package main
import (
        “io”
        “os”
        “fmt”
        “bufio”
        “flag”
)

func cat(r *bufio.Reader) {
        for {
                buf, err := r.ReadBytes(‘\n’)
                if err == io.EOF {
                        break
                }
                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, err := os.Open(flag.Arg(i))
                if err != nil {
                        fmt.Fprintf(os.Stderr, “%s:error reading from %s: %s\n”,
                        os.Args[0], flag.Arg(i), err.Error())
                        continue
                }
                cat(bufio.NewReader(f))
        }
}
```

In § 12.6 we'll see how to do buffered writing.

Exercise 12. 6: `cat_numbered.go`

Extend the example from listing 12.8 to process a flag which indicates that every line read should be preceded by a line number in the output. Test it out with: `cat -n test`

12.6 Reading and writing files with slices

Slices provide the standard Go way to handle I/O buffers, they are used in the following second version of the function `cat`, which reads a file in an infinite for-loop (until end-of-file EOF) in a sliced buffer, and writes it to standard output.

```
func cat(f *os.File) {
        const NBUF = 512
        var buf [NBUF]byte
        for {
            switch nr, err := f.Read(buf[:]); true {
                case nr < 0:
                fmt.Fprintf(os.Stderr, "cat: error reading: %s\n", err.Error())
                os.Exit(1)
                case nr == 0: // EOF
                return
                case nr > 0:
                if nw, ew := os.Stdout.Write(buf[0:nr]); nw != nr {
                       fmt.Fprintf(os.Stderr, "cat: error writing: %s\n",
                       ew.String())
                }
            }
        }
 }
```

This code comes from cat2.go, which uses `os.file` and its Read-method from package os; cat2.go works in the same way as cat.go from listing 12.14:

Listing 12.14—cat2.go:

```
package main
import (
       "flag"
       "fmt"
```

```
      “os”
)

func cat(f *os.File) {
       const NBUF = 512
       var buf [NBUF]byte
       for {
           switch nr, err := f.Read(buf[:]); true {
           case nr < 0:
               fmt.Fprintf(os.Stderr, “cat: error reading: %s\n”, err.Error())
               os.Exit(1)
           case nr == 0: // EOF
               return
           case nr > 0:
               if nw, ew := os.Stdout.Write(buf[0:nr]); nw != nr {
                      fmt.Fprintf(os.Stderr, “cat: error writing: %s\n”,
                      ew.String())
               }
           }
       }
}

func main() {
       flag.Parse() // Scans the arg list and sets up flags
       if flag.NArg() == 0 {
           cat(os.Stdin)
       }
       for i := 0; i < flag.NArg(); i++ {
           f, err := os.Open(flag.Arg(i))
           if f == nil {
                      fmt.Fprintf(os.Stderr, “cat: can’t open %s:          error
                      %s\n”, flag.Arg(i), err)
               os.Exit(1)
           }
           cat(f)
           f.Close()
       }
}
```

12.7 Using defer to close a file

The defer-keyword (see § 6.4) is very useful for ensuring that the opened file will also be closed at the end of the function, like in the following snippet:

```
func data(name string) string {
  f := os.Open(name, os.O_RDONLY, 0)
  defer f.Close()      // idiomatic Go code!
  contents := io.ReadAll(f)
  return contents
}
```

`f.Close()` is executed at `return contents`.

12.8 A practical example of the use of interfaces: fmt.Fprintf

The program `io_interfaces.go` is a nice illustration of the concept of interfaces in io.

Listing 12.15—io_interfaces.go:

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

func main() {
        // unbuffered: os.Stdout implements io.Writer
        fmt.Fprintf(os.Stdout, "%s\n", "hello world! - unbuffered")
        // buffered:
        buf := bufio.NewWriter(os.Stdout)
        // and now so does buf:
        fmt.Fprintf(buf, "%s\n", "hello world! - buffered")
        buf.Flush()
}
```

Output:

```
hello world! - unbuffered
hello world! - buffered
```

Here is the actual signature of **`fmt.Fprintf()`**:

```
func Fprintf(w io.Writer, format string, a ...) (n int, err error)
```

It doesn't write to a file, it writes to a variable of type io.Writer, that is: Writer defined in the io package:

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

`fmt.Fprintf()` writes a string according to a format string to its first argument, which must implement io.Writer. `Fprintf()` can write to any type that has a Write method, including os.Stdout, files (like os.File), pipes, network connections, channels, etc..., and also to write buffers from the bufio package. This package defines a `type Writer struct { ... }`
`bufio.Writer` implements the Write method:

```
func (b *Writer) Write(p []byte) (nn int, err error)
```

It also has a factory: give it an io.Writer, it will return a buffered io.Writer in the form of a bufio. Writer: `func NewWriter(wr io.Writer) (b *Writer)`

Buffering works for anything that Writes.

!! Never forget to use Flush() when terminating buffered writing, else the last ouput won't be written !!

In §15.2-15.8 we use the fmt.Fprint functions to write to a http.ResponseWriter, which also implements io.Writer.

Exercise 12. 7: `remove_3till5char.go`

The following code takes an input file goprogram.go, reads it line by line and writes it to an output file only retaing the 3rd till 5th character of every line. However when you try it out you see that only an empty output file gets written. Find this logical bug, correct it and test it.

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

```
func main() {
	inputFile, _ := os.Open("goprogram.go")
	outputFile, _ := os.OpenFile("goprogramT.go", os.O_WRONLY|os.O_CREATE,
	0666)
	defer inputFile.Close()
	defer outputFile.Close()
	inputReader := bufio.NewReader(inputFile)
	outputWriter := bufio.NewWriter(outputFile)
	for {
		inputString, _, readerError := inputReader.ReadLine()
		if readerError == io.EOF {
			fmt.Println("EOF")
			return
		}
		outputString := string([]byte(inputString)[2:5]) + "\r\n"
		n, err := outputWriter.WriteString(outputString)
		if (err != nil) {
			fmt.Println(err)
			return
		}
	}
	fmt.Println("Conversion done")
}
```

12.9 The json dataformat

To transmit a data structure across a network or to store it in a file, it must be encoded and then decoded again; many encodings exist: JSON, XML, gob, Google's protocol buffers, and others. Go has implementations for all of them; in the next sections we will discuss the first three.

Structs can contain binary data, if this would be printed as text it would not be readable for a human. Moreover the data does not contain the name of the struct field, so we don't know what the data means.

To remedy this a number of formats have been devised, which transform the data into plain text, but annotated by their field names: so humans can read and understand the data. Data in these formats can be transmitted over a network: in this way the data are platform independent and can be used as input/output in all kinds of applications, no matter what the programming language or the operating system is.

The following terms are common here:

> data structure → special format = **marshaling** or **encoding** (before transmission at the sender or source)
> special format → data structure = **unmarshaling** or **decoding** (after transmission at the receiver or destination)

Marshaling is used to convert in memory data to the special format (data -> string), and vice versa for unmarshaling (string -> data structure).

Encoding does the same thing but the output is a stream of data (implementing io.Writer); decoding starts from a stream of data (implementing io.Reader) and populates a data structure.

Well known is the XML-format (see § 12.10); another popular format sometimes preferred for its simplicity is JSON (JavaScript Object Notation, see http://json.org). It is most commonly used for communication between web back-ends and JavaScript programs running in the browser, but it is used in many other places too.

This is a short JSON piece:

```
{ "Person":
        { "FirstName": "Laura",
                "LastName": "Lynn"
        }
}
```

Allthough XML is widely used JSON is less verbose (thus taking less memory space, disk space and network bandwith) and better readable, which explains it's growing popularity.

The json package provides an easy way to read and write JSON data from your Go programs.

We will use it in the following example, and use a simplified version of the structs Address and VCard from exercise 10.1 vcard.go (we leave out most of the error-handling for sake of brevity, but in a real program this must be taken care of, see chapter 13!).

Listing 12.16—json.go:

```
package main
import (
        "fmt"
        "encoding/json"
)
```

```
type Address struct {
        Type          string
        City          string
        Country       string
}

type VCard struct {
        FirstName     string
        LastName      string
        Addresses     []*Address
        Remark        string
}

func main() {
        pa := &Address{"private", "Aartselaar","Belgium"}
        wa := &Address{"work", "Boom", "Belgium"}
        vc := VCard{"Jan", "Kersschot", []*Address{pa,wa}, "none"}
        // fmt.Printf("%v: \n", vc)
        // {Jan Kersschot [0x126d2b80 0x126d2be0] none}:
        // JSON format:
        js, _ := json.Marshal(vc)
        fmt.Printf("JSON format: %s", js)
// using an encoder:
        file, _ := os.OpenFile("vcard.json", os.O_CREATE|os.O_WRONLY, 0)
        defer file.Close()
        enc := json.NewEncoder(file)
        err := enc.Encode(vc)
        if err != nil {
               log.Println("Error in encoding json")
        }
}
```

The `json.Marshal()` function with signature `func Marshal(v interface{}) ([]byte, error)` encodes the data into the following json-text (in fact a []bytes) :

```
{"FirstName":"Jan","LastName":"Kersschot","Addresses":[{"Type":"private","C
ity":"Aartselaar","Country":"Belgium"},{"Type":"work","City":"Boom","Count
ry":"Belgium"}],"Remark":"none"}
```

For security reasons in web applications better use the `json.MarshalForHTML()` function which performs an HTMLEscape on the data, so that the text will be safe to embed inside HTML <script> tags.

The default concrete Go types are:

- bool for JSON booleans,
- float64 for JSON numbers,
- string for JSON strings, and
- nil for JSON null.

Not everything can be json-encoded though: only data structures that can be represented as valid JSON will be encoded:

- JSON objects only support strings as keys; to encode a Go map type it must be of the form map[string]T (where T is any Go type supported by the json package).
- Channel, complex, and function types cannot be encoded.
- Cyclic data structures are not supported; they will cause Marshal to go into an infinite loop.
- Pointers will be encoded as the values they point to (or 'null' if the pointer is nil).

The json package only accesses the exported fields of struct types, only those will be present in the JSON output. This is necessary because json uses reflection on them.

UnMarshal:

The **`UnMarshal()`** function with signature `func Unmarshal(data []byte, v interface{}) error` performs the decoding from json to program data-structures.

First we create a struct Message where the decoded data will be stored: `var m Message`
and call Unmarshal(), passing it a []byte of JSON data b and a pointer to m: `err := json.Unmarshal(b, &m)`

Through reflection it tries to match the json-fields with the destination struct fields; only the matched fields are filled with data. So no error occurs if there are fields that do not match, they are simply disregarded.

(In Ex. 15.2b twitter_status_json.go UnMarshal is used.)

Decoding arbitrary data:

The json package uses map[string]interface{} and []interface{} values to store arbitrary JSON objects and arrays; it will happily unmarshal any valid JSON blob into a plain interface{} value.

Consider this JSON data, stored in the variable b:

```
b == []byte({"Name": "Wednesday", "Age": 6, "Parents": ["Gomez", "Morticia"]})
```

Without knowing this data's structure, we can decode it into an interface{ } value with Unmarshal:

```
var f interface{}
err := json.Unmarshal(b, &f)
```

At this point the value in f would be a map whose keys are strings and whose values are themselves stored as empty interface values:

```
map[string]interface{}{
        "Name": "Wednesday",
        "Age": 6,
        "Parents": []interface{}{
                "Gomez",
                "Morticia",
        },
}
```

To access this data we can use a type assertion to access f's underlying map[string]interface{}:

```
m := f.(map[string]interface{})
```

We can then iterate through the map with a range statement and use a type switch to access its values as their concrete types:

```
for k, v := range m {
        switch vv := v.(type) {
        case string:
                fmt.Println(k, "is string", vv)
        case int:
                fmt.Println(k, "is int", vv)
```

```
            case []interface{}:
                   fmt.Println(k, "is an array:")
            for i, u := range vv {
                   fmt.Println(i, u)
            }
            default:
                   fmt.Println(k, "is of a type I don't know how to handle")
            }
    }
```

In this way you can work with unknown JSON data while still enjoying the benefits of type safety.

Decoding data into a struct: If we would know beforehand the json-data, we could then define an appropriate struct and unmarshal into it. For the example in the previous section, we would define:

```
    type FamilyMember struct {
            Name string
            Age int
            Parents []string
    }
```

and then do the unmarshaling with:

```
    var m FamilyMember
    err := json.Unmarshal(b, &m)
```

thereby allocating a new slice behind the scenes. This is typical of how Unmarshal works with the supported reference types (pointers, slices, and maps).

Streaming Encoders and Decoders

The json package provides Decoder and Encoder types to support the common operation of reading and writing streams of JSON data. The NewDecoder and NewEncoder functions wrap the io.Reader and io.Writer interface types.

```
    func NewDecoder(r io.Reader) *Decoder
    func NewEncoder(w io.Writer) *Encoder
```

To write the json directly to a file, use `json.NewEncoder` on a file (or any other type which implements io.Writer) and call `Encode()` on the program data; the reverse is done by using a `json.Decoder` and the `Decode()` function:

```
func NewDecoder(r io.Reader) *Decoder
func (dec *Decoder) Decode(v interface{}) error
```

See how the use of interfaces generalizes the implementation: the data structures can be everything, because they only have to implement `interface{}`, the targets or sources to/from which the data is encoded must implement io.Writer or io.Reader. Due to the ubiquity of Readers and Writers, these Encoder and Decoder types can be used in a broad range of scenarios, such as reading and writing to HTTP connections, websockets, or files.

12.10 The xml dataformat

The XML equivalent for the json example used in § 12.9 is:

```
<Person>
        <FirstName>Laura</FirstName>
        <LastName>Lynn</LastName>
</Person>
```

Like the json package it contains a Marshal() and UnMarshal()-function to en- and decode data to and from XML; but here they are more general and can also be used to read from and write to files (or other types that implement io.Reader and io.Writer).

In the same way as with json, xml-data can be marshaled or unmarshaled to/from structs; in Listing 15.8 (twitter_status.go) we see this in action.

The `encoding/xml` package also implements a simle xml parser (SAX) to read XML-data and parse it into its constituents. The following code illustrates how this parser can be used:

Listing 12.17—xml.go:

```
package main
import (
        "fmt"
        "strings"
        "os"
        "encoding/xml"
)
```

```
var t, token    xml.Token
var err         error

func main() {
        input :=
"<Person><FirstName>Laura</FirstName><LastName>Lynn</LastName></Person>"
        inputReader := strings.NewReader(input)
        p := xml.NewParser(inputReader)

        for t, err = p.Token(); err == nil; t, err = p.Token() {
                switch token := t.(type) {
                case xml.StartElement:
                        name := token.Name.Local
                        fmt.Printf("Token name: %s\n", name)
                        for _, attr := range token.Attr {
                                attrName := attr.Name.Local
                                attrValue := attr.Value
                                fmt.Printf("An attribute is: %s %s\n", attrName,
                                attrValue)
                                // ...
                        }
                case xml.EndElement:
                        fmt.Println("End of token")
                case xml.CharData:
                        content := string([]byte(token))
                        fmt.Printf("This is the content: %v\n", content )
                        // ...
                default:
                        // ...
                }
        }
}
/* Output:
Token name: Person
Token name: FirstName
This is the content: Laura
End of token
Token name: LastName
This is the content: Lynn
```

```
	End of token
	End of token
	*/
```

The package defines a number of types for XML-tags: StartElement, Chardata (this is the actual text between the start- and end-tag), EndElement, Comment, Directive or ProcInst.

It also defines a struct Parser: the method NewParser takes an io.Reader (in this case a strings. NewReader) and produces an object of type Parser. This has a method Token() that returns the next XML token in the input stream. At the end of the input stream, Token() returns nil (io. EOF).

The XML-text is walked through in a for-loop which ends when the Token() method returns an error at end of file because there is no token left anymore to parse. Through a type-switch further processing can be defined according to the current kind of XML-tag. Content in Chardata is just a [ ]bytes, make it readable with a string conversion.

12.11 Datatransport through gob

gob is Go's own format for serializing and deserializing program data in binary format; it is found in the encoding package. Data in this format is simply called a gob (short for Go binary format). It is similar to Python's "pickle" or Java's "Serialization".

It is typically used in transporting arguments and results of remote procedure calls (RPCs) (see the rpc package § 15.9), and more generally for data transport between applications and machines. In what is it different from json or xml? It was specifically tailored for working in an environment which is completely in Go, for example communicating between two servers written in Go. That way it could be written much more efficient and optimized; it works with the language in a way that an externally-defined, language-independent encoding cannot. That's why the format is binary in the first place, not a text-format like JSON or XML. Gobs are not meant to be used in other languages than Go, because in the encoding and decoding process Go's reflection capability is exploited.

Gob files or streams are completely self-describing: for every type they contain a description of that type and they can always be decoded in Go, without any knowledge of the file's contents.

Only exported fields are encoded, zero values are not taken into account. When decoding structs, fields are matched by name and compatible type, and only fields that exist in both are affected. In this way a gob decoder client will still function when in the source datatype fields have been added: the client will continue to recognize the previously existing fields. Also there is great flexibility provided, e.g. integers are encoded as unsized, variable-length, regardless of the concrete Go type at the sender side.

So if we have at the sender side a struct T:

```
type T struct { X, Y, Z int }
var t = T{X: 7, Y: 0, Z: 8}
```

this can be captured at the receiver side in a variable u of type struct U:

```
type U struct { X, Y *int8 }
var u U
```

At the receiver X gets the value 7, and Y the value 0 (which was not transmitted).

In the same way like json, gob works by creating an Encoder object with a NewEncoder() function and calling Encode(), again completely generalized by using io.Writer; the inverse is done with a Decoder object with a NewDecoder() function and calling Decode(), generalized by using io.Reader. As an example we will write the info of listing 12.12 to a file named vcard.gob. This produces a mixture of readable and binary data as you will see when you try to read it in a text-editor.

In listing 12.18 you find a simple example of decoding and encoding, simulating transmission over a network with a buffer of bytes:

Listing 12.18—gob1.go:

```
package main
import (
        “bytes”
        “fmt”
        “encoding/gob”
        “log”
)

type P struct {
        X, Y, Z int
        Name   string
}

type Q struct {
        X, Y *int32
        Name string
}

func main() {
        // Initialize the encoder and decoder. Normally enc and dec would //
        be bound to network connections and the encoder and decoder    // would
        run in different processes.
        var network bytes.Buffer   // Stand-in for a network connection
        enc := gob.NewEncoder(&network) // Will write to network.
        dec := gob.NewDecoder(&network)      // Will read from network.
        // Encode (send) the value.
        err := enc.Encode(P{3, 4, 5, "Pythagoras"})
        if err != nil {
               log.Fatal("encode error:", err)
        }
        // Decode (receive) the value.
        var q Q
        err = dec.Decode(&q)
        if err != nil {
               log.Fatal("decode error:", err)
        }
        fmt.Printf("%q: {%d,%d}\n", q.Name, *q.X, *q.Y)
}
// Output:      "Pythagoras": {3,4}
```

Listing 12.19 uses encoding to write to a file:

Listing 12.19—gob2.go:

```
package main
import (
        "encoding/gob"
        "log"
        "os"
)

type Address struct {
        Type           string
        City           string
        Country        string
}
```

```
type VCard struct {
        FirstName     string
        LastName             string
        Addresses     []*Address
        Remark        string
}

func main() {
        pa := &Address{"private", "Aartselaar","Belgium"}
        wa := &Address{"work", "Boom", "Belgium"}
        vc := VCard{"Jan", "Kersschot", []*Address{pa,wa}, "none"}
        // fmt.Printf("%v: \n", vc)
        // {Jan Kersschot [0x126d2b80 0x126d2be0] none}:
        // using an encoder:
        file, _ := os.OpenFile("vcard.gob", os.O_CREATE|os.O_WRONLY, 0)
        defer file.Close()
        enc := gob.NewEncoder(file)
        err := enc.Encode(vc)
        if err != nil {
               log.Println("Error in encoding gob")
        }
}
```

Exercise 12. 8: degob.go: Write a program that reads in the file vcard.gob, decodes the contents and prints it.

12.12 Cryptography with go

Data transfer over networks must be encrypted so that no hacker can read or change them and checksums calculated over the data before send and after receive must be identical. Given the business of its mother company, it will come as no surprise to see that Go has more than 30 packages to offer in this field contained in its standard library:

- the `hash` package: implements the `adler32, crc32, crc64` and fnv checksums;
- the crypto package: implements other hashing algorithms like md4, md5, sha1, etc. and complete encryption implementations for `aes, blowfish, rc4, rsa, xtea, etc.`

In the following listings we compute and output some checksums with sha1 and md5.

Listing 12.20—hash_sha1.go:

```
package main
import (
        “fmt”
        “crypto/sha1”
        “io”
        “log”
)

func main() {
        hasher := sha1.New()
        io.WriteString(hasher, “test”)
        b := []byte{}
        fmt.Printf(“Result: %x\n”, hasher.Sum(b))
        fmt.Printf(“Result: %d\n”, hasher.Sum(b))
        hasher.Reset()
        data := []byte(“We shall overcome!”)
        n, err := hasher.Write(data)
        if n!=len(data) || err!=nil {
               log.Printf(“Hash write error: %v / %v”, n, err)
        }
        checksum := hasher.Sum(b)
        fmt.Printf(“Result: %x\n”, checksum)
}
/* Output:
Result: a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
Result: [169 74 143 229 204 177 155 166 28 76 8 115 211 145 233 135 152 47 187
211]
Result: e2222bfc59850bbb00a722e764a555603bb59b2a
*/
```

With `sha1.New()` a new hash.Hash object is made, that can compute the SHA1 checksum. The type Hash is in fact an interface, itself implementing the io.Writer interface:

```
type Hash interface {
    // Write adds more data to the running hash.
    // It never returns an error.
    io.Writer

    // Sum returns the current hash, without changing the
```

```
        // underlying hash state.
        Sum() []byte

        // Reset resets the hash to one with zero bytes written.
        Reset()

        // Size returns the number of bytes Sum will return.
        Size() int
    }
```

Through io.WriteString or hasher.Write the checksum of the given string is computed.

Exercise 12. 9: hash_md5.go: test out the md5 algorithm as in Listing 12.20

Chapter 13—Error-handling and Testing

Go does not have an exception mechanism, like the try/catch in Java or .NET for instance: you cannot throw exceptions. Instead it has a *defer-panic-and-recover* mechanism (§ 13.2-13.3). The Go-designers thought that the try/catch mechanism is overly used and that the throwing of exceptions in lower layers to higher layers of code uses too much resources. The mechanism they devised for Go can 'catch' an exception, but it is much lighter, and even then should only be used as a last resort.

How then does Go deal with normal errors by default ? The Go way to handle errors is for functions and methods to return an error object as their only or last return value—or nil if no error occurred—and for calling functions to always check the error they receive.

!! Never ignore errors, because ignoring them can lead to program crashes !!

Handle the errors and return from the function in which the error occurred with an error message to the user: that way if something does go wrong, your program will continue to function and the user will be notified. The purpose of panic-and-recover is to deal with genuinely exceptional (so unexpected) problems and *not* with normal errors.

Library routines must often return some sort of error indication to the calling function.

In the preceding chapters we saw the idiomatic way in Go to detect and report error-conditions:

- a function which can result in an error returns two variables, a value and an error-code; the latter is nil in case of success, and != nil in case of an error-condition.
- after the function call the error is checked, in case of an error (if error != nil) the execution of the actual function (or if necessary the entire program) is stopped.

In the following code Func1 from package pack1 is tested on its return code:

```
if value, err := pack1.Func1(param1); err != nil {
fmt.Printf("Error %s in pack1.Func1 with parameter %v", err.Error(), param1)
    return    // or: return err
```

```
      }
      // Process(value)
```

Always assign an error to a variable within a compound if-statement, making for clearer code.

Instead of fmt.Printf corresponding methods of the log-package could be used (see §13.3 and §15.2) or even a panic (see the next section) if it doesn't matter that the program stops.

13.1 Error-handling

Go has a *predefined error interface type*:

```
type error interface {
        Error() string
}
```

`error` values are used to indicate an abnormal state; we have seen the standard way of using it in § 5.2. Other examples using it with file-handling can be found in chapter 12; we will see how to use it in network operations in chapter 15. The package `errors` contains an `errorString` struct which implements the error interface. To stop execution of a program in an error-state, we can use `os.Exit(1)`.

13.1.1 Defining errors

Whenever you need a new error-type, make one with the function `errors.New` from the `errors` package (which you will have to import), and give it an appropriate error-string, like following:

```
      err := errors.New("math - square root of negative number")
```

In Listing 13.1 you see a simple example of its use:

Listing 13.1—errors.go:

```
package main

import (
        "errors"
        "fmt"
)

var errNotFound error = errors.New("Not found error")

func main() {
        fmt.Printf("error: %v", errNotFound)
}
// error: Not found error
```

Applied in a function testing the parameter of a square root function, this could be used as:

```
func Sqrt(f float64) (float64, error) {
        if f < 0 {
               return 0, errors.New ("math - square root of negative number")
        }
        // implementation of Sqrt
}
```

You could call this function as follows:

```
if f, err := Sqrt(-1); err != nil {
        fmt.Printf("Error: %s\n", err)
}
```

and because fmt.Printf automatically uses its String()-method (see § 10.7), the error-string "`Error: math - square root of negative number`" is printed out. Because there wil often be a prefix like Error: here, don't start your error string with a capital letter.

In most cases it is interesting to make a custom error struct type, which apart from the (low level) error-message also contains other useful information, such as the operation which was taking place (open file, …), the full path-name or url which was involved, etc. The String() method then provides an informative concatenation of all this information. As an example, see PathError which can be issued from an os.Open:

```
// PathError records an error and the operation and file path that caused it.
type PathError struct {
        Op string    // "open", "unlink", etc.
        Path string  // The associated file.
        Err error  // Returned by the system call.
}

func (e *PathError) String() string {
        return e.Op + " " + e.Path + ": "+ e.Err.Error()
}
```

In case different possible error-conditions can occur, it may be useful to test with a type assertion or type switch for the exact error, and possibly try a remedy or a recovery of the error-situation:

```
//  err != nil
if e, ok := err.(*os.PathError); ok {
  // remedy situation
}
```

Or:

```
switch err := err.(type) {
case ParseError:
        PrintParseError(err)
case PathError:
        PrintPathError(err)
…
default:
        fmt.Printf("Not a special error, just %s\n", err)
}
```

As a 2nd example consider the json package. This specifies a SyntaxError type that the json.Decode function returns when it encounters a syntax error parsing a JSON document:

```
type SyntaxError struct {
    msg    string // description of error
  // error occurred after reading Offset bytes, from which line and columnnr can be obtained
    Offset int64
}

func (e *SyntaxError) String() string { return e.msg }
```

In the calling code you could again test whether the error is of this type with a type assertion, like this:

```
  if serr, ok := err.(*json.SyntaxError); ok {
        line, col := findLine(f, serr.Offset)
        return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
  }
```

A package can also define its own specific Error with additional methods, like net.Error:

```
package net

type Error interface {
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

In §15.1 we see how it can be used.

As you have seen, in all examples the following naming convention was applied: Error types end in "Error" and error variables are called (or start with) "err" or "Err".

syscall is the low-level, external package, which provides a primitive interface to the underlying operating system's calls; these return integer error-codes ; the type syscall.Errno implements the Error interface.

Most syscall functions return a result and a possible error, like:

```
r, err := syscall.Open(name, mode, perm)
if err != 0 {
        fmt.Println(err.Error())
}
```

os also provides a standard set of error-variables like `os.EINVAL`, which come from `syscall` - errors:

```
var (
        EPERM          Error = Errno(syscall.EPERM)
        ENOENT         Error = Errno(syscall.ENOENT)
        ESRCH          Error = Errno(syscall.ESRCH)
        EINTR          Error = Errno(syscall.EINTR)
        EIO            Error = Errno(syscall.EIO)
        …
)
```

13.1.2 Making an error-object with fmt

Often you will want to return a more informative string with the value of the wrong parameter inserted for example: this is accomplished with the `fmt.Errorf()` function: it works exactly like `fmt.Printf()`,taking a format string with one ore more format specifiers and a corresponding number of variables to be substituted. But instead of printing the message it generates an `error` object with that message.

Applied to our Sqrt-example from above:

```
if f < 0 {
        return 0, fmt.Errorf("math: square root of negative number %g", f)
}
```

2nd example: while reading from the command-line we generate an error with a usage message when a help-flag is given:

```
if len(os.Args) > 1 && (os.Args[1] == "-h" || os.Args[1] == "--help") {
        err = fmt.Errorf("usage: %s infile.txt outfile.txt", filepath.Base(os.
Args[0]))
        return
}
```

13.2 Run-time exceptions and panic

When execution errors occur, such as attempting to index an array out of bounds or a type assertion failing, the Go runtime triggers a *run-time panic* with a value of the interface type **`runtime.Error`**, and the program crashes with a message of the error; this value has a RuntimeError()-method, to distinguish it from a normal error.

But a panic can also be initiated from code directly: when the error-condition (which we are testing in the code) is so severe and unrecoverable that the program cannot continue, the **`panic`** function is used, which effectively creates a run-time error that will stop the program. It takes 1 argument of any type, usually a string, to be printed out when the program dies. The Go runtime takes care to stop the program and issuing some debug information. How it works is illustrated in `Listing 13.2—panic.go:`

```
package main
import "fmt"
```

```
func main() {
        fmt.Println("Starting the program")
        panic("A severe error occurred: stopping the program!")
        fmt.Println("Ending the program")
}
```

And the output is:

```
Starting the program

panic: A severe error occurred: stopping the program!

panic PC=0x4f3038
runtime.panic+0x99 /go/src/pkg/runtime/proc.c:1032
        runtime.panic(0x442938, 0x4f08e8)
main.main+0xa5 E:/Go/GoBoek/code examples/chapter 13/panic.go:8
        main.main()
runtime.mainstart+0xf 386/asm.s:84
        runtime.mainstart()
runtime.goexit /go/src/pkg/runtime/proc.c:148
        runtime.goexit()

---- Error run E:/Go/GoBoek/code examples/chapter 13/panic.exe with code Crashed
---- Program exited with code -1073741783
```

A concrete example, checking whether the program starts with a known user:

```
var user = os.Getenv("USER")

func check() {
  if user == "" {
        panic("Unknown user: no value for $USER")
  }
}
```

This could be checked in an init() function of a package which is importing.

Panic can also be used in the error-handling pattern when the error must stop the program:

```
    if err != nil {
            panic("ERROR occurred:" + err.Error())
    }
```

Go panicking:

If `panic` is called from a nested function, it immediately stops execution of the current function, all defer statements are guaranteed to execute and then control is given to the function caller, which receives this call to panic. This bubbles up to the top level, executing defers, and at the top of the stack the program crashes and the error condition is reported on the command-line using the value given to panic: this termination sequence is called *panicking*.

The standard library contains a number of functions whose name is prefixed with Must, like `regexp.MustCompile or template.Must`; these functions panic() when converting the string which into a regular expression or template produces an error.

Of course taking down a program with panic should not be done lightly, so every effort must be exercised to remedy the situation and let the program continue.

13.3 Recover

As the name indicates this built-in function can be used to recover from a panic or an error-condition: it allows a program to regain control of a panicking goroutine, stopping the terminating sequence and thus resuming normal execution.

`recover` is only useful when called *inside* a deferred function (see § 6.4) : it then retrieves the error value passed through the call of panic; when used in normal execution a call to recover will return nil and have no other effect.

Summarized: panic causes the stack to unwind until a deferred recover() is found or the program terminates.

The `protect` function in the example below invokes the function argument `g` and protects callers from run-time panics raised by `g`, showing the message x of the panic:

```
    func protect(g func()) {
            defer func() {
                    log.Println("done")
                    // Println executes normally even if there is a panic
                    if err := recover(); err != nil {
                        log.Printf("run time panic: %v", err)
                }
        }()
        log.Println("start")
        g()    // ← possible runtime-error
}
```

It is analogous to the catch block in the Java and .NET languages.

`log` implements a simple logging package: the default logger writes to standard error and prints the date and time of each logged message. Apart from the `Println` and `Printf` functions, the `Fatal` functions call os.Exit(1) after writing the log message, Exit functions identically. The `Panic` functions call panic after writing the log message; use this when a critical condition occurs and the program must be stopped, like in the case when a webserver could not be started (see for example §15.4).

The `log` package also defines an interface type Logger with the same methods, when you want to define a customized logging system (see http://golang.org/pkg/log/#Logger).

Here is a complete example which illustrates how panic, defer and recover work together:

Listing 13.3—panic_recover.go:

```
package main
import (
        "fmt"
)

func badCall() {
        panic("bad end")
}
func test() {
   defer func() {
    if e := recover(); e != nil {
       fmt.Printf("Panicking %s\r\n", e);
    }

    }()
    badCall()
    fmt.Printf("After bad call\r\n");
}
```

```
func main() {
   fmt.Printf("Calling test\r\n");
   test()
   fmt.Printf("Test completed\r\n");
}
/* Output:
Calling test
Panicking bad end
Test completed
*/
```

Defer, panic- and recover form in a sense also a control-flow mechanism, like if, for, etc.

This mechanism is used at several places in the Go standard library, e.g. in the json package when decoding or in the regexp package in the Compile function. The convention in the Go libraries is that even when a package uses panic internally, a recover is done so that its external API still presents explicit error return values.

13.4 Error-handling and panicking in a custom package

This is a best practice which every writer of custom packages should apply:

1) *always recover from panic in your package:* no explicit panic() should be allowed to cross a package boundary
2) *return errors* as error values *to the callers of your package.*

Within a package, however, especially if there are deeply nested calls to non-exported functions, it can be useful (and improve readability) to use panic to indicate error conditions which should be translated into an error for the calling function.

This is nicely illustrated in the following code. We have a simple parse package (Listing 13.4) which parses input strings to slices of integers; it also contains its specialized ParseError.

This package panics (in function `fields2numbers`) when there is nothing to convert or when the conversion to integer fails. However the exported Parse function can recover from this and returns an error with all info to its caller. To show how it works the package is called from `panic_recover.go` (Listing 13.4); the strings which are not parseable come back with an error which is printed out.

Listing 13.4—parse.go:

```
package parse
import (
        "fmt"
        "strings"
        "strconv"
)

// A ParseError indicates an error in converting a word into an integer.
type ParseError struct {
        Index int      // The index into the space-separated list of words.
        Word  string   // The word that generated the parse error.
        // The raw error that precipitated this error, if any.
        Error err
}

// String returns a human-readable error message.
func (e *ParseError) String() string {
        return fmt.Sprintf("pkg parse: error parsing %q as int", e.Word)
}

// Parse parses the space-separated words in in put as integers.
func Parse(input string) (numbers []int, err error) {
        defer func() {
                 if r := recover(); r != nil {
                        var ok bool
                        err, ok = r.(error)
                         if !ok {
                                err = fmt.Errorf("pkg: %v", r)
                        }
                }
        }()

        fields := strings.Fields(input)
        numbers = fields2numbers(fields) // here panic can occur
        return
}

func fields2numbers(fields []string) (numbers []int) {
        if len(fields) == 0 {
                panic(“no words to parse”)
        }
        for idx, field := range fields {
                num, err := strconv.Atoi(field)
                if err != nil {
                        panic(&ParseError{idx, field, err})
                }
                numbers = append(numbers, num)
        }
        return
}
```

Listing 13.5—panic_package.go:

```
package main
import (
        “fmt”
        “./parse/parse”
)
func main() {
        var examples = []string{
                “1 2 3 4 5”,
                “100 50 25 12.5 6.25”,
                “2 + 2 = 4”,
                “1st class”,
                “”,
        }

        for _, ex := range examples {
                fmt.Printf(“Parsing %q:\n  “, ex)
                nums, err := parse.Parse(ex)
                if err != nil {
                        // here String() method from ParseError is used
                        fmt.Println(err)
                        continue
                }
                fmt.Println(nums)
        }
}
/* Output:
Parsing “w1 2 3 4 5”:
  [1 2 3 4 5]
Parsing “100 50 25 12.5 6.25”:
  pkg parse: error parsing “12.5” as int
Parsing “2 + 2 = 4”:
  pkg parse: error parsing “+” as int
Parsing “1st class”:
  pkg parse: error parsing “1st” as int
Parsing “”:
  pkg: no words to parse
*/
```

13.5 An error-handling scheme with closures

Every time when a function returns we should test whether it resulted in an error: this can lead to repetitive and tedious code. Combining the defer/panic/recover mechanism with closures can result in a far more elegant scheme that we will now discuss. However it is only applicable when all functions have the same signature, which is rather restrictive. A good example of its use is in web applications, where all handler functions are of the following type:

```
func handler1(w http.ResponseWriter, r *http.Request) { … }
```

Suppose all functions have the signature: `func f(a type1, b type2)`

The number of parameters and their types is irrelevant.

We give this type a name: `fType1 = func f(a type1, b type2)`

Our scheme uses 2 helper functions:

i) `check`: a function which tests whether an error occurred, and panics if so:
```
func check(err error) { if err != nil { panic(err) } }
```
ii) `errorhandler`: this is a wrapper function. It takes a function fn of our type `fType1` and returns such a function by calling fn. However it contains the defer/recover mechanism, outlined in § 13.3

```
func errorHandler(fn fType1) fType1 {
    return func(a type1, b type2) {
        defer func() {
            if e, ok := recover().(error); ok {
                log.Printf(“run time panic: %v”, err)
            }
        }()
        fn(a, b)
    }
}
```

When an error occurs it is recovered and printed on the log; apart from simply printing the application could also produce a customized output for the user by using the template package (§ 15.7). The check() function is used in every called function, like this:

```
    func f1(a type1, b type2) {
        …
        f, _, err := // call function/method
        check(err)
        t, err := // call function/method
        check(err)
        _, err2 := // call function/method
        check(err2)
        …
    }
```

The main() or other caller-function should then call the necessary functions wrapped in errorHandler, like this:

```
    func main() {
        errorHandler(f1)
        errorHandler(f2)
        …
    }
```

Using this mechanism all errors are recovered and the error-checking code after a function call is reduced to `check(err)`.In this scheme different error-handlers have to be used for different function types; they could be hidden inside an error-handling package. Alternatively a more general approach could be using a slice of empty interface as parameter and return type.

We will apply this in the web application from § 15.5

EXERCISES:

Exercise 13.1: `recover_dividebyzero.go`
Use the coding scheme from Listing 13.3 to provoke a real runtime panic by letting an integer divide by 0.

Exercise 13.2: `panic_defer.go`
Consider the following complete program. Without executing it, write down what the output of this program will be. Then compile, execute and verify your predictions.

```
package main
import “fmt”

func main() {
        f()
        fmt.Println(“Returned normally from f.”)
}

func f() {
        defer func() {
               if r := recover(); r != nil {
                      fmt.Println(“Recovered in f”, r)
               }
        }()
        fmt.Println(“Calling g.”)
        g(0)
        fmt.Println(“Returned normally from g.”)
}

func g(i int) {
        if i > 3 {
               fmt.Println(“Panicking!”)
               panic(fmt.Sprintf(“%v”, i))
        }
        defer fmt.Println(“Defer in g”, i)
        fmt.Println(“Printing in g”, i)
        g(i + 1)
}
```

Exercise 13.3: `panic_defer_convint.go`

Write a function ConvertInt64ToInt which converts an int64 value to an int, and panics when this goes wrong (hint: see § 4.5.2.1) . Then call this function from a function IntFromInt64 which recovers, and returns an int and an error. Test the function!

13.6 Starting an external command or program

The os package contains the function `StartProcess` to call or start external OS commands or binary executables; its 1st argument is the process to be executed, the 2nd can be used to pass some options or arguments, and the 3rd is a struct which contains basic info about the OS-environment.

It returns the process id (pid) of the started process, or an error if it failed.

The `exec` package contains the structures and functions to accomplish the same task more easily; most important are `exec.Command(name string, arg ...string)` and Run(). The first needs the name of an OS command or executable and creates a Command object, which can then be executed with Run() that uses this object as its receiver. The following program (which only works under Linux because Linux commands are executed) illustrates their use:

Listing 13.6—exec.go:

```
package main
import (
"fmt"
        "os/exec"
        "os"
)

func main() {
// 1) os.StartProcess //
/*********************/
/* Linux: */
        env := os.Environ()
        procAttr := &os.ProcAttr{
                        Env: env,
                        Files: []*os.File{
                               os.Stdin,
                               os.Stdout,
                               os.Stderr,
                        },
                }
        pid, err := os.StartProcess("/bin/ls", []string{"ls", "-l"}, procAttr)
        if err != nil {
                fmt.Printf("Error %v starting process!", err) //
                os.Exit(1)
        }
        fmt.Printf("The process id is %v", pid)
/* Output:
The process id is &{2054 0}total 2056
-rwxr-xr-x 1 ivo ivo 1157555 2011-07-04 16:48 MB1_exec
-rw-r--r-- 1 ivo ivo 2124 2011-07-04 16:48 MB1_exec.go
-rw-r--r-- 1 ivo ivo 18528 2011-07-04 16:48 MB1_exec_go_.6
-rwxr-xr-x 1 ivo ivo 913920 2011-06-03 16:13 panic.exe
-rw-r--r-- 1 ivo ivo 180 2011-04-11 20:39 panic.go
*/

// 2nd example: show all processes
pid, err = os.StartProcess("/bin/ps", []string{"-e", "opid,ppid,comm"}, procAttr)
if err != nil {
fmt.Printf("Error %v starting process!", err) //
os.Exit(1)
}
fmt.Printf("The process id is %v", pid)
// 2) cmd.Run //
/***************/
 cmd := exec.Command("gedit") // this opens a gedit-window
 err := cmd.Run()
 if err != nil {
        fmt.Printf("Error %v executing command!", err)
        os.Exit(1)
 }
 fmt.Printf("The command is %v", cmd)
}
```

13.7 Testing and benchmarking in Go

Every package should in the 1st place contain a certain minimal amount of documentation; 2nd but for some equally important is testing.

In chapter 3 Go's testing tool `gotest` was mentioned, we used it already in § 9.8. Here we will elaborate on its use with some more examples.

A special package called `testing` provides support for automated testing, logging and error reporting. It also contains some functionality for benchmarking functions.

<u>Remark:</u> `gotest` is a Unix bash-script, so under Windows you need a MINGW environment (see § 2.5); for Windows every time you see map pkg/linux_amd64, replace it with pkg/windows.

To *(unit-)test* a package you write a number of tests that you can run frequently (after every update) to check the correctness of your code in small units. For that we will have to write a set of Go source files which will exercise and test our code. These <u>test-programs</u> must be <u>within the same package</u> and the files must have names of the form `*_test.go`, so the test code is separated from the actual code of the package.

These `_test` programs are NOT compiled with the normal Go-compilers, so they are not deployed when you put your application into production; only gotest compiles all programs: the normal—and the test-programs.

In those files which must include `import "testing"`, we write global functions with names starting with `TestZzz,where Zzz is` an alphabetic description of the function to be tested, like TestFmtInterface, TestPayEmployees, etc.

Those functions should have a header of the form: `func TestAbcde(t *testing.T)`

where T is a struct type passed to Test functions that manages test state and supports formatted test logs, like t.Log, t.Error, t.ErrorF, etc. At the end of each of these functions the output is compared with what is expected and if these are not equal an error is logged. A successful test function just returns.

To signal a failure we have the following functions:

i) `func (t *T) Fail()`
   marks the test function as having failed, but continues its execution.

ii) `func (t *T) FailNow()`
   marks the test function as having failed and stops its execution; all other tests in this file are also skipped, execution continues with the next test file

ii) `func (t *T) Log(args …interface{})`

the args are formatted using default formatting and the text is logged in the error-log

iv) `func (t *T) Fatal(args …interface{})`

this has the combined effect of iii) followed by ii)

Run `go test`, this compiles the test-programs, and executes all the TestZzz-functions in the test programs. If all tests are successful the word PASS will be printed.

gotest can also take 1 or more test programs as parameters, and some options.

With the option `--chatty` or `-v` each test function that is run is mentioned and its test-status.

For example:

```
go test fmt_test.go --chatty
=== RUN fmt.TestFlagParser
--- PASS: fmt.TestFlagParser
=== RUN fmt.TestArrayPrinter
--- PASS: fmt.TestArrayPrinter
…
```

The testing package also contains some types and functions for simple benchmarking; the test code must then contain function(s) starting with BenchmarkZzz and take a parameter of type `*testing.B`, e.g.:

```
func BenchmarkReverse(b *testing.B) {
        …
}
```

The command `go test -test.bench=.*` executes all these functions; they will call the functions in the code a very large number of times N (e.g. N = 1000000), show this N and the average execution time of the functions in ns (ns/op). If the functions are called with `testing.Benchmark`, you can also simply run the program.

For a concrete example see §14.16 where benchmarking is performed on an example with goroutines, and Exercise 13.3: `string_reverse_test.go`

13.8 Testing: a concrete example

In Exercise 11.2 you wrote a program `main_oddeven.go` which tests for the first 100 integers whether they are even or not. The function which does the check was contained in a package `even`. Following is a possible solution:

Listing 13.7—main_oddeven.go:

```
package main
import (
     "fmt"
     "./even/even"
)

func main() {
     for i:=0; i<=100; i++ {
            fmt.Printf("Is the integer %d even? %v\n", i, even.Even(i))
     }
}
```

This uses the package even in `even.go`:

Listing 13.8—even/even.go:

```
package even

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

func Odd(i int) bool {
return i%2 != 0
}
```

In the map of the even package, we make a test program `oddeven_test.go`:

Listing 13.9—even/oddeven_test.go:

```
package even
import "testing"

func TestEven(t *testing.T) {
     if !Even(10) {
            t.Log("10 must be even!")
            t.Fail()
     }
```

```
	if Even(7) {
		t.Log("7 is not even!")
		t.Fail()
	}
}

func TestOdd(t *testing.T) {
	if !Odd(11) {
		t.Log("11 must be odd!")
		t.Fail()
	}
	if Odd(10) {
		t.Log("10 is not odd!")
		t.Fail()
	}

}
```

Because testing uses concrete cases of input and we can never test all cases (most likely there is an infinite number) we must give some thought to the test cases we are going to use.

We should at least include:

- a normal case
- abnormal cases (wrong input like negative numbers or letters instead of numbers, no input)
- boundary cases (if a parameter has to have a value in the interval 0-1000, then check 0 and 1000)

Just issue `go install even`, or alternatively we then create a Makefile with content:

```
include $(GOROOT)/src/Make.inc
TARG=even
GOFILES=\
	even.go\

include $(GOROOT)/src/Make.pkg
```

Issuing the command `make (or gomake)` creates the package archive `even.a`

The testcode must not be referred to in the GOFILES parameter, because we don't want our test code in our production program. If you include it, go test will give errors! Gotest will make its own executable containing the test code in a separate map _test.

We can now test our `even` package with the command: `go test (or make test).`

Because the test-functions in Listing 13.5 do not invoke t.Log or t.Fail, this gives us as result: PASS. In this simple example everything works fine.

To see output in case of a failure, change the function TestEven to:

```
func TestEven(t *testing.T) {
         if Even(10) {
            t.Log("Everything OK: 10 is even, just a test to see failed output!")
            t.Fail()
         }
}
```

now invokes t.Log and t.Fail, giving us as result:

```
--- FAIL: even.TestEven (0.00 seconds)
Everything OK: 10 is even, just a test to see failed output!
FAIL
```

Exercise 13.4: `string_reverse_test.go`

Write a unit-test for the program from Exercise 7.11 `string_reverse.go.`

Put string_reverse in its own package `strev`, only containing the exported function reverse.

Go test it !

13.9 Using table-driven tests.

When writing tests it is good practice to use an array to collect the test-inputs and the expected results together: each entry in the table then becomes a complete test case with inputs and expected results, and sometimes with additional information such as a test name to make the test output more informative.

The actual test simply iterates trough all table entries and for each entry performs the necessary tests; this was applied in Exercise 13.4 .

It could be abstracted in the following snippet:

```
var tests = []struct{          // Test table
        in  string
        out string
}{
        {“in1”, “exp1”},
        {“in2”, “exp2”},
        {“in3”, “exp3”},
  ...
}

func TestFunction(t *testing.T) {
        for i, tt := range tests {
                s := FuncToBeTested(tt.in)
                if s != tt.out {
                       t.Errorf(“%d. %q => %q, wanted: %q”, i, tt.in, s, tt.out)
                }
        }
}
```

If a lot of test functions would have to be written that way, it could be useful to write the actual test in a helper function verify:

```
func verify(t *testing.T, testnum int, testcase, input, output, expected string)
{
         if input != output {
                t.Errorf(“%d. %s with input = %s: output %s != %s”, testnum,
                testcase, input, output, expected)
         }
}
```

so that `TestFunction` becomes:

```
func TestFunction(t *testing.T) {
       for i, tt := range tests {
                s := FuncToBeTested(tt.in)
                verify(t, i, “FuncToBeTested: “, tt.in, s, tt.out)
        }
}
```

13.10 Investigating performance: tuning and profiling Go programs

13.10.1 Time and memory consumption

A handy script to measure these is *xtime*:

```
#!/bin/sh
/usr/bin/time -f '%Uu %Ss %er %MkB %C' "$@"
```

Used on the Unix command-line as `xtime goprogexec`, where `progexec` is a Go executable progam, it shows an output like: `56.63u 0.26s 56.92r 1642640kB progexec`
giving respectively the user time, system time, real time and maximum memory usage.

13.10.2 Tuning with go test

If the code used the Go testing package's benchmarking support, we could use the `gotest` standard `-cpuprofile` and `-memprofile` flags, causing a CPU- or memory usage profile to be written to the file specified.

Example of use: `go test -x -v -cpuprofile=prof.out -file x_test.go`
compiles and executes the tests in x_test.go, and writes a cpuprofile to prof.out

13.10.3 Tuning with pprof

For a standalone program `progexec` you have to enable profiling by importing runtime/pprof ; this package writes runtime profiling data in the format expected by the pprof visualization tool. For CPU profiling you have to add a few lines of code:

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to file")

func main() {
    flag.Parse()
    if *cpuprofile != "" {
        f, err := os.Create(*cpuprofile)
        if err != nil {
            log.Fatal(err)
        }
        pprof.StartCPUProfile(f)
```

```
            defer pprof.StopCPUProfile()
        }
        ...
```

This code defines a flag named cpuprofile, calls the Go flag library to parse the command line flags, and then, if the cpuprofile flag has been set on the command line, starts CPU profiling redirected to that file. (`os.Create` makes a file with the given name in which the profile will be written).The profiler requires a final call to StopCPUProfile to flush any pending writes to the file before the program exits; we use defer to make sure this happens as main returns.

Now run the program with this flag: `progexec -`**`cpuprofile`**`=progexec.prof`
and then you can use the gopprof tool as: `gopprof progexec progexec.prof`

The gopprof program is a slight variant of Google's pprof C++ profiler; for more info on this tool, see http://code.google.com/p/google-perftools/.

When CPU profiling is enabled, the Go program stops about 100 times per second and records a sample consisting of the program counters on the currently executing goroutine's stack.

Some of the interesting commands of this tool are:

1) `top`*`N`*
   This shows the top *N* samples in the profile, e.g.: `top5`
   It shows the 10 most heavily used functions during the execution, an output like:
   ```
   Total: 3099 samples
          626 20.2%  20.2%      626 20.2% scanblock
          309 10.0%  30.2%      2839 91.6% main.FindLoops
          …
   ```
   The 5th column is an indicator of how heavy that function is used.

2) web or web funcname
   This command writes a graph of the profile data in SVG format and opens it in a web browser (there is also a gv command that writes PostScript and opens it in Ghostview. For either command, you need graphviz installed). The different functions are shown in rectangles (the more called the bigger), and arrows point in the direction of the function calls.

3) list funcname or weblist funcname

This shows a listing of the code lines in funcname, with in the 2nd column the time spent in executing that line, so this gives a very good indication of what code is most heavy in the execution.

If it is seen that the function `runtime.mallocgc` (which both allocates and runs periodic garbage collections) is heavily used, then it is time for memory profiling. To find out why the garbage collector is running so much, we have to find out what is allocating memory.

For this the following code has to be added at a judicious place:

```
var memprofile = flag.String("memprofile", "", "write memory profile to this
file")
...

CallToFunctionWhichAllocatesLotsOfMemory()
if *memprofile != "" {
        f, err := os.Create(*memprofile)
        if err != nil {
               log.Fatal(err)
        }
        pprof.WriteHeapProfile(f)
        f.Close()
        return
}
```

Now run the program with this flag: `progexec -memprofile=progexec.mprof`
and you can use the gopprof tool again as: `gopprof progexec progexec.mprof`

The same commands apply `top5, list funcname` etc, but now they measure memory allocation in Mb, here is a sample output of top:

```
Total: 118.3 MB
        66.1 55.8% 55.8% 103.7 87.7% main.FindLoops
        30.5 25.8% 81.6% 30.5 25.8% main.*LSG·NewLoop
        …
```

The topmost functions use the most memory, as seen in the 1st column.

An interesting tool that reports object allocation counts is also:

```
gopprof --inuse_objects progexec progexec.mprof
```

For web applications there is also a standard HTTP interface to profiling data. In an HTTP server, adding

```
import _ "http/pprof"
```

will install handlers for a few URLs under /debug/pprof/. Then you can run gopprof with a single argument—the URL to your server's profiling data—and it will download and examine a live profile.

```
gopprof http://localhost:6060/debug/pprof/profile # 30-second CPU profile
gopprof http://localhost:6060/debug/pprof/heap # heap profile
```

A concrete example is analysed in this excellent article on the Go-blog (ref. 15): Profiling Go Programs (Jun 2011).

Chapter 14—Goroutines and Channels

As expected of a 21st century programming language, Go comes with built-in support for *communication* between applications (networking, client-server, distributed computing, see chapter 15) and support for *concurrent* applications. These are programs that execute different pieces of code simultaneously, possibly on different processors or computers. The basic building blocks Go proposes for structuring concurrent programs are **goroutines** and **channels**. Their implementation requires support from the language, the compiler and the runtime. The garbage collection which Go provides is also essential for easy concurrent programming.

Do not communicate by sharing memory. Instead, share memory by communicating.

Communication forces coordination.

14.1 Concurrency, parallelism and goroutines

14.1.1 What are goroutines?

An application is a *process* running on a machine; a process is an independently executing entity that runs in its own address space in memory. A process is composed of one or more operating system *threads* which are simultaneously executing entities that share the same address space. Almost all real programs are *multithreaded,* so as not to introduce wait times for the user or the computer, or to be able to service many requests simultaneously (like web servers), or to increase performance and throughput (e.g. by executing code in parallel on different datasets). Such a *concurrent* application can execute on 1 processor or core using a number of threads, but it is only when the same application process executes at the same point in time on a number of cores or processors that it is truly called *parallelized*.

Parallelism is the ability to make things run quickly by using multiple processors. So concurrent programs may or may not be parallel.

Multithreaded applications are notoriously difficult to get right, the main problem is the shared data in memory, which can be manipulated by the different threads in a non-predictable manner, thereby delivering sometimes irreproducible and random results (called *racing conditions)*.

!! Do not use global variables or shared memory, they make your code unsafe for running concurrently !!

The solution lies in *synchronizing* the different threads, and *locking* the data, so that only one thread at a time can change data. Go has facilities for locking in its standard library in the package sync for when they're needed in lower level code; we have discussed them in § 9.3. But the past experience in software engineering has shown that this leads to complex, error-prone programming and diminishing performance, so this classic approach is clearly not the way to go for modern multicore and multiprocessor programming: the 'thread-per-connection'- model is not nearly efficient enough.

Go adheres to another, in many cases better suited paradigm, which is known as Communicating Sequential Processes (CSP, invented by C. Hoare) or also called the message passing-model (as applied in other languages such as Erlang).

The parts of an application that run concurrently are called *goroutines* in Go, they are in effect concurrently executing computations. There is no one-to-one correspondence between a goroutine and an operating system thread: a goroutine is mapped onto (multiplexed, executed by) one or more threads, according to their availability; this is accomplished by the goroutine-scheduler in the Go runtime.

Goroutines run in the same address space, so access to shared memory must be synchronized; this could be done via the sync package (see § 9.3), but this is highly discouraged: Go use *channels* to synchronize goroutines (see § 14.2 etc.)

When a goroutine is blocked by a system call (e.g. waiting for I/O), other goroutines continue to run on other threads. The design of goroutines hides many of the complexities of thread creation and management.

Goroutines are lightweight, much lighter than a thread. They have a very small footprint (use little memory and resources): they are created with a 4K memory stack-space on the heap. Because they are cheap to create, a great number of them can be started on the fly if necessary (in the order of 100 thousands in the same address space). Furthermore they use a segmented stack for dynamically growing (or shrinking) their memory-usage; stack management is automatic. The stacks are not managed by the garbage collector, instead they are freed directly when the goroutine exits.

Goroutines can run across multiple operating system threads, but crucially, they can also run *within* threads, letting you handle myriad tasks with a relatively small memory footprint. Goroutines time-slice on OS threads as it were, so you can have any number of goroutines being serviced by a smaller number of OS threads, and the Go runtime is smart enough to realize which of those goroutines is blocking something and go off and do something else.

Two styles of concurrency exist: deterministic (well-defined ordering) and non-deterministic (locking/mutual exclusion but order undefined). Go's goroutines and channels promote deterministic concurrency (e.g. channels with one sender, one receiver), which is easier to reason about. We will compare both approaches in a commonly occurring algorithm (the Worker-problem) in § 14.7

A goroutine is implemented as a function or method (this can also be an anonymous or lambda function) and called (invoked) with the keyword **go**. This starts the function running in parallel with the current computation but in the same address space and with its own stack, for example:

```
go sum(bigArray) // calculate sum in the background
```

The stack of a goroutine grows and shrinks as needed, there is no possibility for stack overflow; the programmer needn't be concerned about stack size. When the goroutine finishes it exits silently: nothing is returned to the function which started it.

The `main()` function which every Go program must have can also be seen as a goroutine, although it is not started with go. Goroutines may be run during program initialization (in the `init()` function).

When 1 goroutine is e.g. very processor-intensive you can call `runtime.Gosched()` periodically in your computation loops: this yields the processor, allowing other goroutines to run; it does not suspend the current goroutine, so execution resumes automatically. Using Gosched() computations are more evenly distributed and communication is not starved.

14.1.2 The difference between concurrency and parallelism

Go's concurrency primitives provide the basis for a good concurrency program design: expressing program structure so as to represent independently executing actions; so Go's emphasis is not in the 1st place on parallelism: concurrent programs may or may not be parallel. Parallelism is the ability to make things run quickly by using multiple processors. But it turns out most often that a well designed concurrent program also has excellent performing parallel capabilities.

In the current implementation of the runtime (Jan 2012) Go does not parallelize code by default, only a single core or processor is dedicated to a Go-program, regardless of how many goroutines are started in it; so these goroutines are running *concurrent*, they are not running in parallel: only one goroutine is running at a time.

This will probably change, but until then in order to let your program execute simultaneously by more cores, that is so that goroutines are really running in parallel, you have to use the variable GOMAXPROCS .

This tells the run-time how many goroutines shall execute simultaneously.

Also only the gc-compilers have a true implementation of goroutines, mapping them onto OS threads as appropriate. With the gccgo compiler, an OS thread will be created for each goroutine.

14.1.3 Using GOMAXPROCS

Under the gc compilers (6g or 8g) you must set GOMAXPROCS to more than the default value 1 to allow the run-time support to utilize more than one OS thread, that is all goroutines share the same thread unless GOMAXPROCS is set to a value greater than 1. When GOMAXPROCS is greater than 1, they run on a thread pool with that many threads.With the gccgo compiler GOMAXPROCS is effectively equal to the number of running goroutines. Suppose n is the number of processors or cores on the machine. If you set the environment variable `GOMAXPROCS >= n`, or call `runtime.GOMAXPROCS(n)`, then the goroutines are divided (distributed) among the n processors. More processors however don't mean necessarily a linear improvement in performance, mainly because more communication is needed: the message-passing overhead increases. An experiential rule of thumb seems to be that for n cores setting `GOMAXPROCS` to `n-1` yields the best performance, and the following should also be followed: `number of goroutines > 1 + GOMAXPROCS > 1`

So if there is only one goroutine executing at a certain point in time, don't set GOMAXPROCS!

Here are some other observations from experiments: on a 1 CPU laptop performance improved when GOMAXPROCS was increased to 9. On a 32 core machine, the best performance was reached with GOMAXPROCS=8, a higher number didn't increase performance in that benchmark. Very large values of GOMAXPROCS degraded performance only slightly; using the "H" option to "top" showed only 7 active threads, for GOMAXPROCS=100.

Programs that perform concurrent computation should benefit from an increase in GOMAXPROCS; `see goroutine_select2.go`

Summarized: GOMAXPROCS is equal to the number of (concurrent) threads, on a machine with more than 1 core, as many threads as there are cores can run in parallel.

14.1.4 How to specify the number of cores to be used on the command-line?

Use the flags package, as in:

```
var numCores = flag.Int(“n”, 2, “number of CPU cores to use”)

in main():
flag.Parse()
runtime.GOMAXPROCS(*numCores)
```

A goroutine can stop itself by calling `runtime.`**`Goexit`**`()`, although that’s rarely necessary.

Listing 14.1—goroutine1.go introduces the concept:

```
package main
import (
        “fmt”
        “time”
)

func main() {
        fmt.Println(“In main()”)
        go longWait()
        go shortWait()
        fmt.Println(“About to sleep in main()”)
        // sleep works with a Duration in nanoseconds (ns) !
        time.Sleep(10 * 1e9)  fmt.Println(“At the end of main()”)
}

func longWait() {
        fmt.Println(“Beginning longWait()”)
        time.Sleep(5 * 1e9) // sleep for 5 seconds
        fmt.Println(“End of longWait()”)
}

func shortWait() {
        fmt.Println(“Beginning shortWait()”)
        time.Sleep(2 * 1e9) // sleep for 2 seconds
        fmt.Println(“End of shortWait()”)
}
```

```
Output:  In main()
```

```
About to sleep in main()
Beginning longWait()
Beginning shortWait()
End of shortWait()
End of longWait()
At the end of main() // after 10s
```

The 3 functions main(), longWait() and shortWait() are started in this order as independent processing units, and then work in parallel. Each function outputs a message at its beginning and at the end of its processing. To simulate their processing times, we use the Sleep-function from the time package. Sleep() pauses the processing of the function or goroutine for the indicated amount of time, which is given in nanoseconds (ns, the notation 1e9 represents 1 times 10 to the 9^{th} power, e = exponent).

They print their messages in the order in which we expect, always the same, but we see clearly that they work simultaneously, in parallel. We let main() pause for 10s, so we are sure that it will terminate after the two goroutines. If not (if we let main() stop for only 4s), main() stops execution earlier and longWait() doesn't get the chance to complete. If we do not wait in main(), the program stops and the goroutines die with it.

When the function main() returns, the program exits: it does not wait for other (non-main) goroutines to complete. That is the reason why in server-programs, where each request is handled by a response started as a goroutine, the server() function must be kept live. This is usually done by starting it as an infinite loop.

Moreover goroutines are independent units of execution and when a number of them starts one after the other you cannot depend on when a goroutine will actually be started. *The logic of your code must be independent of the order in which goroutines are invoked.*

To contrast this with a one thread, successive execution, remove the go keywords, and let the program run again.

```
Now the output is:      In main()
                        Beginning longWait()
                        End of longWait()
                        Beginning shortWait()
                        End of shortWait()
                        About to sleep in main()
                        At the end of main() // after 17 s
```

A more useful example of using goroutines could be the search for an item in a very large array.

Divide the array in a number of non-overlapping slices, and start a goroutine on each slice with the search-algorithm. In this way a number of parallel threads can be used for the search-task, and the overall search-time will certainly be decreased (divided by the number of goroutines).

14.1.5 Goroutines and coroutines

Other languages like C#, Lua and Python have a concept of *coroutines*. The name indicates that there is similarity with goroutines, but there are 2 differences:

- goroutines imply parallelism (or can deployed in parallel), coroutines in general do not
- goroutines communicate via channels; coroutines communicate via yield and resume operations

Goroutines are much more powerful than coroutines, and it is easy to port coroutine logic to goroutines.

14.2 Channels for communication between goroutines

14.2.1 Concept

In our first examples the goroutines executed independently, they did not communicate. Of course to be more useful, they have to communicate: *sending* and *receiving* information between them and coordinating / synchronizing their efforts. Goroutines could communicate by using shared variables, but this is highly discouraged because this way of working introduces all the difficulties with shared memory in multi-threading.

Instead Go has a special type, the *channel*, which is a like a conduit (pipe) through which you can send typed values and which takes care of communication between goroutines, avoiding all the pitfalls of shared memory; the very act of communication through a channel guarantees synchronization. Data are passed around on channels: *only one goroutine has access to a data item at any given time: so data races cannot occur, by design*. The ownership of the data (that is the ability to read and write it) is passed around.

A useful analogy is to compare a channel with a conveyor belt in a factory. One machine (the producer goroutine) puts items onto the belt, and another machine (the consumer goroutine) takes them off for packaging.

Channels serve the dual purpose of *communication*—the exchange of a value—with *synchronization*—guaranteeing that two calculations (goroutines) are in a known state at any time.

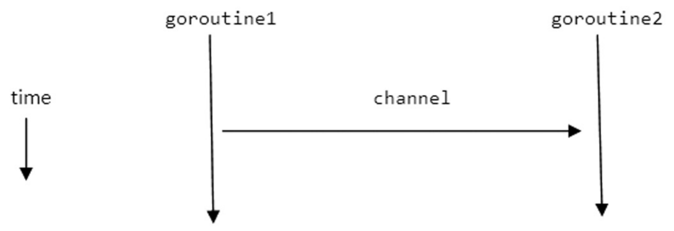


Fig 14.1: Channels and goroutines

The declaration of a channel is in the general format: `var identifier chan datatype`

The value of an uninitialized channel is `nil`.

So a channel can only transmit data-items of one datatype, e.g. `chan int or chan string`, but all types can be used in a channel, also the empty interface{}. It is even possible (and sometimes useful) to create a channel of channels.

A channel is in fact a typed message queue: data can be transmitted through it. It is a First In First Out (FIFO) structure and so they preserve the order of the items that are sent into them (for those who are familiar with it, a channel can be compared to a two-way pipe in Unix shells). A channel is also a *reference* type, so we have to use the make() function to allocate memory for it. Here is a declaration of a channel ch1 of strings, followed by its creation (instantiation):

```
var ch1 chan string
ch1 = make(chan string)
```

But of course this can be shortened to: `ch1 := make(chan string)`

And here we construct a channel of channels of int: `chanOfChans := make(chan chan int)`

Or a channel of functions: `funcChan := chan func()` (for an example of its use see § 14.17).

So channels are first class objects: they can be stored in variables, passed as arguments to functions, returned from functions and sent themselves over channels. Moreover they are typed, allowing the type system to catch programming errors like trying to send a pointer over a channel of integers.

14.2.2 Communication operator <-

This operator represents very intuitively the transmission of data: information flows in the direction of the arrow.

To a channel (*sending*):

`ch <- int1` means: variable int1 is *sent* through the channel ch (binary operator, infix = send)

From a channel (*receiving*), 3 forms:

`int2 = <- ch` means: variable int2 *receives* data (gets a new value) from the channel ch (unary prefix operator, prefix = receive); this supposes int2 is already declared, if not it can be written as: **`int2 := <- ch`**

`<- ch` can on itself be used to take the (next) value from the channel, this value is effectively discarded, but can be tested upon, so the following is legal code:

```
if <-ch != 1000 {
…
}
```

The same operator <- is used for sending and receiving, but Go figures out depending on the operands what to do. Although not necessary, for readability the channel name usually starts with ch or contains 'chan'. *The channel send- and receive operations are atomic: they always complete without interruption.*The use of the communication operator is illustrated in example

Listing 14.2—goroutine2.go:

```
package main
import (
        "fmt"
        "time"
)

func main() {
        ch := make(chan string)
        go sendData(ch)
        go getData(ch)
        time.Sleep(1e9)
}

func sendData(ch chan string) {
        ch <- “Washington”
        ch <- “Tripoli”
        ch <- “London”
        ch <- “Beijing”
        ch <- “Tokio”
}

func getData(ch chan string) {
        var input string
        for {
               input = <-ch
               fmt.Printf(“%s “, input)
        }
}
Output:          Washington Tripoli London Beijing Tokio
```

In main() 2 goroutines are started: sendData() sends 5 strings over channel ch, getData() receives them one by one in order in the string input and prints what is received.

If 2 goroutines have to communicate, you must give them both the same channel as parameter for doing that.

Experiment what happens when you comment out `time.Sleep(1e9)`.

Here we see that synchronization between the goroutines becomes important:

- main() waits for 1 second so that both goroutines can come to completion, if this is not allowed sendData() doesn’t have the chance to produce its output.
- getData() works with an infinite for-loop: this comes to an end when sendData() has finished and ch is empty.
- if we remove one or all go—keywords, the program doesn’t work anymore, the Go runtime throws a *panic*:

```
---- Error run E:/Go/GoBoek/code examples/chapter 14/goroutine2.exe with
code Crashed ---- Program exited with code -2147483645:
```
panic: all goroutines are asleep—deadlock!

Why does this occur? The runtime is able to detect that all goroutines (or perhaps only one in this case) are waiting for something (to be able to read from a channel or write to a channel), which means the program can't possibly proceed. This is a form of *deadlock*, and the runtime is able to detect it for us.

Remark: Don't use print statements to indicate the order of sending to and receiving from a channel: this could be out of order with what actually happens due to the time lag between the print statement and the actual channel sending and receiving.

Exercise 14.4: Explain why if we put time.Sleep(1e9) at the beginning of the function getData(), there is no exception, but no output is produced either.

14.2.3 Blocking of channels

By default, communication is *synchronous* and *unbuffered*: sends do not complete until there is a receiver to accept the value. One can think of an unbuffered channel as if there is no space in the channel for data: there must be a receiver ready to receive data from the channel and then the sender can hand it over directly to the receiver. So channel send/receive operations *block* until the other side is ready:

1) A send operation on a channel (and the goroutine or function that contains it) blocks until a receiver is available for the same channel: if there's no recipient for the value on ch, no other value can be put in the channel: no new value can be sent in ch when the channel is not empty. So the send operation will wait until ch becomes available again: this is the case when the channel-value is received (can be put in a variable).
2) A receive operation for a channel blocks (and the goroutine or function that contains it) until a sender is available for the same channel: if there is no value in the channel, the receiver blocks.

Although this seems a severe restriction, it works well in most practical situations.

This is illustrated in program `channel_block.go`, where the goroutine pump sends integers in an infinite loop on the channel. But because there is no receiver, the only output is the number 0.

Listing 14.3—channel_block.go:

```
package main
import "fmt"

func main() {
        ch1 := make(chan int)
        go pump(ch1)         // pump hangs
        fmt.Println(<-ch1)   // prints only 0
}

func pump(ch chan int) {
        for i:= 0; ; i++ {
               ch <- i
        }
}
```

```
Output: 0
```

The pump() function which supplies the values for the channel is sometimes called a *generator*.

To unblock the channel define a function suck which reads from the channel in an infinite loop, see Listing 14.4—channel_block2.go:

```
func suck(ch chan int) {
        for {
               fmt.Println(<-ch)
        }
}
```

and start this as a goroutine in main():

```
go pump(ch1)
go suck(ch1)
time.Sleep(1e9)
```

Give the program 1s to run: now tens of thousands of integers appear on output.

Exercise 14.1: `channel_block3.go`: Demonstrate the blocking nature of channels by making a channel, starting a go routine which receives the value from the channel, but only after 15s have passed, and then after the goroutine putting a value on the channel. Print messages at the different stages and observe the output.

14.2.4 Goroutines synchronize through the exchange of data on one (or more) channel(s).

Communication is therefore a form of synchronization: two goroutines exchanging data through a channel synchronize at the moment of communication (*the rendez-vous of goroutines*). Unbuffered channels make a perfect tool for synchronizing multiple goroutines.

It is even possible that the two sides block each other, creating what is called a *deadlock situation*. The Go runtime will detect this and panic, stopping the program. A deadlock is almost always caused by bad program design.

We see that channel operations on unbuffered channels can *block*. The way to avoid this is to design the program such that blocking does not occur, or by using buffered channels.

Exercise 14.2: `blocking.go`

Explain why the following program throws a panic: `all goroutines are asleep - deadlock!`

```
package main
import (
        “fmt”
)

func f1(in chan int) {
        fmt.Println(<-in)
}

func main() {
        out := make(chan int)
        out <- 2
        go f1(out)
}
```

14.2.5 Asynchronous channels—making a channel with a buffer

An unbuffered channel can only contain 1 item and is for that reason sometimes too restrictive. We can provide for a buffer in the channel, whose capacity gets set in an extended make command, like this:

```
buf := 100
ch1 := make(chan string, buf)
```

`buf` is the number of elements (here strings) the channel can hold.

Sending to a buffered channel will not block unless the buffer is full (the capacity is completely used), and reading from a buffered channel will not block unless the buffer is empty.

The buffer capacity does not belong to the type, so it is possible (although perhaps dangerous) to assign channels with different capacity to each other, as long as they have the same element type. The built-in `cap` function on a channel returns this buffer capacity.

If the capacity is greater than 0, the channel is *asynchronous*: communication operations succeed without blocking if the buffer is not full (sends) or not empty (receives), and elements are received in the order they are sent. If the capacity is zero or absent, the communication succeeds only when both a sender and receiver are ready.

To synthesize:

```
ch := make(chan type, value)
```

```
value == 0   → synchronous, unbuffered
     (blocking)
value > 0    → asynchronous, buffered
     (non-blocking) up to value elements
```

If you use buffers in the channels, your program will react better to sudden increases in number of 'requests': it will react more elastically, or with the official term: it will be more *scalable*. But design your algorithm in the first place with unbuffered channels, and only introduce buffering when the former is problematic.

Exercise 14.3: `channel_buffer.go`: Demonstrate this by giving the channel in `channel_block3.go` a buffer and observe the difference in output.

14.2.6 Goroutine using a channel for outputting result(s)

In order to know when a calculation is done, pass a channel on which it can report back. In our example of `go sum(bigArray)`, this would be like:

```
ch := make(chan int)
go sum(bigArray, ch) // bigArray puts the calculated sum on ch
// ... do something else for a while
sum := <-ch // wait for, and retrieve the sum
```

We can also use a channel for synchronization purposes, thus effectively using it as what is called a *semaphore* in traditional computing. Or to put it differently: to discover when a process (in a goroutine) is done, pass it a channel with which it can signal it is done.

A common idiom used to let the main program block indefinitely while other goroutines run is to place `select {}` as the last statement in a main function.

But this can also be done by using a channel to let the main program wait until the goroutine(s) completes, the so called semaphore pattern, as discussed in the next section.

14.2.7 Semaphore pattern

This is illustrated in the following snippet: the goroutine `compute` signals its completion by putting a value on the channel ch, the main routine waits on <-ch until this value gets through.

On this channel we would expect to get a result back, like in:

```
func compute(ch chan int) {
    ch <- someComputation()  // when it completes, signal on the channel.
}

func main() {
    ch := make(chan int)    // allocate a channel.
    go compute(ch)          // start something in a goroutine
    doSomethingElseForAWhile()
    result := <-ch
}
```

But the signal could also be something else, not connected to the result, like in this lambda function goroutine:

```
ch := make(chan int)
go func() {
    // doSomething
    ch <- 1  // Send a signal; value does not matter.
}()
doSomethingElseForAWhile()
<-ch   // Wait for goroutine to finish; discard sent value.
```

Or in this snippet where we wait for 2 sort-goroutines, which each sort a part of a slice s, to complete:

```
done := make(chan bool)
// doSort is a lambda function, so a closure which knows the channel done:
```

```
doSort := func(s []int) {
    sort(s)
    done <- true
}
i := pivot(s)
go doSort(s[:i])
go doSort(s[i:])
<-done
<-done
```

In the following code snippet, we have a full-blown *semaphore pattern* where N computations `doSomething()` over a slice of float64's with that size are done in parallel, and a channel sem of exactly the same length (and contaninining items of type empty interface) is signaled (by putting a value on it) when each one of the computations is finished. To wait for all of the goroutines to finish, just make a receiving range-loop over the channel sem:

```
type Empty interface {}
var empty Empty
...
data := make([]float64, N)
res := make([]float64, N)
sem := make(chan Empty, N)     // semaphore
...
for i, xi := range data {
    go func (i int, xi float64) {
        res[i] = doSomething(i,xi)
        sem <- empty
    } (i, xi)
}
// wait for goroutines to finish
for i := 0; i < N; i++ { <-sem }
```

Notice the use of the closure: the current i, xi are passed to the closure as parameters, masking the i, xi variables from the outer for-loop. This allows each goroutine to have its own copy of i, xi; otherwise, the next iteration of the for-loop would update i, xi in all goroutines. On the other hand, the res slice is not passed to the closure, since each goroutine does not need a separate copy of it. The res slice is part of the closure's environment but is not a parameter.

14.2.8 Implementing a parallel for-loop

This is just what we did in the previous code-snippet of §14.2.7 : each iteration in the for-loop is done in parallel:

```
for i, v := range data {
        go func (i int, v float64) {
                doSomething(i, v)
                …
        } (i, v)
}
```

Computing the iterations of a for-loop in parallel could potentially give huge performance gains. But this is only possible when all of the iterations are completely independent of each other. Some languages like Fortress or other parallel frameworks implement this as a separate construct, in Go these are easily implemented with goroutines:

14.2.9 Implementing a semaphore using a buffered channel

Semaphores are a very general synchronization mechanism that can be used to implement mutexes (exclusive locks), limit access to multiple resources, solve the readers-writers problem, etc. There is no semaphore implementation in Go's sync package, but they can be emulated easily using a buffered channel:

- the capacity of the buffered channel is the number of resources we wish to synchronize
- the length (number of elements currently stored) of the channel is the number of resources currently being used
- the capacity minus the length of the channel is the number of free resources (the integer value of traditional semaphores)

We don't care about what is stored in the channel, only its length; therefore, we start by making a channel that has variable length but 0 size (in bytes):

```
type Empty interface {}
type semaphore chan Empty
```

We then can initialize a semaphore with an integer value which encodes the number of available resources N: `sem = make(semaphore, N)`

Now our semaphore operations are straightforward:

```
    // acquire n resources
    func (s semaphore) P(n int) {
        e := new(Empty)
        for i := 0; i < n; i++ {
            s <- e
        }
    }

    // release n resources
    func (s semaphore) V(n int) {
        for i := 0; i < n; i++ {
            <-s
        }
    }
```

This can for example be used to implement a mutex:

```
    /* mutexes */
    func (s semaphore) Lock() {
        s.P(1)
    }

    func (s semaphore) Unlock() {
        s.V(1)
    }

    /* signal-wait */
    func (s semaphore) Wait(n int) {
        s.P(n)
    }

    func (s semaphore) Signal() {
        s.V(1)
    }
```

Exercise 14.5: `gosum.go`: Use this idiom to write a program which starts a goroutine to perform a sum of 2 integers and then waits on the result to print it.

Exercise 14.6: `producer_consumer.go`: Using this idiom write a program with 2 goroutines, the first produces the numbers 0, 10, 20, . . . , 90 and puts them on a channel, the second reads from the channel and prints them. main() waits for both goroutines to have ended.

***IDIOM:* Channel Factory pattern**

Another pattern common in this style of programming goes as follows: instead of passing a channel as a parameter to a goroutine, let the function make the channel and return it (so it plays the role of a *factory*); inside the function a lambda function is called as a goroutine.

Applying this pattern to channel_block2.go gives us `Listing 14.5—channel_idiom.go:`

```
package main
import (
        "fmt"
        "time"
)
func main() {
        stream := pump()
        go suck(stream)
        // the above 2 lines can be shortened to: go suck( pump() )
        time.Sleep(1e9)
}

func pump() chan int {
        ch := make(chan int)
        go func() {
               for i := 0; ; i++ {
                      ch <- i
               }
        }()
        return ch
}

func suck(ch chan int) {
        for {
               fmt.Println(<-ch)
        }
}
```

14.2.10 For—range applied to channels

The range clause on for loops accepts a channel ch as an operand, in which case the for loops over the values received from the channel, like this:

```
for v := range ch {
        fmt.Printf("The value is %v\n",v)
}
```

It reads from the given channel ch until the channel is closed and then the code following for continues to execute. Obviously another goroutine must be writing to ch (otherwise the execution blocks in the for-loop) and must close ch when it is done writing. The function suck can apply this and also launch this action in a goroutine, our former program now becomes:

Listing 14.6—channel_idiom2.go:

```
package main
import (
        "fmt"
        "time"
)

func main() {
        suck(pump())
        time.Sleep(1e9)
}

func pump() chan int {
        ch := make(chan int)
        go func() {
                for i := 0; ; i++ {
                        ch <- i
                }
        }()
        return ch
}

func suck(ch chan int) {
        go func() {
                for v := range ch {
                        fmt.Println(v)
                }
        }()
}
```

***IDIOM:* Channel Iterator pattern**

This pattern uses the previous pattern from Listing 14.6 and can be applied in the common case where we have to populate a channel with the items of a `container` type which contains an index-addressable field `items`. For this we can define a method `Iter()` on the `container` type which returns a read-only channel (see §14.2.8) items, as follows:

```
func (c *container) Iter () <-chan items {
    ch := make(chan item)
    go func () {
        for i := 0; i < c.Len(); i++ {   // or use a for-range loop
            ch <- c.items[i]
        }
    } ()
    return ch
}
```

Inside the goroutine, a for-loop iterates over the elements in the container `c` (for tree or graph algorithms, this simple for-loop could be replaced with a depth-first search).

The code which calls this method can then iterate over the container like:

```
for x := range container.Iter() { … }
```

which can run in its own goroutine, so then the above iterator employs a channel and two goroutines (which may run in separate threads). Then we have a typical *producer-consumer pattern*. If the program terminates before the goroutine is done writing values to the channel, then that goroutine will not be garbage collected; this is by design. This seems like wrong behavior, but channels are for threadsafe communication. In that context, a goroutine hung trying to write to a channel that nobody will ever read from is probably a bug and not something you'd like to be silently garbage-collected.

***IDIOM:* Producer Consumer pattern**

Suppose we have a Produce() function which delivers the values needed by a Consume function. Both could be run as a separate goroutine, Produce putting the values on a channel which is read by Consume. The whole process could take place in an infinite loop:

```
for {
    Consume(Produce())
}
```

14.2.11 Channel directionality

A channel type may be annotated to specify that it may only send or only receive in certain code:

```
var send_only chan<- int            // channel can only receive data
var recv_only <-chan int            // channel can only send data
```

Receive-only channels (`<-chan T`) cannot be closed, because closing a channel is intended as a way for a sender to signal that no more values will be sent to the channel, so it has no meaning for receive-only channels. All channels are created bidirectional, but we can assign them to directional channel variables, like in this code snippet:

```
var c = make(chan int) // bidirectional
go source(c)
go sink(c)

func source(ch chan<- int) {
  for { ch <- 1 }
}

func sink(ch <-chan int) {
  for { <-ch }
}
```

IDIOM: Pipe and filter pattern

A more concrete example would be a goroutine processChannel which processes what it receives from an input channel and sends this to an output channel:

```
sendChan := make(chan int)
reciveChan := make(chan string)
go processChannel(sendChan, receiveChan)

func processChannel(in <-chan int, out chan<- string) {
        for inValue := range in {
                result:= ... // processing inValue
        out <- result
        }
}
```

By using the directionality notation we make sure that the goroutine will not perform unallowed channel operations.

Here is an excellent and more concrete example taken from the Go Tutorial which prints the prime numbers at its output, using filters ('sieves') as its algorithm. Each prime gets its own filter, like in this schema:

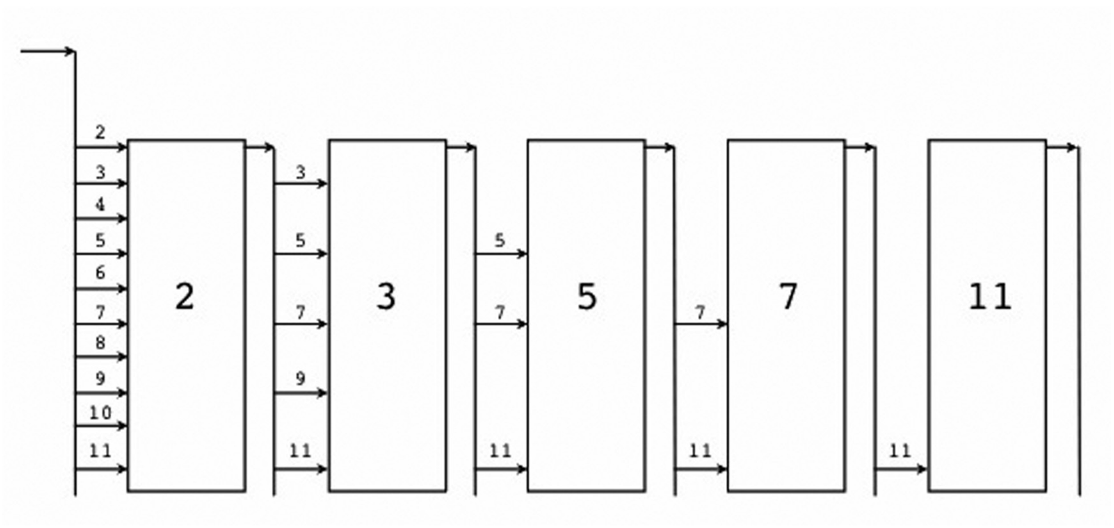


Fig 14.2: The sieve prime-algorithm

Version 1: Listing 14.7—sieve1.go:

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.package main
package main
import "fmt"

// Send the sequence 2, 3, 4, ... to channel ch.
func generate(ch chan int) {
        for i := 2; ; i++ {
                ch <- i // Send i to channel ch.
        }
}

// Copy the values from channel in to channel out,
// removing those divisible by prime.
func filter(in, out chan int, prime int) {
         for {
                i := <-in      // Receive value of new variable i from in.
                if i%prime != 0 {
                       out <- i // Send i to channel out.
                }
         }
}

// The prime sieve: Daisy-chain filter processes together.
func main() {
         ch := make(chan int)  // Create a new channel.
         go generate(ch)       // Start generate() as a goroutine.
         for {
                prime := <-ch
                fmt.Print(prime, " ")
                ch1 := make(chan int)
                go filter(ch, ch1, prime)
                ch = ch1
         }
}
```

The goroutine `filter(in, out chan int, prime int)` copies integers to the output channel discarding anything divisible by prime. So for each prime a new goroutine is launched, working together in the process: the generator and filters execute concurrently.

Output: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013…….

In the second version the idiom described above is applied: the functions `sieve, generate,` and `filter` are factories; they make a channel and return it, and they use lambda functions as goroutines. The main routine is now very short and clear: it calls `sieve()`,which returns a channel containing the primes, and then the channel is printed out via `fmt.Println(<-primes)`.

Version 2: Listing 14.8—sieve2.go:

```
package main
import “fmt”

// Send the sequence 2, 3, 4, ... to returned channel
func generate() chan int {
	ch := make(chan int)
	go func() {
		for i := 2; ; i++ {
			ch <- i
		}
	}()
	return ch
}
// Filter out input values divisible by prime, send rest to returned channel
func filter(in chan int, prime int) chan int {
	out := make(chan int)
	go func() {
		for {
			if i := <-in; i%prime != 0 {
				out <- i
			}
		}
	}()
	return out
}

func sieve() chan int {
	out := make(chan int)
	go func() {
		ch := generate()
		for {
			prime := <-ch
			ch = filter(ch, prime)
			out <- prime
		}
	}()
	return out
}
```

```
func main() {
        primes := sieve()
        for {
              fmt.Println(<-primes)
        }
}
```

14.3 Synchronization of goroutines: closing a channel—testing for blocked channels

Channels can be *closed* explicitly; however they are not like files: you don't usually need to close them. Closing of a channel is only necessary when the receiver must be told there are no more values coming. Only the sender should close a channel, never the receiver.

Continuing with the example goroutine2.go (listing 14.2): how can we signal when sendData() is done with the channel, and how can getData() detect that the channel is closed or blocked ?

The first is done with the function **close**(ch) : this marks the channel as unable to accept more values through a send operation <-; sending to or closing a closed channel causes a run-time panic. It is good practice to do this with a defer-statement immediately after the making of the channel (when it is appropriate in the given situation):

```
ch := make(chan float64)
defer close(ch)
```

The second is done with the comma, ok operator: this tests whether the channel is closed and then returns true, otherwise false.

How can we test that we can receive without blocking (or that channel ch is not closed)?

```
v, ok := <-ch   // ok is true if v received value
```

Often this is used together with an if-statement:

```
if v, ok := <-ch; ok {
       process(v)
}
```

Or when the receiving happens in a for loop, use break when ch is closed or blocked:

```
v, ok := <-ch
if !ok {
    break
}
process(v)
```

We can trigger the behavior of a non-blocking send by writing: _ = ch <- v because the blank identifier takes whatever is send on ch. Using these the program from listing 14.2 is improved to version goroutine3.go, which produces the same output.

To do a non-blocking channel read you need to use select (see § 14.4)

Listing 14.9—goroutine3.go:

```
package main
import "fmt"

func main() {
        ch := make(chan string)
        go sendData(ch)
        getData(ch)
}

func sendData(ch chan string) {
        ch <- "Washington"
        ch <- "Tripoli"
        ch <- "London"
        ch <- "Beijing"
        ch <- "Tokio"
        close(ch)
}
func getData(ch chan string) {
        for {
                input, open := <-ch
                if !open {
                        break
                }
                fmt.Printf("%s ", input)
        }
}
```

Here is what is changed in the code:

- only sendData() is now a goroutine, getData() runs in the same thread as main():
  ```
  go sendData(ch)
  getData(ch)
  ```
- at the end of the function sendData(), the channel is closed:

```
        func sendData(ch chan string) {
                ch <- "Washington"
                ch <- "Tripoli"
                ch <- "London"
                ch <- "Beijing"
                ch <- "Tokio"
                close(ch)
        }
```

- in the for-loop in getData(), before every receive the channel is tested with `if !open`:

```
                for {
                    input, open := <-ch
                    if !open {
                       break
                    }
                    fmt.Printf("%s ", input)
                }
```

It is even better practice to read the channel with a *for-range* statement, because this will automatically detect when the channel is closed:

```
for input := range ch {
        process(input)
}
```

Blocking and the producer-consumer pattern:

In the channel iterator pattern from §14.2.10 the relationship between the two goroutines is such that one is usually blocking the other. If the program runs on a multicore machine, only one processor will be employed most of the time. This can be ameliorated by using a channel with a greater than 0 buffer size. For example, with a buffer of size 100, the iterator can produce at least 100 items from the container before blocking. If the consumer goroutine is running on a separate processor, it is possible that neither goroutine will ever block.

Since the number of items in the container is generally known, it makes sense to use a channel with enough capacity to hold all the items. This way, the iterator will never block (though the consumer goroutine still might). However, this effectively doubles the amount of memory required to iterate over any given container, so channel capacity should be limited to some maximum number. Timing or benchmarking your code will help you find the buffer capacity for minimal memory usage and optimal performance.

14.4 Switching between goroutines with select

Getting the values out of different concurrently executing goroutines can be accomplished with the `select` keyword, which closely resembles the switch control statement (Chapter 5 § 5.3) and is sometimes called the *communications switch*; it acts like an are you ready polling mechanism; `select` listens for incoming data on channels, but there could also be cases where a value is sent on a channel.

```
select {
case u:= <- ch1:
        …
case v:= <- ch2:
        …
        …
default: // no value ready to be received
        …
}
```

The default clause is optional; fall through behavior, like in the normal switch, is not permitted. A select is terminated when a break or return is executed in one of its cases.

What select does is: it chooses which of the multiple communications listed by its cases can proceed.

- if all are blocked, it waits until one can proceed
- if multiple can proceed, it chooses one at random.
- when none of the channel operations can proceed and the default clause is present, then this is executed: the default is always runnable (that is: ready to execute).

Using a send operation in a select statement with a default case guarantees that the send will be non-blocking! If there are no cases, the select blocks execution forever.

The select-statement implements a kind of listener-pattern, and so it is mostly used within a(n infinite) loop; when a certain condition is reached, the loop is exited via a break-statement.

In program `goroutine_select.go` there are 2 channels ch1 and ch2 and 3 goroutines pump1(), pump2() and suck(). This is a typical *producer-consumer* pattern.

In infinite loops ch1 and ch2 are filled with integers through pump1() and pump2();suck() polls for input also in a non ending loop, takes the integers in from ch1 and ch2 in the select clause, and outputs them. The case that is chosen depends upon which channel information is received. The program is terminated in main after 1 second.

Listing 14.10—goroutine_select.go:

```
package main
import (
        “fmt”
        “time”
        “runtime”
)

func main() {
        runtime.GOMAXPROCS(2) // in goroutine_select2.go
        ch1 := make(chan int)
        ch2 := make(chan int)

        go pump1(ch1)
        go pump2(ch2)
        go suck(ch1, ch2)
        time.Sleep(1e9)
}

func pump1(ch chan int) {
        for i:=0; ; i++ {
               ch <- i*2
        }
}

func pump2(ch chan int) {
        for i:=0; ; i++ {
               ch <- i+5
        }
}

func suck(ch1 chan int,ch2 chan int) {
        for {
               select {
               case v:= <- ch1:
```

```
                        fmt.Printf("Received on channel 1: %d\n", v)
                case v:= <- ch2:
                        fmt.Printf("Received on channel 2: %d\n", v)
                }
        }
}
```

Output:

```
Received on channel 2: 5
Received on channel 2: 6
Received on channel 1: 0
Received on channel 2: 7
Received on channel 2: 8
Received on channel 2: 9
Received on channel 2: 10
Received on channel 1: 2
Received on channel 2: 11
…
Received on channel 2: 47404
Received on channel 1: 94346
Received on channel 1: 94348
```

The output produced in 1 s is quite amazing, if we count it (goroutine_select2.go) we get around 90000 numbers.

EXERCISES:

Exercise 14.7:

a) In Exercise 5.4 `for_loop.go` we had a simple for loop printing numbers. Implement the for loop in a function `tel` which is started as a goroutine and in which the numbers are send to a channel. The main() routine takes them from the channel and prints them. Don't use time.Sleep() for synchronization: `goroutine_panic.go`

b) Probably your solution will work, but you get a runtime panic: throw: all goroutines are asleep—deadlock! Why is this ? How could you solve this ? goroutine_close.go

c) Solve the problem from a) in another way: use a second channel which is passed to the goroutine, this signals its end by putting something on that channel. The main() routine checks whether something is send along that channel, and if so stops: goroutine_select.go

Exercise 14.8: Starting from the Fibonacci-program in Listing 6.10, make a solution which isolates the calculation of the Fibonacci-terms in a goroutine, which sends these results on a channel.

Close the channel when finished. The main() function reads from the channel and prints the results: `gofibonacci.go`

Write a shorter variant `gofibonacci2.go` using the algorithm from exercise 6.9

Write a variant which uses the select statement and a quit channel (`gofibonacci_select.go`).

Remark: When timing the results and comparing with 6.10, we see that the overhead of communication via a channel has a slight negative effect; for this simple algorithm using a goroutine is not the best performing choice; but `gofibonacci3.go` is a solution with 2 goroutines which is 3x faster.

Exercise 14.9: Create a random bit generator, that is a program which produces an unending sequence of randomly generated 0's and 1s: random_bitgen.go

Exercise 14.10: `polar_to_cartesian.go`

(This is kind of a synthesis exercise, it uses techniques from chapters 4, 9, 11 and of course this chapter.) Write an interactive console program that asks the user for the polar coordinates of a 2dimensional point (radius and angle (degrees)). Calculate the corresponding Cartesian coordinates x and y, and print out the result. Use structs for polar and Cartesian.

Use channels and a goroutine: a channel1 to receive the polars
a channel2 to receive the Cartesian

The conversion itself must be done with a goroutine, which reads from channel1 and sends to channel2. In reality for such a simple calculation it is not worthwhile to use a goroutine and channels, but if it would be a heavy computation taking some time, this solution-design would be quite appropriate.

Exercise 14.11: `concurrent_pi.go / concurrent_pi2.go`

Calculate pi with the following series using goroutines: start a goroutine to calculate each term in the formule and let it put the result on a channel, the main() function collects and sums the results, and prints the approximation of pi.

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots.$$

Measure the execution time (see § 6.11)

Again this is just for fun and exercising the goroutine concept.

If you need it use Pi from math.pi; and such a series is much more quickly calculated without goroutines. In a second version: use GOMAXPROCS, and start only as much goroutines as you set GOMAXPROCS to.

IDIOM: `Server backend pattern`

Often a server is implemented as a background goroutine which loops forever, in the loop getting and processing values of channels via a select:

```
// Backend goroutine.
   func backend() {
       for {
           select {
           case cmd := <-ch1:
               // Handle ...
           case cmd := <-ch2:
                   ...
           case cmd := <-chStop:
               // stop server
           }
       }
   }
```

Other parts of the application send values on the channels ch1, ch2, etc.; a stop channel is used for a clean termination of the server process.

Another possible (but less flexible) pattern is that all (client)-requests post their request on a chRequest, and the backend routine loops over this channel, processing requests according to their nature in a switch:

```
func backend() {
        for req := range chRequest {
           switch req.Subject() {
            case A1:  // Handle case ...
            case A2:  // Handle case ...
            default:
               // Handle illegal request ...
               // ...
           }
       }
   }
```

14.5 Channels, Timeouts and Tickers

The time package has some interesting functionality to use in combination with channels.

It contains a struct `time.Ticker` which is an object that repeatedly sends a time value on a contained channel C at a specified time interval:

```
type Ticker struct {
    C <-chan Time // the channel on which the ticks are delivered.
    // contains filtered or unexported fields
     …
}
```

The time interval ns is specified (in nanoseconds as an int64) is specified as a variable dur of type Duration in the factory function `time.NewTicker`:   `func NewTicker(dur) *Ticker`

It can be very useful when during the execution of goroutines something (logging of a status, a printout, a calculation, etc.) has to be done periodically at a certain time interval.

A Ticker is stopped with Stop(), use this in a defer statement. All this fits nicely in a select statement:

```
ticker := time.NewTicker(updateInterval)
defer ticker.Stop()
…
select {
case u:= <- ch1:
         …
case v:= <- ch2:
        …
case <- ticker.C:
        logState(status) // call some logging function logState
default: // no value ready to be received
        …
}
```

The `time.Tick()` function with signature `func Tick(d Duration) <-chan Time` is useful when you only need access to the return channel and don't need to shutdown it: it sends out the time on the return channel with periodicity d, which is a number of nanoseconds. Handy to use when you have to limit the rate of processing per unit time like in the following code snippet (the function client. Call( ) is an RPC-call not further specified here (see § 15.9)):

```
import "time"

rate_per_sec := 10
var dur Duration = 1e9 / rate_per_sec
chRate := time.Tick(dur) // a tick every 1/10th of a second
for req := range requests {
    <- chRate // rate limit our Service.Method RPC calls
    go client.Call("Service.Method", req, ...)
}
```

The net effect is that new requests can are only handled at the indicated rate: the channel `chRate` blocks higher rates. The rate per second can be increased or decreased according to the load and / or the resources of the machine.

Question 14.1: Expanding on the snippet above, think about how you could allow handling of periodic bursts in the number of requests (hint: use a buffered channel and a Ticker object).

A `Timer` type looks exactly the same as a `Ticker` type (it is constructed with `NewTimer(d Duration)`) but it sends the time only once, after a Duration d.

There is also a function `time.After(d)` with the signature:

```
func After(d Duration) <-chan Time
```

After Duration d the current time is sent on the returned channel; so this is equivalent to `NewTimer(d).C`; it resembles Tick(), but After() sends the time only once. The following listing shows a very concrete example, and also nicely illustrates the default clause in select:

Listing 14.11: timer_goroutine.go:

```
package main

import (
        "fmt"
        "time"
)

func main() {
        tick := time.Tick(1e8)
        boom := time.After(5e8)
        for {
                select {
                case <-tick:
                        fmt.Println("tick.")
                case <-boom:
                        fmt.Println("BOOM!")
                        return
                default:
                        fmt.Println("    .")
                        time.Sleep(5e7)
                }
        }
}
/* Output:
    .
tick.
    .
    .
tick.
    .
    .
tick.
    .
    .
tick.
    .
    .
tick.
BOOM!
*/
```

IDIOM: Simple timeout pattern

We want to receive from a channel ch, but want to wait at most 1 second for the value to arrive. Start by creating a signalling channel and launching a lambda goroutine that sleeps before sending on the channel:

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1e9) // one second
    timeout <- true
}()
```

Then use a select statement to receive from either ch or timeout: if nothing arrives on ch in the 1 s time period, the timeout case is selected and the attempt to read from ch is abandoned.

```
select {
case <-ch:
    // a read from ch has occurred
case <-timeout:
    // the read from ch has timed out
    break
}
```

2nd variant: Abandon synchronous calls that run too long:

We could also use the time.After() function instead of a timeout-channel. This can be used in a select to signal a timeout or stop an execution of goroutines. When in the following code snippet client.Call does not return a value to channel ch after timeoutNs ns the timeout case is executed in the select:

```
ch := make(chan error, 1)
go func() { ch <- client.Call("Service.Method", args, &reply) } ()
select {
case resp := <-ch:
    // use resp and reply
case <-time.After(timeoutNs):
    // call timed out
    break
}
```

Note that the buffer size of 1 is necessary to avoid deadlock of goroutines and guarantee garbage collection of the timeout channel.

3rd variant: Suppose we have a program that reads from multiple replicated databases simultaneously. The program needs only one of the answers, and it should accept the answer that arrives first. The function Query takes a slice of database connections and a query string. It queries each of the databases in parallel and returns the first response it receives:

```
func Query(conns []Conn, query string) Result {
    ch := make(chan Result, 1)
    for _, conn := range conns {
        go func(c Conn) {
            select {
            case ch <- c.DoQuery(query):
            default:
            }
        }(conn)
    }
    return <- ch
}
```

Here again the result channel ch has to be buffered: this guarantees that the first send has a place to put the value and ensures that it will always succeed, so the first value to arrive will be retrieved regardless of the order of execution. An executing goroutine can always be stopped by calling **`runtime.Goexit()`**

Caching data in applications:

Applications working with data coming from a database (or in general a datastore) will often cache that data in memory, because retrieving a value from a database is a costly operation; when the database value does not change there is no problem with this. But for values that can change we need a mechanism that periodically rereads the value in the database: the cached value in that case becomes invalid (it has expired) and we don't want our application to present an old value to the user. The article discussed at http://www.tideland.biz/CachingValues presents a way to do this with a goroutine and a Ticker object.

14.6 Using recover with goroutines

One application of recover (see § 13.3) is to shut down a failing goroutine inside a server without killing the other executing goroutines.

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)   // start the goroutine for that work
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Printf(“work failed with %s in %v:”, err, work)
        }
    }()
    do(work)
}
```

In the code snippet above, if do(work) panics, the error will be logged and the goroutine will exit cleanly without disturbing the other goroutines.

Because recover always returns nil unless called directly from a deferred function, deferred code can call library routines that themselves use panic and recover without failing. As an example, the deferred function in `safelyDo()` might call a logging function before calling recover, and that logging code would run unaffected by the panicking state. With our recovery pattern in place, the do function (and anything it calls) can get out of any bad situation cleanly by calling panic. But the recovery has to take place inside the panicking goroutine: it cannot be recovered by a different goroutine. A more detailed and in depth treatment can be found at http://www.tideland.biz/SupervisingGoroutines (ref. 43).

14.7 Comparing the old and the new model: Tasks and Worker processes.

Suppose we have to perform a number of tasks; a task is performed by a worker (process). A Task can be defined as a struct (the concrete details are not important here):

```
type Task struct {
    // some state
}
```

1st (old) paradigm: *use shared memory to synchronize*

The pool of tasks is shared memory; in order to synchronize the work and to avoid race conditions, we have to guard the pool with a Mutex lock:

```
type Pool struct {
        Mu    sync.Mutex
        Tasks []Task
}
```

A sync.Mutex (see § 9.3) is a mutual exclusion lock: it serves to guard the entrance to a critical section in code: only one goroutine (thread) can enter that section at one time. If more than one goroutine would be allowed, a race-condition can exist: the Pool struct could no longer be updated correctly. In the traditional model (applied in most classic OO-languages like C++, Java, C#) the Worker process could be coded as:

```
func Worker(pool *Pool) {
    for {
        pool.Mu.Lock()
         // begin critical section:
        task := pool.Tasks[0]         // take the first task
        pool.Tasks = pool.Tasks[1:]   // update the pool of tasks
         // end critical section
        pool.Mu.Unlock()
        process(task)
    }
}
```

Many of these worker processes could run concurrently; they could certainly be started as goroutines. A worker locks the pool, takes the first task from the pool, unlocks the pool, and then processes the task. The lock guarantees that only one worker process at a time can access the pool: a task is assigned to one and only one process. If the lock would not be there, the processing of the worker-routine could be interrupted in the lines task := pool.Tasks[0] and pool.Tasks = pool.Tasks[1:] with abnormal results: some workers would not get a task, some tasks would be obtained by several workers. This locking synchronization works well for a few worker processes, but if the Pool is very big and we assign a large number of processes to work on it, the efficiency of the processing will be diminished by the overhead of the lock-unlock mechanism. This is the bottle-neck: performance will certainly decrease when the number of workers increases, drastically at a certain threshold.

2nd paradigm: *channels*

Now channels of Tasks are used to synchronize: a pending channel receives the requested tasks, a done channel receives the performed tasks (with their results). The worker process are started as goroutines, their number N should be adjusted to the number of tasks.

The main routine, which performs the function of Master, could be programmed as:

```
func main() {
    pending, done := make(chan *Task), make(chan *Task)
    go sendWork(pending)       // put tasks with work on the channel
    for i := 0; i < N; i++ {   // start N goroutines to do work
        go Worker(pending, done)
    }
    consumeWork(done)          // continue with the processed tasks
}
```

The worker process is very simple: take a task from the pending channel, processing it, putting the finished task on the done channel:

```
func Worker(in, out chan *Task) {
         for {
                t := <-in
                process(t)
                out <- t
         }
}
```

There is no locking: the process of getting a new task involves no contention. If the amount of tasks increases, the number of workers can be increased accordingly and the performance will not degrade nearly as badly as in the 1st solution. From the pending channel there is of course only 1 copy in memory, but there is no contention because the first Worker to get at the 1st pending task simply takes it (reading from and sending to a channel are atomic operations: see § 14.2.2) and will process it completely. It is impossible to predict which task will be performed by which process and vice versa. With an increasing number of workers there is also an increasing communication overhead, which has a slight impact on performance.

In this simple example it is perhaps difficult to see the advantage of the 2nd model, but applications with complex lock-situations are very hard to program and to get right, and a great deal of this complexity in the software is not needed in a solution which applies the 2nd model.

Thus not only performance is a major advantage, but the clearer and more elegant code is perhaps an even bigger advantage. It is certainly a Go idiomatic way of working:

IDIOM: **Use an in- and out-channel instead of locking**

```
func Worker(in, out chan *Task) {
```

```
            for {
                    t := <-in
                    process(t)
                    out <- t
            }
    }
```

For any problem which can be modeled as such a Master-Worker paradigm, an analogous solution with Workers as goroutines communicating through channels and the Master as coordinator would be a perfect fit. If the system distributes over several machines, a number of machines could execute the Worker goroutines, and the Master and Workers could communicate amongst themselves through netchan or rpc (see chapter 15).

What to use: a sync.Mutex or a channel?

Although in this chapter we laid strong emphasis on goroutines using channels because this is quite new in system languages, this doesn't mean that the classic approach with locking is now taboo: Go has both and gives you the choice according to the problem being solved: construct the solution which is the most elegant, simple and readable, and in most cases performance will follow automatically. Don't be afraid to use a Mutex if that fits your problem best. Go is pragmatic in letting you use the tools that solve your problem best and not forcing you into one style of code. As a general rule of thumb:

- use locking (mutexes) when:
    - caching information in a shared data structure
    - holding state information, that is context or status of the running application

- use channels when:
    - communicating asynchronous results
    - distributing units of work
    - passing ownership of data

If you find your locking rules are getting too complex, ask yourself whether using channel(s) might not be simpler.

14.8 Implementing a lazy generator

A *generator* is a function that returns the next value in a sequence each time the function is called, like:

```
generateInteger() => 0
generateInteger() => 1
generateInteger() => 2
....
```

It is a producer that only returns the next value, not the entire sequence; this is called *lazy evaluation*: only compute what you need at the moment, saving valuable resources (memory and CPU): it is a technology for the evaluation of expressions on demand. An example would be the generation of an endless sequence of even numbers: to generate it and then use those numbers one by one would perhaps be difficult and certainly would not fit into memory! But a simple function per type with a channel and a goroutine can do the job.

For example in listing 14.12 we see a Go implementation with a channel of a generator of ints. The channel is named yield and resume, the terms commonly used in coroutine code.

Listing 14.12—lazy_evaluation.go

```
package main
import (
        “fmt”
)

var resume chan int

func integers() chan int {
    yield := make (chan int)
    count := 0
    go func () {
        for {
            yield <- count
            count++
        }
    } ()
    return yield
}
func generateInteger() int {
    return <-resume
}

func main() {
    resume = integers()
    fmt.Println(generateInteger()) //=> 0
    fmt.Println(generateInteger()) //=> 1
    fmt.Println(generateInteger()) //=> 2
}
```

A subtle difference is that the value read from the channel could have been generated a while ago, it is not generated at the time of reading. If you need such a behavior, you have to implement a request-response mechanism. When the generator's task is computationally expensive and the order of generating results does not matter, then the generator can be parallelized internally by using goroutines. But be careful that the overhead generated by spawning many goroutines does not outweigh any performance gain.

These principles can be generalized: by making clever use of the empty interface, closures and higher order functions we can implement a generic builder `BuildLazyEvaluator` for the lazy evaluation function (this should best placed inside a utility package). The builder takes a function that has to be evaluated and an initial state as arguments and returns a function without arguments returning the desired value. The passed evaluation function has to calculate the next return value as well as the next state based on the state argument. Inside the builder a channel and a goroutine with an endless loop are created. The return values are passed to the channel from which they are fetched by the returned function for later usage. Each time a value is fetched the next one will be calculated. In the next example this is applied by defining an evenFunc whitch lazily generates even numbers: in main() `we` create the first 10 even numbers, each further call to even() returns the next one. For this purpose we had to specialize our general build function into `BuildLazyIntEvaluator`, and then we were able to define `even` on top of that.

Listing 14.13: general_lazy_evaluation1.go

```
package main
import (
        “fmt”
)

type Any interface{}
type EvalFunc func(Any) (Any, Any)

func main() {
        evenFunc := func(state Any) (Any, Any) {
                os := state.(int)
                ns := os + 2
                return os, ns
        }
        even := BuildLazyIntEvaluator(evenFunc, 0)

        for i := 0; i < 10; i++ {
                fmt.Printf(“%vth even: %v\n”, i, even())
        }
}

func BuildLazyEvaluator(evalFunc EvalFunc, initState Any) func() Any {
        retValChan := make(chan Any)
        loopFunc := func() {
               var actState Any = initState
               var retVal Any
               for {
                      retVal, actState = evalFunc(actState)
                      retValChan <- retVal
               }
        }
        retFunc := func() Any {
               return <-retValChan
        }
        go loopFunc()
        return retFunc
}

func BuildLazyIntEvaluator(evalFunc EvalFunc, initState Any) func() int {
        ef := BuildLazyEvaluator(evalFunc, initState)
        return func() int {
               return ef().(int)
        }
}
/* Output:
0th even: 0
1th even: 2
2th even: 4
3th even: 6
4th even: 8
5th even: 10
6th even: 12
7th even: 14
8th even: 16
9th even: 18
*/
```

Exercise 14.12: `general_lazy_evaluation2.go`

Use the general builder from Listing 14.12 to calculate the first 10 Fibonacci numbers.

Hint: Because these numbers grow quickly, use the uint64 type.

Note: This calculation is typically defined as a recursive function, but in languages without *tail recursion* such as Go this can lead to a stack overflow, but with Go's extensible stacks the optimization is less critical. The trick here is to use an imperative way for the calculation together with the lazy evaluation. The gccgo compiler does implement tail recursion in some cases.

14.9 Implementing Futures

A related idea is that of *futures*: sometimes you know you need to compute a value before you need to actually use the value. In this case, you can potentially start computing the value on another processor and have it ready when you need it.

Futures are easy to implement via closures and goroutines, the idea is similar to generators, except a future needs only to return one value.

An excellent example is given in ref. 18: suppose we have a type Matrix and we need to calculate the inverse of the product of 2 matrices a and b, first we have to invert both of them through a function Inverse(m), and then take the Product of both results. This could be done with the following function InverseProduct():

```
func InverseProduct(a Matrix, b Matrix) {
    a_inv := Inverse(a)
    b_inv := Inverse(b)
    return Product(a_inv, b_inv)
}
```

In this example it is known initially that the inverse of both a and b must be computed. Why should the program wait for a_inv to be computed before starting the computation of b_inv? These inverse computations can be done in parallel. On the other hand, the call to Product needs to wait for both a_inv and b_inv to finish. This can be implemented as follows:

```
func InverseProduct(a Matrix, b Matrix) {
    a_inv_future := InverseFuture(a)  // started as a goroutine
    b_inv_future := InverseFuture(b)  // started as a goroutine
    a_inv := <-a_inv_future
    b_inv := <-b_inv_future
    return Product(a_inv, b_inv)
}
```

where InverseFuture() launches a closure as a goroutine, which puts the resultant inverse matrix on a channel future as result:

```
func InverseFuture(a Matrix) {
    future := make(chan Matrix)
    go func() { future <- Inverse(a) }()
    return future
}
```

When developing a computationally intensive package, it may make sense to design the entire API around futures. The futures can be used within your package while maintaining a friendly API. In addition, the futures can be exposed through an asynchronous version of the API. This way the parallelism in your package can be lifted into the user's code with minimal effort. (See the discussion in ref. 18: http://www.golangpatterns.info/concurrency/futures)

14.10 Multiplexing

14.10.1 A typical client-server pattern

Client-server applications are the kind of applications where goroutines and channels shine.

A *client* can be any running program on any device that needs something from a server, so it sends a *request*. The *server* receives this request, does some work, and then sends a *response* back to the client. In a typical situation there are many clients (so many requests) and one (or a few) servers. An example we use all the time is the client browser, which requests a web page. A web server responds by sending the web page back to the browser.

In Go a server will typically perform a response to a client in a goroutine, so a goroutine is launched for every client-request. A technique commonly used is that the client-request itself contains a channel, which the server uses to send in its response.

For example the request is a struct like the following which embeds a reply channel:

```
type Request struct {
    a, b    int;
    replyc  chan int;  // reply channel inside the Request
}
```

Or more generally:

```
type Reply struct { ... }
type Request struct {
arg1, arg2, arg3 some_type
replyc chan *Reply
}
```

Continuing with the simple form, the server could launch for each request a function run() in a goroutine that will apply an operation op of type binOp to the ints and then send the result on the reply channel:

```
type binOp func(a, b int) int

func run(op binOp, req *Request) {
    req.replyc <- op(req.a, req.b)
}
```

The server routine loops forever, receiving requests from a `chan *Request` and, to avoid blocking due to a long-running operation, starting a goroutine for each request to do the actual work:

```
func server(op binOp, service chan *Request) {
    for {
        req := <-service; // requests arrive here
  // start goroutine for request:
        go run(op, req);  // don't wait for op to complete
    }
}
```

The server is started in its own goroutine by the function startServer:

```
func startServer(op binOp) chan *Request {
    reqChan := make(chan *Request);
    go server(op, reqChan);
    return reqChan;
}
```

startServer will be invoked in the main routine.

In the following test-example, 100 requests are posted to the server, only after they all have been sent do we check the responses in reverse order:

```
func main() {
        adder := startServer(func(a, b int) int { return a + b })
        const N = 100
        var reqs [N]Request
        for i := 0; i < N; i++ {
               req := &reqs[i]
               req.a = i
               req.b = i + N
               req.replyc = make(chan int)
               adder <- req  // adder is a channel of requests
        }
        // checks:
        for i := N - 1; i >= 0; i-- { // doesn't matter what order
               if <-reqs[i].replyc != N+2*i {
                      fmt.Println("fail at", i)
                      } else {
fmt.Println("Request ", i, "is ok!")
}
        }
        fmt.Println("done")
}
```

The code can be found in Listing 14.13—multiplex_server.go, the output is:

```
Request 99 is ok!
Request 98 is ok!
…
Request 1 is ok!
Request 0 is ok!
Done
```

This program only starts 100 goroutines. Execute the program for 100000 goroutines, and even then one sees that it finishes within a few seconds. This demonstrates how lightweight goroutines are: if we would start the same amount of real threads, the program quickly crashes.

Listing 14.14—multiplex_server.go:

```
package main
import "fmt"

type Request struct {
        a, b   int
        replyc chan int // reply channel inside the Request
}

type binOp func(a, b int) int


func run(op binOp, req *Request) {
        req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *Request) {
        for {
            req := <-service // requests arrive here
            // start goroutine for request:
            go run(op, req) // don’t wait for op
        }
}

func startServer(op binOp) chan *Request {
        reqChan := make(chan *Request)
        go server(op, reqChan)
        return reqChan
}

func main() {
        adder := startServer(func(a, b int) int { return a + b })
        const N = 100
        var reqs [N]Request
        for i := 0; i < N; i++ {
               req := &reqs[i]
               req.a = i
               req.b = i + N
               req.replyc = make(chan int)
               adder <- req
        }
        // checks:
        for i := N - 1; i >= 0; i-- { // doesn’t matter what order
               if <-reqs[i].replyc != N+2*i {
                      fmt.Println(“fail at”, i)
               } else {
                      fmt.Println(“Request “, i, “is ok!”)
               }
        }
        fmt.Println(“done”)
}
```

14.10.2 Teardown: shutdown the server by signaling a channel

In the previous version the server does not a clean shutdown when main returns; it is forced to stop. To improve this we can provide a second, quit channel to the server:

```
func startServer(op binOp) (service chan *Request, quit chan bool) {
        service = make(chan *Request)
        quit = make(chan bool)
        go server(op, service, quit)
        return service, quit
}
```

The server function then uses a select to choose between the service channel and the quit channel:

```
func server(op binOp, service chan *request, quit chan bool) {
        for {
               select {
               case req := <-service:
                      go run(op, req)
               case <-quit:
                      return
               }
        }
}
```

When a true value enters the quit channel, the server returns and terminates.

In main we change the following line:

```
adder, quit := startServer(func(a, b int) int { return a + b })
```

At the end of main we place the line: `quit <- true`

The complete code can be found in `multiplex_server2.go`, with the same output.

Listing 14.15—multiplex_server2.go:

```
package main
import "fmt"

type Request struct {
        a, b   int
        replyc chan int // reply channel inside the Request
}
```

```
type binOp func(a, b int) int

func run(op binOp, req *Request) {
	req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *Request, quit chan bool) {
	for {
		select {
		case req := <-service:
			go run(op, req)
		case <-quit:
			return
		}
	}
}

func startServer(op binOp) (service chan *Request, quit chan bool) {
	service = make(chan *Request)
	quit = make(chan bool)
	go server(op, service, quit)
	return service, quit
}

func main() {
	adder, quit := startServer(func(a, b int) int { return a + b })
	const N = 100
	var reqs [N]Request
	for i := 0; i < N; i++ {
		req := &reqs[i]
		req.a = i
		req.b = i + N
		req.replyc = make(chan int)
		adder <- req
	}
	// checks:
	for i := N - 1; i >= 0; i-- { // doesn’t matter what order
		if <-reqs[i].replyc != N+2*i {
			fmt.Println(“fail at”, i)
		} else {
```

```
                        fmt.Println("Request ", i, "is ok!")
                }
        }
        quit <- true
        fmt.Println("done")
}
```

Exercise 14.13: `multiplex_server3.go`: Using the former example, write a variant with a String() method on the Request struct which hows the server output; test the program with 2 requests:

```
req1 := &Request{3, 4, make(chan int)}
req2 := &Request{150, 250, make(chan int)}
…
// show the output:
fmt.Println(req1,"\n",req2)
```

14.11 Limiting the number of requests processed concurrently

This is easily accomplished using a channel with a buffer (see § 14.2.5), whose capacity is the maximum number of concurrent requests. The program `max_tasks.go` does nothing useful but contains the technique to do just that: no more than MAXREQS requests will be handled and processed simultaneously, because when the buffer of the channel sem is full, the function handle blocks and not other request can start, until a request is removed from sem. sem acts like a semaphore, a technical term for a flag variable in a program which signals a certain condition: hence the name.

Listing 14.16—max_tasks.go:

```
package main

const (
  AvailableMemory = 10 << 20  // 10 MB, for example
  AverageMemoryPerRequest = 10 << 10  // 10 KB
  MAXREQS = AvailableMemory / AverageMemoryPerRequest // here amounts to 1000
)

var sem = make(chan int, MAXREQS)

type Request struct {
        a, b   int
        replyc chan int
}

func process(r *Request) {
         // Do something
         // May take a long time and use a lot of memory or CPU
}

func handle(r *Request) {
    process(r)
    // signal done: enable next request to start
    // by making 1 empty place in the buffer
    <-sem
}

func Server(queue chan *Request) {
   for {
        sem <- 1
    // blocks when channel is full (1000 requests are active)
    // so wait here until there is capacity to process a request
    // (doesn't matter what we put in it)
request := <-queue
        go handle(request)
   }
}

func main() {
         queue := make(chan *Request)
         go Server(queue)
}
```

In this way the application makes optimal use of a limited resource like memory, by having goroutines synchronize their use of that resource using a buffered channel (the channel is used as a semaphore).

14.12 Chaining goroutines

The following demo-program `chaining.go` demonstrates again how easy it is to start a huge number of goroutines. Here this happens in a for-loop in main. After the loop 0 is inserted in the rightmost channel, the 100000 goroutines execute, and the result which is 100000 is printed in less than 1.5 s.

This program also demonstrates how the number of goroutines can be given on the command-line and parsed in through `flag.Int`, e.g. chaining -n=7000 generates 7000 goroutines.

Listing 14.17—chaining.go:

```
package main
import (
        "flag"
        "fmt"
)

var ngoroutine = flag.Int("n", 100000, "how many goroutines")

func f(left, right chan int) { left <- 1+<-right }

func main() {
        flag.Parse()
        leftmost := make(chan int)
        var left, right chan int = nil, leftmost
        for i := 0; i < *ngoroutine; i++ {
               left, right = right, make(chan int)
               go f(left, right)
        }
        right <- 0      // start the chaining
        x := <-leftmost // wait for completion
        fmt.Println(x)  // 100000, approx. 1,5 s
}
```

14.13 Parallelizing a computation over a number of cores

Suppose we have NCPU number of CPU cores: `const NCPU = 4 // e.g. 4 for a quadqore processor` and we want to divide a computation in NCPU parts, each running in parallel with the others.

This could be done schematically (we leave out the concrete parameters) as follows:

```
func DoAll() {
    sem := make(chan int, NCPU)  // Buffering optional but sensible.
    for i := 0; i < NCPU; i++ {
        go DoPart(sem)
    }
    // Drain the channel sem, waiting for NCPU tasks to complete
    for i := 0; i < NCPU; i++ {
        <-sem    // wait for one task to complete
    }
    // All done.
}

func DoPart(sem chan int) {
    // do the part of the computation
    }
    sem <- 1    // signal that this piece is done
}

func main() {
  runtime.GOMAXPROCS = NCPU
  DoAll()
}
```

- The function DoAll() makes a channel sem upon which each of the parallel computations will signal its completion; in a for loop NCPU goroutines are started, each performing 1/NCPU —th part of the total work. Each DoPart() goroutine signals its completion on sem.
- DoAll() waits in a for-loop until all NCPU goroutines have completed: the channel sem acts like a *semaphore;* this code shows a typical semaphore pattern (see § 14.2.7) .

In the present model of the runtime you also have to set GOMAXPROCS to NCPU (see § 14.1.3).

14.14 Parallelizing a computation over a large amount of data

Suppose we have to process a large number of data-items independent of each other, coming in through an in channel, and when completely processed put on an out channel, much like a factory pipeline. The processing of each data-item will also probably involve a number of steps: Preprocess / StepA / StepB / … / PostProcess

A typical sequential *pipelining algorithm* for solving this executing each step in order could be written as follows:

```
    func SerialProcessData (in <- chan *Data, out <- chan *Data) {
            for data := range in {
                    tmpA := PreprocessData(data)
```

```
                tmpB := ProcessStepA(tmpA)
                tmpC := ProcessStepB(tmpB)
                out <- PostProcessData(tmpC)
            }
        }
```

Only one step is executed at a time, and each item is processed in sequence: processing the 2nd item is not started before the 1st item is postprocessed and the result put on the out channel.

If you think about it, you will soon realize that this is a gigantic waste of time.

A much more efficient computation would be to let each processing step work independent of each other as a goroutine. Each step gets its input data from the output channel of the previous step. That way the least amount of time will be lost, and most of the time all steps will be busy executing:

```
func ParallelProcessData (in <- chan *Data, out <- chan *Data) {
        // make channels:
        preOut := make(chan *Data, 100)
        stepAOut := make(chan *Data, 100)
        stepBOut := make(chan *Data, 100)
        stepCOut := make(chan *Data, 100)
        // start parallel computations:
        go PreprocessData(in, preOut)
        go ProcessStepA(preOut, stepAOut)
        go ProcessStepB(stepAOut, stepBOut)
        go ProcessStepC(stepBOut, stepCOut)
        go PostProcessData(stepCOut, out
}
```

The channels buffer capacities could be used further to optimize the whole process.

14.15 The leaky bucket algorithm

Consider the following client-server configuration: the client goroutine performs an infinite loop receiving data from some source, perhaps a network; the data are read in buffers of type Buffer. To avoid too much allocating and freeing buffers, it keeps a free list of them, and uses a buffered channel to represent it: `var freeList = make(chan *Buffer, 100)`

This queue of reusable buffers is shared with the server. When receiving data the client tries to take a buffer from `freeList`; but if this channel is empty, a new buffer gets allocated. Once the message buffer is loaded, it is sent to the server on serverChan:

```
var serverChan = make(chan *Buffer)
```

Here is the algorithm for the client code:

```
func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not
        select {
case b = <-freeList:
      // Got one; nothing more to do
default:
      // None free, so allocate a new one
      b = new(Buffer)
        }
        loadInto(b)          // Read next message from the network
        serverChan <- b      // Send to server
    }
}
```

The server loop receives each message from the client, processes it, and tries to return the buffer to the shared free list of buffers:

```
func server() {
    for {
        b := <-serverChan      // Wait for work.
        process(b)
        // Reuse buffer if there’s room.
        select {
         case freeList <- b:
                // Reuse buffer if free slot on freeList; nothing more to do
         default:
                // Free list full, just carry on: the buffer is ‘dropped’
        }
    }
}
```

But this doesn’t work when `freeList` is full, in which case the buffer is ‘dropped on the floor’ (hence the name ‘leaky bucket’) to be reclaimed by the garbage collector.

14.16 Benchmarking goroutines.

In §13.7 we mentioned the principle of performing benchmarks on your functions in Go. Here we apply it to a concrete example of a goroutine which is filled with ints, and then read. The functions are called N times (e.g. N = 1000000) with `testing.Benchmark`, the `BenchMarkResult` has a `String()` method for outputting its findings. The number N is decided upon by gotest, judging this to be high enough to get a reasonable benchmark result.

Of course the same way of benchmarking also applies to ordinary functions.

If you want to exclude certain parts of the code or you want to be more specific in what you are timing, you can stop and start the timer by calling functions `testing.B.StopTimer()` and `testing.B.StartTimer()` as appropriate. The benchmarks will only be run if all your tests pass!

Listing 14.18—benchmark_channels.go:

```
package main
import (
        “fmt”
        “testing”
)

func main() {
        fmt.Println(“sync”, testing.Benchmark(BenchmarkChannelSync).String())
        fmt.Println(“buffered”,      testing.Benchmark(BenchmarkChannelBuffere
        d).String())
}

func BenchmarkChannelSync(b *testing.B) {
        ch := make(chan int)
        go func() {
               for i := 0; i < b.N; i++ {
                      ch <- i
               }
               close(ch)
        }()
        for _ = range ch {
        }
}
```

```
func BenchmarkChannelBuffered(b *testing.B) {
        ch := make(chan int, 128)
        go func() {
               for i := 0; i < b.N; i++ {
                      ch <- i
               }
               close(ch)
        }()
        for _ = range ch {
        }
}
/* Output:
Windows:        N              Time 1 op       Operations per sec
sync            1000000        2443 ns/op  -->  409 332 / s
buffered        1000000        4850 ns/op  -->  810 477 / s
Linux:

*/
```

14.17 Concurrent acces to objects by using a channel.

To safeguard concurrent modifications of an object instead of using locking with a sync Mutex we can also use a backend goroutine for the sequential execution of anonymous functions.

In the following program we have a type Person which now contains a field chF, a channel of anonymous functions. This is initialized in the constructor-method NewPerson, which also starts a method backend() as a goroutine.This method executes in an infinite loop all the functions placed on chF, effectively serializing them and thus providing safe concurrent access. The methods that change and retrieve the salary make an anonymous function which does that and put this function on chF, and backend() will sequentially execute them. Notice how in the method Salary the created closure function includes the channel fChan.

This is of course a simplified example and it should not be applied in such cases, but it shows how the problem could be tackled in more complex situations.

Listing 14.19—conc_access.go:

```
package main
import (
        “fmt”
        “strconv”
)
```

```
type Person struct {
        Name   string
        salary float64
        chF    chan func()
}

func NewPerson(name string, salary float64) *Person {
        p := &Person{name, salary, make(chan func())}
        go p.backend()
        return p
}

func (p *Person) backend() {
        for f := range p.chF {
               f()
        }
}

// Set salary.
func (p *Person) SetSalary(sal float64) {
        p.chF <- func() { p.salary = sal }
}

// Retrieve salary.
func (p *Person) Salary() float64 {
        fChan := make(chan float64)
        p.chF <- func() { fChan <- p.salary }
        return <-fChan
}

func (p *Person) String() string {
        return "Person - name is: " + p.Name + " - salary is: " +   strconv.
        FormatFloat(p.Salary(), 'f', 2, 64)
}

func main() {
        bs := NewPerson("Smith Bill", 2500.5)
        fmt.Println(bs)
        bs.SetSalary(4000.25)
        fmt.Println("Salary changed:")
        fmt.Println(bs)
}
/* Output       Person - name is: Smith Bill - salary is: 2500.50
                Salary changed:
                Person - name is: Smith Bill - salary is: 4000.25 */
```

Chapter 15—Networking, templating and web-applications

Go is very usable for writing web applications. Because there is no GUI framework for Go as yet, making html-screens with strings or templating is the only way to build Go applications with screens today.

15.1 A tcp-server

In this paragraph we will develop a simple client-server application using the TCP-protocol and the goroutine-paradigm from chapter 14. A (web) server application has to respond to requests from many clients simultaneously: in Go for every client-request a goroutine is spawned to handle the request. We will need the package **net** for networking communication functionality. It contains methods for working with TCP/IP and UDP protocols, domain name resolution, etc.

The server-code resides in its own program Listing 15.1—server.go:

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

func main() {
        fmt.Println("Starting the server ...")
        // create listener:
        listener, err := net.Listen("tcp", "localhost:50000")
        if err != nil {
               fmt.Println("Error listening", err.Error())
               return // terminate program
        }
        // listen and accept connections from clients:
        for {
               conn, err := listener.Accept()
```

```
                if err != nil {
                        fmt.Println("Error accepting", err.Error())
                        return // terminate program
                }
                go doServerStuff(conn)
        }
}

func doServerStuff(conn net.Conn) {
        for {
                buf := make([]byte, 512)
                _, err := conn.Read(buf)
                if err != nil {
                        fmt.Println("Error reading", err.Error())
                        return // terminate program
                }
                fmt.Printf("Received data: %v", string(buf))
        }
}
```

In `main()` we make a `net.Listener` variable `listener`, which is the basic function of a server: to listen for and accepting incoming client requests (on localhost which is IP-address 127.0.0.1 on port 50000 via the TCP-protocol). This Listen() function can return a variable err of type error. The waiting for client requests is performed in an infinite for-loop with `listener.Accept()`. A client request makes a connection variable conn of type net.Conn. On this connection a separate goroutine `doServerStuff()` is started which reads the incoming data in a buffer of size 512 bytes and outputs them on the server terminal; when all the data from the client has been read the goroutine stops. For each client a separate goroutine is created. The server-code must be executed before any client can run.

The code for the client is in a separate file client.go:

`Listing 15.2—client.go:`

```
package main
import (
        "fmt"
        "os"
        "net"
        "bufio"
        "strings"
)

func main() {
        // open connection:
        conn, err := net.Dial("tcp", "localhost:50000")
        if err != nil {
               // No connection could be made because the target machine
               actively refused it.
               fmt.Println("Error dialing", err.Error())
               return // terminate program
        }

        inputReader := bufio.NewReader(os.Stdin)
        fmt.Println("First, what is your name?")
        clientName, _ := inputReader.ReadString('\n')
        // fmt.Printf("CLIENTNAME %s",clientName)
        trimmedClient := strings.Trim(clientName, "\r\n") // "\r\n" on Windows,
        "\n" on Linux
        // send info to server until Quit:
        for {
               fmt.Println("What to send to the server? Type Q to quit.")
               input, _ := inputReader.ReadString('\n')
               trimmedInput := strings.Trim(input, "\r\n")
               // fmt.Printf("input:--%s--",input)
               // fmt.Printf("trimmedInput:--%s--",trimmedInput)
               if trimmedInput == "Q" {
                      return
               }
               _, err = conn.Write([]byte(trimmedClient + " says: " +
               trimmedInput))
        }
}
```

The client establishes a connection with the server through `net.Dial`

He receives input from the keyboard `os.Stdin` in an infinite loop until "Q" is entered. Notice the trimming of \n and \r (both only necessary on Windows). The trimmed input is then transmitted to the server via the Write-method of the connection.

Of course the server must be started first, if he is not listening, he can't be dialed by a client.

If a client process would start without a server listening, the client stops with the following error-message : `Error dialing dial tcp 127.0.0.1:50000: No connection could be made because the target machine actively refused it.`

Open a command-prompt in the directory where the server- and client-executables are, type server.exe (or just server) on Windows, ./server on Linux and ENTER.

The following message appears in the console: `Starting the server ...`

On Windows this process can be stopped with CTRL/C .

Then open 2 or 3 separate console-windows, in each a client process is started:
type client and ENTER.

Here is some output of the server (after removing the empty space from the 512 byte string):

```
Starting the server ...
Received data: IVO says: Hi Server, what's up ?
Received data: CHRIS says: Are you busy server ?
Received data: MARC says: Don't forget our appointment tomorrow !
```

When a client enters Q and stops, the server outputs the following message:

```
Error reading WSARecv tcp 127.0.0.1:50000: The specified network name is no longer
available.
```

The `net.Dial` function is one of the most important functions in networking. 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 on port 80 over TCP, then UDP and lastly TCP over IPv6 looks like this:

Listing 15.3—dial.go:

```
// make a connection with www.example.org:
package main

import (
        "fmt"
        "net"
        "os"
)

func main() {
         conn, err:= net.Dial("tcp", "192.0.32.10:80")      // tcp ipv4
         checkConnection(conn, err)

         conn, err = net.Dial("udp", "192.0.32.10:80")      // udp
         checkConnection(conn, err)

         conn, err = net.Dial("tcp", "[2620:0:2d0:200::10]:80") // tcp ipv6
         checkConnection(conn, err)
}

func checkConnection(conn net.Conn, err error) {
         if err!= nil {
                fmt.Printf("error %v connecting!")
                os.Exit(1)
         }

         fmt.Println("Connection is made with %v", conn)
}
```

The following program is another illustration of the use of the net package for opening, writing to and reading from a *socket*:

Listing 15.4—socket.go:

```
package main
import (
  "fmt"
  "net"
  "io"
)

func main() {
  var (
    host = "www.apache.org"
    port = "80"
    remote = host + ":" + port
    msg string = "GET / \n"
    data = make([]uint8, 4096)
    read = true
    count = 0
  )
  // create the socket
  con, err := net.Dial("tcp", remote)
  // send our message.  an HTTP GET request in this case
  io.WriteString(con, msg)
  // read the response from the webserver
  for read {
    count, err = con.Read(data)
    read = (err == nil)
    fmt.Printf(string(data[0:count]))
  }
  con.Close()
}
```

Exercise 15.1: Write new versions of client and server (client1.go / server1.go):

a) do the error-checking in a separate function checkError(error); discuss the advantage / disadvantage of a possible solution: why would this refactoring perhaps not be that optimal? Examine how it is solved in listing 15.14
b) give the client the possibility to shutdown the server by sending the command SH
c) let the server keep a list of all the connected clients (their names); when a client sends the WHO—command, the server will display this list:

```
-----------------------------------
This is the client list: 1=active, 0=inactive
User IVO is 1
User MARC is 1
User CHRIS is 1
-----------------------------------------
```

Remark: When the server is running, you cannot compile/link a new version of its source code in the same directory, because server.exe is in use by the operating system and cannot be replaced with a new version.

The following version `simple_tcp_server.go` has a much better structure and improves on our first example of a tcp-server `server.go` in many ways, using only some 80 lines of code!

`Listing 15.5—simple_tcp_server.go:`

```
// Simple multi-thread/multi-core TCP server.
package main
import (
        “flag”
        “net”
        “os”
        “fmt”
)

const maxRead = 25

func main() {
        flag.Parse()
        if flag.NArg() != 2 {
                panic(“usage: host port”)
        }
        hostAndPort := fmt.Sprintf(“%s:%s”, flag.Arg(0), flag.Arg(1))
        listener := initServer(hostAndPort)
        for {
                conn, err := listener.Accept()
                checkError(err, “Accept: “)
                go connectionHandler(conn)
        }
}

func initServer(hostAndPort string) *net.TCPListener {
        serverAddr, err := net.ResolveTCPAddr(“tcp”, hostAndPort)
        checkError(err, “Resolving address:port failed: `” + hostAndPort + “’”)
        listener, err := net.ListenTCP(“tcp”, serverAddr)
        checkError(err, “ListenTCP: “)
        println(“Listening to: “, listener.Addr().String())
        return listener
}

func connectionHandler(conn net.Conn) {
        connFrom := conn.RemoteAddr().String()
        println(“Connection from: “, connFrom)
        sayHello(conn)
        for {
                var ibuf []byte = make([]byte, maxRead + 1)
                length, err := conn.Read(ibuf[0:maxRead])
                ibuf[maxRead] = 0 // to prevent overflow
                switch err {
                case nil:
                        handleMsg(length, err, ibuf)
                case os.EAGAIN: // try again
                        continue
                default:
                        goto DISCONNECT
                }
        }

DISCONNECT:
        err := conn.Close()
        println("Closed connection: ", connFrom)
        checkError(err, "Close: ")
}

func sayHello(to net.Conn) {
        obuf := []byte{'L', 'e', 't', '\'', 's', ' ', 'G', 'O', '!', '\n'}
        wrote, err := to.Write(obuf)
        checkError(err, "Write: wrote " + string(wrote) + " bytes.")
}

func handleMsg(length int, err error, msg []byte) {
        if length > 0 {
                print("<", length, ":")
                for i := 0; ; i++ {
                        if msg[i] == 0 {
                                break
                        }
                        fmt.Printf("%c", msg[i])
                }
                print(">")
        }
}

func checkError(error error, info string) {
        if error != nil {
                panic("ERROR: " + info + " " + error.Error()) // terminate
                program
        }
}
```

What are the improvements?

(1) The server address and port are not hard-coded in the program, but given on the command-line and read via the flag-package. Note the use of flag.NArg() to signal when the expected 2 arguments are not given:

```
if flag.NArg() != 2 {
        panic(“usage: host port”)
}
```

The arguments are then formatted into a string via the fmt.Sprintf-function:

```
hostAndPort := fmt.Sprintf(“%s:%s”, flag.Arg(0), flag.Arg(1))
```

(2) The server address and port are controlled in the function `initServer` through `net.ResolveTCPAddr`, and this function return a `*net.TCPListener`
(3) For each connection the function `connectionHandler` is started as a goroutine. This begins with showing the address of the client with `conn.RemoteAddr()`
(4) It writes a promotional Go-message to the client with the function `conn.Write`
(5) It reads from the client in chuncks of 25 bytes and prints these one by one; in case of an error in the read the infinite read-loop is left via the default switch clause and that client-connection is closed. In case the OS issues an EAGAIN error, the read is retried.
(6) All error-checking is refactored in a separate function checkError which issues a panic with a contextual error-message in the case of an error occurring.

Start this server-program on the command-line with: `simple_tcp_server localhost 50000` and start a few clients with client.go in separate command-windows. A typical server output from 2 client-connections follows, where we see that the clients each have their own address:

```
E:\Go\GoBoek\code examples\chapter 14>simple_tcp_server localhost 50000
Listening to: 127.0.0.1:50000
Connection from: 127.0.0.1:49346
<25:Ivo says: Hi server, do y><12:ou hear me ?>
Connection from: 127.0.0.1:49347
<25:Marc says: Do you remembe><25:r our first meeting serve><2:r?>
```

net.Error:

The net package returns errors of type error, following the usual convention, but some of the error implementations have additional methods defined by the *net.Error* interface:

```
package net

type Error interface {
        Timeout() bool // Is the error a timeout?
        Temporary() bool      // Is the error temporary?
        …
}
```

Client code can test for a net.Error with a type assertion and then distinguish transient network errors from permanent ones. For instance, a web crawler might sleep and retry when it encounters a temporary error and give up otherwise.

```
// in a loop - some function returns an error err
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
        time.Sleep(1e9)
        continue   // try again
}
if err != nil {
        log.Fatal(err)
}
```

15.2 A simple webserver

Http is a higher protocol than tcp, and it describes how a webserver communicates with client-browsers. Go has its `net/http`-package, which we will now explore. We will start with some really simple things, first let's write a "Hello world!" webserver: see listing 15.6

We import "http", and our webserver is started, analogous to the `net.Listen("tcp", "localhost:50000")` function for our tcp server in § 15.1, with the function `http.ListenAndServe("localhost:8080", nil)` which returns nil if everything is OK or an error otherwise (localhost can be omitted from the address, 8080 is the chosen port number).

A web-address is represented by the type **`http.URL`** which has a Path field that contains the url as a string; client-requests are described by the type **`http.Request`**, which has a URL field.

If the request req is a POST of an html-form, and "var1" is the name of an html input-field on that form, then the value entered by the user can be captured in Go-code with: `req.`**`FormValue`**`("var1")` (see § 15.4). An alternative is first to call **`request.ParseForm()`** and the value can then be retrieved as the 1st return parameter of **`request.Form`**`["var1"]`, like in:

```
	var1, found := request.Form["var1"]
```

The 2nd parameter found is then true, if var1 was not on the form found becomes false.

The Form field is in fact of type map[string][]string. The webserver sends an **`http.Response,`** its output is send on an **`http.ResponseWriter`** object. This object assembles the HTTP server's response; by writing to it, we send data to the HTTP client.

Now we still have to program what the webserver must do, how it *handles* a request. This is done through the function **`http.HandleFunc,`** which in this example says that if the root "/" (the url http://localhost:8080) is requested (or any other address on that server) the function `HelloServer` is called. This function is of the type `http.HandlerFunc`, and they are most often named Prefhandler with some prefix Pref.

`http.HandleFunc` *registers a handler function* (here `HelloServer`) for incoming requests on /.

The / can be replaced by more specific urls like /create, /edit, etc.; for each specific url you can then define its corresponding handler-function. This function has as 2nd parameter: the request req; its first parameter is the ResponseWriter w, to which it writes a string composed of Hello and **`r.URL.Path[1:]`** : the trailing [1:] means "create a sub-slice of Path from the 1st character to the end.", this drops the leading "/" from the path name. This writing is done with the function `fmt.Fprintf()` (see § 12.8); another possibility is `io.WriteString(w, "hello, world!\n")`

Summarized: the 1st parameter is a requested path and the 2nd parameter is a reference to a function to call when the path is requested.

Listing 15.6—hello_world_webserver.go:

```
	package main
	import (
		"fmt"
		"net/http"
	"log"
	)

	func HelloServer(w http.ResponseWriter, req *http.Request) {
		fmt.Println("Inside HelloServer handler")
		fmt.Fprint(w, "Hello," + req.URL.Path[1:])
```

```
}

func main() {
        http.HandleFunc("/",HelloServer)
        err := http.ListenAndServe("localhost:8080", nil)
               if err != nil {
                      log.Fatal("ListenAndServe: ", err.Error())
        }
}
```

Start the program on the command-line, this opens a command-window with the text:

```
Starting Process E:/Go/GoBoek/code_examples/chapter_14/hello_world_webserver.exe
 ...
```

Then open your browser with the address(url): http://localhost:8080/world and in the browser window the text: `Hello, world` appears. The webserver serves what you type in after :8080/

The fmt.Println statement prints on the server console; somewhat more useful could be to log inside every handler what was requested.

Remarks:

1) The first 2 lines (without the error-handling) can be replaced by this line:
   `http.ListenAndServe(":8080", http.HandlerFunc(HelloServer)`

2) `fmt.Fprint` and more so `fmt.Fprintf` are good functions to use to write to the `http.ResponseWriter` (which implements io.Writer).

   For example: we could use
   `fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", title, body)`
   to construct a very simple web page where the values title and body are inserted.

   If you need to do more sophisticated substitutions, use the templating package (see § 15.7).

3) If you need the security of https, use `http.`**`ListenAndServeTLS`**`()` instead of `http.ListenAndServe()`
4) Instead of `http.HandleFunc("/", HFunc)`

   where `HFunc` is a handler function with the signature:

```
func HFunc(w http.ResponseWriter, req *http.Request) {
…
}
```

this form can also be used: `http.Handle("/", http.HandlerFunc(HFunc))`

`HandlerFunc` is just a type name for the signature above with definition:

```
type HandlerFunc func(ResponseWriter, *Request)
```

It is an adapter to allow the use of ordinary functions as HTTP handlers. If f is a function with the appropriate signature, HandlerFunc(f) is a Handler object that calls f.

The 2nd argument to http.Handle can also be an object obj of type T: `http.Handle("/", obj)`

provided that T has a ServeHTTP method, implementing the `Handler` interface of http:

```
func (obj *Typ) ServeHTTP(w http.ResponseWriter, req *http.Request) {
   …
}
```

This is used in the webserver in §15.8 for the types Counter and Chan. So the package http serves HTTP requests using any value that implements `http.Handler`.

Exercise 15.2: `webhello2.go`

Write a webserver which listens on port 9999, with the following handler functions:
 i) When http://localhost:9999/hello/Name is requested, responds with: hello Name
    (where Name is a variable first name, like Chris or Madeleine)
 ii) When http://localhost:9999/shouthello/Name is requested, responds with: hello NAME

Exercise 15.3: `hello_server.go`

Make an hello struct with no fields and let it implement http.Handler. Start a webserver and test it out.

15.3 Polling websites and reading in a web page

In the following program all url's in an array are polled: a simple **`http.Head()`** request is send to them to see how they react; its signature is: `func Head(url string) (r *Response, err error)`

The Status of the Response resp is printed.

Listing 15.7—poll_url.go:

```
package main
import (
        "fmt"
        "net/http"
)

var urls = []string{
        "http://www.google.com/",
        "http://golang.org/",
        "http://blog.golang.org/",
}

func main() {
        // Execute an HTTP HEAD request for all url's
        // and returns the HTTP status string or an error string.
        for _, url := range urls {
               resp, err := http.Head(url)
               if err != nil {
                      fmt.Println("Error:", url, err)
               }
               fmt.Print(url, ": ", resp.Status)
        }
}
```

The output is:

```
http://www.google.com/ : 302 Found
http://golang.org/ : 200 OK
http://blog.golang.org/ : 200 OK
```

In the following program we show the html content of a web page with http.Get(); the response res returned from Get has the content in a field Body, which is read with ioutil.ReadAll:

Listing 15.8—http_fetch.go:

```
package main
import (
        "fmt"
        "net/http"
        "io/ioutil"
        "log"
)

func main() {
        res, err := http.Get("http://www.google.com")
        CheckError(err)
        data, err := ioutil.ReadAll(res.Body)
        CheckError(err)
        fmt.Printf("Got: %q", string(data))
}
func CheckError(err error) {
        if err != nil {
               log.Fatalf("Get: %v", err)
        }
}
```

Here is a sample error output from CheckError when trying to read a non-existing web's site homepage:

```
2011/09/30 11:24:15 Get: Get http://www.google.bex: dial tcp www.google.bex:80:
GetHostByName: No such host is known.
```

In the following program we get the twitter-status of a certain user, and unmarshall its status via the xml package into a struct:

Listing 15.9—twitter_status.go:

```
package main
import (
        "net/http"
        "fmt"
        "encoding/xml"
)
/* these structs will house the unmarshalled response.
   they should be hierarchically shaped like the XML
   but can omit irrelevant data. */
type Status struct {
        Text string
}

type User struct {
```

```
         XMLName xml.Name
         Status  Status
}
// var user User

func main() {
         // perform an HTTP request for the twitter status of user: Googland
         response, _ := http.Get("http://twitter.com/users/Googland.xml")
         // initialize the structure of the XML response
         user := User{xml.Name{"", "user"}, Status{""}}
         // unmarshal the XML into our structures
         xml.Unmarshal(response.Body, &user)
         fmt.Printf("status: %s", user.Status.Text)
}
/* Output:
status: Robot cars invade California, on orders from Google: Google has been testing
self-driving cars ... http://bit.ly/cbtpUN http://retwt.me/97p<exit code="0"
msg="process exited normally"/>
*/
```

Other useful functions in the http package which we will be using in § 15.4 are:

- `http.`**`Redirect`**`(w ResponseWriter, r *Request, url string, code int)`: this redirects the browser to url (can be a path relative to the request path) and a statuscode code.
- `http.`**`NotFound`**`(w ResponseWriter, r *Request)`: this replies to the request with an HTTP 404 not found error.
- `http.`**`Error`**`(w ResponseWriter, error string, code int)`: this replies to the request with the specified error message and HTTP code.
- a useful field of an `http.Request object req is: req.`**`Method`**, this is a string which contains "GET" or "POST" according to how the web page was requested.

All HTTP status codes are defined as Go-constants, for example:

```
http.StatusContinue           = 100
http.StatusOK                 = 200
http.StatusFound              = 302
http.StatusBadRequest         = 400
http.StatusUnauthorized       = 401
http.StatusForbidden          = 403
http.StatusNotFound           = 404
http.StatusInternalServerError       = 500
```

You can set the content header with **`w.Header().Set("Content-Type", "../..")`**

e.g. when sending html-strings in a web application, execute `w.Header().Set("Content-Type", "text/html")` before writing the output (this is normally not necessary).

Exercise 15.4: Extend http_fetch.go so that the url is read in from the console, apply the console input methods from §12.1 (`http_fetch2.go`).

Exercise 15.5: Fetch the twitter status in json-format and show the status, just like in Listing 15.9 (`twitter_status_json.go`)

15.4 Writing a simple web application

The following program starts a webserver on port 8088; the url /test1 will be handled by SimpleServer which outputs hello world in the browser. The url /test2 will be handled by FormServer: if the url is requested by the browser initially, then the request is of method GET, and the response is the constant form, which contains the html for a simple input form with text box and submit button. When entering something in the text box and clicking the button a POST request is issued. The code for FormServer uses a switch to distinguish between the 2 possibilities. In the POST case the content of the textbox with name inp is retrieved with: request.FormValue("inp") and written back to the browser page. Start the program in a console and open a browser with the url http://localhost:8088/test2 to test this program:

Listing 15.10—simple_webserver.go:

```
package main
import (
     "net/http"
     "io"
)

const form = `<html><body><form action="#" method="post" name="bar">
              <input type="text" name="in"/>
               <input type="submit" value="Submit"/>
        </form></html></body>`

/* handle a simple get request */
func SimpleServer(w http.ResponseWriter, request *http.Request) {
     io.WriteString(w, "<h1>hello, world</h1>")
}
```

```
/* handle a form, both the GET which displays the form
   and the POST which processes it.*/
func FormServer(w http.ResponseWriter, request *http.Request) {
     w.Header().Set("Content-Type", "text/html")
     switch request.Method {
     case "GET":
                  /* display the form to the user */
                  io.WriteString(w, form );
     case "POST":
                  /* handle the form data, note that ParseForm must
                     be called before we can extract form data*/
                  //request.ParseForm();
                  //io.WriteString(w, request.Form["in"][0])
              io.WriteString(w, request.FormValue("in"))
     }
}

func main() {
     http.HandleFunc("/test1", SimpleServer)
     http.HandleFunc("/test2", FormServer)
     if err := http.ListenAndServe(":8088", nil); err != nil {
                 panic(err)
     }
}
```

Remark: When using constant strings which represent html-text it is important to include the

```
<html><body>… </html></body>
```

to let the browser know it receives html.

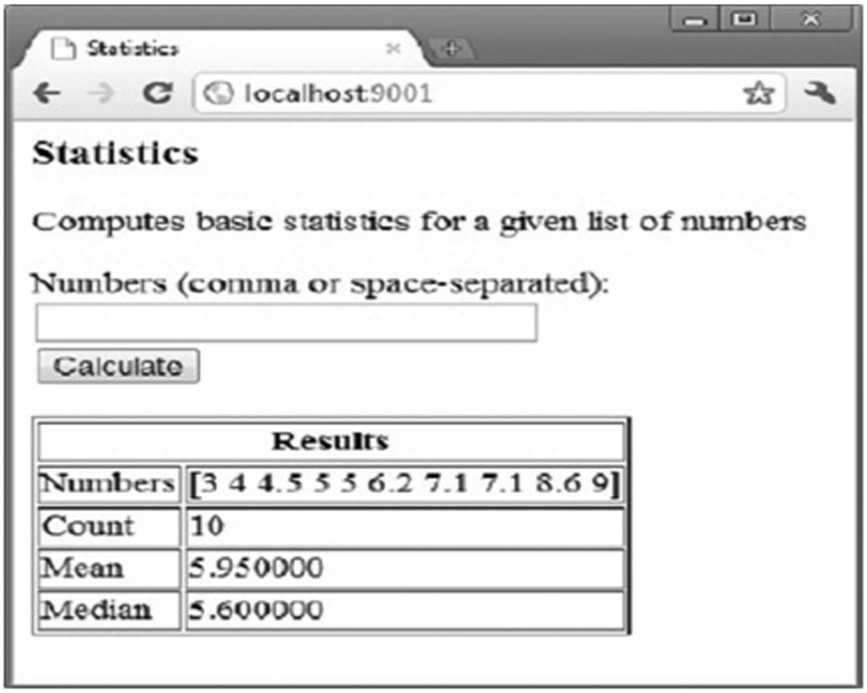
Even safer is it to set the header with the content-type text/html before writing the response in the handler: w.Header().Set("Content-Type", "text/html")

The content-type the browser thinks it receives can be retrieved with the function:

```
http.DetectContentType([]byte(form))
```

Exercise 15.6: `statistics.go`

Develop a web application that lets the user put in a series of numbers, and that prints them out, their number, the mean and the median, like in the following screen:



Statistics

Computes basic statistics for a given list of numbers

Numbers (comma or space-separated):

Results	
Numbers	[3 4 4.5 5 5 6.2 7.1 7.1 8.6 9]
Count	10
Mean	5.950000
Median	5.600000

Fig 15.1—Screen of exercise 15.6

15.5 Making a web application robust

When a handler function in a web application panics our webserver simply terminates. This is not good: a webserver must be a robust application, able to withstand what perhaps is a temporary problem.

A first ideao could be to use defer/recover in every handler-function, but this would lead to much duplication of code. Apply the error-handling scheme with closures from § 13.5 is a much more elegant solution. We show this mechanism applied here to the simple webserver from the previous §, but it can just as easily be applied in any webserver program.

To make the code more readable we create a function type for a page handler-function:

```
    type HandleFnc func(http.ResponseWriter,*http.Request)
```

Our errorHandler function from §13.5 applied here becomes the function logPanics:

```
    func logPanics(function HandleFnc) HandleFnc {
        return func(writer http.ResponseWriter, request *http.Request) {
            defer func() {
                if x := recover(); x != nil {
                    log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
                }
            }()
            function(writer, request)
        }
    }
```

And we wrap our calls to the handler functions within logPanics:

```
    http.HandleFunc("/test1", logPanics(SimpleServer))
    http.HandleFunc("/test2", logPanics(FormServer))
```

The handler-functions then should contain panic calls, or a kind of check(error) function as in §13.5; the complete code is listed here:

Listing 15.11—robust_webserver.go:

```
    package main
    import (
         "net/http"
         "io"
         "log"
    )

    type HandleFnc func(http.ResponseWriter,*http.Request)

    // …  same code as in listing 15.10

    func main() {
         http.HandleFunc("/test1", logPanics(SimpleServer))
         http.HandleFunc("/test2", logPanics(FormServer))
         if err := http.ListenAndServe(":8088", nil); err != nil {
             panic(err)
         }
    }

    func logPanics(function HandleFnc) HandleFnc {
        return func(writer http.ResponseWriter, request *http.Request) {
```

```
        defer func() {
            if x := recover(); x != nil {
                log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
            }
        }()
        function(writer, request)
    }
}
```

15.6 Writing a web application with templates

The following program is a working web-application for a wiki, that is a collection of pages that can be viewed, edited and saved, in under 100 lines of code. It is the codelab wiki tutorial from the Go site, one of the best Go-tutorials I know of; it is certainly worthwhile to work through the complete codelab to see and better understand how the program builds up(http://golang.org/doc/codelab/wiki/). Here we will give a complementary description of the program in its totality, from a top-down view. The program is a webserver, so it must be started on the command-line, let's say on port 8080. The browser can ask to view the content of a wiki-page with a url like: http://localhost:8080/view/page1.

The text of this page is then read from a file and shown in the web page; this includes a hyperlink to edit the wiki page (http://localhost:8080/edit/page1).The edit page shows the contents in a textframe, the user can change the text and save it to its file with a submit button Save; then the same page with the modified content is viewed. If the page that is asked for viewing does not exist (e.g. http://localhost:8080/edit/page999), the program detects this and redirects immediately to the edit-page, so that the new wiki-page can be made and saved.

The wiki-page needs a title and a text-content; it is modeled in the program by the following struct, where the content is a slice of bytes named Body):

```
type Page struct {
Title string
Body []byte
}
```

In order to maintain our wiki-pages outside of the running program, we will use simple text-files as persistent storage. The program and the necessary templates and text files can be found in the map `code_examples\chapter 15\wiki`.

Listing 15.12—wiki.go:

```
package main
import (
        "net/http"
        "io/ioutil"
        "log"
        "regexp"
        "text/template"
)

const lenPath = len("/view/")

var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")
var templates = make(map[string]*template.Template)
var err error

type Page struct {
        Title string
        Body []byte
}

func init() {
    for _, tmpl := range []string{"edit", "view"} {
        templates[tmpl] = template.Must(template.ParseFiles(tmpl + ".html"))
    }
}

func main() {
        http.HandleFunc("/view/", makeHandler(viewHandler))
        http.HandleFunc("/edit/", makeHandler(editHandler))
        http.HandleFunc("/save/", makeHandler(saveHandler))
        err := http.ListenAndServe(":8080", nil)
            if err != nil {
                log.Fatal("ListenAndServe: ", err.Error())
        }
}

func makeHandler(fn func(http.ResponseWriter, *http.Request, string)) http.
HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                title := r.URL.Path[lenPath:]
                if !titleValidator.MatchString(title) {
                        http.NotFound(w, r)
                        return
                }
                fn(w, r, title)
        }
}

func viewHandler(w http.ResponseWriter, r *http.Request, title string) {
        p, err := load(title)
        if err != nil {  // page not found
                http.Redirect(w, r, "/edit/" + title, http.StatusFound)
                return
        }
        renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request, title string) {
        p, err := load(title)
        if err != nil {
                p = &Page{Title: title}
        }
        renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request, title string) {
        body := r.FormValue("body")
        p := &Page{Title: title, Body: []byte(body)}
        err := p.save()
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        http.Redirect(w, r, "/view/" + title, http.StatusFound)
}

func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
        err := templates[tmpl].Execute(w, p)
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
        }
}

func (p *Page) save() error {
        filename := p.Title + ".txt"
// file created with read-write permissions for the current user only
        return ioutil.WriteFile(filename, p.Body, 0600)
}
func load(title string) (*Page, error) {
        filename := title + ".txt"
        body, err := ioutil.ReadFile(filename)
                if err != nil {
                        return nil, err
                }
                return &Page{Title: title, Body: body}, nil
}
```

Let us go through the code:

- First we import the necessary packages, http of course because we are going to construct a webserver, but also io/ioutil for easy reading and writing of the files, `regexp` for validating title-input, and `template` for dynamically creating our html-files; we use os for the errors.
- We want to stop hackers input which could harm our server, so we will check the user input in the browser url (which is the title of the wiki-page) with the following regular expression:
  `var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")`
  This will be controlled in the function `makeHandler.`
- We must have a mechanism to insert our Page structs into the title and content of a web page, this is done as follows with the use of the template package:

  i) first make the *html-templatefile(s)* in an editor, e.g. `view.html`:
     ```
     <h1>{{.Title |html}}</h1>
     <p>[<a href="/edit/{{.Title |html}}">edit</a>]</p>
     <div>{{printf "%s" .Body |html}}</div>
     ```
     the fields which are to be inserted from a data-structure are put between {{ }}, here `{{.Title |html}}` and `{{printf "%s" .Body |html}}` from the Page struct (of course this can be very complex html, but here it is simplified as much as possible to show the principle (for `|html` and `printf "%s"` see the following §).

  ii) The template.Must(template.ParseFiles(tmpl + ".html")) function transforms this into a *template.Template, for efficiency reasons we only do this parsing once in our program, the init() function is a convenient place to do that. The template-objects are kept in memory in a map indexed by the name of the html-file:

```
templates = make(map[string]*template.Template)
```

This technique is called *template caching* and is a recommended best practice.

iii) In order to construct the page out of the template and the struct, we must use the function: `templates[tmpl].Execute(w, p)`
It is called upon a template, gets the Page-struct p to be substituted in the template as a parameter, and writes to the ResponseWriter w. This function must be checked on its error-output; in case there is a/n error we call http.Error to signal this. This code will be called many times in our application, so we extract it into a separate function renderTemplate.

- In `main()` our webserver is started with `ListenAndServe` on port 8080; but as in §15.2 we first define some Handler functions for urls which start with view, edit and save after `localhost:8080/`. In most webserver applications this forms a series of url-paths with handler-functions, analogous to a routing table in MVC frameworks like Ruby and Rails, Django or ASP.NET MVC. The request url is matched with these paths, the longer paths match first; if not matched with anything else, the handler for / is called.

  Here we define 3 handlers, and because this setting up contains repetitive code, we isolate this in a `makeHandler` function. This is a rather special higher-order function which is well worth studying: it has a function as its first parameter, and it returns a function which is a closure:

```
func  makeHandler(fn  func(http.ResponseWriter,  *http.Request,  string))
http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                title := r.URL.Path[lenPath:]
                if !titleValidator.MatchString(title) {
                        http.NotFound(w, r)
                        return
                }
                fn(w, r, title)
        }
}
```

- This closure takes the enclosing function variable fn in order to construct its return value; but before that it validates the input `title with titleValidator.MatchString(title)`. If the title does not consist out of letters and digits, a NotFound error is signaled.

  (test this with e.g. `localhost:8080/view/page++`);viewhandler, edithandler and savehandler which are the parameters for makeHandler in main() must all be of the same type as fn.

- The viewhandler tries to read a text file with the given title; this is done through the load() function which constructs the filename and reads the file with `ioutil.ReadFile`; if the file is found its contents goes into a local string body. A pointer to a Page struct is made literally with it: `&Page{Title: title, Body: body}`

  and this is returned to the caller together with nil for the error. The struct is then merged with the template with renderTemplate.

  In case of an error, which means the wiki-page does not yet exist on disk, the error is returned to viewHandler(), where an automatic redirect is done to request an edit-page with that title.

- The edithandler is almost the same: try to load the file, if found render the edit-template with it; in case of an error make a new Page object with that title and render it also.
- Saving of a page content is done through the Save-button in the edit-page; this button resides in the html-form which starts with:
  ```
  <form action="/save/{{.Title}}" method="POST">
  ```

  This means that when posting a request with a url of the form http://localhost/save/{Title} (with the title substituted through the template) is sent to the webserver. For such a url we have defined a handler function: `saveHandler()`.With the FormValue() method of the request it extracts the contents of the textarea-field named body, constructs a Page object with this info and tries to store this page with the `save()` function. In case this fails an http.Error is returned to be shown in the browser, when it succeeds the browser is redirected to viewing the same page. The `save()` function is very simple: write the Body field of the Page struct in a file called filename with the function `ioutil.WriteFile()`. It uses `{{ printf "%s" .Body|html}}`.

15.7 Exploring the template package

(The documentation for the template package is available at http://golang.org/pkg/template/)

In the preceding paragraph, we used templates to merge data from a (data) struct(ure)s with html-templates. This is very useful indeed for building web applications, but the template techniques are more general than this: data-driven templates can be made for generating textual output, and HTML is only a special case of this.

A template is executed by merging it with a data structure, in many cases a struct or a slice of structs. It rewrites a piece of text on the fly by substituting elements derived from data items passed to `templ.Execute()`. Only the exported data items are available for merging with the template. Actions can be data evaluations or control structures and are delimited by "{{" and "}}". Data items may be values or pointers; the interface hides the indirection.

15.7.1. Field substitution: {{.FieldName}}

To include the content of a field within a template, enclose it within double curly braces and add a dot at the beginning, e.g. if Name is a field within a struct and its value needs to be substituted while merging, then include the text {{.Name}} in the template; this also works when Name is a key of a map.A new template is created with **`template.New`** whitch takes the template name as a string parameter. As we have already encountered in § 15.5, the **`Parse`** methods generate a template as an internal representation by parsing some template definition string, use **`ParseFile`** when the parameter is the path to a file with the template definition. When there was a problem with the parsing their second return parameter is an Error != nil. In the last step the content of a datastructure is merged with the template through the **`Execute`** method, and written to its 1st argument which is an io.Writer; again an error can be returned. This is illustrated in the following program, where the output is written to the console through os.Stdout:

Listing 15.13—template_field.go:

```
package main
import (
    "os"
    "text/template"
)

type Person struct {
    Name string
}

func main() {
    t := template.New("hello")
    t, _ = t.Parse("hello {{.Name}}!")
    p := Person{Name:"Mary", nonExportedAgeField: "31"} // data
    if err := t.Execute(os.Stdout, p); err != nil {
        fmt.Println("There was an error:", err.Error())
    }
}
// Output: hello Mary!
```

Our datastructure contains a non exported field, and when we try to merge this through a definition string like

```
t, _ = t.Parse("your age is {{.nonExportedAgeField}}!")
```

the following error occurs: `There was an error: template: nonexported template hello:1: can’t evaluate field nonExportedAgeField in type main.Person.`

If you simply want to substitute the 2nd argument of Execute() use **{{.}}**

When this is being done in a browser context, first filter the content with the html filter, like this: **`{{html .}}`** or with a field `FieldName` **`{{ .FieldName |html }}`**

the |html part asks the template engine to pass the value of `FieldName` through the htmlformatter before outputting it, which escapes special HTML characters (such as replacing > with &gt;). This will prevent user data from corrupting the form HTML.

15.7.2. Validation of the templates

To check whether the template definition syntax is correct, use the **Must** function executed on the result of the Parse. In the following example tOk is correct, tErr has a validation error and causes a runtime panic!s

Listing 15.14—template_validation.go:

```
package main
import (
        “text/template”
        “fmt”
)

func main() {
        tOk := template.New(“ok”)
        //a valid template, so no panic with Must:
        template.Must(tOk.Parse(“/* and a comment */ some static text: {{ .Name
        }}”))
        fmt.Println(“The first one parsed OK.”)
        fmt.Println(“The next one ought to fail.”)
        tErr := template.New(“error_template”)
        template.Must(tErr.Parse(“ some static text {{ .Name }”))
}
/* Output:
The first one parsed OK.
The next one ought to fail.
panic: template: error_template:1: unexpected “}” in command
*/
```

Errors in template syntax should be uncommon, therefore use the defer/recover mechanism outlined in § 13.3 to report this error and correct it.

It is common to see in code the 3 basic functions being chained, like:

```
var strTempl = template.Must(template.New("TName").Parse(strTemplateHTML))
```

Exercise 15.7: `template_validation_recover.go`

Implement the defer/recover mechanism in the example above.

15.7.3 If-else

The output from a template resulting from Execute contains static text, and text contained within {{ }} which is called a *pipeline*. For example, running this code (program `Listing 15.15 pipeline1.go`):

```
	t := template.New("template test")
	t = template.Must(t.Parse("This is just static text. \n{{\"This is pipeline data—because it is evaluated within the double braces.\"}} {{`So is this, but within reverse quotes.`}}\n"))
	t.Execute(os.Stdout, nil)
```

gives this output:

```
This is just static text.
This is pipeline data—because it is evaluated within the double braces. So is this,
but within reverse quotes.
```

Now we can condition the output of pipeline data with if-else-end: if the pipeline is empty, like in: `{{if ``}} Will not print. {{end}}`
then the if condition evaluates to false and nothing will be output, but with this:

```
{{if `anything`}} Print IF part. {{else}} Print ELSE part.{{end}}
```

`Print IF part` will be output. This is illustrated in the following program:

Listing 15.16—template_ifelse.go:

```
package main
import (
	"os"
```

```
        “text/template”
)

func main() {
        tEmpty := template.New(“template test”)
        tEmpty = template.Must(tEmpty.Parse(“Empty pipeline if demo: {{if ``}}
        Will not print. {{end}}\n”)) //empty pipeline following if
        tEmpty.Execute(os.Stdout, nil)

        tWithValue := template.New(“template test”)
        tWithValue = template.Must(tWithValue.Parse(“Non empty pipeline if demo:
        {{if `anything`}} Will print. {{end}}\n”)) //non empty pipeline
        following if condition
        tWithValue.Execute(os.Stdout, nil)

        tIfElse := template.New(“template test”)
        //non empty pipeline following if condition
        tIfElse = template.Must(tIfElse.Parse(“if-else demo: {{if `anything`}}
        Print IF part. {{else}} Print ELSE part.{{end}}\n”))
        tIfElse.Execute(os.Stdout, nil)
}
/* Output:
Empty pipeline if demo:
Non empty pipeline if demo: Will print.
if-else demo: Print IF part.
*/
```

15.7.4 Dot and with-end

The dot (.) is used in Go templates: its value **{{.}}** is set to the current pipeline value.

The with statement sets the value of dot to the value of the pipeline. If the pipeline is empty, then whatever is between the with-end block is skipped; when nested, the dot takes the value according to closest scope. This is illustrated in the following program:

Listing 15.17—template_with_end.go:

```
package main
import (
        “os”
        “text/template”
)

func main() {
        t := template.New("test")
        t, _ = t.Parse("{{with `hello`}}{{.}}{{end}}!\n")
        t.Execute(os.Stdout, nil)

        t, _ = t.Parse("{{with `hello`}}{{.}} {{with `Mary`}}{{.}}{{end}}
        {{end}}!\n")
        t.Execute(os.Stdout, nil)
}
/* Output:
hello!
hello Mary!
*/
```

15.7.5 Template variables $

You can create local variables for the pipelines within the template by prefixing the variable name with a "$" sign. Variable names have to be composed of alphanumeric characters and the underscore. In the example below I have used a few variations that work for variable names.

Listing 15.18—template_variables.go:

```
package main
import (
        "os"
        "text/template"
)

func main() {
        t := template.New("test")
        t = template.Must(t.Parse("{{with $3 := `hello`}}{{$3}}{{end}}!\n"))
        t.Execute(os.Stdout, nil)

        t = template.Must(t.Parse("{{with $x3 := `hola`}}{{$x3}}{{end}}!\n"))
        t.Execute(os.Stdout, nil)

        t = template.Must(t.Parse("{{with $x_1 := `hey`}}{{$x_1}} {{.}} {{$x_1}}
        {{end}}!\n"))
        t.Execute(os.Stdout, nil)
}

/* Output:
hello!
```

```
hola!
hey hey hey!*/
```

15.7.6 Range-end

This construct has the format: `{{range pipeline}} T1 {{else}} T0 {{end}}`

range is used for looping over collections: the value of the pipeline must be an array, slice, or map. If the value of the pipeline has length zero, dot is unaffected and T0 is executed; otherwise, dot is set to the successive elements of the array, slice, or map and T1 is executed.

If t is the template:

```
{{range .}}
{{.}}
{{end}}
```

then this code:

```
s := []int{1,2,3,4}
t.Execute(os.Stdout, s)
```

will output:

```
1
2
3
4
```

For a more useful example, see § 20.7 where data from the App Engine datastore is shown through a template with:

```
{{range .}}
  {{with .Author}}
    <p><b>{{html .}}</b> wrote:</p>
  {{else}}
    <p>An anonymous person wrote:</p>
  {{end}}
  <pre>{{html .Content}}</pre>
  <pre>{{html .Date}}</pre>
{{end}}
```

`range .` here loops over a slice of structs, each containing an Author, Content and Date field.

15.7.7 Predefined template functions

There are also a few predefined template functions that you can use within your code, e.g. the printf function which works similar to the function `fmt.Sprintf`:

Listing 15.19—predefined_functions.go:

```
package main
import (
        "os"
        "text/template"
)

func main() {
        t := template.New("test")
        t = template.Must(t.Parse("{{with $x := `hello`}}{{printf `%s %s` $x
        `Mary`}}{{end}}!\n"))
        t.Execute(os.Stdout, nil)
}
// hello Mary!
```

This was also used in §15.6: `{{ printf "%s" .Body|html}}`
otherwise the bytes of Body are printed out as numbers.

15.8 An elaborated webserver with different functions

To further deepen your understanding of the http-package and how to build webserver functionality, study and experiment with the following example: first the code is listed, then the usage of the different functionalities of the program and its output are shown in a table.

Listing 15.20—elaborated_webserver.go:

```
package main
import (
        "bytes"
        "expvar"
        "flag"
        "fmt"
        "net/http"
        "io"
        "log"
        "os"
        "strconv"
)

// hello world, the web server
var helloRequests = expvar.NewInt("hello-requests")
```

```
// flags:
var webroot = flag.String("root", "/home/user", "web root directory")
// simple flag server
var booleanflag = flag.Bool("boolean", true, "another flag for testing")

// Simple counter server. POSTing to it will set the value.
type Counter struct {
        n int
}

// a channel
type Chan chan int

func main() {
        flag.Parse()
        http.Handle("/", http.HandlerFunc(Logger))
        http.Handle("/go/hello", http.HandlerFunc(HelloServer))
        // The counter is published as a variable directly.
        ctr := new(Counter)
        expvar.Publish("counter", ctr)
        http.Handle("/counter", ctr)
        http.Handle("/go/", http.StripPrefix("/go/", http.FileServer(http.
        Dir(*webroot))))
        http.Handle("/flags", http.HandlerFunc(FlagServer))
        http.Handle("/args", http.HandlerFunc(ArgServer))
        http.Handle("/chan", ChanCreate())
        http.Handle("/date", http.HandlerFunc(DateServer))
        err := http.ListenAndServe(":12345", nil)
        if err != nil {
               log.Panicln("ListenAndServe:", err)
        }
}

func Logger(w http.ResponseWriter, req *http.Request) {
        log.Print(req.URL.String())
        w.WriteHeader(404)
        w.Write([]byte("oops"))
}

func FlagServer(w http.ResponseWriter, req *http.Request) {
```

```
         w.Header().Set("Content-Type", "text/plain; charset=utf-8")
         fmt.Fprint(w, "Flags:\n")
         flag.VisitAll(func(f *flag.Flag) {
         if f.Value.String() != f.DefValue {
                fmt.Fprintf(w, "%s = %s [default = %s]\n", f.Name, f.Value.
                String(), f.DefValue)
         } else {
                fmt.Fprintf(w, "%s = %s\n", f.Name, f.Value.String())
                }
         })
}

// simple argument server
func ArgServer(w http.ResponseWriter, req *http.Request) {
         for _, s := range os.Args {
                fmt.Fprint(w, s, " ")
         }
}

func HelloServer(w http.ResponseWriter, req *http.Request) {
         helloRequests.Add(1)
         io.WriteString(w, "hello, world!\n")
}

// This makes Counter satisfy the expvar.Var interface, so we can export
// it directly.
func (ctr *Counter) String() string { return fmt.Sprintf("%d", ctr.n) }

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
         switch req.Method {
         case "GET": // increment n
                ctr.n++
         case "POST": // set n to posted value
                buf := new(bytes.Buffer)
                io.Copy(buf, req.Body)
                body := buf.String()
                if n, err := strconv.Atoi(body); err != nil {
                  fmt.Fprintf(w, "bad POST: %v\nbody: [%v]\n", err, body)
                } else {
                  ctr.n = n
                  fmt.Fprint(w, “counter reset\n”)
                }
         }
         fmt.Fprintf(w, “counter = %d\n”, ctr.n)
}

func ChanCreate() Chan {
         c := make(Chan)
         go func(c Chan) {
                for x := 0; ; x++ {
                       c <- x
                }
         }(c)
         return c
}

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
         io.WriteString(w, fmt.Sprintf(“channel send #%d\n”, <-ch))
}

// exec a program, redirecting output
func DateServer(rw http.ResponseWriter, req *http.Request) {
         rw.Header().Set(“Content-Type”, “text/plain; charset=utf-8”)
         r, w, err := os.Pipe()
         if err != nil {
                fmt.Fprintf(rw, “pipe: %s\n”, err)
                return
         }

p, err := os.StartProcess(“/bin/date”, []string{“date”}, &os.ProcAttr{Files:
[]*os.File{nil, w, w}})
         defer r.Close()
         w.Close()
         if err != nil {
                fmt.Fprintf(rw, “fork/exec: %s\n”, err)
                return
         }w
         defer p.Release()
         io.Copy(rw, r)
         wait, err := p.Wait(0)
```

```
        if err != nil {
                fmt.Fprintf(rw, "wait: %s\n", err)
                return
        }
        if !wait.Exited() || wait.ExitStatus() != 0 {
                fmt.Fprintf(rw, "date: %v\n", wait)
                return
        }
}
```

handler /call with url: *response in browser*

Logger: http://localhost:12345/ (root) *oops*

The Logger prints a 404 Not Found header with w.WriteHeader(404).

This technique is generally useful, whenever an error is encountered in the web processing code, it can be applied like this:

```
if err != nil {
        w.WriteHeader(400)
        return
}
```

It also prints date + time through the logger function and the url for each request on the command window of the webserver

HelloServer: http://localhost:12345/go/hello *hello, world!*

The package `expvar` makes it possible to create variables (of types Int, Float, String) and make them public by publishing them. It exposes these variables via HTTP at /debug/vars in JSON format. It is typically used for e.g. operation counters in servers; `helloRequests` is such an int64 variable, this handler adds 1 to it, and then writes "hello, world!" to the browser.

Counter:	http://localhost:12345/counter	*counter = 1*
	refresh (= GET)	*counter = 2*

The counter object ctr has a String() method and thus implements the Var interface. This makes it publishable, although it is a struct. The function ServeHTTP is a handler for ctr because it has the right signature.

FileServer: http://localhost:12345/go/ggg.html   *404 page not found*

FileServer returns a handler that serves HTTP requests with the contents of the file system rooted at root. To use the operating system's file system, use http.Dir, as in:

```
http.Handle("/go/", http.FileServer(http.Dir("/tmp")))
```

FlagServer: http://localhost:12345/flags
Flags: boolean = true   root = /home/rsc

This handler uses the flag.VisitAll function to loop through all the flags, prints their name, value and default value (if different from value).

ArgServer: http://localhost:12345/args   *./elaborated_webserver.exe*

This handler loops through `os.Args` to prints out all command-line arguments; if there are none only the program name (the path to the executable) is printed.

Channel: http://localhost:12345/chan   *channel send #1*
Refresh: *channel send #2*

The channel's ServeHTTP method shows the next integer from the channel at each new request. So a webserver can take its response from a channel, populated by another function (or even a client). The following snippet shows a handler function which does exactly that, but also times out after 30 s:

```
func ChanResponse(w http.ResponseWriter, req *http.Request) {
         timeout := make (chan bool)

          go func () {
                 time.Sleep(30e9)
                 timeout <- true
         }()

          select {
          case msg := <-messages:
                io.WriteString(w, msg)
          case stop := <-timeout:
                return
          }
}
```

DateServer: http://localhost:12345/date *shows current datetime*
(works only on Unix because calls /bin/date)

Possible output: `Thu Sep 8 12:41:09 CEST 2011`
`os.Pipe()` returns a connected pair of Files; reads from r, and returns bytes written to w. It returns the files and an error, if any: `func Pipe() (r *File, w *File, err error)`

15.9 Remote procedure calls with rpc

Go programs can communicate with each other through the net/rpc-package, so this is another application of the client-server paradigm. It provides a convenient means of making function calls over a network connection. Of course this is only useful when the programs run on different machines. The rpc package builds on gobs (see § 12.11) to turn its encode/decode automation into transport for method calls across the network.

A server registers an object, making it visible as a *service* with the type-name of the object: this provides access to the exported methods of that object across a network or other I/O connection for remote clients. It is all about exposing methods on types over a network.

The package uses the http- and tcp-protocol, and the gob package for data-transport. A server may register multiple objects (services) of different types but it is an error to register multiple objects of the same type.

Here we discuss a simple example: we define a type Args and a method Multiply on it, preferably in its own package; the methods have to return a possible error

`Listing 15.21—rpc_objects.go:`

```
package rpc_objects
import “net”

type Args struct {
        N, M int
}

func (t *Args) Multiply(args *Args, reply *int) net.Error {
        *reply = args.N * args.M
        return nil
}
```

The server makes an object `calc` of that type and registers it with rpc.Register(object), calls HandleHTTP(), and starts listening with net.Listen on an address. You can also register the object by name like: `rpc.RegisterName("Calculator", calc)`

For every listener that comes in, a goroutine http.Serve(listener, nil) is started, creating a new service thread for each incoming HTTP-connection. We have to keep the server awake for a specified period with e.g. `time.Sleep(1000e9)`.

Listing 15.22—rpc_server.go:

```
package main
import (
    "net/http"
    "log"
    "net"
    "net/rpc"
    "time"
    "./rpc_objects"
)

func main() {
    calc := new(rpc_objects.Args)
    rpc.Register(calc)
    rpc.HandleHTTP()
    listener, e := net.Listen("tcp", "localhost:1234")
    if e != nil {
        log.Fatal("Starting RPC-server -listen error:", e)
    }
    go http.Serve(listener, nil)
    time.Sleep(1000e9)
}
/* Output:
Starting Process E:/Go/GoBoek/code_examples/chapter_14/rpc_server.exe ...
** after 5 s: **
End Process exit status 0
*/
```

The client has to know the definition of the object type and its methods. It calls rpc.DialHTTP(), and when the connection client is made, can invoke remote methods upon it with client.Call("Type. Method", args, &reply), where Type and Method is the remotely defined Type and the Method you want to call, args is an initialized object of that type, and reply is a variable which has to be declared before and in which the result of the method call will be stored.

Listing 15.23—rpc_client.go:

```
package main
import (
        "fmt"
        "log"
        "net/rpc"
        "./rpc_objects"
)
const serverAddress = "localhost"

func main() {
        client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
        if err != nil {
                log.Fatal("Error dialing:", err)
        }
        // Synchronous call
        args := &rpc_objects.Args{7, 8}
        var reply int
        err = client.Call("Args.Multiply", args, &reply)
        if err != nil {
                log.Fatal("Args error:", err)
        }
        fmt.Printf("Args: %d * %d = %d", args.N, args.M, reply)
}
```

First start the server, and then a client process: this then gets the following result:

```
/* Output:
Starting Process E:/Go/GoBoek/code_examples/chapter_14/rpc_client.exe ...

Args: 7 * 8 = 56
End Process exit status 0
*/
```

This call is synchronous, so waits for the result to come back. An asynchronous call can be made as follows:

```
call1 := client.Go("Args.Multiply", args, &reply, nil)
replyCall := <- call1.Done
```

If the last argument has value nil, a new channel will be allocated when the call is complete.

If you have a Go server running as root and want to run some of your code as a different user, the package go-runas by Brad Fitz uses the rpc package to accomplish just that: https://github.com/bradfitz/go-runas. We will see an application of rpc in a full fledged project in chapter 19.

15.10 Channels over a network with netchan

Remark: The Go team decided to improve and rework the existing version of the package `netchan`. This package has been moved to `old/netchan, and the old/` package hierarchy, which holds deprecated code, will not be part of Go 1. This § discusses the concept of the `netchan` package for backward compatibility reasons.

A technique closely related to rpc is the usage of channels over a network. The channels as used in chapter 14 are local, they exist only in the memory space of the machine on which they are executed. The package netchan implements type-safe networked channels: it allows the two ends of a channel to appear on different computers connected by a network. It does this by transporting data sent to a channel on one machine so it can be received by a channel of the same type on the other computer. An exporter publishes a (set of) channel(s) by name. An importer connects to the exporting machine and imports the channel(s) by name. After importing the channel(s), the two machines can use the channel(s) in the usual way. Networked channels are not synchronized, they act like buffered channels.

On the sending machine, the code goes schematically like this:

```
exp, err := netchan.NewExporter(“tcp”, “netchanserver.mydomain.com:1234”)
if err != nil {
        log.Fatalf(“Error making Exporter: %v”, err)
}
ch := make(chan myType)
err := exp.Export(“sendmyType”, ch, netchan.Send)
if err != nil {
        log.Fatalf(“Send Error: %v”, err)
}
```

And on the receiving side:

```
imp, err := netchan.NewImporter(“tcp”, “netchanserver.mydomain.com:1234”)
if err != nil {
        log.Fatalf(“Error making Importer: %v”, err)
}
ch := make(chan myType)
err = imp.Import("sendmyType", ch, netchan.Receive)
if err != nil {
        log.Fatalf("Receive Error: %v", err)
}
```

15.11 Communication with websocket

Remark: The Go team decided for Go 1 to move the package `websocket` out of the Go standard library to the subrepository `websocket of code.google.com/p/go`. It is also expected to change significantly in the near future.

The line `import "websocket"` will then become:

```
import websocket "code.google.com/p/go/websocket"
```

The websocket protocol in contrast to the http-protocol is based upon a lasting connection between client and server throughout their dialogue, but otherwise it functions in almost the same way as http. In Listing 15.24 we have a typical websocket server, which is started on its own and listens to websocket-clients dialing in. Listing 15.25 shows code for such a client that terminates after 5 s. When the connection comes in, the server first prints: `new connection`; when the client stops, the server prints: `EOF => closing connection`

Listing 15.24—websocket_server.go:

```
package main
import (
        "fmt"
        "net/http"
        "websocket"
)

func server(ws *websocket.Conn) {
        fmt.Printf("new connection\n")
        buf := make([]byte, 100)
        for {
                if _, err := ws.Read(buf); err != nil {
                        fmt.Printf("%s", err.Error())
                        break
                }
        }
        fmt.Printf(“ => closing connection\n”)
        ws.Close()
}

func main() {
        http.Handle(“/websocket”, websocket.Handler(server))
        err := http.ListenAndServe(“:12345”, nil)
        if err != nil {
               panic(“ListenAndServe: “ + err.Error())
        }
}
```

Listing 15.25—websocket_client.go:

```
package main
import (
        “fmt”
        “time”
        “websocket”
)

func main() {
        ws, err := websocket.Dial(“ws://localhost:12345/websocket”, “”,
               “http://localhost/”)
        if err != nil {
               panic(“Dial: “ + err.Error())
        }
        go readFromServer(ws)
        time.Sleep(5e9)
        ws.Close()
}

func readFromServer(ws *websocket.Conn) {
        buf := make([]byte, 1000)
        for {
               if _, err := ws.Read(buf); err != nil {
                      fmt.Printf(“%s\n”, err.Error())
                      break
               }
        }
}
```

15.12 Sending mails with smtp

The package smtp implements the Simple Mail Transfer Protocol for sending mails. It contains a type Client that represents a client connection to an SMTP server:

- Dial returns a new Client connected to an SMTP server
- Set Mail (=from) and Rcpt (= to)
- Data returns a writer that can be used to write the data, here with buf.WriteTo(wc)

Listing 15.26—smtp.go:

```
package main
import (
        “bytes”
        “log”
        “net/smtp”
)

func main() {
        / Connect to the remote SMTP server.
        client, err := smtp.Dial(“mail.example.com:25”)
        if err != nil {
               log.Fatal(err)
        }
        // Set the sender and recipient.
        client.Mail(“sender@example.org”)
        client.Rcpt(“recipient@example.net”)
        // Send the email body.
        wc, err := client.Data()
        if err != nil {
               log.Fatal(err)
        }
        defer wc.Close()
        buf := bytes.NewBufferString(“This is the email body.”)
        if _, err = buf.WriteTo(wc); err != nil {
               log.Fatal(err)
        }
}
```

The function SendMail can be used if authentication is needed and when you have a number of recipients. It connects to the server at addr, switches to TLS (Transport Layer Security encryption and authentication protocol) if possible, authenticates with mechanism a if possible, and then sends an email from address from, to addresses to, with message msg:

```
func SendMail(addr string, a Auth, from string, to []string, msg []byte) error
```

Listing 15.27—smtp_auth.go:

```
package main
import (
        “log”
        “smtp”
)

func main() {
        // Set up authentication information.
        auth := smtp.PlainAuth(
                “”,
                “user@example.com”,
                “password”,
                “mail.example.com”,
        )
        // Connect to the server, authenticate, set the sender and recipient,
        // and send the email all in one step.
        err := smtp.SendMail(
                “mail.example.com:25”,
                auth,
                “sender@example.org”,
                []string{“recipient@example.net”},
                []byte(“This is the email body.”),
        )
        if err != nil {
                log.Fatal(err)
        }
}
```

PART 4

APPLYING GO

Chapter 16—Common Go Pitfalls or Mistakes

In the previous text we sometimes warned with !! … !! for Go misuses. So be sure to look for the specific section in the book on that subject when you encounter a difficulty in a coding situation like that. Here is an overview of pitfalls for your convenience, refering to where you can find more explanations and examples:

- Never use var p*a to not confuse pointer declaration and multiplication (§ 4.9)
- Never change the counter-variable in the for-loop itself (§ 5.4)
- Never use the value in a for-range loop to change the value itself (§ 5.4.4)
- Never use goto with a preceding label (§ 5.6)
- Never forget () after a function-name (chapter 6), specifically when calling a method on a receiver or invoking a lambda function as a goroutine
- Never use new() with maps (§ 8.1), always make
- When coding a String() method for a type, don't use fmt.Print or alike in the code (§ 10.7)
- Never forget to use Flush() when terminating buffered writing (§ 12.2.3)
- Never ignore errors, ignoring them can lead to program crashes (§ 13.1)
- Do not use global variables or shared memory, they make your code unsafe for running concurrently (§ 14.1)
- Use println only for debugging purposes

<u>Best practices:</u> In contrast use the following:

- Initialize a slice of maps the right way (§ 8.1.3).
- Always use the comma, ok (or checked) form for type assertions (§ 11.3).
- Make and initialize your types with a factory (§10.2-§18.4).
- Use a pointer as receiver for a method on a struct only when the method modifies the structure, otherwise use a value (§ 10.6.3).

In this chapter we assemble some of the most common mistakes or do-not's in Go-programming. We often refer to the full explanation and examples in previous chapters. Read the paragraph titles as what you should not do!

16.1 Hiding (shadowing) a variable by misusing short declaration.

```
var remember bool = false
if something {
        remember := true      // Wrong.
}
// use remember
```

In the previous code-snippet the variable remember will never become true outside of the if-body. If something is true, inside the if-body a new remember variable which hides the outer remember is declared because of :=, and there it will be true. But after the closing } of if remember regains its outer value false. So write it as:

```
if something {
        remember = true
}
```

This can also occur with a for-loop, and can be particularly subtle in functions with named return variables, as the following snippet shows:

```
func shadow() (err error) {
        x, err := check1() // x is created; err is assigned to
        if err != nil {
               return // err correctly returned
        }
        if y, err := check2(x); err != nil { // y and inner err are created
               return // inner err shadows outer err so nil is wrongly returned!
        } else {
               fmt.Println(y)
        }
        return
}
```

16.2 Misusing strings.

When you need to do a lot of manipulations on a string, mind that strings in Go (like in Java and C#) are immutable. String concatenations of the kind a += b are inefficient, especially when performed inside a loop. They cause numerous reallocations and copying of memory. *Instead one should use a bytes.Buffer to accumulate string content*, like in the following snippet:

```
var b bytes.Buffer
...
for condition {
        b.WriteString(str)  // appends string str to the buffer
}
return b.String()
```

Remark: Due to compiler-optimizations and depending on the size of the strings using a Buffer only starts to become more efficient when the number of iterations is > 15.

16.3 Using defer for closing a file in the wrong scope.

Suppose you are processing a range of files in a for-loop, and you want to make sure the files are closed after processing by using defer, like this:

```
for _, file := range files {
    if f, err = os.Open(file); err != nil {
        return
    }
    // This is /wrong/. The file is not closed when this loop iteration ends.
    defer f.Close()
    // perform operations on f:
    f.Process(data)
}
```

But at the end of the for-loop defer is not executed, so the files are not closed! Garbage collection will probably close them for you, but it can yield errors. Better do it like this:

```
for _, file := range files {
    if f, err = os.Open(file); err != nil {
        return
    }
    // perform operations on f:
    f.Process(data)
    // close f:
    f.Close()
}
```

Defer is only executed at the return of a function, not at the end of a loop or some other limited scope.

16.4 Confusing new() and make()

We have talked about this and illustrated it with code already at great length in § 7.2.4 and § 10.2.2. The point is:

- *for slices, maps and channels: use make*
- *for arrays, structs and all value types: use new*

16.5 No need to pass a pointer to a slice to a function

A slice, as we saw in § 4.9, is a pointer to an underlying array. Passing a slice as a parameter to a function is probably what you always want: namely passing a pointer to a variable to be able to change it, and not passing a copy of the data.

So you want to do this:

```
      func findBiggest( listOfNumbers []int ) int {}
```

not this:

```
      func findBiggest( listOfNumbers *[]int ) int {}
```

Do not dereference a slice when used as a parameter!

16.6 Using pointers to interface types

Look at the following program: nexter is an interface with a method next() meaning read the next byte. nextFew1 has this interface type as parameter and reads the next num bytes, returning them as a slice: this is ok. However nextFew2 uses a pointer to the interface type as parameter: when using the next() function, we get a clear compile error: `n.next undefined (type *nexter has no field or method next)`

Listing 16.1 pointer_interface.go (does not compile!):

```
      package main
      import (
          “fmt”
      )
      type nexter interface {
          next() byte
      }
      func nextFew1(n nexter, num int) []byte {
          var b []byte
          for i:=0; i < num; i++ {
              b[i] = n.next()
          }
          return b
      }
      func nextFew2(n *nexter, num int) []byte {
          var b []byte
          for i:=0; i < num; i++ {
      b[i] = n.next() // compile error:
                      // n.next undefined (type *nexter has no field or method next)
          }
          return b
      }
      func main() {
          fmt.Println("Hello World!")
      }
```

So *never use a pointer to an interface type, this is already a pointer*!

16.7 Misusing pointers with value types

Passing a value as a parameter in a function or as receiver to a method may seem a misuse of memory, because a value is always copied. But on the other hand values are allocated on the stack, which is quick and relatively cheap. If you would pass a pointer to the value instead the Go compiler in most cases will see this as the making of an object, and will move this object to the heap, so also causing an additional memory allocation: therefore nothing was gained in using a pointer instead of the value!

16.8 Misusing goroutines and channels

For didactic reasons and for gaining insight into their working, a lot of the examples in chapter 14 applied goroutines and channels in very simple algorithms, for example as a generator or iterator. In practice often you don't need the concurrency, or you don't need the overhead of the goroutines with channels, passing parameters using the stack is in many cases far more efficient.

Moreover it is likely to leak memory if you break or return or panic your way out of the loop, because the goroutine then blocks in the middle of doing something. In real code, it is often better to just write a simple procedural loop. *Use goroutines and channels only where concurrency is important!*

16.9 Using closures with goroutines

Look at the following code:

Listing 16.2 closures_goroutines.go:

```
package main
import (
        “fmt”
        “time”
)

var values = [5]int{10, 11, 12, 13, 14}

func main() {
        // version A:
        for ix := range values {  // ix is the index
               func() {
                      fmt.Print(ix, “ ”)
               }()             // call closure, prints each index
        }
        fmt.Println()
        // version B: same as A, but call closure as a goroutine
        for ix := range values {
               go func() {
                      fmt.Print(ix, “ ”)
               }()
        }
        fmt.Println()
        time.Sleep(5e9)
        // version C: the right way
        for ix := range values {
               go func(ix interface{}) {
                      fmt.Print(ix, “ ”)
               }(ix)
        }
        fmt.Println()
        time.Sleep(5e9)
        // version D: print out the values:
        for ix := range values {
               val := values[ix]
               go func() {
                      fmt.Print(val, " ")
               }()
        }
        time.Sleep(1e9)
}
/* Output:      0 1 2 3 4
               4 4 4 4 4
               1 0 3 4 2
               10 11 12 13 14
*/
```

Version A calls 5 times a closure which prints the value of the index, version B does the same but invokes each closure as a goroutine, argumenting that this would be faster because the closures execute in parallel. If we leave enough time for all goroutines to execute, the output of version B is: 4 4 4 4 4 . Why is this ? The ix variable in the above loop B is actually a single variable that takes on the index of each array element. Because the closures are all only bound to that one variable, there is a very good chance that when you run this code you will see the last index (4) printed for every iteration instead of each index in sequence, because the goroutines will probably not begin executing until after the loop, when ix has got the value 4.

The right way to code that loop is version C: invoke each closure with ix as a parameter. ix is then evaluated at each iteration and placed on the stack for the goroutine, so each index is available to the goroutine when it is eventually executed. Note that the output can be 0 2 1 3 4 or 0 3 1 2 4 or . . . , depending on when each of the goroutines can start.

In version D we print out the values of the array; why does this work and version B does not ?

Because variables declared within the body of a loop (as val here) are not shared between iterations, and thus can be used separately in a closure.

16.10 Bad error handling

Stick to the patterns described in chapter 13 and summarized in §17.1 and § 17.2 (4).

16.10.1 Don't use booleans:

Making a boolean variable whose value is a test on the error-condition like in the following is superfluous: `var good bool`

```
        // test for an error, good becomes true or false
        if !good {
                return errors.New("things aren't good")
        }
```

Test on the error immediately:

```
... err1 := api.Func1()
if err1 != nil { … }
```

16.10.2 Don't clutter your code with error-checking:

Avoid writing code like this:

```
... err1 := api.Func1()
if err1 != nil {
   fmt.Println("err: " + err.Error())
   return
}

err2 := api.Func2()
if err2 != nil {
...
   return
}
```

First include the call to the functions in an initialization statement of the if's.

But even then the errors are reported (by printing them) with if-statements scattered throughout the code. With this pattern, it is hard to tell what is normal program logic and what is error checking/reporting. Also notice that *most* of the code is dedicated to error conditions at any point in the code. A good solution is to wrap your error conditions in a closure wherever possible, like in the following example:

```
func httpRequestHandler(w http.ResponseWriter, req *http.Request) {
    err := func () error {
        if req.Method != "GET" {
            return errors.New("expected GET")
        }

        if input := parseInput(req); input != "command" {
            return errors.New("malformed command")
        }
  // other error conditions can be tested here
```

```
        } ()

        if err != nil {
            w.WriteHeader(400)
            io.WriteString(w, err)
            return
        }
        doSomething() ...
```

This approach clearly separates the error checking, error reporting, and normal program logic (for a more elaborated treatment see § 13.5).

Answer the following questions before starting to read Chapter 17:

Question 16.1: Sum up all occurrences of the comma,ok pattern you can remember.
Question 16.2: Sum up all occurrences of the defer pattern you can remember.

Chapter 17—Go Language Patterns

17.1 The comma, ok pattern

While studying the Go-language in parts 2 and 3, we encountered several times the so called *comma, ok Idiom*: an expression returns 2 values, the first of which is a value or nil, and the second is true/false or an error. An if-condition with initialization and then testing on the second-value leads to succinct and elegant code. This is a very important pattern in idiomatic Go-code. Here are all cases summarized:

(1) Testing for errors on function return (§ 5.2):

```
if value, err := pack1.Func1(param1); err != nil {
fmt.Printf("Error %s in pack1.Func1 with parameter %v", err.Error(), param1)
              return err
       }
// no error in Func1:
       Process(value)

e.g.: os.Open(file) strconv.Atoi(str)
```

The function in which this code occurs returns the error to the caller, giving it the value nil when the normal processing was successful and so has the signature:

```
func SomeFunc() error {
       …
       if value, err := pack1.Func1(param1); err != nil {
              …
              return err
       }
       …
       return nil
}
```

The same pattern is used when recovering from a panic with defer (see §17.2 (4)).

A good pattern for clean error checking is using closures, see § 16.10.2

(2) Testing if a key-value item exists in a map (§ 8.2): does map1 have a value for key1?

```
if value, isPresent = map1[key1]; isPresent {
        Process(value)
}
// key1 is not present
…
```

(3) Testing if an interface variable varI is of type T: type-assertions (§ 11.3):

```
if value, ok := varI.(T); ok {
        Process(value)
}
// varI is not of type T
```

(4) Testing if a channel ch is closed (§ 14.3):

```
        for input := range ch {
        Process(input)
        }
```

or:

```
        for {
            if input, open := <-ch; !open {
        break  // channel is closed
            }
            Process(input)
        }
```

17.2 The defer pattern

Using defer ensures that all resources are properly closed or given back to the 'pool' when the resources are not needed anymore. Secondly it is paramount to recover from panicking.

(1) Closing a file stream: (see § 12.7)

```
// open a file f
defer f.Close()
```

(2) Unlocking a locked resource (a mutex): (see §9.3)

```
mu.Lock()
defer mu.Unlock()
```

(3) Closing a channel (if necessary):

```
ch := make(chan float64)
defer close(ch)
```

or with 2 channels:

```
answerα, answerβ := make(chan int), make(chan int)
defer func() { close(answerα); close(answerβ) }()
```

(4) Recovering from a panic: (see §13.3)

```
defer func() {
       if err := recover(); err != nil {
              log.Printf("run time panic: %v", err)
}
```

(5) Stopping a Ticker: (see §14.5)

```
tick1 := time.NewTicker(updateInterval)
defer tick1.Stop()
```

(6) Release of a process p: (see §13.6)

```
p, err := os.StartProcess(…, …, …)
defer p.Release()
```

(7) Stopping CPUprofiling and flushing the info: (see § 13.10)

```
pprof.StartCPUProfile(f)
```

```
defer pprof.StopCPUProfile()
```

It can also be used for not forgetting to print a footer in a report.

17.3 The visibility pattern

In § 4.2.1 we saw how the simple Visibility rule dictates the access-mode to types, variables and functions in Go. § 10.2.1 showed how you can make the use of the Factory function mandatory when defining types in separate packages.

17.4 The operator pattern and interface

An operator is a unary or binary function which returns a new object and does not modify its parameters, like + and *. In C++, special infix operators (+, -, *, etc) can be overloaded to support math-like syntax, but apart from a few special cases Go does not support operator overloading: to overcome this limitation operators must be simulated with functions. Since Go supports a procedural as well as an object-oriented paradigm there are 2 options:

17.4.1 Implement the operators as functions

The operator is implemented as a package-level function to operate on one or two parameters and return a new object, implemented in the package dedicated to the objects on which they operate. For example if we implement matrix manipulation in a package matrix, this would contain addition of matrices Add() and multiplication Mult() which result in a matrix. These would be called on the package name itself, so that we could make expressions of the form: `m := matrix.Add(m1, matrix.Mult(m2, m3))`

If we would like to differentiate between different kinds of matrices (sparse, dense) in these operations because there is no function overloading, we would have to give them different names, as in:

```
func addSparseToDense (a *sparseMatrix, b *denseMatrix) *denseMatrix
func addDenseToDense (a *denseMatrix, b *denseMatrix) *denseMatrix
func addSparseToSparse (a *sparseMatrix, b *sparseMatrix) *sparseMatrix
```

This is not very elegant, and the best we can do is hide these as private functions and expose as public API a single public function Add(). This can operate on any combination of supported parameters by type-testing them in a nested type switch:

```
func Add(a Matrix, b Matrix) Matrix {
    switch a.(type) {
    case sparseMatrix:
        switch b.(type) {
        case sparseMatrix:
            return addSparseToSparse(a.(sparseMatrix), b.(sparseMatrix))
        case denseMatrix:
            return addSparseToDense(a.(sparseMatrix), b.(denseMatrix))
…
    default:
        // unsupported arguments
        …
    }
}
```

But the more elegant and preferred way is to implement the operators as methods, as it is done everywhere in the standard library. More information on this package by Ryanne Dolan implementing linear algebra can be found at: http://code.google.com/p/gomatrix/

17.4.2 Implement the operators as methods

Methods can be differentiated according to their receiver type, so instead of having to use different function names, we can simply define an Add method for each type:

```
func (a *sparseMatrix) Add(b Matrix) Matrix
func (a *denseMatrix) Add(b Matrix) Matrix
```

Each method returns a new object which becomes the receiver of the next method call, so we can make *chained* expressions: `m := m1.Mult(m2).Add(m3)`
which is shorter and cleared than the procedural form of §17.4.1

The correct implementation can again be selected at runtime based on a type-switch:

```
func (a *sparseMatrix) Add(b Matrix) Matrix {
    switch b.(type) {
    case sparseMatrix:
            return addSparseToSparse(a.(sparseMatrix), b.(sparseMatrix))
    case denseMatrix:
            return addSparseToDense(a.(sparseMatrix), b.(denseMatrix))
         …
    default:
        // unsupported arguments
        …
        }
    }
```

again easier than the nested type switch of §17.4.1

17.4.3 Using an interface

When operating with the same methods on different types, the concept of creating a generalizing interface to implement this polymorphism should come to mind.

We could for example define the interface Algebraic:

```
type Algebraic interface {
    Add(b Algebraic) Algebraic
    Min(b Algebraic) Algebraic
    Mult(b Algebraic) Algebraic
      …
    Elements()
}
```

and define the methods Add(), Min(), Mult(), … for our matrix types.

Each type which implements the Algebraic interface above will allow for method chaining. Each method implementation should use a type-switch to provide optimized implementations based on the parameter type. Additionally, a default case should be specified which relies only on the methods in the interface:

```
func (a *denseMatrix) Add(b Algebraic) Algebraic {
    switch b.(type) {
    case sparseMatrix:
        return addDenseToSparse(a, b.(sparseMatrix))
    default:
        for x in range b.Elements() …
    …
}
```

If a generic implementation cannot be implemented using only the methods in the interface, you probably are dealing with classes that are not similar enough, and this operator pattern should be abandoned. For example, it does not make sense to write a.Add(b) if a is a set and b is a matrix; therefore, it will be difficult to implement a generic a.Add(b) in terms of set and matrix operators. In this case, split your package in two and provide separate AlgebraicSet and AlgebraicMatrix interfaces.

Chapter 18—Useful Code Snippets—Performance Advice

18.1 Strings

(1) How to change a character in a string:

```
str:="hello"
c:=[]byte(s)
c[0]='c'
s2:= string(c) // s2 == "cello"
```

(2) How to take a part(substring) of a string str:

```
substr := str[n:m]
```

(3) How to loop over a string str with for or for-range:

```
// gives only the bytes:
for i:=0; i < len(str); i++ {
 … = str[i]
}

// gives the Unicode characters:
for ix, ch := range str {
…
}
```

(4) Number of bytes in a string str: `len(str)`
Number of characters in a string str:
FASTEST: `utf8.RuneCountInString(str)`

`len([]int(str))`

(5) Concatenating strings:

```
FASTEST:           with a bytes.Buffer (see § 7.2.6)

Strings.Join() (see § 4.7.10)
+=
```

(6) How to parse command-line arguments: use the os or flag package

see examples § 12.4

18.2 Arrays and and slices

Making:	`arr1 := new([len]type)`
	`slice1 := make([]type, len)`
Initialization:	`arr1 := [...]type{i1, i2, i3, i4, i5}`
	`arrKeyValue := [len]type{i1: val1, i2: val2}`
	`var slice1 []type = arr1[start:end]`

(1) How to cut the last element of an array or slice line:

```
line = line[:len(line)-1]
```

(2) How to loop over an array(or slice) arr with for or for-range:

```
for i:=0; i < len(arr); i++ {
     … = arr[i]
}

for ix, value := range arr {
     …
    }
```

(3) Searching for a value V in a 2 dimensional array/slice arr2Dim:

```
found := false
Found: for row := range arr2Dim {
    for column := range arr2Dim[row]
        if arr2Dim[row][column] == V
            found = true
            break Found
            }
        }
    }
```

18.3 Maps

Making: `map1 := make(map[keytype]valuetype)`
Initialization: `map1 := map[string]int{“one”: 1, “two”: 2}`

(1) How to looping over a map map1 with for, range:

```
for key, value := range map1 {
…
}
```

(2) Testing if a key value key1 exists in a map1:

```
val1, isPresent = map1[key1]
which gives: val or zero-value, true or false
```

(3) Deleting a key in a map:

```
delete(map1, key1)
```

18.4 Structs

Making:
```
type struct1 struct {
        field1 type1
        field2 type2
        …
}
ms := new(struct1)
```
Initialization: `ms := &struct1{10, 15.5, “Chris”}`

Capitalize the first letter of the struct name to make it visible outside its package.

Often it is better to define a factory function for the struct, and force using that (see §10.2): `ms := Newstruct1{10, 15.5, “Chris”}`

```
func Newstruct1(n int, f float32, name string) *struct1 {
```

```
                        return &struct1{n, f, name}
                }
```

18.5 Interfaces

(1) How to test if a value v implements an interface Stringer:

```
if v, ok := v.(Stringer); ok {
fmt.Printf("implements String(): %s\n", v.String());
}
```

(2) A type classifier:

```
func classifier(items ...interface{}) {
    for i, x := range items {
        switch x.(type) {
        case bool:       fmt.Printf("param #%d is a bool\n", i)
        case float64:    fmt.Printf("param #%d is a float64\n", i)
        case int, int64: fmt.Printf("param #%d is an int\n", i)
        case nil:        fmt.Printf("param #%d is nil\n", i)
        case string:     fmt.Printf("param #%d is a string\n", i)
        default:         fmt.Printf("param #%d's type is unknown\n", i)
        }
    }
}
```

18.6 Functions

How to use recover to stop a panic terminating sequence (see § 13.3):

```
func protect(g func()) {
    defer func() {
            log.Println("done")
              // Println executes normally even if there is a panic
            if x := recover(); x != nil {
                   log.Printf("run time panic: %v", x)
            }
    }()
    log.Println("start")
    g()
}
```

18.7 Files

(1) How to open and read a file:

```
file, err := os.Open("input.dat")
       if err!= nil {
              fmt.Printf("An error occurred on opening the inputfile\n" +
                 "Does the file exist?\n" +
                 "Have you got acces to it?\n")
              return
       }
       defer file.Close()
       iReader := bufio.NewReader(file)
for {
       str, err := iReader.ReadString('\n')
       if err!= nil {
              return // error or EOF
       }
       fmt.Printf("The input was: %s", str)
}
```

(2) How to read and write a file with a sliced buffer:

```
func cat(f *file.File) {
        const NBUF = 512
        var buf [NBUF]byte
        for {
            switch nr, er := f.Read(buf[:]); true {
            case nr < 0:
               fmt.Fprintf(os.Stderr, "cat: error reading from %s: %s\n",
               f.String(), er.String())
               os.Exit(1)
            case nr == 0: // EOF
               return
            case nr > 0:
               if nw, ew := file.Stdout.Write(buf[0:nr]); nw != nr {
               fmt.Fprintf(os.Stderr, "cat: error writing from %s: %s\n",
               f.String(), ew.String())
               }
            }
        }
 }
```

18.8 Goroutines and channels

Performance advice:

A rule of thumb if you use parallelism to gain efficiency over serial computation: the amount of work done inside goroutine has to be much higher than the costs associated with creating goroutines and sending data back and forth between them.

1- Using buffered channels for performance:

   A buffered channel can easily double its throughput, depending on the context the performance gain can be 10x or more. You can further try to optimize by adjusting the capacity of the channel.

2- Limiting the number of items in a channel and packing them in arrays:

   Channels become a bottleneck if you pass a lot of individual items through them. You can work around this by packing chunks of data into arrays and then unpacking on the other end. This can be a speed gain of a factor 10x.

Making: `ch := make(chan type, buf)`

(1) How to loop over a channel ch with a for—range:

```
for v := range ch {
       // do something with v
}
```

(2) How to test if a channel ch is closed:

```
//read channel until it closes or error-condition
for {
    if input, open := <-ch; !open {
        break
    }
    fmt.Printf("%s ", input)
}
```

Or use (1) where the detection is automatic.

(3) How to use a channel to let the main program wait until the goroutine completes? (Semaphore pattern):

```
ch := make(chan int) // Allocate a channel.
// Start something in a goroutine; when it completes, signal on the channel.
go func() {
    // doSomething
    ch <- 1 // Send a signal; value does not matter.
}()
doSomethingElseForAWhile()
<-ch // Wait for goroutine to finish; discard sent value.
```

If the routine must block forever, omit ch <- 1 from the lambda function.

(4) Channel Factory pattern: the function is a channel factory and starts a lambda function as goroutine populating the channel

```
func pump() chan int {
       ch := make(chan int)
       go func() {
              for i := 0; ; i++ {
                     ch <- i
              }
       }()
       return ch
}
```

(5) Channel Iterator pattern:

(6) How to limit the amount of requests processed concurrently: see § 14.11

(7) How to parallelize a computation over a number of cores: see § 14.13

(8) Stopping a goroutine: runtime.Goexit()

(9) Simple timeout pattern:

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1e9) // one second
```

```
        timeout <- true
    }()

    select {
    case <-ch:
        // a read from ch has occurred
    case <-timeout:
        // the read from ch has timed out
    }
```

(10) How to use an in- and out-channel instead of locking:

```
    func Worker(in, out chan *Task) {
        for {
            t := <-in
            process(t)
            out <- t
        }
    }
```

(11) How to abandon synchronous calls that run too long: see § 14.5 2nd variant

(12) How to use a Ticker and Timers with channels: see § 14.5

(13) Typical server backend patterns: see § 14.4

18.9 Networking and web applications

18.9.1. Templating:

- make, parse and validate a template:

```
var strTempl = template.Must(template.New("TName").Parse(strTemplateHTML))
```

- when used in a web context: use the html filter to escape HTML special characters: {{html .}} or with a field `FieldName` {{ `.FieldName` |html }}
- use template-caching (see § 15.7)

18.10 General

```
How to stop a program when an error occurs:
      if err != nil {
                fmt.Printf("Program stopping with error %v", err)
                os.Exit(1)
      }

      or:       if err != nil {
      panic("ERROR occurred: " + err.Error())
      }
```

18.11 Performance best practices and advice

(1) Use the initializing declaration form := wherever possible (in functions).
(2) Use bytes if possible instead of strings
(3) Use slices instead of arrays.
(4) Use arrays or slices instead of a map where possible (see ref. 15)
(5) Use for range over a slice if you only need the value and not the index; this is slightly faster than having to do a slice lookup for every element.
(6) When the array is sparse (containing many 0 or nil-values), using a map can result in lower memory consumption.
(7) Specify an initial capacity for maps.
(8) When defining methods: use a pointer to a type (struct) as a receiver.
(9) Use constants or flags to extract constant values from the code.
(10) Use caching whenever possible when large amounts of memory are being allocated.
(11) Use template caching (see §15.7)

Chapter 19—Building a complete application

19.1 Introduction

In this chapter we will develop a complete program: *goto*, a URLShortener web application, because the web is all pervasive. The example is taken from the excellent lecture from Andrew Gerrand (see ref. 22). We will do this in 3 stages, each stage has more functionality and shows progressively more features of the Go language. We will draw heavily on what we have learned on web applications in chapter 15.

Version 1: a map and a struct are used, together with a Mutex from the sync package and a struct factory.
Version 2: the data is made persistent because written to a file in gob-format.
Version 3: the application is rewritten with goroutines and channels (see chapter 14)
Version 4: what to change if we want a json-version?
Version 5: a distributed version is made with the rpc protocol.

Because the code changes frequently it is not printed here, but instead you are referred to the download site.

19.2 Introducing Project UrlShortener

You certainly know that some addresses in the browser (called URLs) are (very) long and/or complex, and that there are services on the web which turn these into a nice short url, to be used instead. Our project is like that: it is *a web service* with 2 functionalities:

Add: given a long URL, it returns a short version, e.g.:

http://maps.google.com/maps?f=q&source=s_q&hl=en&geocode=&q=tokyo&sll=37.0625,-95.677068&sspn=68.684234,65.566406&ie=UTF8&hq=&hnear=Tokyo,+Japan&t=h&z=9

(A) becomes: http://goto/UrcGq

(B) and stores this pair of data

Redirect: whenever a shortened URL is requested, it redirects the user to the original, long URL: so if you type (B) in a browser, it redirects you to the page of (A).

VERSION 1—DATA STRUCTURE AND WEB SERVER FRONTEND:

The code of the 1st version *goto_v1* (discussed in §19.3 and §19.4) can be found in the map code_examples\chapter_19\goto_v1

19.3 Data structure

[the code for this § can be found in `goto_v1\store.go`]

When our application is running in production, it will receive many requests with short urls, and a number of requests to turn a long URL into a short one. But in which data structure will our program stock these data ? Both url-types (A) and (B) from § 19.2 are strings, moreover they relate to each other: given (B) as key we need to fetch (A) as value, they *map* to each other. To store our data in memory we need such a structure, which exists in nearly all programming languages under different names as hash table, dictionary, etc.

Go has such a `map built-in: a map[string]string.`

The key type is written between [ ], and the value type follows; we will learn all about maps in chapter 8. In any non trivial program it is useful to give the special types that you use a name. In Go we do this with the keyword type, so we define a: `type URLStore map[string]string`

It maps short URLs to long URLs, both are strings.

To make a variable of that type named m, just use: `m := make(URLStore)`

Suppose we want to store the mapping of http://goto/a to http://google.com/ in m, we can do this with the statement: `m["a"] = "http://google.com/"`

(We just store the suffix of http://goto/ as key, what comes before the key stays always the same). To retrieve its corresponding long url given "a", we write: `url := m["a"]`
and then the value of url is equal to "http://google.com/".

Note that with := we don't need to say that url is of type string: the compiler deduces the type from the value on the right side.

Making it thread-safe:

Our URLStore variable is the central memory datastore here: once we get some traffic there will be many requests of type Redirect. These are in fact only read operations: read with the given short URL as key and return the long URL as value. But requests of type Add are different, they change our URLStore adding new key-value pairs. When our service gets many of that update type-requests at once, the following problem could arise: the add operation could be interrupted by another request of the same type, and the long URL would perhaps not be written as value. Also there could be modifications together with reads, resulting perhaps in wrong values read. Our map does not guarantee that once an update starts it will terminate completely before a new update begins; in other words: a map is not thread-safe, goto will serve many requests concurrently, so we must make our URLStore type safe to access from separate threads. The simplest and classical way to do this is to add a lock to it. In Go this is a Mutex type from the sync package in the standard library which we now must import into our code (for locking in detail see §9.3).

We change the type definition of our **URLStore** to a struct type (which is just a collection of fields like in C or Java, we explain structs in Chapter 10) with two fields: the map and a RWMutex from the sync package:

```
import “sync”

type URLStore struct {
        urls map[string]string       // map from short to long URLs
        mu  sync.RWMutex
}
```

An RWMutex has two locks: one for readers, and one for writers. Many clients can take the read lock simultaneously, but only one client can take the write lock (to the exclusion of all readers), effectively serializing the updates, making them take place consecutively.

We will implement our read request type Redirect in a **Get** function, and our write request type Add as a **Set** function. The Get function looks like this:

```
func (s *URLStore) Get(key string) string {
        s.mu.RLock()
        url := s.urls[key]
        s.mu.RUnlock()
        return url
}
```

It takes a key (short URL) and returns the corresponding map value as url. The function works on a variable s, which is a pointer (see § 4.9) to our URLStore. But before reading the value we set a read-lock with s.mu.RLock(), so that no update can interrupt the read. After the read we unlock so that pending updates can start. What if the key is not present in the map? Then the zero value for the string type (an empty string) will be returned. Notice the .-notation familiar from OO languages: the method RLock() is called on the field mu of s.

The **Set** function needs both a key and url, and has to use the write lock Lock() to exclude any other updates at the same time. It returns a boolean true or false value to signal whether the Set was successful or not:

```
func (s *URLStore) Set(key, url string) bool {
        s.mu.Lock()
        _, present := s.urls[key]
        if present {
                s.mu.Unlock()
                return false
        }
        s.urls[key] = url
        s.mu.Unlock()
        return true
}
```

With the form `_, present := s.urls[key]` we can test to see whether our map already contains the key, then present become true, otherwise false. This is the so called comma, ok form which we will encounter frequently in Go code. If the key is already present Set returns a boolean false value and the map is not updated because we return from the function (so we don't allow shorts urls to be reused). If the key is not present, we add it to the map and return true. The _ on the left side is a placeholder for the value and we indicate that we will not use it because we assign it to _ . Note that as soon as possible (after the update) we Unlock() our URLStore.

<u>Using defer to simplify the code:</u>

In this case the code is still simple and it was easy to remember to do the Unlock(). However in more complex code this might be forgotten or put in the wrong place, leading to problems often difficult to track down. For these kinds of situations Go has a special keyword defer (see § 6.4), which allows in this case to signal the Unlock immediately after the Lock, however its effect is that the Unlock() will only be done just before returning from the function.

Get can be simplified to (we have eliminated the local variable url):

```
func (s *URLStore) Get(key string) string {
        s.mu.RLock()
        defer s.mu.RUnlock()
        return s.urls[key]
}
```

The logic for Set also becomes somewhat clearer (we don't need to think about unlocking anymore):

```
func (s *URLStore) Set(key, url string) bool {
        s.mu.Lock()
        defer s.mu.Unlock()
        _, present := s.urls[key]
        if present {
                return false
        }
        s.urls[key] = url
        return true
}
```

A factory for URLStore:

The URLStore struct contains a map field, which must be initialized with make before it can be used. Making an instance of a struct is done in Go by defining a function with prefix New, that returns an initialized instance of the type (here, and in most cases, a pointer to it):

```
func NewURLStore() *URLStore {
        return &URLStore{ urls: make(map[string]string) }
}
```

In the return we make a URLStore literal with our map initialized, the lock doesn't need to be specifically initialized; this is the standard way in Go of making struct objects. & is the address-of operator, to turn what we return into a pointer because NewURLStore returns a pointer `*URLStore`. We just call this function to make a URLStore variable s: `var store = NewURLStore()`

Using our URLStore:

To add a new short/long URL pair to our map, all we have to do is call the Set method on s, and since this was a boolean, we can immediately wrap it in an if-statement:

```
if s.Set("a", "http://google.com") {
        // success
}
```

To retrieve the long URL given a short URL we call the Get method on s and put the result in a variable url:

```
if url := s.Get(“a”); url != “” {
        // redirect to url
} else {
        // key not found
}
```

Here we make use of the fact that in Go an if can start with an initializing statement before the condition. We also need a method Count to give us the number of key-value pairs in the map, this is given by the built in len function:

```
func (s *URLStore) Count() int {
        s.mu.RLock()
        defer s.mu.RUnlock()
        return len(s.urls)
}
```

How will we compute the short URL given the long URL? For this we use a function `genKey(n int) string {…}` and for its integer parameter we give it the current value of s.Count().

[The exact algorithm is of little importance, an example code can be found in key.go.]

We can now make a **Put** method that takes a long URL, generates its short key with genKey, stores the URL under this (short url) key with the Set method, and returns that key:

```
func (s *URLStore) Put(url string) string {
        for {
                key := genKey(s.Count())
                if s.Set(key, url) {
                        return key
                }
        }
        // shouldn’t get here
        return “”
}
```

The for loop retries the Set until that is successful (meaning that we have generated a not yet existing short URL). Until now we have defined our datastore and functions to work with it (the code can be found in store.go). But this in itself doesn't do anything, we will have to define a webserver to deliver the Add and the Redirect services.

19.4 Our user interface: a web server frontend

[the code for this § can be found in `goto_v1\main.go`]

We haven't yet coded the function with which our program must be started up. This is (always) the function main() as in C, C++ or Java; in it we will start our webserver, e.g. we can start a local webserver on port 8080 with the command: `http.ListenAndServe(“:8080”, nil)`

(The functionality of a webserver comes from the package http, we treat this in depth in chapter 15). The webserver listens for incoming requests in an infinite loop, but we must also define how this server must respond to these requests. We do this by making so called HTTP handlers with the function HandleFunc, for example by coding:

```
    http.HandleFunc(“/add”, Add)
```

we say that every request which ends in /add will call a function Add (still to be made).

Our program will have two HTTP handlers:

- Redirect, which redirects short URL requests, and
- Add, which handles the submission of new URLs.

Schematically:

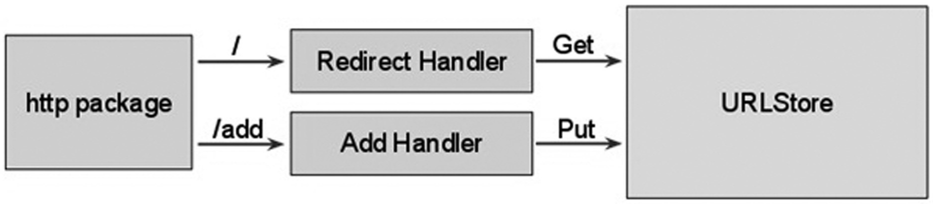


Fig 19.1: Handler functions in goto

Our minimal main() could look like:

```
    func main() {
            http.HandleFunc(“/”, Redirect)
            http.HandleFunc(“/add”, Add)
```

```
		http.ListenAndServe(":8080", nil)
	}
```

Requests to /add will be served by the Add handler; all other requests will be served by the Redirect handler. Handler functions get information from an incoming request (a variable r of type *http.Request), and they make and write their response to a variable w of type http.ResponseWriter .

What must our Add function do ?

i) reading in the long URL, that is: read the url from an html-form contained in an HTTP request with r.FormValue("url")
ii) put it into the store using our Put method on store
iii) send the corresponding short URL to the user

Each requirement translates in one codeline:

```
	func Add(w http.ResponseWriter, r *http.Request) {
		url := r.FormValue("url")
		key := store.Put(url)
		fmt.Fprintf(w, "http://localhost:8080/%s", key)
	}
```

The function Fprintf of the fmt package is used here to substitute key in the string containing %s, and then sending that string as response back to the client. Notice that Fprintf writes to a ResponseWriter, in fact Fprintf can write to any data structure that implements io.Writer(), which means that it implements a Write() method. io.Writer() is what is called in Go an interface, and we see that through the use of interfaces Fprintf is very general, it can write to a lot of different things. The use of interfaces is pervasive in Go, and makes code more generally applicable (see Chapter 11).

But we still need a form, we can display a form by using Fprintf again, this time writing a constant to w. Let's modify Add to display an HTML form when no url is supplied:

```
	func Add(w http.ResponseWriter, r *http.Request) {
		url := r.FormValue("url")
		if url == "" {
			fmt.Fprint(w, AddForm)
			return
		}
		key := store.Put(url)
```

```
                fmt.Fprintf(w, "http://localhost:8080/%s", key)
        }

        const AddForm = `
        <form method="POST" action="/add">
        URL: <input type="text" name="url">
        <input type="submit" value="Add">
        </form>
        `
```

In that case we send the constant string AddForm to the client, which is in fact the html necessary for creating a form with an input field url, and a submit button, which when pushed will post a request ending in /add. So the Add handler function is again invoked, now with a value for url from the text field. (The `` are needed to make a raw string, otherwise strings are enclosed in "" as usual).

The Redirect function finds the key in the HTTP request path (the short url key is the request path minus the first character, this can be written in Go as [1:]; for the request "/abc" the key would be "abc"), retrieves the corresponding long URL from the store with the Get function, and sends an HTTP redirect to the user. If the URL is not found, a 404 "Not Found" error is sent instead:

```
func Redirect(w http.ResponseWriter, r *http.Request) {
        key := r.URL.Path[1:]
        url := store.Get(key)
        if url == "" {
                http.NotFound(w, r)
                return
        }
        http.Redirect(w, r, url, http.StatusFound)
}
```

(http.NotFound and http.Redirect are helpers for sending common HTTP responses.)

Now we have gone through all the code of `goto_v1`

<u>Compiling and running:</u>

The executable is present in the map, so you can skip this section and test immediately if you want to. Within the map with the 3 go source-files there is also a Makefile, with which the application can be compiled and linked, just do:

for Linux, OS X: start make in the map in a terminal window or build the project in LiteIDE
for Windows: start the MINGW-environment via Start, All Programs, MinGW, MinGW Shell (see § 2.5 (5)), type: `make` in the command-window and enter:
    the source files are compiled and linked into a native .exe

The result is the executable program: `goto` in Linux/OS X or `goto.exe` in Windows.

To run it and thus start the webserver, give

> for Linux, OS X: the command `./goto`
> for Windows: start the application from GoIde (if Windows Firewall blocks the starting program: set allow this program)

Testing the program:

Open a browser and request the url: http://localhost:8080/add

This starts our Add handler function. There isn't yet any url variable in the form, so the response is the html-form with asks for input:

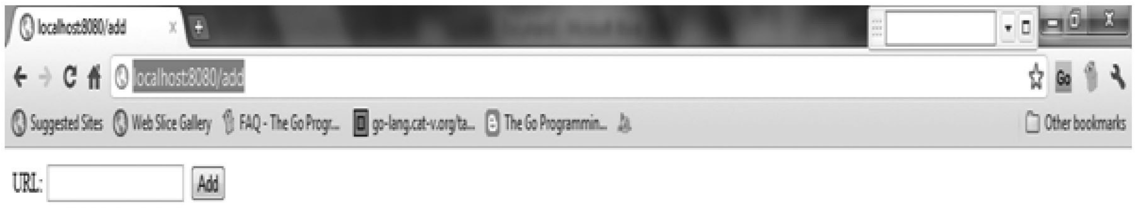
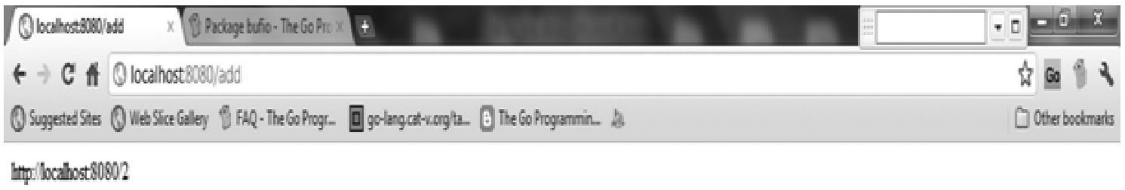


Fig 19.2: The Add handler

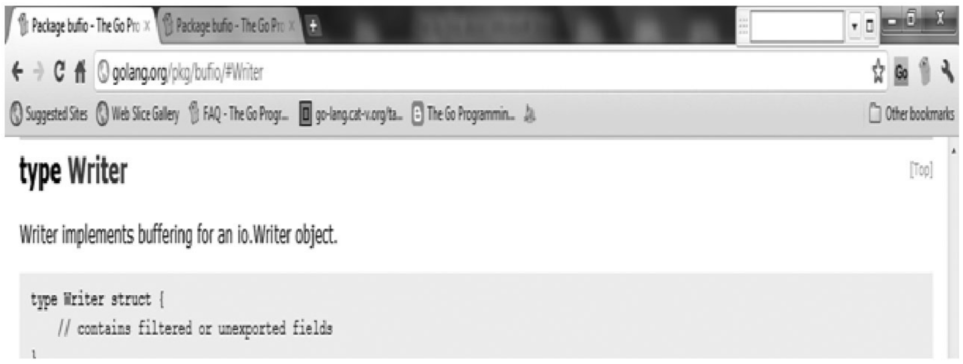
Add a (long) URL for which you want a short equivalent, like http://golang.org/pkg/bufio/#Writer and press the button. The application makes a short URL for you and prints that, e.g. http://localhost:8080/2



http://localhost:8080/2

<u>Fig 19.3</u>: The response of the Add handler

Copy and paste that URL in your browser address box and request it. The result is the Redirect handler in action, and the page of the long URL is shown.



type Writer

Writer implements buffering for an io.Writer object.

```
type Writer struct {
    // contains filtered or unexported fields
}
```

<u>Fig 19.4</u>: The response of the Redirect handler

<u>VERSION 2—ADDING PERSISTENT STORAGE:</u>

The code of the 2nd version *goto_v2* (discussed in §19.5) can be found in the map code_examples\chapter_19\goto_v2

19.5 Persistent storage: gob

[the code for this § can be found in `goto_v2\store.go and main.go`]

When the goto process (the webserver on port 8080) ends, and this has to happen sooner or later, the map with the shortened URLs in memory will be lost. To preserve our map data, we need to save them in a disk file. We will modify URLStore so that it writes its data to a file, and restores that data on goto start-up. For this we will use Go's `encoding/gob` package: this is a serialization and deserialization package that turns data structures into arrays (or more accurately slices) of bytes and vice versa (see § 12.11).

With the gob package's NewEncoder and NewDecoder functions you can decide where you write the data to or read it from. The resulting Encoder and Decoder objects provide Encode and Decode methods for writing and reading Go data structures to and from files. By the way: Encoder also implements the Writer interface, and so does Decoder for the Reader interface. We will add to URLStore a new file field (of type *os.File) that will be a handle to an open file that can be used for writing and reading.

```
type URLStore struct {
        urls          map[string]string
        mu            sync.RWMutex
        file          *os.File
}
```

We will call this file `store.gob` and give that name as a parameter when we instantiate the URLStore: `var store = NewURLStore("store.gob")`

Now we have to adapt our NewURLStore function:

```
func NewURLStore(filename string) *URLStore {
        s := &URLStore{urls: make(map[string]string)}
        f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
        if err != nil {
               log.Fatal("URLStore:", err)
        }
        s.file = f
        return s
}
```

The NewURLStore function now takes a filename argument, opens the file (see Chapter 12), and stores the *os.File value in the file field of our URLStore variable store, here locally called s.

The call to OpenFile can fail (our disk file could be removed or renamed for example).

It can return an error err, notice how Go handles this:

```
f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
if err != nil {
        log.Fatal("URLStore:", err)
}
```

When err is not nil, meaning there really was an error, we stop the program with a message. This is one way of doing it, mostly the error is returned to the calling function, but this pattern of testing errors again is ubiquitous in Go code. After the } we are certain the file is opened.

We open this file with writing enabled, more exactly in append-mode. Each time a new (short, long) URL pair is made in our program, we will store it through gob in the file "store.gob".

For that purpose we define a new struct type record:

```
type record struct {
        Key, URL string
}
```

and a new save method that writes a given key and url to disk as a gob-encoded record:

```
func (s *URLStore) save(key, url string) error {
        e := gob.NewEncoder(s.file)
        return e.Encode(record{key, url})
}
```

At the start of goto our datastore on disk must be read into the URLStore map, for this we have a load method

```
func (s *URLStore) load() error {
        if _, err := s.file.Seek(0, 0); err != nil {
                return err
        }
        d := gob.NewDecoder(s.file)
        var err error
        for err == nil {
                var r record
                if err = d.Decode(&r); err == nil {
                        s.Set(r.Key, r.URL)
                }
        }
        if err == io.EOF {
                return nil
        }
        return err
}
```

The new load method will Seek to the beginning of the file, read and Decode each record, and store the data in the map using the Set method. Again notice the all pervasive error-handling. The decoding of the file is an infinite loop which continues as long as there is no error:

```
for err == nil {
        …
}
```

If we get an error, it could be because we have just decoded the last record and then the error io.EOF (EndOfFile) occurs; if this is not the case we had an error while decoding and this is returned with return err. This method must be added to NewURLStore:

```
func NewURLStore(filename string) *URLStore {
        s := &URLStore{urls: make(map[string]string)}
        f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
        if err != nil {
               log.Fatal("Error opening URLStore:", err)
        }
        s.file = f
        if err := s.load(); err != nil {
               log.Println("Error loading data in URLStore:", err)
        }
        return s
}
```

Also in the Put function when we add a new url-pair to our map, this should also be saved immediately to the datafile:

```
func (s *URLStore) Put(url string) string {
        for {
               key := genKey(s.Count())
               if s.Set(key, url) {
                      if err := s.save(key, url); err != nil {
                             log.Println("Error saving to URLStore:", err)
                      }
                      return key
               }
        }
        panic("shouldn't get here")
}
```

Compile and test this 2nd version or simply use the executable that is present, and verify that it still knows the short urls after closing down the webserver (you can stop this process with CTRL/C in the terminal window). The 1st time goto is started the file store.gob doesn't yet exist, so when loading you get the error: `2011/09/11 11:08:11 Error loading URLStore: open store.gob: The system cannot find the file specified.`

Stop the process and restart, and then it works. Alternatively you can simply make an empty store.gob file before starting goto.

Remark: When starting goto the 2nd time, you probably get the error:

```
    Error loading URLStore: extra data in buffer
```

This is because gob is a stream based protocol that doesn't support restarting. In Version 4 we will remedy this situation by using json as storage protocol.

VERSION 3—ADDING GOROUTINES:

The code of the 3rd version *goto_3* (discussed in §19.6) can be found in the map code_examples\chapter_19\goto_v3

19.6 Using goroutines for performance

There is still a performance problem with the 2nd version if too many clients attempt to add URLs simultaneously: our map is safely updated for concurrent access thanks to the locking mechanism, but the immediate writing of each new record to disk is a bottleneck. The disk writes may happen simultaneously and depending on the characteristics of your OS, this may cause corruption. Even if the writes do not collide, each client must wait for their data to be written to disk before their Put function will return. Therefore, on a heavily I/O-loaded system, clients will wait longer than necessary for their Add requests to go through.

To remedy these issues, we must *decouple* the Put and save processes: we do this by using Go's concurrency mechanism. Instead of saving records directly to disk, we send them to a *channel*, which is a kind of buffer, so the sending function doesn't have to wait for it.

The `save` process which writes to disk reads from that channel and is started on a separate thread by launching it as a *goroutine* called `saveloop`. The main program and saveloop are now executed concurrently, so there is no more blocking.

We replace the file field in URLStore by a channel of records: `save chan record.`

```
type URLStore struct {
        urls   map[string]string
        mu     sync.RWMutex
        save   chan record
}
```

A channel, just like a map, must be made with make; we will do this in our changed factory NewURLStore and give it a buffer length of 1000, like: `save := make(chan record, saveQueueLength)`. To remedy our performance situation instead of making a function call to save each record to disk, Put can send a record to our buffered channel `save`:

```
func (s *URLStore) Put(url string) string {
        for {
                key := genKey(s.Count())
                if s.Set(key, url) {
                        s.save <- record{key, url}
                        return key
                }
        }
        panic("shouldn't get here")
}
```

At the other end of the save channel we must have a receiver: our new method saveLoop will run in a separate goroutine; it receives record values and writes them to a file. saveLoop is also started in the NewURLStore() function with the keyword go, and we can remove the now-unnecessary file opening code. Here is the modified NewURLStore():

```
const saveQueueLength = 1000

func NewURLStore(filename string) *URLStore {
        s := &URLStore{
                urls: make(map[string]string),
                save: make(chan record, saveQueueLength),
        }
        if err := s.load(filename); err != nil {
                log.Println("Error loading URLStore:", err)
        }
        go s.saveLoop(filename)
        return s
}
```

Here is the code for the method saveloop:

```
func (s *URLStore) saveLoop(filename string) {
        f, err := os.Open(filename, os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0644)
        if err != nil {
                log.Fatal("URLStore:", err)
        }
        defer f.Close()
e := gob.NewEncoder(f)
        for {
// taking a record from the channel and encoding it
r := <-s.save
        if err := e.Encode(r); err != nil {
                        log.Println("URLStore:", err)
                }
            }
    }
```

Records are read from the save channel in an infinite loop and encoded to the file.

In chapter 14 we study goroutines and channels in depth, but here we have seen a useful example for better managing the different parts of a program. Notice also that now we only make our Encoder object once, instead of each save, which also conserves some memory and processing.

Another amelioration can be made to make goto more flexible: instead of coding the filename, the listener address and the host name hard-coded or as constants in the program, we can define them as *flags*. That way they can be given new values if these are typed in on the command line when starting the program, if this is not done the default value from the flag will be taken. This functionality comes from a different package, so we have to: `import "flag"` (for more info on this package, see §12.4).

We first create some global variables to hold the flag values:

```
var (
        listenAddr = flag.String("http", ":8080", "http listen address")
        dataFile   = flag.String("file", "store.gob", "data store file name")
        hostname   = flag.String("host", "localhost:8080", "host name and port")
)
```

For processing command-line parameters we must add flag.Parse() to the main function, and instantiate the URLStore after the flags have been parsed, once we know the value of dataFile (in the code *dataFile is used, this is because a flag is a pointer and must be dereferenced to get the value, see §4.9):

```
var store *URLStore
func main() {
        flag.Parse()
        store = NewURLStore(*dataFile)
        http.HandleFunc("/", Redirect)
        http.HandleFunc("/add", Add)
        http.ListenAndServe(*listenAddr, nil)
}
```

In the Add handler we must now substitute *hostname instead of `localhost:8080`:

```
    fmt.Fprintf(w, "http://%s/%s", *hostname, key)
```

Compile and test this 3rd version or use the executable that is present.

VERSION 4—PERSISTENT STORAGE WITH JSON:

The code of the 4th version *goto_4* (discussed in §19.7) can be found in the map code_examples\chapter_19\goto_v4.

19.7 Using json for storage

If you are a keen tester you will perhaps have noticed that when goto is started 2 times, the 2nd time it has the short urls and works perfectly, however from the 3rd start onwards, we get the error: `Error loading URLStore: extra data in buffer`. This is because gob is a stream based protocol that doesn't support restarting. We will remedy this situation here by using json as storage protocol (see § 12.9), which stores the data as plain text, so it can also be read by processes written in other languages than Go. This also shows how easy it is to change to a different persistent protocol, because the code dealing with the storage is cleanly isolated in 2 methods, namely `load and saveLoop`.

Start by creating a new empty file store.json, and change the line in main.go where the variable for the file is declared:

```
    var dataFile = flag.String("file", "store.json", "data store file name")
```

In store.go import json instead of gob. Then in saveLoop the only line that has to be changed is:

```
    e := gob.NewEncoder(f)
```

We change it in: `e := json.NewEncoder(f)`

Similarly in the load method the line `d := gob.NewDecoder(f)`
is changed to `d := json.NewDecoder(f).`

This is everything we need to change! Compile, start and test: you will see that the previous error does not occur anymore.

VERSION 5—DISTRIBUTING THE PROGRAM:

The code of the 5th version *goto_5* (discussed in §19.8 and § 19.9) can be found in the map code_examples\chapter_19\goto_v5. This version continues with the gob-storage, but could be easily adapted to use json as demonstrated in Version 4.

19.8 Multiprocessing on many machines

So far goto runs as a single process, but even with using goroutines a single process running on one machine can only serve so many concurrent requests. A URL Shortener typically serves many more Redirects (reads, using Get()) than it does Adds (writes, using Put()). Therefore we can create an arbitrary number of read-only slaves that serve and cache Get requests, and pass Puts through to the master, like in the following schema:

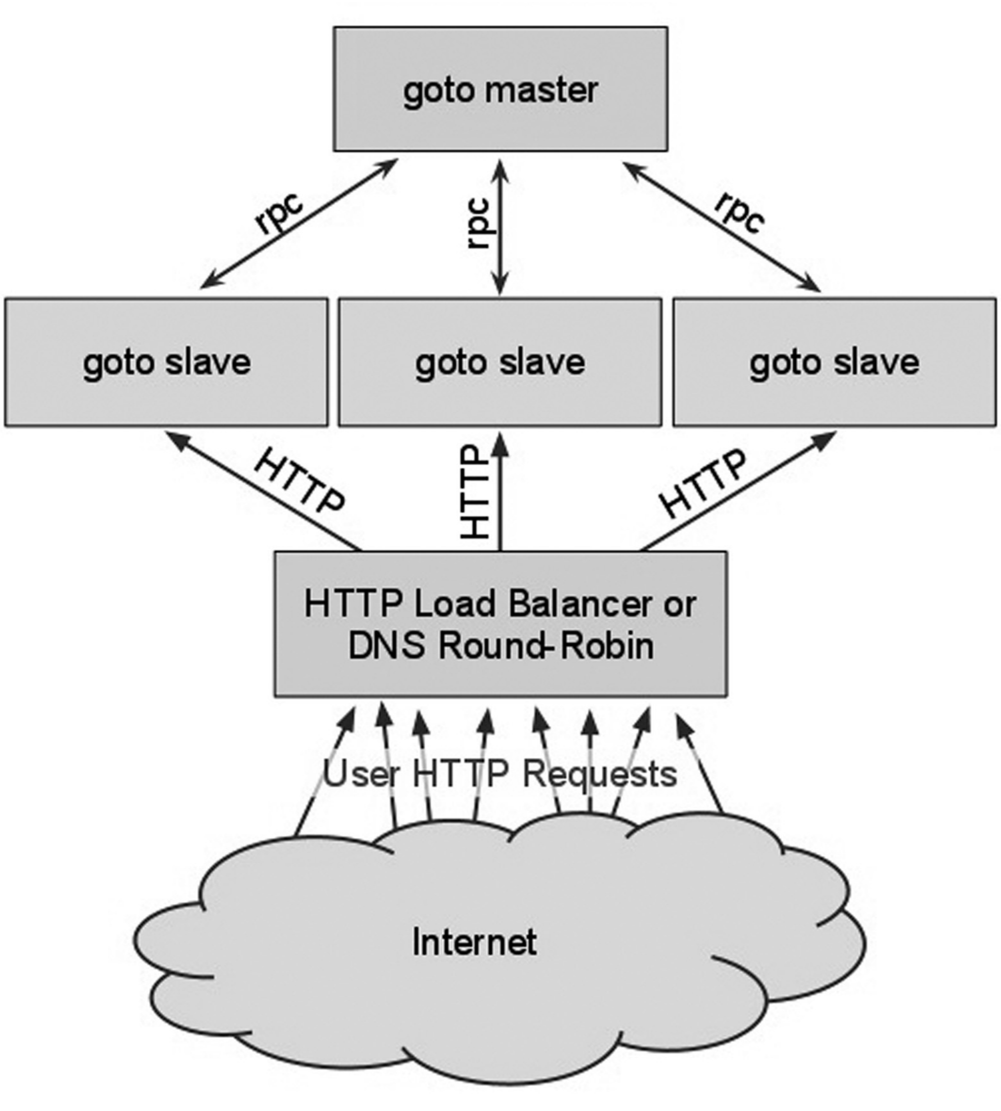


Fig 19.5: Distributing the work load over master- and slave computers

To run a master instance of the application goto on another computer in a network than the slave(s) process(es), they have to be able to communicate with each other. Go's rpc package provides a convenient means of making function calls over a network connection, making URLStore an RPC service (we discuss rpc in detail in §15.9). The slave process(es) will handle Get requests to deliver the long urls. When a new long url has to be transformed into a short one (using a Put() method) they delegate that task to the master process through an rpc-connection; so only the master will have to write to the datafile.

The basic Get() and Put() methods of URLStore until now have the signature:

```
func (s *URLStore) Get(key string) string
func (s *URLStore) Put(url string) string
```

RPC can only work through methods with the form (t is a value of type T):

```
func (t T) Name(args *ArgType, reply *ReplyType) error
```

To make URLStore an RPC service we need to alter the Put and Get methods so that they match this function signature. This is the result:

```
func (s *URLStore) Get(key, url *string) error
func (s *URLStore) Put(url, key *string) error
```

The code for Get() becomes:

```
func (s *URLStore) Get(key, url *string) error {
        s.mu.RLock()
        defer s.mu.RUnlock()
        if u, ok := s.urls[*key]; ok {
                *url = u
                return nil
        }
        return errors.New(“key not found”)
}
```

where now, because key and url are pointers, we must take their value by prefixing them with *, like in `*key` ; u is a value, we can assign it to pointer u with: `*url = u`

And the same goes for the code of Put():

```
func (s *URLStore) Put(url, key *string) error {
        for {
                *key = genKey(s.Count())
                if err := s.Set(key, url); err == nil {
                        break
                }
        }
        if s.save != nil {
                s.save <- record{*key, *url}
        }
        return nil
}
```

Because Put() calls Set(), the latter also has to be adapted to the fact that key and url are now pointers, and that it must return an error instead of a boolean:

```
func (s *URLStore) Set(key, url *string) error {
        s.mu.Lock()
        defer s.mu.Unlock()
        if _, present := s.urls[*key]; present {
                return errors.New("key already exists")
        }
        s.urls[*key] = *url
        return nil
}
```

For the same reason, when we call Set() from load(), the call must be adapted to:

```
s.Set(&r.Key, &r.URL)
```

We must also modify the HTTP handlers to accommodate the changes to URLStore. The Redirect handler now returns the error string provided by the URLStore:

```
func Redirect(w http.ResponseWriter, r *http.Request) {
        key := r.URL.Path[1:]
        var url string
        if err := store.Get(&key, &url); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
        http.Redirect(w, r, url, http.StatusFound)
}
```

The Add handler changes in much the same way:

```
func Add(w http.ResponseWriter, r *http.Request) {
        url := r.FormValue("url")
        if url == "" {
               fmt.Fprint(w, AddForm)
               return
        }
        var key string

if err := store.Put(&url, &key); err != nil {
               http.Error(w, err.Error(), http.StatusInternalServerError)
               return
        }
        fmt.Fprintf(w, "http://%s/%s", *hostname, key)
}
```

To make our application flexible, as we have done in the previous section, we can add a command-line flag to enable the RPC server in main():

```
var rpcEnabled = flag.Bool("rpc", false, "enable RPC server")
```

To make rpc work, we then have to register the URLStore with the rpc package and set up the RPC-over-HTTP handler with HandleHTTP, like this:

```
func main() {
        flag.Parse()
        store = NewURLStore(*dataFile)
        if *rpcEnabled { // flag has been set
               rpc.RegisterName("Store", store)
               rpc.HandleHTTP()
        }
        ... (set up http like before)
}
```

19.9 Using a ProxyStore

Now that we have the URLStore available as an RPC service, we can build another type that represents the rpc-client and that will forward requests to the RPC server; we call it ProxyStore:

```
type ProxyStore struct {
        client *rpc.Client
}
```

An rpc-client has to use the DialHTTP() method to connect to an rpc-server, so we incorporate that in the function `NewProxyStore` which makes our ProxyStore object:

```
func NewProxyStore(addr string) *ProxyStore {
        client, err := rpc.DialHTTP("tcp", addr)
        if err != nil {
               log.Println("Error constructing ProxyStore:", err)
        }
        return &ProxyStore{client: client}
}
```

This ProxyStore has Get and Put methods that pass the requests directly to the RPC server using the Call method of the rpc client:

```
func (s *ProxyStore) Get(key, url *string) error {
        return s.client.Call("Store.Get", key, url)
}

func (s *ProxyStore) Put(url, key *string) error {
        return s.client.Call("Store.Put", url, key)
}
```

A caching ProxyStore:

But if the slave(s) simply delegate all work to the master, there is no benefit! We intended the slaves to handle the Get requests. To be able to do that, they must have a copy (a cache) of the URLStore with the map. So we expand the definition of our ProxyStore to include URLStore:

```
type ProxyStore struct {
        urls   *URLStore
        client *rpc.Client
}
```

and NewProxyStore must be changed as well:

```
func NewProxyStore(addr string) *ProxyStore {
        client, err := rpc.DialHTTP("tcp", addr)
        if err != nil {
                log.Println("ProxyStore:", err)
        }
```

```
		return &ProxyStore{urls: NewURLStore(“”), client: client}
	}
```

And we must modify the URLStore so that it doesn’t try to write to or read from disk if an empty filename is given:

```
	func NewURLStore(filename string) *URLStore {
		s := &URLStore{urls: make(map[string]string)}
		if filename != “” {
			s.save = make(chan record, saveQueueLength)
			if err := s.load(filename); err != nil {
				log.Println(“Error loading URLStore: “, err)
			}
			go s.saveLoop(filename)
		}
		return s
	}
```

Our Get method needs to be expanded: it should first check if the key is in the cache. If present, Get returns the cached result. If not, it should make the RPC call, and update its local cache with the result:

```
	func (s *ProxyStore) Get(key, url *string) error {
		if err := s.urls.Get(key, url); err == nil { // url found in local map
			return nil
		}
		// url not found in local map, make rpc-call:
		if err := s.client.Call(“Store.Get”, key, url); err != nil {
			return err
		}
		s.urls.Set(key, url)
		return nil
	}
```

Similarly the Put method need only update the local cache when it performs a successful RPC-Put:

```
		func (s *ProxyStore) Put(url, key *string) error {
			if err := s.client.Call(“Store.Put”, url, key); err != nil {
				return err
			}
			s.urls.Set(key, url)
			return nil
		}
```

To summarize: the slave(s) use the ProxyStore, only the master uses the URLStore. But the way we made them, they look very much alike: they both implement Get and Put methods with the same signature, so we can specify an interface Store to generalize their behavior:

```
type Store interface {
        Put(url, key *string) error
        Get(key, url *string) error
}
```

Now our global variable store can be of type Store: `var store Store`

Finally we adapt our main() function so that either a slave- or a master process is started up (and we can only do that because store is now of the interface type Store!).

To that end we add a new command line `flag masterAddr` with no default value.

```
var masterAddr = flag.String("master", "", "RPC master address")
```

If a master address is given, we start a slave process and make a new ProxyStore; otherwise we start a master process and make a new URLStore:

```
func main() {
        flag.Parse()
        if *masterAddr != "" { // we are a slave
              store = NewProxyStore(*masterAddr)
        } else {      // we are the master
              store = NewURLStore(*dataFile)
        }
        ...
}
```

This way we have enabled ProxyStore to use the web front-end, in the place of URLStore.

The rest of the front-end code continues to work as before. It does not need to be aware of the Store interface. Only the master process will write to the datafile.

Now we can launch a master and several slaves, and stress-test the slaves.

Compile this 4th version or use the executable that is present.

To test it first start the master on a command-line with:

```
./goto -http=:8081 -rpc=true   (or goto replacing ./goto on Windows)
```

providing 2 flags: the master listens on port 8081 and rpc is enabled.

A slave is started up with: `./goto -master=127.0.0.1:8081`

It gets its master-address and will receive client requests on port 8080.

Included in the code map is the following shell script demo.sh to do an automated start up in Unix like systems:

```
#!/bin/sh
gomake
./goto -http=:8081 -rpc=true &
master_pid=$!
sleep 1
./goto -master=127.0.0.1:8081 &
slave_pid=$!
echo "Running master on :8081, slave on :8080."
echo "Visit: http://localhost:8080/add"
echo "Press enter to shut down"
read
kill $master_pid
kill $slave_pid
```

To test under Windows, start a MINGW shell and start the master, then per slave start a new MINGW shell and start the slave process.

19.10 Summary and enhancements

By gradually building up our goto application we encountered practically all Go's important features.

While this program does what we set out to do, there are a few ways it could be improved:

- *Aesthetics*: the user interface could be (much) prettier. For this you would use Go's template package (see §15.7).

- *Reliability*: the master/slave RPC connections could be more reliable: if the client-server connection goes down, the client should attempt to re-dial. A "dialer" goroutine could manage this.
- *Resource exhaustion*: as the size of the URL database grows, memory usage might become an issue. This could be resolved by dividing (sharding) the master servers by key.
- *Deletion*: to support deletion of shortened URLs, the interactions between master and slave would need to be made more complex.

Chapter 20—Go in Google App Engine

20.1 What is Google App Engine ?

Google App Engine (from now on shortened to GAE) is the Google way of *cloud computing*: executing your web applications and storing your data on the vast Google infrastructure, without having to worry about the servers, the network, the operating system, the data store, and so on. This collection of resources is commonly referred to as the cloud, and its maintenance is the sole responsibility of Google itself. For you as developer only your application counts and the *services* it can deliver to its users, who can work with and run your application on any device which can connect to the internet. You only pay for the resources (CPU processing time, network bandwidth, disk storage, memory, etc.) your software really needs. When there are peak moments the cloud platform automatically increases resources for your application, and decreases them when they are no longer needed: *scalability* is one of the biggest advantages of cloud computing. Collaborative applications (where groups of people work together on, share data, communicate, etc.), applications that deliver services and applications that perform large computations are excellent candidates for cloud computing. The typical user interface for a cloud application is a *browser* environment.

GAE was launched with support for Python apps in 2008 and added Java support in 2009; since 2011 there is also Go support. The start page is: http://code.google.com/appengine/

> Google's App Engine provides a reliable, scalable and easy way to build and deploy applications for the web. Over a hundred thousand apps are hosted at appspot.com and custom domains using the App Engine infrastructure. It is a "platform-as-as-service" environment that operates on a higher level than an "infrastructure cloud" like Amazon EC2, attempting to share resources with even greater efficiency.

The Sandbox:

Your applications run in a secure environment called the *Sandbox*, that provides limited access to the underlying operating system. These limitations allow App Engine to distribute web requests for the application across multiple servers, and start and stop servers to meet traffic demands. The sandbox isolates your application in its own secure, reliable environment that is independent of the hardware, operating system and physical location of the web server. Some of the limitations are:

- The application cannot write to the file system of the server; only files uploaded within the application can be read. It must use the *App Engine datastore*, *memcache* or other services for all data that persists between requests.
- Code runs only in response to a web request, a queued task, or a scheduled task, and the response must be within 60 seconds; a request handler cannot spawn a sub-process or execute code after the response has been sent.
- It can only access other computers on the Internet through the provided URL fetch and email services. Other computers can only connect to the application by making HTTP (or HTTPS) requests on the standard ports.

<u>An overview of its services:</u>

1) <u>Data</u> is stored in the GAE *Datastore* based on Google's Bigtable: this is a distributed data storage service that features a query engine and transactions; it grows automatically with your data. It is not a traditional relational database, so the classic SQL and joins are not allowed; but it provides you with a SQL-like query language, called *GQL*. Data objects, called *entities*, have a *kind* and a set of properties. Queries can retrieve entities of a given kind filtered and sorted by the values of the properties. Property values can be of any of the supported property value types. Entities can be grouped, and this is the unit on which transactions work: a transaction must be within a group. There is no database schema for your entities: any structure between the entities must be provided and enforced by your application code. Updates use *optimistic concurrency control*, meaning the last update changes the data.
2) <u>User authentication</u> of the app can integrate with Google Accounts
3) <u>URL Fetch</u>: through this service your app can access resources on the Internet, such as web services or other data.
4) <u>Mail</u>: is also a built-in service for use in apps.
5) <u>Memcache</u>: a high performance in-memory key-value cache; it is useful for data that does not need the persistence and transactional features of the datastore, such as temporary data or data copied from the datastore to the cache for high speed access.
6) <u>Image Manipulation</u>:
7) <u>Scheduled tasks and task queues</u> (cron jobs): an appl can perform tasks besides responding to web requests; it can perform these tasks on a schedule that you configure, such as on a daily or hourly basis. Alternatively the application can perform tasks added to a queue by the application itself, such as a background task created while handling a request.

20.2 Go in the cloud

Go support for GAE was first announced on May 10 2011 at the Google I/O conference. First experimental and for registered testers only, it was completely opened for every developer on July 21 2011. At the time of writing (Jan 2012) the current **Go App Engine SDK** is **1.6.1** (released 2011-12-13); it exists only for Linux and Mac OS X (10.5 or greater), both 32 and 64 bit. The supporting Go tool chain is release r60.3; some changes are backwards incompatible, the SDK api_version is 3.

When Go app runs on App Engine it is compiled with the 64-bit x86 compiler (6g). Only one thread is run in a given instance. That is, all goroutines run in a single operating system thread, so there is no CPU parallelism available for a given client request.

Go is the first compiled language to run on AppEngine. It shines because it is positioned so well compared to the other two runtimes:

- against Java: Go has much better instance start-up time and more concurrency possibilities.
- against Python: Go has a much more better speed of execution.

20.3 Installation of the Go App Engine SDK: the development environment for Go

20.3.1. Installation

Download the GAE SDK zip file appropriate to your system from the Google App Engine downloads page: http://code.google.com/appengine/downloads.html

e.g. if your OS is a 64 bit Linux Ubuntu 11.10 then download `go_appengine_sdk_linux_amd64-1.6.1.zip.`

Open with Archive Manager and extract this file to a directory of your choice (e.g. your home directory): it will make 1 big folder called `google_appengine` which contains the entire AppEngine for Go development environment, e.g. under `/home/user/google_appengine` or in general `“install root”/google_appengine/goroot`.

This environment contains everything you need to develop, build and test your Apps locally: it includes an *AppEngine server* to test your applications with, a DataStore where you can store data similar to how you would eventually do with a live app hosted on the AppEngine servers in the cloud, and other API support and tools that allow you to mimic the real AppEngine for purposes of development and testing. Since this AppEngine environment is for Go, it also contains the appropriate Go compilers, packages, and tools as part of the download.

Differences between GAE-Go and regular Go:

The GAE-Go runtime provides the full Go language and almost all the standard libraries, except for a few things that don't make sense in the App Engine environment:

- there is no unsafe package and the syscall package is trimmed.
- it does not support cgo (no interaction with C libraries), even more so: you cannot use any binary libraries (Go or otherwise) in a GAE project. You need source for everything since GAE compiles/links it all.
- go install is not supported
- GAE often lags the main distribution by one or more major versions

Furthermore the limitations of the Sandbox environment (§ 20.1) have to be taken into account. So for example attempts to open a socket or write to a file will return an os.EINVAL error.

Therefore treat your GAE and non-GAE Go tools as completely separate; if you're only doing GAE development, you could do without the standard tools altogether.

Under `google_appengine` reside a few Python scripts, the basic workhorses of Google App Engine. Make sure that they are executable (if not issue the chmod +x *.py command in the map). Also add their path to the PATH variable in order that you don't have to include the complete path when invoking them: e.g. if you have a bash shell, do this by adding the line: `export PATH=/home/user/google_appengine:$PATH to your .bashrc or .profile file.`

Remarks:

1) If you already have a working Go environment (as you would when working through this book), this AppEngine installation is outside and parallel to it, not affecting it; in particular you don't have to change your Go environment variables in your OS. Go on AppEngine has its own completely separate environment containing its own Go version, which it picks up from `"install root"/google_appengine/goroot`
2) It is also a good idea to download the documentation, so you can browse that when offline: download `google-appengine-docs-20111011.zip` and extract.
3) GAE heavily uses Python, by default installed on Mac OS X and Linux; if for some reason this is not the case, download and install Python 2.5 from www.python.org

4) Source code: the libraries and SDK are open source, hosted at http://code.google.com/p/appengine-go/.
   Download it with: `hg clone https://code.google.com/p/appengine-go/`
5) All Go packages for a given app are built into a single executable, and request dispatch is handled by the Go program itself; this is unlike what happens in the Java and Python SDK's.

In § 20.8 we will see how to connect to the GAE cloud to deploy your application. But before you are ready to do that, you will develop, test and run your app in the local GAE environment which you just installed, which is as good a simulation of the production environment as it gets.

20.3.2. Checking and testing

Check the installation:

To control that everything works fine go in a console to the map `google_appengine` and invoke the local AppEngine server by invoking `dev_appserver.py`

If you see text starting with:

```
Invalid arguments
Runs a development application server for an application.
dev_appserver.py [options]

Application root must be …
```

everything is fine.

Running a demo app:

There are a few demo apps along with the SDK bundle. Let's check one to ensure things are going good so far.

- Go to the map `google_appengine/demos`: there you see a few folders, e.g. helloworld, guestbook, etc.
- From within the demos directory, execute the command: `dev_appserver.py helloworld`

Note that this automatically compiles, links and runs the Go program.

- There are a few WARNINGs and INFOs, but if the last line is: Running `application helloworld on port 8080: http://localhost:8080`, we are good. At this point the helloworld application has been instantiated within the local AppEngine server and is ready to serve the user on your machine at the port 8080.
- Open a browser and navigate to `http://localhost:8080`

If you see a webpage with
```
Hello, World! 세상아 안녕!!
```
You are successfully running a Go web application executing on the local AppEngine.

Which Go code have you just run ? This is its source code:

<u>`Listing 20.1 helloworld.go:`</u>

```
package helloworld
import (
        "fmt"
        "net/http"
)

func init() {
        http.HandleFunc("/", handle)
}

func handle(w http.ResponseWriter, r *http.Request) {
        // some Chinese characters after World!
        fmt.Fprint(w, "<html><body>Hello, World! 세상아 안녕!! </body></html>")
}
```

It is a simple web application (see chapter 15), that starts the overall handler in the init() function. Also note that it is contained in its own package.

20.4 Building your own Hello world app

Let us now build the same application as the demo we saw in § 20.3, but this time exploring a bit deeper.

20.4.1 Map structure—Creating a simple http-handler

Create a directory and give it a name characteristic for your app, like `helloapp`. All files for this application reside in this directory, and inside it create another directory named `hello`. This will contain the Go source files for our hello package. Then inside the hello directory, create a file named `helloworld2.go`, and give it the following contents (in fact almost the same as the demo app):

Listing 20.2 helloworld2_version1.go:

```
package hello

import (
    “fmt”
    “net/http”
)

func init() {
    http.HandleFunc(“/”, handler)
}

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, “Hello, world!”)
}
```

Mind the package name: when writing a stand-alone Go program we would place this code in package main, but the Go GAE Runtime provides the main package and HTTP Listener, so you should put your code in a package of your choice, in this case, hello. Secondly, since Go App Engine applications communicate with the outside world via a web server writing them is very much like writing a stand-alone Go web application (see chapter 15). So we import the http package and define handler-functions for the different url-patterns used in our app. We have no main() function, so the handler's setup has to move to the init() function! Also the starting of the webserver itself is done by GAE for us. Our Go package hello responds to any request by sending a response containing the message “Hello, world!”

20.4.2 Creating the configuration file app.yaml

All GAE apps need a yaml configuration file `app.yaml`, it contains app metadata for GAE (yaml is a file format for text-files often used in open source projects, for more info see www.yaml.org). Among other things, this file tells the App Engine service which runtime to use and which URLs should be handled by our Go program. You can take a copy of the app.yaml file from the demo app, placing it inside the map helloapp, and removing the handler for favicon.ico.

So the applications map/file-structure should be:

```
helloapp\           // map of the GAE application
app.yaml            // configuration file
hello\              // map containing the source files
helloworld2.go
```

Only app.yaml is a required name, the names of the maps, Go files and packages can be chosen differently, but by convention their names are the same or analogous, and the root-map has suffix `app`.

`app.yaml` is read and interpreted by the AppEngine that hosts and executes your programs when at several times:

- when you upload your application to the AppEngine for it to be hosted;
- when it is executed;
- when users access it;

It can contain comments preceded by a #, and contains the following statements:

```
application: helloworld
version: 1
runtime: go
api_version: 3

# routing-table: routing of different urls to different types of handlers
handlers:
- url: /.*
  script: _go_app
```

The *application*: value helloworld in app.yaml is your *application identifier.* This value can be anything during development; later when registering your application with App Engine, you will select a unique identifier (unique among all GAE applications!) and update this value.

version indicates which version of your app is running: indeed GAE can run several versions of your app in parallel, but one of them must be designated as the default. It can contain alphanumeric characters, and hyphens. So you can have running a testversion like T2-31 and a production version P2-1.

runtime is the language in which the app is written (other allowed values are Java and Python). If you adjust this before uploading new versions of your application software, App Engine will retain previous versions, and let you roll back to a previous version using the administrative console.

api_version is the version of the Go API's used in this SDK; they are probably incompatible with previous versions. You could have build earlier versions of your app in a previous api_version SDK; if GAE still permits they can continue to run, but usually there is a time limit for this, and you should update your app to the new api version: the gofix tool in the bin map will probably be able to do most of the updating required.

The *handlers* section is the *routing table*: it tells GAE how to map requests that are sent to the server on to the code. For every incoming request the *url* pattern (the part that comes after http://localhost:8080/ when developing locally or after http://appname.appspot.com/ when running in the cloud) is matched with the regular expressions after url:

For the first url-pattern that matches, the corresponding *script* is executed. In our case every request to a URL whose path matches the regular expression /.* (that is: all URLs) should be handled by the Go program. The `_go_app` value is a magic string recognized by the dev_appserver.py; it is ignored by the production App Engine servers.

If you look at the app.yaml file for the demo helloworld application, you will see that it contains an initial section in handlers:

```
handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: /.*
  script: _go_app
```

Some files (*static_files*) like images do not change (in this case the image favicon.ico). These files could be put in a sort of common cache across different AppEngine servers, allowing them to be served faster to the user. If your application has a number of them, put them in a separate directory which is by convention named static.

upload indicates what must be uploaded to the cloud when you deploy your app; for example if it contained images/(*.ico|*.gif|*.jpg), it will upload all these types of files within the local images directory onto the AppEngine server.

Most GAE applications as we will see also use template files, these can be stored in the root app map, or in a special directory `tmpl`.

So a general structure for a GAE application could be:

```
      yourapp\                      // map of the GAE application
                      app.yaml             // configuration file
                      yourpackage\         // map containing the source files
                            package1.go
                            …
                     tmpl\                 // map containing template files
                            root.html
                            update.html
                            …
                     static\       // map containing static files
                            yourapp.ico
                            …
```

As with the demo go in a console window to the map containing the map helloapp, and issue the command: `dev_appserver.py helloapp`

Alternatively you could get a console window in any map and invoking:

```
      dev_appserver.py /path_to_map_helloapp/helloapp
```

In both cases the web server is now running, listening for requests on port 8080. Test the application by visiting the following URL in your web browser: http://localhost:8080/

And you should see: `Hello, world!`

In the server console the following text appears:

```
$ dev_appserver.py helloapp
INFO     2011-10-31 08:54:29,021 appengine_rpc.py:159] Server: appengine.google.com
INFO     2011-10-31 08:54:29,025 appcfg.py:463] Checking for updates to the SDK.
INFO     2011-10-31 08:54:29,316 appcfg.py:481] The SDK is up to date.
WARNING  2011-10-31 08:54:29,316 datastore_file_stub.py:512] Could not read datastore
data from /tmp/dev_appserver.datastore
INFO     2011-10-31 08:54:29,317 rdbms_sqlite.py:58] Connecting to SQLite database ‘’
with file ‘/tmp/dev_appserver.rdbms’
INFO     2011-10-31 08:54:29,638 dev_appserver_multiprocess.py:637] Running application
helloworld on port 8080: http://localhost:8080
              <-(A)
INFO     2011-10-31 08:56:13,148 __init__.py:365] building _go_app
              <-(B)
```

```
INFO     2011-10-31 08:56:15,073 __init__.py:351] running _go_app
INFO     2011-10-31 08:56:15,188 dev_appserver.py:4143] "GET / HTTP/1.1" 200 -
               <-(C)
```

At <-(A) the server is ready, at <-(B) the server compiles and runs the Go program, at <-(C) the request for our app came in and the HTML output page is served.

When the server is terminated or not yet started and a client requests the url http://localhost:8080/ the browser prints a message like this in FireFox: `Unable to connect Firefox can't establish a connection to the server at localhost:8080.`

20.4.3 Iterative development

The development app server watches for changes in your file: as you update your source (edit + save!), it recompiles them and relaunches your local app; there is no need to restart dev_appserver.py!

Try it now: leave the web server running, then edit helloworld2.go to change "Hello, world!" to something else. Reload http://localhost:8080/ to see the change: this works just as dynamically as writing a Rails- or Django-application!

To shut down the web server, make sure the terminal window is active, and then press Control-C (or the appropriate "break" key for your console).

```
INFO   2011-10-31 08:56:21,420 dev_appserver.py:4143] "GET / HTTP/1.1" 200 -
INFO   2011-10-31 08:57:59,836 __init__.py:365] building _go_app <-(D)
INFO   2011-10-31 08:58:00,365 __init__.py:351] running _go_app
INFO   2011-10-31 08:58:00,480 dev_appserver.py:4143] "GET / HTTP/1.1" 200 -
^CINFO 2011-10-31 08:58:32,769 dev_appserver_main.py:665] Server interrupted by user,
terminating                                                <-(E)
```

This can be seen in the listing above of server console output which comes after the first listing: at (D) the appserver sees that the Go source has been changed and recompiles; at (E) the server is terminated.

20.4.4. Integrating with the GoClipse IDE

a) Window / Preferences / Go:

point everything to the Go root of GAE

b) Run / External Tools / External Tools Configuration / select Program
 Make New Configuration: click New Button,
  Name: GAE
  Location: `/home/user/google_appengine/dev_appserver.py`
  Working Directory: `/home/user/workspace/bedilly/src/pkg/helloapp`
  Arguments: `home/user/workspace/bedilly/src/pkg/helloapp`
 Apply / Run

Deploying your app is also easy by configuring an external tool: http://code.google.com/p/goclipse/wiki/DeployingToGoogleAppEngineFromEclipse

20.5 Using the Users service and exploring its API

GAE provides several useful services based on Google infrastructure. As mentioned in § 20.1, GAE offers a Users service, which lets your application integrate with Google user accounts. With the Users service, your users can employ the Google accounts they already have to sign in to your application. The Users service makes it easy to personalize this application's greeting.

Edit the file `helloworld2.go` and replace it with the following Go code:

Listing 20.3 helloworld2_version2.go:
```
package hello

import (
    "appengine"
    "appengine/user"
    "fmt"
    "net/http"
)
func init() {
    http.HandleFunc("/", handler)
}

func handler(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    u := user.Current(c)
    if u == nil {
        url, err := user.LoginURL(c, r.URL.String())
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        w.Header().Set("Location", url)
        w.WriteHeader(http.StatusFound)
        return
    }
    fmt.Fprintf(w, "Hello, %v!", u)
}
```

Test it out by reloading the page in your browser. Your application presents you with a link that, when followed, will redirect you to the local version of the Google sign-in page suitable for testing your application. You can enter any username you like in this screen, and your application will see a fake user.User value based on that username. When your application is running on App Engine, users will be directed to the Google Accounts sign-in page, and then redirected back to your application after successfully signing in or creating an account.

The Users API:

In order to access this we need to import some Go packages specifically targeted at GAE, namely the general `appengine and appengine/user.`

In the handler we first need to make a Context object associated with the current request r. This is done in the line: `c := appengine.NewContext(r)`

The appengine.NewContext function returns an appengine.Context value named c here: this is a value used by many functions in the Go App Engine SDK to communicate with the App Engine service. Then from this context we test whether there is already a user logged in at this point with: `u := user.Current(c)`

If this is the case user.Current returns a pointer to a **user.User** value for the user; otherwise it returns nil. If the user has not yet signed in, that is when u == nil, redirect the user's browser to the Google account sign-in screen by calling:

```
url, err := user.LoginURL(c, r.URL.String())
```

The 2nd parameter `r.URL.String()` is the currently requested url so that the Google account sign-in mechanism can perform a *redirection* after successful login: it will send the user back here after signing in or registering for a new account. The sending of the login screen is finished by setting a Location header and returning an HTTP status code of 302 "Found".

The LoginURL() function returns an error value as its 2nd argument. Though an error is unlikely to occur here, it is good practice to check it and display an error to the user, if appropriate (in this case, with the http.Error helper):

```
if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
}
```

When the user has signed in, we display a personalized message using the name associated with the user's account: `fmt.Fprintf(w, "Hello, %v!", u)`

In this case, the fmt.Fprintf function calls *user.User's String method to get the user's name in string form. More information can be found at this reference: http://code.google.com/appengine/docs/go/users/

20.6 Handling forms

As we saw in chapter 15 § 15.6/7 the template package is frequently used in web-applications, so also in GAE apps. The following app lets a user input a text. First a guestbook form is shown (via the / root handler), and when this is posted the sign handler substitutes this text in the resulting html response. The sign function gets the form data by calling `r.FormValue` and passes it to `signTemplate.Execute` that writes the rendered template to the `http.ResponseWriter`.

Edit the file `helloworld2.go`, replace it with the following Go code and try it out:

Listing 20.4 helloworld2_version3.go:

```
package hello
import (
    "fmt"
    "net/http"
    "template"
)

const guestbookForm = `
<html>
  <body>
    <form action="/sign" method="post">
      <div><textarea name="content" rows="3" cols="60"></textarea></div>
      <div><input type="submit" value="Sign Guestbook"></div>
    </form>
  </body>
</html>
`
const signTemplateHTML = `
<html>
  <body>
    <p>You wrote:</p>
    <pre>{{html .}}</pre>
  </body>
</html>
`
var signTemplate = template.Must(template.New("sign").Parse(signTemplateHTML))

func init() {
    http.HandleFunc("/", root)
    http.HandleFunc("/sign", sign)
}

func root(w http.ResponseWriter, r *http.Request) {
    // w.Header().Set("Content-Type", "text/html")
    fmt.Fprint(w, guestbookForm)
}

func sign(w http.ResponseWriter, r *http.Request) {
    // w.Header().Set("Content-Type", "text/html")
    err := signTemplate.Execute(w, r.FormValue("content"))
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

20.7 Using the datastore

We now have a way with html-forms to collect information from the user. Often we want to make this information persistent: we need a place to put it and a way to get it back. GAE here offers us its DataStore facility: a non-relational database which persists your data across webservers and even across machines. Indeed the next request of a user could very well go to a different webserver running on a different computer, but GAE's infrastructure takes care of all of the distribution, replication and load balancing of data behind a simple API—and you get a powerful query engine as well.

We will now expand our example a bit and making a Greeting struct, that can contain the author, the content and the time of the greeting, and we want to store them. This is the 1st thing you have to do: make a suitable data structure for your program *entity* (that is the kind of object your program handles), most of the time this will be a struct. The in-memory values of this struct in the running program will contain the data from the DataStore of that entity.

The following version of our program:

(A) url: / : retrieves all stored greetings and shows them through the template package (see § 15.7)
(B) url: /sign: stores a new greeting in the DataStore

We now also need to import the `appengine/datastore` package.

Listing 20.5 helloworld2_version4.go:

```
package hello

import (
        “appengine”
        “appengine/datastore”
        “appengine/user”
        “net/http”
        “template”
        “time”
)

const guestbookTemplateHTML = `
<html>
  <body>
    {{range .}}
      {{with .Author}}
        <p><b>{{html .}}</b> wrote:</p>
      {{else}}
        <p>An anonymous person wrote:</p>
      {{end}}
      <pre>{{html .Content}}</pre>
    {{end}}
    <form action="/sign" method="post">
      <div><textarea name="content" rows="3" cols="60"></textarea></div>
      <div><input type="submit" value="Sign Guestbook"></div>
    </form>
  </body>
</html>
`
var guestbookTemplate =
template.Must(template.New("book").Parse(guestbookTemplateHTML))

type Greeting struct {
         Author  string
         Content string
         Date    datastore.Time
}

func init() {
         http.HandleFunc("/", root)
         http.HandleFunc("/sign", sign)
}

func root(w http.ResponseWriter, r *http.Request) {
         // w.Header().Set("Content-Type", "text/html")
         c := appengine.NewContext(r)
         q := datastore.NewQuery("Greeting").Order("-Date").Limit(10)
         greetings := make([]Greeting, 0, 10)
         if _, err := q.GetAll(c, &greetings); err != nil {
             http.Error(w, err.Error(), http.StatusInternalServerError)
             return
         }
         if err := guestbookTemplate.Execute(w, greetings); err != nil {
             http.Error(w, err.Error(), http.StatusInternalServerError)
         }
}

func sign(w http.ResponseWriter, r *http.Request) {
         // w.Header().Set("Content-Type", "text/html")
         c := appengine.NewContext(r)
         g := Greeting{
                Content: r.FormValue("content"),
                 Date:    datastore.SecondsToTime(time.Seconds()),
         }
         if u := user.Current(c); u != nil {
                g.Author = u.String()
         }
         _, err := datastore.Put(c, datastore.NewIncompleteKey(c, "Greeting",
         nil), &g)
         if err != nil {
             http.Error(w, err.Error(), http.StatusInternalServerError)
             return
         }
         http.Redirect(w, r, "/", http.StatusFound)
}
```

The sign-handler (B) constructs a Greeting value g from the form- and context data, and then stores this with `datastore.Put()`. DataStore internally generates its own unique keys for data records; in order to do this we call the Put() function with `datastore.NewIncompleteKey(c, "Greeting", nil)` as 2nd argument (this function needs the name of the entity `Greeting`). The 3rd argument to Put() `&g` is the struct which contains the values.

The `datastore` package provides a Query type for querying the datastore and iterating over the results. The root handler does just that by constructing a query q that requests Greeting objects from DataStore in Date-descending order, with a limit of 10:

```
q := datastore.NewQuery("Greeting").Order("-Date").Limit(10)
```

We need a data structure to store the results of our query, this is `greetings`, a slice of `Greeting` values. The call to `q.GetAll(c, &greetings)` retrieves the data and stores them in our slice; of course we check for a possible error which could have occurred in the retrieval.

When everything is ok we show the data by merging with our template:

```
guestbookTemplate.Execute(w, greetings)
```

which is performed by a range-construct (see § 15.7.6).

Again test it out by editing the file `helloworld2.go`, replace it with the code from listing 20.5; closing your browser session in between greetings, so that you can check that they are persisted.

Clearing the Development Server Datastore:

The development web server uses a local version of the datastore for testing your application, using temporary files. The data persists as long as the temporary files exist, and the web server does not reset these files unless you ask it to do so. If you want the development server to erase its datastore prior to starting up, use the `--clear_datastore` option when starting the server: `dev_appserver.py --clear_datastore helloapp/`

Debugging:

The gdb debugger works with Go (see http://golang.org/doc/debugging_with_gdb.html), and you can attach gdb to an existing process. Thus: start the dev_appserver.py as usual, and browse to localhost:8080 to start your Go app. Then do: `$ ps ax | grep _go_app` to find the PID and path of the _go_app. If you attach gdb to that, then the next HTTP request you make to the dev appserver should hit any breakpoints you can set in your code. Remember that if you modify your Go source, then the dev appserver will recompile and exec a different _go_app.

20.8 Uploading to the cloud

Our guest book application authenticates users using Google accounts, lets them submit messages, and displays messages other users have left, let us call it base feature complete: we will now deploy it in the cloud. If our application would get immensely popular, we don't need to change anything because GAE handles scaling automatically.

But first you need to have a Google account, like a gmail-address; you can make one quickly at www.google.com/accounts

Creating and managing App Engine web applications takes place from the *App Engine Administration Console* at this URL: https://appengine.google.com/

After a quick SMS verification procedure you get the "Create an Application" page. Choose an *application identifier* (which is unique for all GAE applications) like `ib-tutgae.appspot.com`; with prefix http:// this becomes the url of your application. This identifier cannot be changed afterwards, prefixing it with your initials for private apps or your company name for commercial apps is a good idea. Then choose an *application title,* this is visible in your app and can be changed afterwards, e.g. "Tutorial GAE App". Leave the defaults Google Authentication and High Replication Datastore as they are. Below certain quotas GAE runs your applications at no cost! After clicking the button Create Application, a screen will appear with the message "Application Registered Successfully".

To upload your app in the cloud, do the following:

1) Edit the `app.yaml` file changing the value of the application: setting from `helloworld` to your registered application `ib-tutgae`
2) To upload and configure your application in GAE use the script `appcfg.py` by issuing the command: `appcfg.py update helloapp/`

Verification is done by asking your Google account data and if everything succeeds, your application is now deployed on App Engine!

Step 2) has to be performed every time you upload a new version of your application.

If you see compilation errors, fix the source and rerun appcfg.py; it won't launch (or update) your app until compilation is successful.

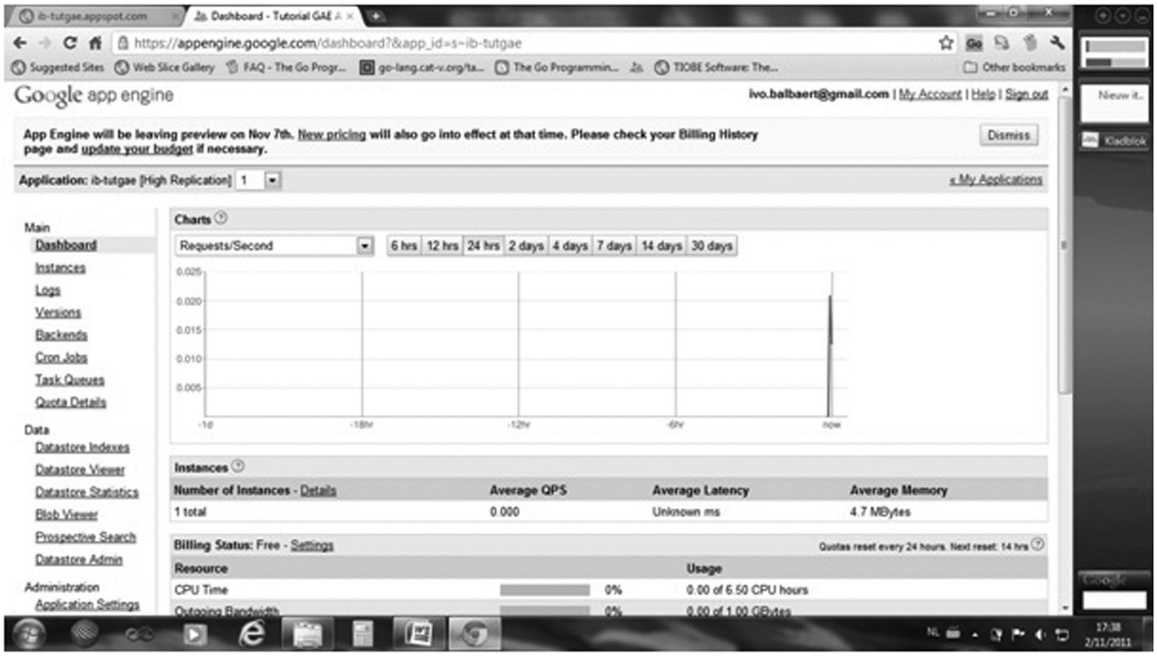
Test it out in the cloud with: `http://`*`application-id`*`.appspot.com`

using your own unique `application-id`, in our case http://ib-tutgae.appspot.com

This also works in a browser on a Windows platform, not only on Linux or OS X.

Monitoring your application:

Visiting https://appengine.google.com/ again will now show a list of your applications. Clicking the link of your application shows its *Control Panel*, which serves to monitor your application:



App Engine will be leaving preview on Nov 7th. New pricing will also go into effect at that time. Please check your Billing History page and update your budget if necessary.

Application: ib-tutgae [High Replication] 1

Instances

Number of Instances - Details	Average QPS	Average Latency	Average Memory
1 total	0.000	Unknown ms	4.7 MBytes

Billing Status: Free - Settings — Quotas reset every 24 hours. Next reset: 14 hrs

Resource		Usage
CPU Time	0%	0.00 of 6.50 CPU hours
Outgoing Bandwidth	0%	0.00 of 1.00 GBytes

Fig 20.1: The Application Control Panel

This is very important because your application runs in the cloud and this is your only means of accessing it (apart from uploading a new version with app_cfg)! You cannot profile or debug your code yourself when it is running in the cloud. There is a graphical representation of the load of your application (amount of requests per sec), how much resources it has consumed (cpu usage, bandwidth, storage, replicated data, backend usage) and how this is billed. There is also a load view: per uri pattern what was the number of requests and the cpu load, and very important: an Errors view: summary information about the errors that occurred in your application. The Data panel, and more specifically the Datastore Viewer, lets you visualize and query your stored data. In addition there are specific views for Administration and links to the GAE documentation. Main / Logs gives you access to the application log where every request and error/exception is logged (exceptions are not shown to the users)

Chapter 21—Real World Uses of Go

In the following sections we discuss some real use cases of Go: we explore a number of Go applications used in businesses today, and point out why Go was chosen in these domains. This is quite impressive taking into account the language is only been public since 2 years and that established businesses building initial projects in a new language often don't want this to be public knowledge.

21.1 Heroku—a highly available consistent data store in Go.

http://www.heroku.com/ (ref. 39)

Heroku is a Silicon Valley company based in San Francisco U.S. and recently purchased by Salesforce.com which provides powerful, scalable and above all very manageable cloud hosting for Ruby and Rails, Java, Clojure and node.js applications. Two engineers from Heroku, Keith Rarick and Blake Mizerany, designed an open source "distributed init system" called **Doozer** for managing processes across clusters of machines, and recovering gracefully from instance failures and network partitions. One of the requirements was that they needed to reliably synchronize and share information among many servers.

Every server in the system needs to have a lot of information (configuration data, locks, etc.) about the system as a whole in order to be able to coordinate, and that information needs to be consistent and available even during data store failures, so they needed a data store with solid consistency guarantees. For that purpose they developed Doozer, a new, consistent, highly-available data store written in Go, modeled after Google's (closed source) Chubby program for managing their back-end infrastructure.

Doozer is based upon Paxos, a family of protocols for solving consensus in an unreliable network of unreliable nodes. While Paxos is essential to running a fault-tolerant system, it is notorious for being difficult to implement. Even example implementations that can be found online are complex and hard to follow, despite being simplified for educational purposes. Existing production systems have a reputation for being worse.

Doozer is developed as a rock-solid basis for building distributed systems:

- A highly available (works during network partitions),
- Consistent (no inconsistent writes),
- Data store (for small amounts of data).

As the developers say: "Doozer is where you put the family jewels."

It provides a single fundamental synchronization primitive: compare-and-set.

Example use cases:

- Database master election
- Name service
- Configuration

Why Go was chosen and how did Go's characteristics make it a successful product:

Paxos is defined in terms of independent, concurrent processes that communicate via passing messages. This is the kind of problem where *Go's concurrency primitives* (goroutines and channels, see chapter 14) excel. In Doozer, these processes are implemented as *goroutines*, and their communications as *channel operations*. In the same way that Go's *garbage collector* keeps memory usage to a minimum, the developers of Doozer found that goroutines and channels improve upon the lock-based approach to concurrency. These tools let them avoid complex bookkeeping and stay focused on the problem at hand. *They are still amazed at how few lines of code it took to achieve something renowned for being difficult.*

The *standard packages* in Go were another big win for Doozer, most notably the *websocket* package.

Here follows some of the afterthoughts of the developers themselves:

". . . For instance, a package we quickly found useful was websocket. Once we had a working data store, we needed an easy way to introspect it and visualize activity. Using the websocket package, Keith was able to add the web viewer on his train ride home and without requiring external dependencies. This is a real testament to how well Go mixes systems and application programming.

Deploying Doozer was satisfyingly simple. Go builds statically linked binaries which means Doozer has no external dependencies; it's a single file that can be copied to any machine and immediately launched to join a cluster of running Doozers."

Finally, Go's maniacal focus on simplicity and orthogonality aligns with our view of software engineering. Like the Go team, we are pragmatic about what features go into Doozer. We sweat the details, preferring to change an existing feature instead of introducing a new one. In this sense, Go is a perfect match for Doozer. We already have future projects in mind for Go. Doozer is just the start of much bigger system."

They also liked the automatic formatting tool *gofmt,* to achieve consistent code styling and layout, avoiding discussions about these topics.

Other languages provide somewhat similar mechanisms for concurrency—such as Erlang and Scala—but Go is designed to provide maximum efficiency and control, as well. In another article (ref. 12) Keith Rarick states:

"Go comes from a line of systems programming languages like C and C++, so it gives you the ability to really control the performance characteristics. When it comes time to measure things and make sure they run fast, you have the flexibility to really get in there and do what you need. And when you figure out why your program is being slow, you really have the control you need to fix it. Go gives you a unique combination: C gives you control, but it's not good for concurrency. It doesn't even give you garbage collection. Go gives you concurrency and garbage collection, but it still gives you control over memory layout and resource use."

In Doozer Go is primarily used as a systems programming language. A more technical description can be found in ref. 38; the code can be found at https://github.com/ha/doozer

21.2 MROffice—a VOIP system for call centers in Go.

http://mroffice.org/

This example demonstrates that Go is also suitable for simple, reliable applications programming. MROffice is a New Zealand based company specialized in market research software. They have built a telephony solution on top of Freeswitch for market research call centers using Go.Kees Varekamp who is the developer has a background in market research software. He found most existing software in that space to be pretty bad and launched MROffice in 2010 to provide better software to the marketrResearch industry.

His flagship product is named **Dialer** (http://mroffice.org/telephony.html).

What does Dialer primarily do?

- It connects interviewers in a call center to interviewees.

- It provides a bridge between interview platforms (that provide scripts and collect statistics) and a VoIP dialer (to do the actual telephony).

Why Go?

The 1st version of Dialer was written in Python, but his experience was that Python as a dynamic scripting language was perhaps not such a good choice for long-running server processes: lots of runtime errors occurred that could have been caught at compile time.

As Mr. Varekamp stated at the Sydney Go user group (March 2011): *"When Go came out it immediately made sense to me: type safe, compiled, feels like a scripting language."* So he ported the Python code to Go. *Go's concurrency model* suited the problem: a goroutine is started to handle each call, interviewer, and interviewee, all communicating via channels.The *http* and *websocket* libraries made it easy to write a management UI.

The product is now running in multiple call centers, and work is being done on a predictive dialer design that uses neural networks.

21.3 Atlassian—a virtual machine cluster management system.

http://www.atlassian.com/

At Atlassian Go is used in utility programming with concurrency support for provisioning and monitoring test servers. They make development and collaboration tools for software developers and are primarily a Java shop. They have a testing cluster of virtual machines (VMs), run over a large number of diskless hosts. Its provisioning and monitoring system is written in Go; the system consists of 3 parts:

- Agent processes that run on each server, broadcasting the state of their VMs.
- A manager process that listens to the agent's broadcasts and takes action if a VM fails to report
- A command-line tool for issuing commands to the manager.

The Agent uses protocol buffer to encode the state information it has read, and broadcasts this info via UDP. The Manager reads a configuration file and launches one goroutine for each VM in the cluster. Each goroutine listens for announcements from its corresponding VM, and issues instructions (shell commands) to keep it in the correct state.

Why Go works here: One goroutine per VM maps nicely to their configuration.

The system is also *easy to deploy* because they can ship binaries with no dependencies.

As Dave Cheney, Atlassian Engineer states it: *"The agent process runs on machines that netboot and run completely from RAM. A single static binary was a major saving, compared to a JVM or Python runtime."*

21.4 Camlistore—a content addressable storage system.

http://camlistore.org/

In Camlistore "full stack" programming is happening in Go, from the data store to the UI.

The system is being developed by Brad Fitzpatrick is a system for storing personal data in the cloud and sharing that data with friends and the public. It is composed out of a content-addressable data store, a synchronization and access-control mechanism, an API, a user interface, a personal "home directory for the web".

It is programming language-agnostic bu the largest parts of it are written in Go. They comprise a blob dataserver, an http-server, an http user interface, and a number of command-line tools.

It could be used for:

- Personal backups automatically synced to remote servers.
- Dropbox-style file synchronization across machines.
- Photo management and sharing.
- Web site content management.

Here are some comments from Brad about this Go-project:

"I bust out lots of fast, correct, maintainable testable code in very small amounts of time, without much code.I haven't been this excited about a language in ages.I had the idea for Camlistore a long time ago, but before I learned Go it had always seemed too painful."

21.5 Other usages of the Go language.

In the preceding sections we discussed only a few of the many places where Go is already used in business environments. Some other organizations using Go are:

1) **Canonical—**the Ubuntu company **(http://www.canonical.com/):** developing backend infrastructure using Go**,** with main developer Gustavo Niemeyer, e.g. the project Ensemble (see ref. 30)
2) **FeedBooks** (http://www.feedbooks.com/): distributing e-books with Go.

   FeedBooks is a distributor of e-books, it uses Go and mgo to serve more than a million book covers a day. A comment from Benoît Larroque, R&D Engineer at Feedbooks:

   "mgo (a Go library for talking to MongoDB) enables us to blazingly serve more than 1.000.000 book covers a day while reducing our servers load "

3) **Anchor-Orchestra** (http://www.anchor.com.au/): a distributed execution framework using Go. The hallmark of this company is high level server support, configuring application settings, caching and resolve scalability issues. They can also work with other site hosting companies to expertly set up load balancing, database clusters and virtual environments. For that purpose they developed and use the **Orchestra** distributed execution framework using Go.

   (More info: http://www.anchor.com.au/blog/2011/08/the-automation-waltz/)

4) **Open Knowledge Foundation** (http://eris.okfn.org/ww/2011/03/gockan):

This organization uses Go for (meta) data catalogue aggregation and linked data. All of the existing software was written in Python, so the developer could compare both and he concludes:

- Go is *simple*. Once the initial, shallow, learning curve is passed it is about as convenient and comfortable to work in as Python. The only disadvantage is that there are not as many libraries as with Python.
- Go is a *statically typed language*. This may seem like an esoteric detail but it has some important consequences. Much programming in Python involves extensive unit and functional tests. This can get to be quite burdensome, the CKAN test suite, despite some major improvements takes quite a long time to run. It soon becomes clear, however, that many of these tests are basically testing duck typing and variable existence (such as when you rename a variable in refactoring and aren't sure you renamed everything properly). These of things are caught by the compiler in a language like Go and do not need separate tests. This means *you can write fewer tests because the compiler itself is quite a formidable test suite.*
- Even though it is a compiled language, the compilation process is very fast and *the write-compile-test loop is no slower than the write-test loop in Python*. Because fewer tests need to be run, see above, this loop is further compressed.

- Go can be *much more memory efficient* than Python . . . , the difference is striking.
- Go programs are *fast compared* to Python ones, being a compiled and type-checked language.
- Go is not object-oriented, at least not in the same sense as Python is. Instead it has a concept of interfaces. This makes for *cleaner design* because it does not encourage an elaborate multiply inheriting class hierarchy . . . . , interfaces just feel cleaner.
- Go has *built-in concurrency*. There are many opportunities for parallelism in this work and this is nice to have.

5) **Tinkercad Inc.**(http://tinkercad.com/) : this Finnish company started by Kai Backman is designing software for 3D solid modeling and printing in the browser/cloud, using WebGL for rendering on the client. Watch this video http://www.youtube.com/watch?v=5aY4a9QnLhw for a tech talk on the subject. A phrase from Kai: "At this time (2011) Go is probably the best language for writing concurrent servers in."
6) **Clarity Services Inc**. (http://www.clarityservices.com) : this company is a real-time credit bureau and uses Go for event based post-processing of credit applications
7) **Cablenet Communication Systems Ltd** (http://www.cablenet.com.cy/en/): this Cyprus cablenet provider developed an in-house Provisioning System in Go.
8) **Tonika** (http://pdos.csail.mit.edu/~petar/5ttt.org/): is an open source secure social networking platform developed in Go.
9) **Medline** (http://eris.okfn.org/ww/2011/05/medline/): uses Go's XML parser to transform compressed XML from Medline (data from medical journals) into RDF
10) **Iron.io** (www.iron.io): building cloud infrastructure software.

Its first product developed in Go is **SimpleWorker**, a large scale background processing and scheduling system; they are using Go for other services as well.

11) **SmartTweets** (http://www.facebook.com/apps/application.php?id=135488932982): a Facebook application built in Go. This application re-posts your Twitter status updates to your Facebook profile, and allows filtering retweets, mentions, hashtags, replies, and more. The application has now over 120,000 users. "It's a stable language," Michael Hoisie says. "It can handle the load."
12) At **Sandia National Laboratories** (http://www.sandia.gov/about/index.html) a U.S. government institute to develop science-based technologies that support national security, a good number of people who used to program using C, C++, Perl, Python or whatever for the HPC management software, have moved to Go and have no intention of going back: Go hits a good place between efficiency and language capability and ease of writing code (according to Ron Minnich).
13) **Carbon Games** (http://carbongames.com/): an online game company, uses Go for their backend server stuff.

14) **Vaba Software** (http://vabasoftware.com/): rewrote their message and storage engines in Go.
15) **Institute for Systems Biology** (http://systemsbiology.org/): developed the *Golem* (see http://code.google.com/p/golem/) distributed computational analysis system in Go.
16) **Second Bit** (http://www.secondbit.org/): using Go to power their 2cloud service.
17) **Numerotron Inc** (http://www.stathat.com/): developed their *StatHat* statistics and event tracking system in Go.

Last but not least there is **Google Inc**. itself, the home of (the inventors) of Go.

The usage of Go within Google itself is kept rather secret. But in May 2010 Rob Pike declared that Google's back-end infrastructure was running applications built with Go (ref. 27). Go is being used in a number of systems (web servers, but also storage systems and databases) that play a role in the distributed infrastructure that spans Google's worldwide network of data centers. Go probably will become the standard back-end language at Google over the next few years. Andrew Gerrand also says Google employees are using Go to simply grab information from servers. "Google has people who administer apps and services, and they need to write tools that scrape a few thousand machines statuses and aggregate the data," he says. "Previously, these operations people would write these in Python, but they're finding that Go is much faster in terms of performance and time to actually write the code."

A comprehensive list of usage of Go in organizations can be found at http://go-lang.cat-v.org/organizations-using-go

APPENDICES

(A) CODE REFERENCE

gotemplate.go:

```
package main

import (
“fmt”
)

const c = “C”

var v int = 5

type T struct{}

func init() {
        // initialization of package
}

func main() {
        var a int
        Func1()
        // ...
        fmt.Println(a)
}

func (t T) Method1() {
        //...
}

func Func1() { // exported function Func1
```

```
            //...
    }
```

Format specifiers:

%t booleans
%c character (like 's')
%U Unicode code point (U+hhhh notation)
%s strings or []bytes: shows raw bytes
%n.mf   n is the minimum width of the string, and m the maximum width.
%**q** for strings and []byte produces a quoted string format (%#**q** uses backquotes)
%**b** bit representation (base 2)
%d integers (base 10) (**%x** or %**X** is hexadecimal (base 16) notation (can also be used for strings and []byte), %**o** is octal (base 8) notation,), (see example in § 5.4.4)
%f of **%g** floats or complex (**%e** is scientific notation), or their equivalent uppercase versions **%F, %G, %E**
%n.mf n is the maximum amount of digits shown, and m the number of digits after the decimal point.
%p pointers (hexadecimal address with prefix 0x)
%v default format, when String() exists, this is used.
%+v gives us a complete output of the instance with its fields
%#v gives us a complete output of the instance with its fields and qualified type name
%T gives us the complete type specification
Write %% if you want to print a literal % sign
Default is right-justified output, a—between % and the letter makes it left justified.
A 0 between % and the letter shows leading 0s instead of spaces.

Operator precedence:

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

Value types	**Reference types**
`new(T) returns type *T (allocated)`	`make(T) returns type T (initialized)`
strings (immutable)	func
array (mutable)	slice
struct (mutable)	map
	channel

index-notation [i] is used for strings, arrays, slices, maps

Nice to know:

`math.MaxInt32` is the largest integer (also equal to int(^uint(0) >> 1))
other related constants can be found in the math package.

Overview for-range construct (iterator):

If only one value is used on the left of a range expression, it is the 1st value in this table; T, K and V are types.

Range expression	1st value	2nd value	notes
`string s string type`	`index i int`	`rune int`	`range iterates over Unicode code points, not bytes`
`array or slice a [n]T, *[n]T, or []T`	`index i int`	`a[i] T`	
`map m map[K]V`	`key k K`	`value m[k] V`	
`channel c chan T`	`element e T`	`none`	

Some elementary idiomatic code snippets:

(1) Distributed keyword:

```
import (
       "fmt"
       "os"
)
```

```
type (
       Protocol string
       Hostname string
)

var (
       a int
       b bool
       str string
)

const (
       Unknown = 0
       Female  = 1
       Male    = 2
)
```

(2) Enums:

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

(3) Multiple declarations of variables of the same type on a single line

```
var a,b,c int
```

(4) Declaration and assignment together:

```
a := 100
```

(5) Multiple assignments of variables on a single line:

```
a,b,c = 5,7,9                Swapping: a,b = b,a
a,b,c := 5,7,9
```

(6) Conversions to typeB: `valueOfTypeB = typeB(valueOfTypeA)`

(7) No else branch:

```
if condition {
        return x
}
return y
```

(8) Initialization in if:

```
if value := process(data); value > max {
        …
}
```

(B) CUTE GO QUOTES.

1) *Go allows me to say: "I would actively encourage my competition to use Java."*
2) *if _, ok := reality.(Reasonable); !ok {*
   panic(reality)
   }
   (Eleanor McHugh—Games With Brains—http://feyeleanor.tel)

3) "*After Go, programming in anything else seems as difficult as balancing the State of California's budget.*" -- Charles Thompson
4) "*Go is like a better C, from the guys that didn't bring you C++*" -- Ikai Lan
5) "*Go seems to be a counterpoint to the old stroustop adage 'There are only two kinds of languages: the ones people complain about and the ones nobody uses.' Go seems to be a language people complain about without being used.*" -- tef in reddit.
6) *"From the tutorial: "The language forces the brace style to some extent." Well, that's it. If I can't have a brace-war tearing the dev group apart for months and kill productivity, I want nothing to do with that language."* -- SoftwareMaven in hackernews
7) *Programming in Go is like being young again (but more productive!).*"—Anneli
8) *Four out of five language designers agree: Go sucks. The fifth was too busy [to answer] actually writing code [in Go].*"—aiju
9) *[Go] is a WTF-language. Compiler* fails *(not warns) if you have any unused variables, even though it has garbage collection* -- @myfreeweb
10) *everytime I get [a variable not used] error I'm like "come on compiler, give me a break. ok, ok, you're right, thanks for not letting me code too much like a pig.*"

GO QUOTES: TRUE BUT NOT SO CUTE.

1) *Most of the appeal for me is not the features that Go has, but rather the features that have been intentionally left out."*—from Hacker News
2) "*Go is not meant to innovate programming theory. It's meant to innovate programming practice.*" -- Samuel Tesla
3) "*One of the reasons I enjoy working with Go is that I can mostly hold the spec in my head—and when I do misremember parts it's a few seconds' work to correct myself. It's quite possibly the only non-trivial language I've worked with where this is the case.*" -- Eleanor McHugh
4) "*In Go, the code does exactly what it says on the page.*" -- Andrew Gerrand
5) "*[the Go authors] designed a language that met the needs of the problems they were facing, rather than fulfilling a feature checklist*" -- ywgdana in reddit
6) "*I have reimplemented a networking project from Scala to Go. Scala code is 6000 lines. Go is about 3000. Even though Go does not have the power of abbreviation, the flexible type system seems to out-run Scala when the programs start getting longer. Hence, Go produces much shorter code asymptotically.*" -- Petar Maymounko
7) "*Go doesn't implicitly anything.*" -- Steven in golang-nuts
8) "*Porting my code review tools to Go from Python. Surprised to see a reduction in line counts.*" -- Scott Dunlop
9) "*Why would you have a language that is not theoretically exciting? Because it's very useful.*" -- Rob Pike paraphrased by Roger Peppe
10) *Porting a Google App Engine app written in Python to Go: Although I'm new to Go I get much quicker results than I've got with Python. Never used a language before that empowers you to solve problems as quick as Go does*" -- Alexander Orlov

(C) LIST OF CODE EXAMPLES (Listings)

Chapter 2—Installation and Runtime Environment

Chapter 3—Editors, IDE's and other tools

Exercise 5.5—for_character.go	§ 5.4
Exercise 5.6—bitwise_complement.go	§ 5.4
Exercise 5.7—fizzbuzz.go	§ 5.4
Exercise 5.8—rectangle_stars.go	§ 5.4
Exercise 5.9—output.go	§ 5.4

Chapter 6—Functions

Exercise 6.1—mult_returnval.go	§ 6.3
Exercise 6.2—error_returnval.go	§ 6.3
Exercise 6.3—varargs.go	§ 6.3
Exercise 6.4—fibonacci2.go	§ 6.6
Exercise 6.5—10to1_recursive.go	§ 6.6
Exercise 6.6—factorial.go	§ 6.6
Exercise 6.7—strings_map.go	§ 6.7
Exercise 6.8—lambda_value.go	§ 6.8
Exercise 6.9—fibonacci_closure.go	§ 6.8
Exercise 6.10—compose.go	§ 6.8

Chapter 7—Arrays and Slices

Exercise 7.1—array_value.go	§ 7.1
Exercise 7.2—for_array.go	§ 7.1
Exercise 7.3—fibonacci_array.go	§ 7.1
Exercise 7.4—fibonacci_funcarray.go	§ 7.2
Exercise 7.5—append(slice, data[]byte) []byte	§ 7.2
Exercise 7.6—buffer split	§ 7.2
Exercise 7.7—sum_array.go	§ 7.3
Exercise 7.8—min_max.go	§ 7.3
Exercise 7.9—magnify_slice.go	§ 7.5
Exercise 7.10—filter_slice.go	§ 7.5
Exercise 7.11—insert_slice.go	§ 7.5
Exercise 7.12—remove_slice.go	§ 7.5
Exercise 7.13—string_split.go	§ 7.6
Exercise 7.14—string_split2.go	§ 7.6
Exercise 7.15—string_reverse.go	§ 7.6
Exercise 7.16—uniq.go	§ 7.6
Exercise 7.17—bubblesort.go	§ 7.6
Exercise 7.18—map_function.go	§ 7.6

Chapter 8—Maps

Exercise 8.1—map_days.go	§ 8.2
Exercise 8.2—map_drinks.go	§ 8.3

Chapter 9—Packages

Exercise 9.1—dlinked_list.go	§ 9.1
Exercise 9.2—size_int.go	§ 9.2
Exercise 9.3—main_greetings.go	§ 9.5
greetings\greetings.go	
Exercise 9.4—main_oddeven.go	§ 9.5
even.go	
Exercise 9.5—main_fibo.go	§ 9.5
fibo\fibonacci.go	
Exercise 9.6—package strev	§ 9.8

Chapter 10—Structs and Methods

Exercise 10.1—vcard.go	§ 10.1
Exercise 10.2—personex1.go	§ 10.1
Exercise 10.3—point.go	§ 10.1
Exercise 10.4—rectangle.go	§ 10.1
Exercise 10.5—anonymous_struct.go	§ 10.4
Exercise 10.6—employee_salary.go	§ 10.4
Exercise 10.7—iteration_list.go	§ 10.4
Exercise 10.8—inheritance_car.go	§ 10.4
Exercise 10.9—point_methods.go	§ 10.5
Exercise 10.10—inherit_methods.go	§ 10.5
Exercise 10.11—magic.go	§ 10.5
Exercise 10.12—type_string.go	§ 10.6
Exercise 10.13—celsius.go	§ 10.6
Exercise 10.14—days.go	§ 10.6
Exercise 10.15—timezones.go	§ 10.6
Exercise 10.16—stack_arr.go	§ 10.6
stack_struct.go	
Exercise 10.17—stack_struct.go	§ 10.6
main_struct.go	

Chapter 11—Interfaces

Exercise 11.1 - simple_interface.go	§ 11.1
Exercise 11.2-3—interfaces_poly2/3.go	§ 11.1
Exercise 11.4 - simple_interface2.go	§ 11.1
Exercise 11.5—interfaces_ext.go	§ 11.6
Exercise 11.6—point_interfaces.go	§ 11.6
Exercise 11.7—float_sort.go / float_sortmain.go	§ 11.6
Exercise 11.8—sort.go / sort_persons.go	§ 11.6
Exercise 11.9 - simple_interface3.go	§ 11.7
Exercise 11.10 -min_interface.go / minmain.go	§ 11.7
Exercise 11.11—map_function_interface.go	§ 11.8
Exercise 11.12—map_function_interface_var.go	§ 11.8
Exercise 11.13 - main_stack.go—stack/stack_general.go	§ 11.8

Chapter 12—Reading and writing

Exercise 12.1 - word_letter_count.go	§ 12.1
Exercise 12.2 - calculator.go	§ 12.1
Exercise 12.3—read_csv.go	§ 12.4
Exercise 12.4—wiki_part1.go	§ 12.4
Exercise 12.5—hello_who.go	§ 12.5
Exercise 12.6—cat_numbered.go	§ 12.5
Exercise 12.7—remove_3till5char.go	§ 12.5
Exercise 12.8—degob.go	§ 12.11
Exercise 12.9—hash_md5.go	§ 12.12

Chapter 13—Error-handling and Testing

Exercise 13.1 - recover_dividebyzero.go	§ 13.3
Exercise 13.2 - panic_defer.go	§ 13.3
Exercise 13.3 - panic_defer_convint.go	§ 13.3
Exercise 13.4 - string_reverse_test.go	§ 13.6
Exercise 13.5—panic_defer.go	§ 13.3

Chapter 14—Goroutines and Channels

Exercise 14.1—channel_block3.go	§ 14.2
Exercise 14.2 - blocking.go	§ 14.2

Exercise 14.3—channel_buffer.go	§ 14.2
Exercise 14.4—give explanation	§ 14.2
Exercise 14.5—gosum.go	§ 14.2
Exercise 14.6—producer_consumer.go	§ 14.6
Exercise 14.7—goroutine_panic.go	§ 14.4
goroutine_close.go	
goroutine_select.go	
Exercise 14.8—go_fibonacci.go	§ 14.4
go_fibonacci2.go / go_fibonacci3.go	
Exercise 14.9—random_bitgen.go	§ 14.4
Exercise 14.10 -polar_to_cartesian.go	§ 14.4
Exercise 14.11 -concurrent_pi.go	§ 14.4
concurrent_pi2.go	
Exercise 14.12—general_lazy_evaluation2.go	§ 14.8
Exercise 14.13—multiplex_server3.go	§ 14.6

Chapter 15—Networking, templating and web applications

Exercise 15.1—client1.go / server1.go	§ 15.1
Exercise 15.2—webhello2.go	§ 15.2
Exercise 15.3—hello_server.go	§ 15.2
Exercise 15.4—http_fetch2.go	§ 15.3
Exercise 15.5—twitter_status_json.go	§ 15.3
Exercise 15.6—statistics.go	§ 15.4
Exercise 15.7—template_validation_recover.go	§ 15.7

(E) References in the text to Go—packages

appengine	NewContext	§ 20.5
appengine/user	Current / LoginURL	
appengine/datastore	NewQuery / Order / Limit / GetAll / Put /	
	NewIncompleteKey / SecondsToTime	§ 20.7
bufio	Reader / NewReader / ReadString/ ReadLine	§ 12.1
	Writer / New Writer / WriteString / Flush	§ 12.3
	WriteTo	§ 15.9
bytes	Buffer	
	WriteString / String	§ 7.2.6 / § 9.3 / § 11.1 / § 12.1 / § 15.5-6 / § 16.2
compress/gzip	NewReader	§ 12.2.2
crypto/sha1	New / Write / Sum / Reset	§ 12.12

encoding/gob	NewEncoder / Encode / NewDeoder / Decode	§ 12.11
encoding/json	Marshal / NewEncoder / Encode	§ 12.9
encoding/xml	Token / NewParser / Name / Value / Attr	
	StartElement / EndElement / CharData	§ 12.10
	UnMarshal	§ 14.3
expvar	NewInt / Publish	§ 15.6
filepath	Base	§ 13.1
flag	Parse / NArg	§ 12.4
	Bool / Arg	§ 12.5
	Int	§ 14.8
	Name / Value / DefValue / VisitAll	§ 15.6
fmt	Print / Println / Printf	§ 4.4.3
	Sprintf	§ 4.4.3 / § 5.1
	Fprint / Fprintf	§ 12.2.2 § 12.8 / throughout chapter 15
	Scanln / Scanf / Scanf	§ 12.1
	Fscanln	§ 12.2
	Errorf	§ 4.5.2.1 / § 13.1
io	Writer	everywhere
	ReadAll	§ 12.7
	WriteString / Copy	§12.12 / § 15.1 § 15.6
io/ioutil	ReadFile / WriteFile	§ 12.2 § 15.4
	ReadAll	§ 15.3
log	SetFlags	§ 6.10
	Print / Println / Printf	§ 13.3
	Panicln / Exit	§ 15.6
	Fatalf	§ 15.8
math	Atan	§ 4.4.5
	Sqrt	§ 5.2
	Pi	§ 11.2.3
math/big	Int / Rat	§ 4.5.2.5 / § 9.4
math/rand	Int / Intn / Seed / Float32 / Float64	§ 4.5.2
net	Listener / Conn / Listen / Accept / Read / Error	§ 15.1
	Dial / RemoteAddr / TCPListener / ResolveTCPAddr / ListenTCP / Addr	
net/http	Request / Response / ResponseWriter / HandleFunc / ListenAndServe	
		§ 15.2
	Get / Redirect / NotFound / Error / status constants	§ 15.3
	DetectContentType	§ 15.4
	Request.Method / Handle / HandlerFunc	§ 15.6
	Serve	§ 15.7
net/rpc	Register / HandleHTTP / DialHTTP / Call	§ 15.9

net/smtp	`Dial / Mail / Rcpt / Data / SendMail / PlainAuth`	§ 15.12
netchan	`NewExporter / Export / NewImporter / Import`	§ 15.10
os	`GetEnv`	4.4.1 / § 13.2
	`Error / Exit / Open`	§ 5.2
	`Stdin / Stdout / Sterr / EOF`	§ 12.1
	`File`	§ 12.2
	`Errno`	§ 12.4
	`Args`	§ 12.4 § 15.6
	`NewError`	§ 13.1
	`StartProcess`	§ 13.4 § 15.6
	`Pipe / Close / Release / Wait`	§ 15.6
os/exec	`Command / Run`	§ 13.4
reflect	`StructType / Typeof / Field`	§ 10.4
	`ValueOf / Type / Kind / NumField / Field / Method / Call`	§ 10.5
regexp	`Match / Compile / ReplaceAllString / ReplaceAllStringFunc`	§ 11.10
	`MustCompile`	§ 15.4
runtime	`Version`	§ 2.3
	`Caller`	§ 6.10
	`SetFinalizer / GC / MemStats`	§ 10.7
	`Error`	§ 13.2
	`Gosched / Goexit`	§ 14.1 / § 14.5
sort	`SortInts / IntsAreSorted / SortFloat64s / Strings /SearchInts`	§ 7.6.5
strconv	`IntSize`	
	`Itoa / Atoi / FormatFloat / ParseFloat`	§ 4.7.8
strings	`HasPrefix / HasSuffix / Contains / Index / LastIndex / Count / Repeat`	§ 4.7
	`ToLower / ToUpper / Trim / Fields`	
	`IndexFunc / Map`	§ 6.7
	`Join`	§ 12.4
	`NewReader`	§ 4.7.11 / § 12.10
sync	`Mutex  / RWMutex / Once / Do`	§ 9.3 § 14.13
syscall	`Stdin / Stdout / Sterr / Open / Close / Read / Write`	§ 12.4
text/template	`Template / Execute`	§ 15.7
	`Parse / ParseFiles / Must`	
time	`Time / Duration / Location`	§ 4.5.6.2 / § 6.9
	`Now / Format / Add / Sub`	§ 4.5.6
	`Sleep`	§ 14.1
	`Ticker / Tick / Stop / Timer / After`	§ 14.5
unicode	`IsLetter / IsDigit / IsSpace`	§ 4.5.3
unicode/utf8	`RuneCountInString`	Ex 4.6 § 4.6

unsafe	`Sizeof`	`§ 10.2.1 / ex. 9.2`
websocket	`Handler / Conn / Read`	`§ 15.14`

(F) References in the text to Go—tools

go doc	§1.1	godoc in the golang website
	§3.6	documenting code
	§4.2.3	extracting comments from code with godoc
	§9.6	documenting your own packages
go fix	§3.7	updating code
go fmt	§3.5	formatting code
		imposing standard formatting (various places)
go install	§ 3.7	installing packages outside of standard library
	§ 3.9	using with cgo
	§ 9.7	installing custom packages
gomake	§3.4	building a program
	§11.1	building and installing a package
	§13.7	testing and benchmarking in Go
go test	§13.7	testing and benchmarking in Go

(G) Answers to Questions

Question 4.1: int and int64 are different types, in order to mix variables of these types, you have to use explicit conversions.

Question 4.2:
- multiplication of numeric variables: 3 * 4
- declaration of a pointer type: *float64
- dereferencing a pointer to obtain the value

Question 5.1:
```
was <= 6
was <= 7
was <= 8
default case
```

Question 5.2: 1) prints increasing value of i infinitely, since the middle check statement does not exist;

2) prints "value of i is: 0" infinitely since the incrementing part is not there and i can never reach 3;
3) here both initialization and incrementing is missing in the for statement, but it is managed outside of it:

```
a
aa
aaa
aaaa
```

4)
```
Value of i, j, s: 0 5 a
Value of i, j, s: 1 6 aa
Value of i, j, s: 2 7 aaa
(see listing multiple_for.go)
```

Question 5.3: 1)
```
Value of i is: 0
Value of i is: 1
Value of i is: 2
A statement just after for loop.
```
2)
```
Odd: 1
Odd: 3
Odd: 5
```

Question 6.1: `The compiler gives the error: "function ends without a return statement"`
`The for loop contains a return, but this executes only after the if condition is satisfied; the compiler reasons that it could be that no if condition is true, so return never executes.`
`Correction: place a return 0 after the for-loop:`

```
func (st *Stack) Pop() int {
        v := 0
        for ix:= len(st)-1; ix>=0; ix-- {
                if v=st[ix]; v!=0 {
                        st[ix] = 0
                        return v
                }
        }
        return 0
}
```

Question 6.2: In case A) both a and b are pointers to the same value of type A.

If through b in DoSomething the value is changed, a sees that change.

In case B) a is passed by value, so a copy of the value is made and b is a pointer to that copy.

So a and b refer to different copies of the (same) value of type A.

If through b in DoSomething the value is changed, a doesn't see that change, the value of a doesn't change.

Question 7.1:

```
Array item 0 is a
 Array item 1 is b
 Array item 2 is c
 Array item 3 is d
```

Question 7.2: Given the slice of bytes `b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}`

```
b[1:4] == []byte{'o', 'l', 'a'} // this slice has length 3 and indices 0,
1 and 2
b[:2] == []byte{'g', 'o'}
b[2:] == []byte{'l', 'a', 'n', 'g'}
b[:] == b
```

Question 7.3:

```
Given s := make([]byte, 5), len(s) == cap(s) is 5
s = s[2:4], len(s) is 2 and cap(s) is 3
```

Question 7.4: Suppose `s1 := []byte{'p', 'o', 'e', 'm'} and s2 := d[2:]`
What is the value of s2 ? `s2 == []byte{'e', 'm'}`
We do: `s2[1] == 't'`, what is now the value of s1 and s2 ?

```
      s2 == []byte{'e', 't'} and s1 == []byte{'p', 'o', 'e', 't'}
```

Question 7.5: a) items will be unchanged, because the local variable item is a copy of the real value, so the real values cannot be changed through it!

b)
```
for ix := range items {
    items[ix] *= 2
}
```

Question 7.6: 1) in variadic functions: `func myFunc(a, b, arg ...int) {}`

```
myFunc receives a variable number of values in arg
```

the built-in append function is an example

2) when defining an array without indicating the length

```
arrLazy = [...]int{5, 6, 7, 8, 22}
```

3) when calling a variadic function with an array/slice:

```
myFunc(5, 10, sl1...)
```

Question 7.7: 1) the length of s[n:n] is 0, this slice is empty.

2) the length of s[n:n+1] is 1, this slice contains the element at index n.

Question 8.1:

```
Map item: Capital of Japan is Tokyo
Map item: Capital of Italy is Rome
Map item: Capital of France is Paris
```

Question 9.1:

a) Can a package be divided over multiple source files ? yes
b) Can a single source file contain multiple packages ? no

Question 10.1:

```
1) packagename.Constant or packagename.Method()
2) struct.field
3) variable.method()
```

Question 10.2: a) Suppose we define: `type Integer int`
Fill in the body of the get() function: `func (p Integer) get() int { return in(p) }`

b) Defined are: `func f(i int) { }`

```
var v Integer
```

How would you call f with as parameter v ? `f(int(v))`
`f(v)` is not valid, because type int and type Integer are not the same!

c) Suppose Integer is define as : `type Integer struct { n int }`
Now fill in the body of the get() function: func `(p Integer) get() int { return p.n }`

d) Same question as in b) for the Integer struct type.
How would you call f with as parameter v ? `f(v.n))`

Question 14.1: Use a buffered channel throttle and a NewTicker object tick:

```
import "time"
rate_per_sec := 10
burst_limit := 100
tick := time.NewTicker(1e9 / rate_per_sec)
defer tick.Stop()
throttle := make(chan int64, burst_limit)
go func() {
  for ns := range tick {
    select {
      case: throttle <- ns
      default:
    }
  }  // exits after tick.Stop()
}()
for req := range requests {
  <- throttle  // rate limit our Service.Method RPCs
  go client.Call("Service.Method", req, ...)
}
```

Question 17.1: see § 17.1

Question 17.2: see § 17.2

(H) ANSWERS TO EXERCISES

Exercise 4.4:

```
type Rope string
var r1 Rope = "Admiral Blake"
```

Exercise 5.1:

```
anInt, err := strconv.Atoi(origStr)
The declarations var anInt int and var err error can then be omitted
```

Exercise 5.3:

```
The variable i in the last line of main() is out of scope, it was only
known within the body of the for-loop where it was declared.
We can make it work by declaring I before the loop, like:

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

Exercise 5.6: `0 0 0 0 0  (think about the initialization)`

Exercise 7.5:

```
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // reallocate
    // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
// The copy function is predeclared and works for
// any slice type.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
    return slice
}
```

Exercise 7.6: `header, tail := buf[0:n], buf[n:len(buf)]`

Exercise 10.2: Difference in behavior: only the copied value in upPerson is changed, not the value in main()

Exercise 10.7: The type list.List is defined in the package container/list, so it is not local (belonging to the current package); therefore you cannot define a new method like Iter() on it in the local package.

Exercise 10.8: `magic.go`

The output is:

```
voodoo magic
base magic base magic
```

So when MoreMagic() is called the Magic() method is also called on Base.

This composition is achieved via delegation, not inheritance. Once the anonymous member's method has been called, flow has been delegated to that method entirely. So you cannot simulate a type hierarchy

Exercise 12. 7: `remove_3till5char.go`
The logical error is that in the if test on EOF, a return is done: this returns from main(), so Flush() doesn't get executed. For small files the whole output contents is still in the flush buffer, so nothing gets written.

That no message "`Conversion done`" was printed should have alerted you.

Solution: use a break instead of return, this jumps out of the for-loop, so the Flush() gets executed.

```
if readerError == io.EOF {
      fmt.Println("EOF")
      break
}
```

Exercise 13.1: The output is:

```
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.
```

Remember: defer is executed in LIFO-order, panic returns the 'wrong' variable value with Sprintf, and because the panic occurs in g, "Returned normally from g." is not printed!

Remove the defer function in f() and see what happens: the program crashes and the output sequence is the same except for:

```
…
Defer in g 0
panic: 4
runtime.panic+0x9e /go/src/pkg/runtime/proc.c:1060
runtime.panic(0x43fa1c, 0x12740260)
…
```

Exercise 14.2: `blocking.go`

out is an unbuffered channel, out <- 2 blocks because the goroutine which must do the receiving has not been started yet: so we have a deadlock situation.

Solutions:
1) start the go routine before putting something on out
2) make out a buffered channel: out:=make(chan int, 1) then the send can proceed without blocking.

Exercise 14.3: The main() routine ends before getData() can receive and print the data.

(I) BIBLIOGRAPHY (Resources and References)

1) The official Go language website: http://golang.org/
2) The official Go Language FAQ: http://golang.org/doc/go_faq.html
3) The official Go Language Specification: http://golang.org/doc/go_spec.html
4) The official Go-Tutorial: http://golang.org/doc/go_tutorial.html
5) Effective Go—http://golang.org/doc/effective_go.html
6) The Go Programming language Blog: http://blog.golang.org/
7) ARTICLE: "Go For C++ Programmers"—http://golang.org/doc/go_for_cpp_programmers.html
8) The French Go-wiki: http://fr.wikibooks.org/wiki/Programmation_en_Go
9) ARTICLE: "CodeLab: Writing Web Applications"—http://golang.org/doc/codelab/wiki/
10) Wiki-list of Go articles: http://code.google.com/p/go-wiki/w/list
11) Website: The Computer Language Benchmark Game: http://shootout.alioth.debian.org/
12) ARTICLE: "Google Go boldly goes where no code has gone before"—The Register, May 2011—http://www.theregister.co.uk/2011/05/05/google_go/
13) BOOK: "Go Programming"—John P. Baugh—jun 2010—La Vergne TN USA—ISBN: 1453636676 / EAN-13: 9781453636671
14) ARTICLE: "Go Data Structures"—Russ Cox—http://research.swtch.com/2009/11/go-data-structures.html
15) ARTICLE: "Profiling Go Programs "—Russ Cox—The Go Programming Language Blog, Jun 2011—http://blog.golang.org/2011/06/profiling-go-programs.html
16) VIDEO: Go Programming—Russ Cox / Rob Pike—Google IO 2010

17) ARTICLE: “Benchmarking Go and Python webservers.”—Michał Derkacz http://ziutek.github.com/web_bench/
18) WEB SITE: “Go Language Patterns”—Ryanne Dolan—http://www.golangpatterns.info/
19) VIDEO: Real World Go—Andrew Gerrand—Google IO BootCamp 2011
20) VIDEO: “Writing Go Packages”—Andrew Gerrand
21) VIDEO: “Testing Go Packages”—Andrew Gerrand
22) VIDEO: Practical Go Programming—Andrew Gerrand—FOSSDEM 2011
23) ARTICLE: “C? Go? Cgo!”—Andrew Gerrand—The Go Programming Language Blog, Mar 2011—http://blog.golang.org/2011/03/c-go-cgo.html
24) WEB SITE: “Go Snippets”—Andrew Gerrand—http://gosnip.posterous.com/
25) PDF: “Learning Go”—Miek Gieben—oct 2010: http://www.miek.nl/files/go
26) ARTICLE: “Loop Recognition in C++/Java/Go/Scala”—Robert Hundt—http://research.google.com/pubs/pub37122.html
27) ARTICLE: “Google programming Frankenstein is a Go—Python-C++ crossbreed lands on Goofrastructure”—Cade Metz—The Register, May 2010—http://www.theregister.co.uk/2010/05/20/go_in_production_at_google/
28) BOOK: “Systemprogrammierung in Google Go—Skalierbarkeit—Performanz—Sicherheit”—Frank Müller—jan 2011—dpunkt.verlag—ISBN 978-3-89864-712-0
29) WEBSITE: “Coding in Go”—Frank Müller—http://www.tideland.biz/CodingInGo
30) ARTICLE: “Ensemble, Go, and MongoDB at Canonical”—Gustavo Niemeyer—http://blog.labix.org/2011/08/05/ensemble-go-and-mongodb-at-canonical#more-706
31) Go-course given at Google by Rob Pike (oct 2009): slides of Day 1, 2, 3: http://golang.org/doc/GoCourseDay1.pdf, etc.
32) ARTICLE: “Google Discloses New Go Language” -- Nikkei Electronics Asia -- January 2010—Tech-On!
33) VIDEO: The Go Programming Language—Rob Pike—30/10/2009 (the ‘initial’ talk)
34) VIDEO: Public static void—Rob Pike—OSCON 2010
35) VIDEO: Another Go at language design—Rob Pike—OSCON 2010
36) VIDEO: Go—Rob Pike—Emerging languages OSCON 2010
37) ARTICLE: “The Laws of Reclection”—Rob Pike—The Go Programming Language Blog, Sep 2011—http://blog.golang.org/2011/09/laws-of-reflection
38) ARTICLE: “Introducing Doozer”—Keith Rarick—http://xph.us/2011/04/13/introducing-doozer.html
39) ARTICLE: “Go at Heroku”—Keith Rarick—The Go Programming Language Blog, Apr 2011—http://blog.golang.org/2011/04/go-at-heroku.html
40) BOOK: “Programming in Go: Creating Applications for the 21st Century”—Mark Summerfield—May 2012 (estimated)—Addison-Wesley Professional—ISBN-10: 0-321-77463-9 / ISBN-13: 978-0-321-77463-7

41) WEB SITE: “Go Lang Tutorials”—Sathish VJ—http://golangtutorials.blogspot.com/2011/05/table-of-contents.html
42) ARTICLE: “Google Go: A Primer”—Samuel Tesla, InfoQ Jan 2010
43) WEB SITE: “PhatGoCode”—Chris Umbel—http://www.phatgocode.com/
44) ARTICLE: J. N. White—http://jnwhiteh.net/posts/2010/09/go-examples-1-channels-and-goroutines.html

INDEX